Model checking *Circus*

Leonardo Freitas

Thesis submitted to the degree of
Doctor of Philosophy

University of York
YO10 5DD York, UK
Department of Computer Science

October 2005

To Sig and Jac,
$\Lambda o\gamma\acute{o}s + \text{'}A\nu\alpha\gamma\chi\eta \sqsubseteq \text{''}E\rho\omega\varsigma + \Psi\upsilon\chi\eta'$
Amalá Xangô, Kaô Kabiesilê

Acknowledgments

It was rather difficult for me to appositely write this part, as there are so many to mention, always with an existing story behind it. So, promising to be brief, I decided to present a chronological tale from my early years in research to present time.

My warmest acknowledgment goes to my treasured friend Dr. Adolfo Duran, when during his encouragement as a former teacher, he led me to enter the world of formal methods and academic research back in the late nineties. This fruitful experience went on during my MSc. in Brazil with Dr. Augusto Sampaio and Dr. Ana Cavalcanti, once more via suggestion and indication of Adolfo. Together with Augusto and Ana, I got to know and recognise how much I liked research in this field, as well as how to better think and focus on a task at hand. In this time, I am particularly indebted to Ana, for her patience and continuous encouragement towards my poor "abstraction" writing skills, whilst still correcting my reports and drafts carefully and attentively.

During these early days, the influence of Augusto's encouragement, and the surgical precision of Ana's comments, allowed me to build confidence to embrace the challenge of such a herculean task that a doctorate is; yet it was something I always wanted to do since even before the undergraduate course. At this point, I was immensely grateful to Ana for her indication as a PhD student to my current supervisor, as well as Augusto's suggestion for working on model checking; and a whole new journey begins.

Throughout the doctorate, working with Prof. Jim Woodcock has been a fulfilling experience. From his encompassing ability as an academic and instructor, I will always be very thankful. Even with his very busy schedule, he was able to give me the opportunity to broaden my horizons, both personally and professionally, with noteworthy expertise and commitment. Not surprisingly, I could find some of his advice in the classics that I like entertaining myself with reading, such as Immanuel Kant's Logic (*e.g.*, "a good educator does not teach thoughts; he teaches how to think") [Kan65, p.174], and Bertrand Russell's Sceptical Essays (*e.g.*, "the fundamental argument for freedom of opinion is the doubtfulness of all our beliefs") [Rus04, p.168].

For helping me with technical problems of various forms, I am very grateful to various persons without whom this work would not be possible in the first place. To Ana and Jim goes again my greatest acknowledgement for their precise and extensive commentary and incentive. Dr. Michael Goldsmith provided valuable sources for the understanding of FDR at the beginning of the work. Throughout the doctorate, Dr. Mark Saaltink offered valuable technical assistance about ***Z/Eves*** and proof planning. Ana provided invaluable advice for the operational semantics of *Circus* as well as the use of refinement laws. Dr. Peter Mosses added useful insight to solve difficulties in the operational semantics of *Circus* regarding local variables. In our research group, Marcel Oliveira was always available for discussion related to *Circus*, refinement, and another various related topics. Angela Freitas and Manuela Xavier allowed good discussions about parsing and typechecking for *Circus*. During the implementation stage, I am gratified for the helpful discussions and suggestions from Dr. Mark Utting, Dr. Petra Malik, and Dr. Tim Milner of the *Community Z Tools* (CZT) project, for their suggestions on how to integrate and extend *Circus* within the CZT Z Standard compliant framework. Rodolfo Gomez from Kent helped me to understand some difficult issues related to automata theory through profitable discussions. Together with Jim, Dr. Malcolm Wren was of great support in clarifying and explaining tricky language and writing style issues with incredible precision and patience. I am also thankful to the Universities of Kent and York for indispensable

full funding for my studies.

Finally, to emotionally cope with this strenuous journey, I fostered life-long friendships full of unforgettable moments with Dr. Alex Ferreira, Dr. Miguel Leon-Ladesma, and others at Kent. I have also carefully preserved old friendships with Adolfo, Demian Lessa, Orlando Campos, and many others from Brazil. To Helson Ramos and the *Moebius* group I am specially pleased for introducing me to another journey in the world of psychoanalysis. From Dede, Guiga, Clo, and Virgínea goes my particular acknowledgement for their everlasting source of "emotional anchor" and "safe-harbour". In the last stage of the journey since I arrived at York, I have found and nurtured love and peace of mind with Lulu!

Abstract

As software complexity increases, so does the need for precision. For some areas, such as high-integrity and safety-critical domains, this precision is imperative rather than optional. To address this issue, both academia and industry have been applying formal methods and formal verification techniques, where model checking and theorem proving are the most successful.

Model checking is a verification technique that exhaustively searches the state space of a system represented by some formal notation. It became a successful technique applied by both academia and industry, due to its high level of automation and the ability to provide counter-examples as a debugging device in the case of failure. The difficulty of applying this technique is the state explosion problem that often happens in software verification, hence making such technique unsuitable for representation by computer. In order to finitely represent infinite state systems to be analysed by computer, one needs to resort to the more powerful technique of theorem proving. It allows precise description with less compromise on the state representation, as it uses symbols and quantifiers rather than actual values. The problem is that the higher the expressiveness of a notation, the lower the level of automation that it supports and greater the demand for user expertise and interactivity.

The maturity of these techniques, as well as the wide availability of tools, pushed the demand for more expressive techniques with powerful automation tool support. Thus, combination of formalisms and their tools have become a topic of great interest in current research in formal methods. The combination of formalisms usually involve blending mature techniques that cover different aspects of software development in a common semantical framework. For instance, combining data oriented languages, such as Z and B, with behaviour oriented languages, such as CCS and CSP has been the focus of considerable research. The next step in this direction is to provide tool support for these combined languages. The combination of model checking and theorem proving have become the state-of-the-art in terms of tool development for formal verification techniques, as it combines expressiveness with high levels of automation.

In this thesis, our main goal is to provide model checking support with integrated theorem proving for *Circus*, a concurrent language for refinement that combines Z, CSP, and the refinement calculus. Its semantic model is based on Hoare and He's Unifying Theories of Programming (UTP), which provides an integrated theoretical framework for development and extension of different programming paradigms. From the partnership of our research group with QinetiQ Malvern, it is clear that there is demand for integrated formalisms and respective tool support. Our aim is to provide tool support for *Circus*, in order to allow its use in real applications, where we are able to formally specify different aspects of systems including, but not limited to, data and behaviour. As *Circus* is based in UTP, it is possible to integrate other aspects, such as mobility, and real-time, and research in these fronts is well advanced.

To fulfill our goal, we provided an operational semantics for model checking *Circus*, which enables the representation of *Circus* programs as automata, as well as a search algorithm enabling us to establish refinement between two programs. Throughout the development process, we have decided to take our own medicine and use formal specification and verification, in order to increase the levels of integrity of our tools and techniques. The semantics and the underlying automata theory has been formally defined and mechanised in the ***Z/Eves*** theorem prover. Next, we proposed a model checking architecture, which integrates theorem proving facilities, and is implemented as a model checker prototype in Java. This architecture has been formally

defined in *Circus* itself, and we augmented the Java code with JML annotations and assertions representing our findings from the formal specification. Finally, from the abstract *Circus* specification of the model checker architecture, we calculated a sequential refinement search algorithm using *Circus* refinement laws, where generated proof obligations have been discharged using ***Z/Eves*** again. This effort gave rise to a prototype model checking tool for *Circus*, which integrates refinement model checking with theorem proving in an extensible framework compliant with the Z Standard.

Declaration

I hereby declare that the research work present in this thesis is original unless otherwise indicated in the text. I have acknowledged external sources through bibliographic referencing. The material of Chapter 3 is an extended and revised version of the work published in [WCF05b]. An adapted and extended version of Chapter 5 was also accepted for a journal publication to appear in April 2006 [FW06].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Truth is a far land where mathematics is the shortest path".
Jacques Lacan Seminars

Hardware technologies and software systems are growing in complexity. Often, the requirements demand high-integrity software, and therefore a rigorous development process, guaranteed reliability, reduced errors, and well-defined documentation.

Formal methods are particularly suitable for complex and critical systems, where costs of system modelling, as well as safety concerns come into place. The tasks involved in the application of formal methods are typically the specification of the system design, formal verification of desired properties, formal proof of correctness of a design or implementation with respect to the specification, and so on [FME, FMA].

Specifying a system involves providing a precise, correct, and concise description for it. Due to the complexity involved, specification languages are normally designed to describe some specific aspects of systems. The growth in the maturity of specific methods along the years, however, motivated the idea of linking theories. The objective was to support system description with more accurate models involving several aspects of the problem. Following this idea, language integration is now a common-place. Some examples are CSP ‖ B [ST02], CSP-Z and CSP-OZ [Fis00, Fis97a, Fis97b, MS01], LOTOS and Z [BBDS97], ACT ONE LOTOS [vEVD89, LOT], RAISE [Gro95, Gro92, RTE, RUN, RFA], ProB [LB03, BL05], TCOZ [QDC03, MD00], *Circus* [WC01a, WC02], and others.

This thesis is concerned with the *Circus* language, which combines the model-based specification language Z [Spi98, WD96, Pan00], the process algebra CSP [Hoa85, Ros97, Sch00], and specification statements found in refinement calculi [Cav97, Mor94, BvW98]. Other executable commands are also available, such as assignments, conditionals, and loops [Dij76]. The semantic model of *Circus* is based on the Unifying Theories of Programming (UTP) proposed by Hoare and He [HJ98]. The result is a unified programming language that can be used for developing concurrent programs through refinement [Mor94, BvW98, WD96, Cav97, CSW02], and that also allows room for extension [Tan05, SJ02, Woo02]. Here we assume previous knowledge of Z, CSP, guarded commands, and the notion of program development through refinement.

Verifying a system requires a notion of correctness with respect to a set of desired properties. In our application domain of concurrent or reactive systems, due to the complexity of the interaction of components, establishing correctness requires enormous effort in calculations to be performed efficiently and reliably, and so tool support is imperative. The main concern is the compromise between expressiveness of a specification language, and the clarity of the result obtained with respect to the necessary human interaction. The most successful techniques are model checking and

theorem proving [Sha02].

Model checking [CGP00, McM93a, Ros94b] is an automatic verification technique that can prove properties about formal specifications through exhaustive search over possible observations, provided these can be finitely represented. The major advantages are the high level of automation, and the availability of debugging information to reason about design flaws at early stages. The main challenge is to deal with the state explosion problem that occurs due to the (concurrent) interaction of components in a design. The main approaches involve an optimised data structure to represent the system behaviours, together with an efficient exhaustive search strategy, where abstraction over the state space is always a concern. Theorem provers [MS97, Lem03] use rewriting rules and axioms of a theory. The main advantages of theorem proving are expressiveness, and the ability to handle infinite state systems via symbolic reasoning. The price to pay is that it demands expertise from users, and is normally a time-consuming task.

Formal analysis is usually divided into three stages: (i) modelling, where the informal requirements are converted to a formal specification language; (ii) specifying, where the properties the design must satisfy are given in a formal language as well; and (iii) verifying, where specified properties are formally checked for the design. The result is either a successful report guaranteeing the correctness of the design with respect to the properties, or information witnessing the nature of the failure. This witness is often either a path explaining the possible cause of failure, or an illustration of a mistake in the specification. Known as a counter-example, it is used as a debugging device to reason about failure, and to provide the basis for an appropriate solution; however, an important requirement is that a counter-example ought to be easy-to-use.

Our main objective is to provide tool support for verifying properties of *Circus* programs through the integration of model checking and theorem proving verification techniques. In doing so, we believe that applying formal methods throughout the development process is very important. For this reason, a series of techniques have been applied, including *Circus* itself, the refinement calculus, and so on. We have also used the ***Z/Eves*** theorem prover [Saa99b, MS97] throughout the development process to discharge generated proof obligations.

1.1 Motivation

There are two main model checking schools: (i) classical model checking using temporal logic [CGP00, McM93a], and (ii) refinement model checking using automata theoretic methods [Ros94b, Gol01, Ros97]. Our strategy for model checking *Circus* programs is based on the latter school, since *Circus* is strongly based on the notion of program development through refinement.

The idea of refinement model checking has been studied for many years for CSP, and successful industrial-scale tools already exist [Low97, For00, JG98, Sca92], where the most successful example is the CSP model checker FDR [Gol00]. So, why should one spend time to develop new tools in a established field? Why not adapt the *Circus* language to use such tools?

One of the major concerns when applying formal techniques is the analysis of design properties to guarantee correctness. During analysis, tool support availability sets the trend of specification languages. As we are interested in a deep combination of different languages, namely Z and CSP, in such a way that projections of specifications do not necessarily provide meaningful Z and CSP models, current tool support is

not sufficient. For the implementation of a model checker for *Circus*, we face many challenges, such as

- How to represent Z schemas without loosing their characteristic abstraction?
- How to represent predicate calculus finitely in order to allow model checking?
- How to effectively address the state explosion problem in a state-rich language?
- How to model check behavioural and data aspects of systems?
- How to maximise the levels of automation whenever theorem proving is required?

One of the greatest challenges in combining different programming paradigms is the provision of a suitable semantic model. The semantic model behind *Circus* is different from those in available tools. In Z, refinement is established via abstract data types, such as sets, being rewritten in more concrete structures, such as sequences or arrays, where operations have their preconditions weakened and their post conditions strengthened [WD96]. In CSP, refinement is established in three different set-theoretic models related to traces, nondeterminism, and divergence [Ros97]. In *Circus*, refinement is expressed as a relation of improvement between two specifications with respect to their levels of nondeterminism [CSW02]. It is given as universal reverse implication in a model of alphabetised relations [WC01a, SWC02, HJ98], which is capable of embracing both notions from CSP and Z, as well as embedded imperative features [Cav97, Mor94].

For an integrated language like *Circus*, a model checker for CSP such as FDR is not enough. It can handle behavioural aspects of a system design in terms of its interactions with the environment and parallel composition. Although FDR does have a powerful functional programming language capable of representing state operations defined in Z, *Circus* requires the abstraction and nondeterminism usually found in Z specifications. On the other hand, although theorem provers, such as ***Z/Eves*** and ProofPowerZ [Lem03], handle these Z features well, they cannot handle the behavioural aspects from CSP. Therefore, the direct use of these tools separately available for CSP and Z is not possible.

Let us present one simple example showing, some of our challenges that cannot currently be handled by FDR. Suppose we want to represent loosely defined components such as

$$size, max : \mathbb{N} \mid size \leq max$$

That is, the information we know about *size* and *max* are their types and a property relating them, rather than their actual values. This example involves looseness in Z, but a similar problem can occur in CSP. FDR cannot deal with a process such as `(c?x:T->P(x))` unless `T` is suitably bounded. FDR cannot model check a script containing loosely defined constants. Symbolic model checkers exist that directly address this problem [CCO+05a], but none of them are adapted for use with CSP.

As FDR requires explicit enumeration of values, we must find a suitable bounded abstraction of all infinite data types before we can perform any check. These abstractions are generally not easy to find and often demand either theorem proving, or a compromise in the expressiveness of the system being modelled, even at the abstract specification level.

Our work aims at providing model checking support for a subset of *Circus*; necessarily our approach has to integrate theorem proving facilities to handle refinement of state operations, or infinite/unbounded data types. Different from other approaches that try to adapt their theory to existing tools [Fis97a, ST02], we believe that there is enough theoretical understanding behind *Circus* that enables us to break this convention, and address the challenge to build a new tool capable of dealing with state-rich specifications. In summary, while model checking *Circus* specifications, theorem proving is a clear concern not yet addressed in available tools, and we are interested in solutions to this problem both in terms of impacts on the theory, or in the actual development process.

In the existing research in the specification of systems using *Circus* [WC01b, CSW02, AKW03, WC01b], due to the lack of adequate tools, adaptations and simplifications have been carried out in order to make the descriptions suitable for analysis using FDR. A list of possible application areas in which it might be interesting to use *Circus* is mentioned in [FME]. There is also interest from industry in the availability of tool support for *Circus*, so that complex systems such as control law diagrams for avionics can be analysed [CCO05b]. This shows that there is demand for tool support for *Circus*. This thesis aims at addressing this demand.

Some tools for *Circus* are already available, such as a parser [BC02], which is compliant with the Z Standard [MU05, Pan00]. Other tools for *Circus* are also under development, such as a typechecker [Xav06], a translator to Java [Fre05a], and a mechanised proof environment for the alphabetised relational calculus [Nuk05], which is to the basis of a theorem prover under development for UTP and *Circus* [Oli06].

1.2 Thesis proposition

This thesis aims at providing tool support for *Circus*, a concurrent language for refinement that combines Z, CSP, and the refinement calculus. More precisely, we want to study and provide the necessary theory and practice for refinement model checking *Circus* programs with integrated theorem proving support, where both behavioural and data aspects of systems are present.

In this direction, this thesis presents in Chapter 3 an operational semantics to represent *Circus* programs as automata. For this we mechanised a theory of automata needed to represent the state-rich aspects of the language that is present in [Fre04b]. A formalised model checking architecture with tool support for *Circus* is given in Chapter 4 and Chapter 5. The thesis demonstrates the entire development process that uses the ***Z/Eves*** theorem prover [Saa99b] for mechanisation and verification.

1.3 Objectives

As mentioned by Clarke [CGP00, CW96], there are two major concerns when performing model checking: (i) a suitable theory of automata [HMU01] that enables the appropriate representation of the language under concern; and (ii) an efficient exhaustive search strategy over that data structure, which also enables gathering information for debugging purposes. In the light of this argument, the main objective of this work is threefold: theory, practice, and tool support. On the theoretic side, we need a suitable theory of automata that satisfies all requirements for model checking a *Circus* specification, as well as a precise and correct description of an exhaustive search algorithm over those automata (see Section 4.4, and [Fre04b] for the theory of automata,

and Section 4.8 and [Fre04d] for the search algorithm). On the practical side, we need an architecture for a refinement model checker. Our proposed architecture is similar to the one used by the model checker for CSP (FDR) [Ros94b, Gol01], but also includes the necessary integration with theorem proving (see Section 4.3). Regarding tool support, we offer a *Circus* model checker prototype implemented in Java, which covers an expressive subset of *Circus* (see Sections 2.2 and 5.3.3).

To provide a formal design of a refinement model checking strategy for *Circus*, we have used *Circus* itself as a formal specification language. To represent the specifications as automata we have defined an operational semantics (see Chapter 3), which unfolds the process algebra into a specialised labelled transition system [Fre04b]. To represent our findings at the level of the implementation as much as possible, we have documented parts of the Java code with JML annotations [LBR98, LBR99, LPC+04], hence enabling extended static checking for exception freedom and partial code correctness [DLNS98, ECGN01, Hui01, BRL03]; however, further experiments on this front are beyond the scope of this thesis. The search algorithm used to check refinement was calculated from the abstract *Circus* specification of the model checking architecture using *Circus* refinement laws [CSW02, Oli06, Cav97]. We have used ***Z/Eves*** throughout these stages to formalise and mechanically discharge consistency (or domain) checks, applicability theorems, and proof obligations. The use of a theorem prover from the very beginning of the specification and refinement calculation processes ensured a greater degree of confidence in the correctness and precision of both the properties of the theory of automata and the refinement search algorithm, as well as a clearer understanding of the subtleties of the problem we were dealing with. This work also shows the expressiveness of *Circus* itself, as well as the expressiveness of using a concurrent refinement language for program development.

We illustrate our ideas in more detail using a specification in *Circus* of a connection pool acting as the communication layer between a web-server and a transaction system, which has been inspired by [Law04]. In this presentation, we exemplify some of the problems related to model checking state-rich specifications mentioned earlier, where available tools do not cover all the language features that we require.

For this thesis, we are interested in an expressive subset of *Circus*. It includes basic actions, schema expressions, all forms of communication via prefixing, guarded actions, recursion and action call, commands, such as variable declaration and assignment, sequential composition, internal and external choice, parallelism, hiding, and so on. In the next section, we further justify our choice for *Circus*.

1.4 Combining data and behaviour—why *Circus*?

In this section we briefly summarise alternative combinations of specification languages for the description of concurrent and reactive safety-critical systems. We show why *Circus* is an interesting choice for specifying such systems, and why it has Z and CSP as its basis.

1.4.1 Data related languages

Among the specification languages that focus on data aspects of systems, one can find Z, VDM [Jon90, VDM], Abstract State Machines [BS03, ASM], B [Abr96, Sch02, BA], and others. Usually, specifications in Z, VDM, and B use a model-based approach, where mathematical objects from set theory form the basis of the specifications. Nevertheless, none of these languages has specific constructs to describe behavioural as-

pects, such as choice, parallelism, sequence, and so on. Specification of behaviour in these languages is possible, but in a rather restricted manner, where, for instance, parallelism is modelled as conjunction, and choice as disjunction. Behavioural descriptions are not so clear as they would be if behavioural constructs existed.

Refinement in these language is established either through calculation or through verification. The calculational approach is based on laws to guide the proof that designs refine the more abstract specifications. Once the calculation is performed, one ends with both the derived program and its proof of correctness with respect to the original specification. In the verification approach, the designer is required to (manually) provide a (guessed) refinement of a program, and tools can later be applied to guarantee the correctness. These verification tasks can be more easily integrated with common software development processes.

1.4.2 Behaviour related languages

Among the languages primarily concerned with modelling behavioural aspects of systems, one can find Petri Nets [Pet81, PNW], CSP [Hoa85, Ros97, Sch00], CCS [Mil80, Mil90], and others. A machine readable version of CSP contains constructs and data structures supported by programming languages [Sca98]. They do not, however, support an abstract, precise, and elegant description of data aspects of systems.

An essential distinction between these languages is the way in which equivalence between processes is defined. Typically, processes are modelled using labelled transition systems (LTS) [MK99] or directed graphs [HMU01]. In CCS and Petri Nets, process equivalence is given by behavioural equivalence between LTSs (synchronisation or communication trees) through bisimulation relations. CSP, however, deals with process equivalence in a different way. In CSP, process equivalence is decided by different refinement models for traces, stable-failures, and failures-divergences [Ros97, Chapter 8], which are not primarily based on transition systems, but defined by subset inclusion on each denotational model. In CSP, the traces model is concerned with safety properties: it answers the question "what is the system expected to do?" The stable-failures model is an extension of the traces model to deal with nondeterminism. It does not take divergence into account, and is capable of answering questions such as "when does the system refuse to do something?" Finally, the failures-divergences model extends the stable-failures model with divergences, answering the question "under which circumstances is the system behaviour unpredictable or chaotic?"

1.4.3 Combined languages

We can also find languages where aspects of data and behaviour have already been combined, such as ACT ONE LOTOS [vEVD89, LOT], CSP-Z and CSP-OZ [Fis00, Fis97a, MS01, Fis97b], CSP || B [ST02, ST04], or RAISE [Gro95, Gro92, RTE, RUN, RFA]. In these languages, data and behaviour are dealt with orthogonally. We are interested in languages where data and behaviour are freely intermixed and there is a notion of refinement.

LOTOS is based on a combination of CCS with ACT ONE [EFH83]. Since CCS is based in the notion of observational equivalence, the idea of refinement is not present. Furthermore, ACT ONE does not provide a model-based style of refinement.

One of the main aims of RAISE is to integrate behavioural and data aspects into a single specification framework, and it is based on a theory of verification through refinement. The behavioural constructs of RAISE are based on Hoare's CSP [Hoa85],

whereas for the data constructs, there is a choice between an applicative style and an algebraic style, as well as facilities for under-specification. Although RAISE is based on CSP and a language similar to Z, its refinement approach is different. In program development in RAISE, the approach to refinement is to fill undefinedness, rather than to reduce nondeterminism.

Although CSP-Z and CSP-OZ are based on languages tailored for refinement, such as CSP and Z, they contain some shortcomings in the way the combined semantic models are interpreted, as pointed out in [CSW02]. Basically, the semantic model insists on identifying CSP events with Z operations that change the state, a requirement not always suitable for programming, as it is not a natural way for a language to deal with concurrency. Furthermore, one needs to choose which interpretation to give for a Z operation executing outside its precondition: either deadlock (blocking semantics) or divergence (non-blocking semantics) [Fis98]. These decisions have been made for reasons of convenience, while trying to integrate the combination of Z and CSP with FDR [MS01, Fis97b, Mot97].

In CSP || B, both languages are combined orthogonally. This allows one to use FDR and tools for B, such as the B toolkit [BT], for analysis and verification. Nevertheless, restrictions on the way B machines can be composed with CSP programs must be enforced in order to use available tools.

Other combinations closer to *Circus* are TCOZ [MD00], and ProB [LB03]. TCOZ also has its background in UTP [QDC03], and ProB does not enforce compromises on the combined language, such as in the way communications are related to operations. In TCOZ, the use of Object-Z instead of Z, compromises a compositional approach to refinement. Although ProB (version 1.1.4) has already combined CSP and B [BL05], support for refinement checking is still limited, as shown in [LBP05].

1.4.4 *Circus*

The objective of the design of *Circus* was to enable the specification of both data and behavioural aspects of concurrent and reactive systems, as well as to support stepwise development through refinement. The idea was to provide a method of program development that is based on refinement laws, and is calculational in style. Refinement laws have been proposed for *Circus* in [CSW02, SWC02, Oli06].

The choice of *Circus* as the basis for our work can be justified by its suitability for the calculational style of development of concurrent programs through stepwise refinement. It also has a strong appeal to high-integrity software development, both in industry and academia. The former occurs in a close cooperation in research projects with QinetiQ Malvern [Qin04, CCO05b], whereas the latter occurs as part of one of the grand challenges of computer science in dependable systems evolution [Woo03].

The combination of Z with a process algebra has already been studied in [Fis98]. The main reason for the selection of Z and CSP as the basis for the design of *Circus* is their notion of refinement. Furthermore, having UTP as its semantic model gives *Circus* flexibility for further extension. For instance, the UTP semantic model of *Circus* is being extended for real-time applications [SJ02], mobile processes [Tan05], object orientation [CSW05, JLL02], and others [AKW03, Nuk05]. Moreover, the semantic model of *Circus* offers a solid foundation for language understanding, reasoning, extension, and tool construction. Further up-to-date details about *Circus* can be found in its web-site [Oli03].

1.5 Overview

This section provides a reading guideline for this thesis. In Chapter 2, we briefly explain the syntax of the subset of *Circus* with which we are concerned, and present examples of *Circus* programs. Next, Chapter 3 presents the operational semantics of this subset of *Circus*, which is the theoretical basis of the *Circus* compiler (see Section 5.3.1). The compiler builds up a typechecked abstract syntax tree (AST) [WB00] from a *Circus* program given in LaTeX markup. This AST is an extension of that for the Z Standard implemented by the *Community Z Tools* (CZT) project [MU05, MFMU05]. After that, Chapter 4 presents an abstract specification for a model checking strategy for *Circus* described in *Circus* itself. Together with the operational semantics of Chapter 3, this constitutes the major theoretical contribution of this thesis. In Chapter 4, an sketch of the theory of automata, its properties, and the refinement search algorithm calculated from the abstract specification are also explained. Further details on the formal material, proofs, and refinement calculations performed are given separately in Appendix A as a CD-ROM.

Chapter 5 presents the architecture and design patterns of the model checker prototype implementation in Java. This includes a short user's guide, software engineering considerations, and some discussion about extensions. We give an overview of the process we used to build the model checker: our experience with the ***Z/Eves*** theorem prover, the use of JML to represent the formal material in Java as precisely as possible, and the application of framework design strategies [Rie00, Fre02, Sil99] and design patterns [GHJV95, HFR99, MRB98, VCK96, SSRB00, Ris00, Lea00].

Finally, in Chapter 6, we present our conclusions and summarise the work providing a comparative overview with other related tools and technologies already available. Additionally, we present our ideas for future research in the development of an efficient industrial-scale tool from the prototype, as well as some known restrictions.

Due to space constraints, the appendices in this thesis contain pointers for further material in the companion CD-ROM, such as the complete proof scripts used to prove the theorems presented in Chapters 3 and 4, an extended ***Z/Eves*** tutorial, the complete theory of automata we have developed, and so on.

Chapter 2

Circus actions

> ***"In a proposition, a thought can be expressed in such a way that elements of the proposition sign correspond to the objects of the thought precisely".***
> ***Ludwig Wittgenstein [Wit21, p.14]***

In this chapter, we briefly introduce the UTP semantic model of *Circus*, and present a subset of *Circus* through an example. The aim is to explain the key issues related to the use of the UTP as a semantic framework and the links to the semantics of *Circus*, as well as to give an example of the use of *Circus*.

In the next section, we briefly introduce the UTP and its relationship to *Circus*. Next, in Section 2.2, the *Circus* syntax is introduced and briefly explained. Section 2.3 presents an example of a specification and its properties in *Circus*. Finally, Section 2.4 presents a summary and some final considerations.

2.1 *Circus* and the UTP

The semantic model proposed for *Circus* [WC02] is based on the Unifying Theories of Programming (UTP) of Hoare and He [HJ98], where both state and communication aspects of concurrent systems are captured by observational variables. The UTP provides a single theoretical framework based on an alphabetised relational calculus that can be used for unification of many language paradigms. A theory in UTP consists of an alphabet of names, a signature of language constructs, and a set of properties (called healthiness conditions) restricting the possible relations to consider. The CSP theory, in particular, represents an enriched failures-divergences model through a set of healthiness conditions that restrict the predicates to consider, as shown in Figure 2.1. Programs, designs, and specifications are all interpreted as relations between an initial and a subsequent (intermediate or final) observation of behaviour. Through the application of appropriate healthiness conditions, one can go from the general "world" of predicates to alphabetised relations, designs with pre and postcondition specifications, reactive or CSP processes, and so on.

In the literature, there are several UTP models for different programming paradigms, such as imperative and sequential [HJ98, Chapters 5-6], reactive and parallel [HJ98, Chapters 7-8] [Woo02], higher-order and declarative [HJ98, Chapter 9], object-oriented [CSW05, JLL02], real-time [SJ02, QDC03], mobility [Tan05], and so on. Other languages have also used UTP as its semantics background [QDC03]. These theories (or paradigms) are differentiated in the framework of alphabetised relations by varying their alphabet, signature, and healthiness conditions. A brief introduction to each of these three concepts is given next. More details about the UTP, *Circus*, and its healthiness conditions can be found in [CW05, WC04, Woo05].

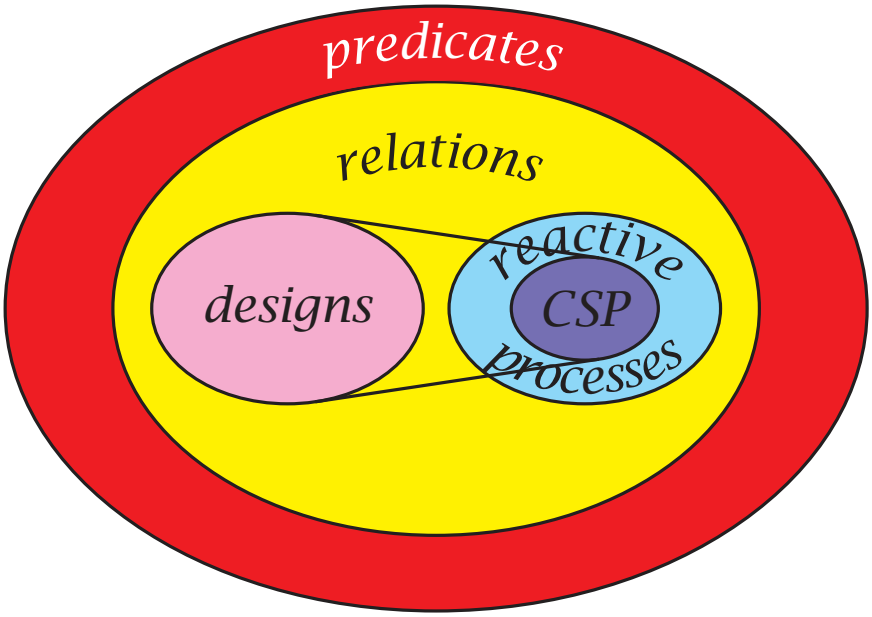

Figure 2.1: Relational calculus restrictions through healthiness conditions

An alphabet identifies observational variables whose values are relevant to characterise system behaviours. Intermediate (or final) observations are possible through the use of dashed variables representing the current (or final) value of the original initial undecorated variable. The alphabet of each theory contains variables relevant to the description of its programs. The variables relevant to the definition of *Circus* actions are summarised below.

- The boolean variable *okay* represents stabilised behaviour. In the model of designs [HJ98, Chapter 3], where there are no intermediate state, only final states, its value being ***true*** or ***false*** characterises whether the initial action has started or not. The later observation $okay'$ records whether the program has terminated. In the theory of reactive processes, there are intermediate states, where a process is waiting for interaction with its environment, as well as final states, where the process has terminated. Thus, if $okay'$ is ***true***, then either the process has terminated, or it has reached an intermediate state. On the other hand, if $okay'$ is unconstrained, then it is possible that the program may never reach a stable state; that is, it may never reach an intermediate, or a final state.

- The boolean variable *wait* is used to distinguish intermediate from final states, and it is important to describe reactive behaviour. In an intermediate state, the value of $wait'$ is ***true***, whereas in a final state, its value is ***false***. Although it is not necessary in the theory of sequential programs, for concurrent and reactive systems, where intermediate observations are relevant, this variable is crucial.

- The variable $tr \in \text{seq } \Sigma$ is a sequence of events that cumulatively records all synchronisations and communications in which a program has engaged, where Σ is the set of all visible events. The variable tr refers to the trace of events before the program has started, whereas tr' refers to trace of events at the moment where a later observation takes place.

- The variable $ref \in \mathbb{P}\,\Sigma$ represents the set of events which can be refused. It allows the analysis of responsiveness properties of an action, and is needed since not always all synchronisations or communications are possible, despite the will of the external environment. Refusal sets are relevant at intermediate waiting states. That is, ref' refers to refused events when $wait'$ holds. Also note that because of healthiness condition **CSP3** (see Table 2.1 below), the value of ref is only relevant for the sequential composition of predicates.
- The user state components correspond to a list of all other program variables v defined by either: (i) schema components, such as variables, inputs, and outputs; or (ii) local declarations, such as implicit variables, or action parameters. As before, dashed values refer to later observations.

Through these variables, it is possible to express desired features of programming languages in an elegant and concise way. Other variables representing program control, real-time clock, or resource availability can also be included as needed, but we are not interested in them for the subset of *Circus* we are working on. Specialised theories select the appropriate relevant subset of variables on the alphabet that is sufficient to represent intended behaviours. As far as our subset of *Circus* is concerned, the interesting combinations of these variables on an intermediate (or final) state are:

- Non-divergent action waiting for interaction—$okay' \land wait'$
- Non-divergent action that has terminated—$okay' \land \neg wait'$
- Divergent action—unconstrained value of $okay'$

The signature of a theory is the language syntax. The meaning of every specification is given as a predicate restricted to the selected alphabet and signature.

Healthiness conditions are given as properties that restrict predicates to reflect constraints of the systems under consideration. Healthiness conditions are useful for unification of theories, differentiation of paradigms into families, clarification of choices in a programming language design, and so forth (see Figure 2.1). The model of *Circus* satisfies at least eight healthiness conditions: the first three are related to reactive processes, and the last five to CSP. These laws are informally summarised in Table 2.1. More details on the healthiness conditions for *Circus* can be found in [HJ98, Appendix 3], [WC01a, Section 4], and [CW05].

Reactive	**Healthiness condition**
R1	An event can never be undone; *i.e.,* tr prefix tr'.
R2	Behaviour is oblivious to the past.
R3	A process does not make progress in the waiting state of another process.
CSP	**Healthiness condition**
CSP1	Behaviour of a process not yet started is unpredictable.
CSP2	One cannot require a process to abort; *i.e.,* monotonicity of the specification with respect to $okay'$.
CSP3	Outside a stable state, refusals are irrelevant.
CSP4	Refusals are irrelevant after termination.
CSP5	Refusals must be subset closed.

Table 2.1: *Circus* healthiness conditions

Refining *Circus* actions amounts to reducing their nondeterminism. In UTP, the

refinement ordering between a specification S and an implementation I, denoted by $S \sqsubseteq I$, is formally defined in terms of universal reverse implication [HJ98, Section 1.5].

$$S \sqsubseteq I \mathrel{\widehat=} \forall\, v : \alpha\, S \bullet I \Rightarrow S, \text{ provided } \alpha\, I = \alpha\, S$$

The set $\alpha\, P$ contains the variables in the alphabet of the process P. In the denotational UTP model of *Circus* [WC01a, WC02], Z itself was used as the metalanguage for a Z state-based failures-divergences model of *Circus* with embedded imperative features. In this way, every *Circus* program denotes a Z specification: each process of a *Circus* program is characterised by a definition of UTP observational variables in Z. A similar approach is taken for the definition of an operational semantics for model checking *Circus* defined in the next chapter. Apart from elegance and precision, another important practical advantage of using Z as a metalanguage is that the resulting model of *Circus* is suitable for mechanical analysis with Z theorem provers, such as ***Z/Eves*** [Saa99b] and ProofPowerZ [Lem03], as well as other Z tools such as the Jaza Z animator [Utt05].

Circus is not just a syntactic combination of formalisms. The role of the UTP as the semantic background for *Circus* is fundamental as it allows precise description of different programming paradigms in an incremental fashion, hence leaving room for extension. Since *Circus* is based on the UTP, the extension of the alphabet, signature, and set of healthiness conditions enables theoretical and practical extensions to the language, as well as an elegant, concise, and powerful reasoning framework. Thus, the UTP gives a theoretical account of the language semantics in a very expressive and extensive way. Together with the central notion of stepwise program development through refinement, this semantic expressiveness is another strong argument in favour of using the UTP as the semantic background for *Circus*, as well as the choice of *Circus* as the target language for our model checker. The UTP semantic model is a clear theoretical distinction between *Circus* and other available combined formalisms Section 1.4.3, except for TCOZ, which is also based on UTP.

2.2 *Circus* specifications

In this section, we briefly present the main constructs of a *Circus* program: it is usually formed by a series of paragraphs. Specifications of data operations are based on the use of Z constructs, and specification statements. These constructs can be combined with executable commands, like assignments, conditionals, and loops. Reactive behaviour such as communication, parallelism, and choice, are defined with the use of CSP operators. Existing combinations of Z with a process algebra model concurrent programs as communicating abstract data types [Fis98, Fis00, Fis97a], but *Circus* does not identify events with operations that change the state: Z is freely intermixed with CSP. The result is a general and extensible programming language, with a strong theoretical background, that is appropriate for developing complex systems via refinement. *Circus* has a well-defined syntax that is fully presented with clarifying examples in [CSW02, WC01a, BC02]. The BNF syntax description of the subset of *Circus* for which we implemented the prototype model checker is given in Figure 2.2. To explain the main constructs of *Circus*, we provide an example that generates the series of factorials of natural numbers in ascending order, as given in Figure 2.3.

$$
\begin{array}{lcl}
\mathsf{Program} & ::= & \mathsf{CircusPar}^* \\
\mathsf{CircusPar} & ::= & \mathsf{ZParagraph} \mid \mathsf{ChanDecl} \mid \mathsf{CSetDecl} \mid \mathsf{ProcDecl} \\
\mathsf{ChanDecl} & ::= & \textbf{channel}\ \mathsf{CDecl} \\
\mathsf{SeqCDecl} & ::= & \mathsf{CDecl} \mid \mathsf{SeqCDecl};\ \mathsf{CDecl} \\
\mathsf{CDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{ZExpr} \\
\mathsf{CSDecl} & ::= & \textbf{chanset}\ \mathsf{N} == \mathsf{CSExpr} \\
\mathsf{CSExpr} & ::= & \{|\,|\} \mid \{|\ \mathsf{N}^+\ |\} \mid \mathsf{N} \mid \mathsf{CSExpr} \setminus \mathsf{CSExpr} \\
 & \mid & \mathsf{CSExpr} \cup \mathsf{CSExpr} \mid \mathsf{CSExpr} \cap \mathsf{CSExpr} \\
\mathsf{PDecl} & ::= & \textbf{process}\ \mathsf{N}\ \widehat{=}\ \mathsf{ProcDef} \\
\mathsf{PDef} & ::= & \textbf{begin}\ \mathsf{PPar}^*\textbf{state}\ \mathsf{Schema\text{-}Expr}\ \mathsf{PPar}^* \bullet \mathsf{Act}\ \textbf{end} \\
\mathsf{PPar} & ::= & \mathsf{ZParagraph} \mid \mathsf{N}\ \widehat{=}\ \mathsf{Act} \mid \mathsf{NameSetDecl} \\
\mathsf{Act} & ::= & \mathsf{Schema\text{-}Expr} \mid \mathsf{CSPAct} \\
\mathsf{CSPAct} & ::= & \mathsf{Skip} \mid \mathsf{Stop} \mid \mathsf{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Act} \mid \mathsf{Cmd} \mid \mathsf{ZPred}\ \&\ \mathsf{Act} \\
 & \mid & \mathsf{N} \mid \mu\ \mathsf{N} \bullet \mathsf{Act} \mid \mathsf{Act}\,;\ \mathsf{Act} \mid \mathsf{Act} \sqcap \mathsf{Act} \mid \mathsf{Act} \Box \mathsf{Act} \\
 & \mid & \mathsf{Act}\ [\![\ \mathsf{NSExpr} \mid \mathsf{CSExpr} \mid \mathsf{NSExpr}\ ]\!]\ \mathsf{Act} \mid\ \mathsf{Act} \setminus \mathsf{CSExpr} \\
 & \mid & |||\ \mathsf{ZDecl} \bullet \mathsf{Act} \\
\mathsf{Cmd} & ::= & \textbf{var}\ \mathsf{x} : \mathsf{ZExpr} \bullet \mathsf{Act} \mid \mathsf{N}^+ := \mathsf{ZExpr}^+ \\
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParam}^* \\
\mathsf{CParam} & ::= & ?\,\mathsf{N} \mid ?\,\mathsf{N} : \mathsf{ZPred} \mid !\,\mathsf{ZExpr} \mid .\,\mathsf{ZExpr} \\
\mathsf{NSDecl} & ::= & \textbf{nameset}\ \mathsf{N} == \mathsf{NSExpr} \\
\mathsf{NSExpr} & ::= & \{\,\} \mid \{\,\mathsf{N}^+\,\} \mid \mathsf{N} \mid \mathsf{NSExpr} \setminus \mathsf{NSExpr} \\
 & \mid & \mathsf{NSExpr} \cup \mathsf{NSExpr} \mid \mathsf{NSExpr} \cap \mathsf{NSExpr}
\end{array}
$$

Figure 2.2: Simplified *Circus* BNF syntax

2.2.1 Programs

Programs in *Circus* are formed by zero or more elements of CircusPar, denoted by an asterisk in the BNF syntax. They can be a Z paragraph, a channel declaration, a channel set declaration, or a process definition [Ros97]. Z paragraphs are defined according to the Z Standard. As one would expect, these include given sets, schema expressions, axiomatic descriptions, and all other familiar Z constructs.

2.2.2 Channels

As in CSP, a channel defines communication points to which processes may refer. In *Circus*, however, channels are strongly typed, which means that a type restricts the possible values allowed for communication. In our example, the channel used to output the calculated factorial is declared as *out*, and its type insists that those values must belong to $\mathbb{N}$. Channels can also be declared without the definition of a type; in this case they represent synchronisation points, where no values are communicated. In the BNF, the notation N^+ represents a non-empty list of comma-separated identifiers.

Channel sets are used in expressions involving CSP operators. A channel set definition gives its name and a defining set of previously declared channels. Channel sets can be empty, channel enumerations, or set expressions involving usual set operators.

2.2.3 Processes

In our subset of *Circus*, a process definition gives its name and its explicit description mostly through action definitions. We do not consider process operators, and do not describe them here.

$$fac : \mathbb{N}_1 \pfun \mathbb{N}_1$$

$$\mathrm{dom}\ fac = \mathbb{N}_1 \setminus \{ 1 \}$$
$$\forall n : \mathbb{N}_1 \mid n > 1 \bullet fac\, n = n * fac\, (n - 1)$$

channel $out : \mathbb{N}_1$
process *Factorial* $\widehat{=}$ **begin**
state
FacState $\widehat{=}$ $[x : \mathbb{N}]$
InitFacSt $\widehat{=}$ $[FacState' \mid x' = 2]$
InitFac $\widehat{=}$ $out!1 \rightarrow out!1 \rightarrow InitFacSt$
CalcFac $\widehat{=}$ $[\Delta FacState;\ next! : \mathbb{N}_1 \mid next! = fac\, x \land x' = x + 1]$
OutFac $\widehat{=}$ $\mu X \bullet (\ \mathbf{var}\ next : \mathbb{N}_1 \bullet CalcFac\, ;\, out!next \rightarrow X\ )$
$\bullet\ InitFac\, ;\, OutFac$
end

Figure 2.3: Factorial generator for $\mathbb{N}$

Explicitly defined processes are specified by a sequence of zero, one, or more process paragraphs, as in our factorial example. These can be Z paragraphs, actions, or name set definitions. Every explicitly declared process has a distinguished nameless action at the end defining the behaviour of the entire process.

For explicitly declared processes, a Z schema may be given in order to describe the process internal state, in which case it is specially labelled with the **state** keyword. In Figure 2.3, the internal state of the *Factorial* process contains a natural number x as its single element: it represents the value of the last calculated factorial. The state has an invariant implicitly declared by the type of x being $\mathbb{N}$: it says that $(x \geq 0)$. For technical reasons related to schema text manipulation in the prototype implementation (see Chapter 5), we strongly suggest the use of maximal types with appropriate restrictions, such as $[x : \mathbb{Z} \mid x \geq 0]$, in order to achieve higher levels of automation. As the prototype has very basic rules for reasoning about schemas, this is a suggestion rather than a restriction, and is motivated by increased levels of automation. Once more powerful theorem proving is integrated, such a suggestion is no longer relevant.

2.2.4 Actions

Actions are defined within processes and have access to the process's local state. Actions can be either Z schemas, CSP operators, or guarded commands. For instance, a schema expression defining an operation over the process state is an action. It changes the state, but does not communicate any value. After the operation is performed, the behaviour of the schema action is equivalent to the primitive action Skip that successfully terminates without communicating any value. Actions defined as schema expressions diverge if executed outside their preconditions. This approach, combining state operations with communications, is different from that where communications and state operations are tightly bound in a one-to-one correspondence [Fis00, Fis97a].

In our example, after the process state, we have action definitions. They have names that other actions can refer to, and they are local to the process that contains them. Action *InitFacSt* is a schema that initialises the state. Action *InitFac* outputs

the first value corresponding to the factorial of 0 through channel *out*, initialising the process state afterwards. The next two actions define the effect of the factorial calculation upon the state and the output of the results, respectively. Action *CalcFac* calculates the next factorial value to output using function *fac* previously defined through a Z axiomatic definition. The result of the calculation is recorded in the output variable *next*!, and the action also increments the after value of x by one. Action *OutFac* performs the calculation via sequential composition of action *CalcFac*, followed by the output of the result through channel *out*, and then it recurses. The nameless action at the end describes the main behaviour of the whole *Factorial* process as the sequential composition of the actions *InitFac* and *OutFac*.

CSP actions describe the behavioural aspects of process specifications. They are composed using operators similar to those of CSP, such as sequence, recursion, communication via prefixing, external and internal choice, parallelism, interleaving, and hiding. As already said, the primitive action Skip terminates immediately without communicating any value or changing the state. The primitive action Stop deadlocks leaving the state unconstrained, whereas the action Chaos diverges: it has chaotic behaviour, and only the trace history and the state invariant are guaranteed.

Except for interleaving and parallelism, all operators are as in CSP [Ros97]. At the level of actions, the parallel operator must deal with concurrent state changes. To use the parallel operator we need to define sets partitioning all the variables in scope. These are defined using name set definitions. Similarly to channel sets definitions, a name set definition consists of a name and its defining expression. Name sets can be empty, state or local variables enumerations, or set expressions.

Actions composed in parallel can read the initial value of any variable in scope at any time, but upon termination of the parallel composition, the after-state of each action is merged according to the restrictions of the partitions in order to form the after-state of the parallel action without concurrent writing. For instance, we write $(\mathsf{A} \,[\![\, \mathsf{ns0} \mid \mathsf{cs} \mid \mathsf{ns1} \,]\!]\, \mathsf{B})$ for the parallel composition of actions A and B synchronising on the channels in the channel set cs. The state merge for parallelism ensures that actions A and B can modify only the variables in the name set ns0 and ns1 respectively. Both A and B have access to the initial value of the variables in ns0 and ns1, though. As interleaving is a kind of parallelism with an empty synchronisation set, it is therefore required for interleaving to define the state partitions of each operand as well. If any of the partitions are empty, the corresponding action cannot update the state of the parallelism. Moreover, action definitions can also have formal parameters.

In the definition of an action, all free variables have to be in the scope of the process. The state components are always into (global) scope, and input communications or schema input and output variables introduce further variables in (local) scope. The prefixing operator for communications is standard, but can be associated with a guard, which is a Z predicate. For instance, the action $(\mathsf{g} \,\&\, \mathsf{c?x} : \mathsf{P} \rightarrow \mathsf{A})$ inputs a value x through channel c, where the type of c has been restricted according to predicate P; afterwards, it assigns this value to the implicitly declared variable x, and finally behaves like action A within a scope containing x. All this occurs provided that the guard g is ***true***; otherwise, the prefixing deadlocks as the guard is an enabling condition. The input communication implicitly declares variable x with the same type of c, assigning the communicated value to that variable which is in scope for A.

As in CSP, a communication models the synchronous passage of value v through a channel c, the value of v must belong to the type T of c, and v can contain expressions with (possibly unevaluated) symbols. The question mark represents inputs, whereas the exclamation mark and the period, outputs. An input can have an associated Z

predicate that further restricts the possible values communicated. Shriek and dot are equivalent and can be used interchangeably. Replicated versions of action operators are also available in *Circus*. For the sake of simplicity, we define only a simplified version of replicated interleaving needed for the example of the next section, where the partitions on the state are all empty.

2.3 Example: connection pool

In this section, we present the specification in *Circus* of a multi-threaded connection pooling mechanism, which is part of a communication adapter between a web-server and a transaction processing system. It has been inspired by a similar CSP specification [Law04] of part of an IBM product that was analysed with FDR. The aim is to present a real system specification simple enough for presentation, and yet interesting enough for comparison with other tools and techniques.

2.3.1 Overview

The connection pool is a middleware between a web-server and a transaction processing system (IBM CICS). It allows multiple requests of service through system calls. The pool is responsible for establishing and managing a connection to the underlying resources linking the two systems. The connection management is an important aspect of the pool, as creating and destroying connections may generate undesired bottlenecks or deadlocks. Moreover, no connections should be concurrently allocated to different system calls from the web-server. The pool is responsible for: (i) keeping an adequate number of connections open; (ii) establishing links between the web-server and the transaction system; and (iii) avoiding interference between system calls on a single connection with the transaction system.

There are three possible outcomes for a system call from the web-server: (i) access to a connection is granted; (ii) an error while trying to establish the link with a connection with granted access is signalled; or (iii) access to a connection is denied due to an excessive number of already allocated connections. The pool has a limit on the number of concurrent system call requests it can handle, which is left undefined in our specification. Whenever the limit is reached, the connection pool returns a *full* response. The decision to deny access depends only on whether the pool is full or not; however, even if the pool grants a connection, the transaction system could still signal an error due to, say, an I/O problem. Since linking is allowed only when access has already been granted, an attempt to linking does raise a *full* response.

2.3.2 Connection pool specification

A graphic representation of the abstract connection pool is given in Figure 2.4. It represents a web-server making system calls to a transaction system through a connection pool middleware, which returns the status of linkage between the two systems.

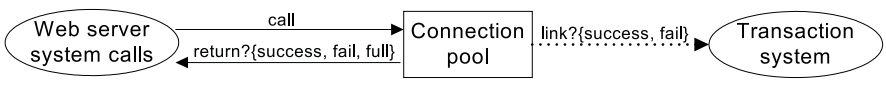

Figure 2.4: Connection pool abstract specification

Firstly, we define some data types used by the specification as Z paragraphs. System call requests are identified with elements of the *ID* given set.

$$[ID]$$

This introduces a loosely defined component that fully abstracts the concept of an identifier from our application. Although Z can be represented using FDR's functional language, here is an example FDR is not capable of representing, as FDR does not allow loosely defined components. The abbreviation *ValidIds* defines a finite set of non-empty identifiers in use; it introduces an unboundedly finite state space.

$$ValidIDs == \mathbb{F}_1\ ID$$

The abstract system has three channels: *call*, *link*, and *return*. Channel *call* is used by the web-server to request a connection to the pool, in order to link with the transaction system. Each request is identified with an appropriate *ID*.

$$\textbf{channel}\ call : ID$$

The responses from the connection pool are defined using a free-type *Response*, with one value for each possible outcome, as described above.

$$Response ::= success \mid fail \mid full$$

The channel *link* is used by the connection pool to request a link to the transaction system resource using a connection with access already granted. The channel carries a *Response* for each request made to the transaction system identified by a valid *ID*. Similarly, channel *return* carries the answer from the connection pool to the web-server about the link request to the transaction system.

$$\textbf{channel}\ link, return : ID \times Response$$

The abstract connection pool is specified by the process *PoolSpec* below. It does not have a state and can handle as many system calls concurrently as the number of *ValidIDs*. The main action is defined as the interleaving of the basic pool interface as defined by action *PoolIntf*, replicated up to the maximum number of requests allowed.

$$\begin{array}{l}
\textbf{process}\ PoolSpec\ \widehat{=}\ \textbf{begin} \\
\quad ReqGranted \ \widehat{=}\ (i : ID) \bullet call.i \rightarrow link.i?r : (r \neq full) \rightarrow \\
\qquad\qquad\qquad\qquad return.i.r \rightarrow PoolIntf\ (i) \\
\quad ReqDenied \ \widehat{=}\ (i : ID) \bullet call.i \rightarrow return.i.full \rightarrow PoolIntf\ (i) \\
\quad PoolIntf \ \widehat{=}\ (i : ID) \bullet ReqGranted\ (i) \sqcap ReqDenied\ (i) \\
\bullet\ |||\ id : ValidIDs \bullet PoolIntf\ (id) \\
\textbf{end}
\end{array}$$

The pool interface is defined by action *PoolIntf* as the nondeterministic (internal) choice between granting or denying a system call request to link to the underlying resource, for the given system call identifier i. The former models the possibility of linking with the transaction system resources, whereas the latter models the pool reaching its threshold of open connections available, when further identified requests must be rejected. For the abstract specification, this maximum number of connections to accept is left unspecified and allowed to be nondeterministically chosen.

The behaviour when access is granted is defined by the action *ReqGranted* as a sequence of three communications: (i) a system call from the web-server requesting

a connection, (ii) a request from the pool to link to an underlying resource on the transaction system, and (iii) a pool response to the web-server with the answer of the link operation from the transaction system. As access to the connection has already been granted after the system call occurs through channel *call*, the response on channel *link* can be only either *sucess* for a successfully established link, or *fail* due to an error. This restriction is specified with a predicate on the input prefixing via channel *link*, enforcing that the allowed answers from the transaction system do not include *full*. The action *ReqGranted* returns to the web-server the answer from the transaction system through channel *return*, and then behaves like the pool interface again. Denied requests are handled by action *ReqDenied*: after a system call from the web-server, where the returned response *full* is given through channel *return*, this action then recurses and behaves as the connection pool interface on the same *ID*.

Once our abstract system has been specified, we need to make sure it behaves as intended in the informal requirements. There are two simple properties of interest one can state: deadlock and divergence freedom. The former ensures the responsiveness of the connection pool, whereas the latter ensures its robustness. Just like in CSP [RSR+01], one can also use a *Circus* program to specify properties of interest.

2.3.3 Connection pool design

We describe a design for the connection pool as a multi-threaded application. Each thread is monitored by a supervisor that guarantees an efficient number of open connections. Apart from the maximum number of connections handled by the pool, the design also allows exceeding threads to be suspended and put into a bounded queue waiting to be served. It also keeps some connections permanently opened. The maximum number of system call requests that can be made is described by this extended value of extra waiting connections. Any further request beyond this boundary is rejected.

A thread models the expected behaviour of the pool from the web-server point of view: it accepts system calls for links with the transaction system, and returns each of the possible responses according to particular conditions on the state monitored by the *Supervisor* action defined below.

Definitions

Firstly, we define some further data types used by the design. The given set *ConnID* contains valid identifiers for connections. The abbreviation *ValidConnIDs* defines a non-empty finite set of connection identifiers in use; its value is left unspecified.

$$[ConnID]$$
$$ValidConnIDs == \mathbb{F}_1\ ConnId$$

The constant *maxreq* represents the maximum number of system call requests that can be made concurrently. Its value is given as the cardinality of the finite set *ValidIDs*, meaning that the maximum number of requests is bounded by the number of valid identifiers in use.

$$maxreq : \mathbb{N}_1$$
$$\hline$$
$$maxreq = \#\ ValidIDs$$

In the abstract specification we were concerned only with the maximum number of valid system call requests the web-server could make. In the design we are concerned

not only with the maximum number of requests, but also with the maximum number of established connections, the number of permanently maintained connections, and the maximum number of threads allowed to be queued waiting for the next available connection. These are specified with constants *maxconn*, *poolsize*, and *queuesize*.

$$
\begin{array}{|l}
maxconn : \mathbb{N}_1; \\
poolsize, queuesize : \mathbb{N} \\
\hline
maxconn = \# \mathit{ValidConnIDs} \land poolsize \leq maxconn \\
queuesize + poolsize \leq maxreq
\end{array}
$$

The maximum number of connections the system can handle is the number of valid connection identifiers in use: the cardinality of the finite set *ValidConnIDs*. This might be different from the number of connections to be kept open as defined by the natural number in *poolsize*. It is left unspecified with a value that must be no bigger than the maximum number of connections the pool can handle. Moreover, threads can either have an established connection, or be waiting in a queue. The queue size is modelled as a natural number that we leave unspecified. The number of opened connections plus the size of the queue for suspended threads waiting for their requests cannot be bigger than the maximum number of requests the pool design can handle.

Channels

The assembled connection pool is shown in Figure 2.5. It consists of independent threads receiving system calls on one side, and linking through connections with the transaction system on the other side, where each thread handles a connection. This behaviour is monitored by a supervisor via internal channels so that the design can meet the abstract specification.

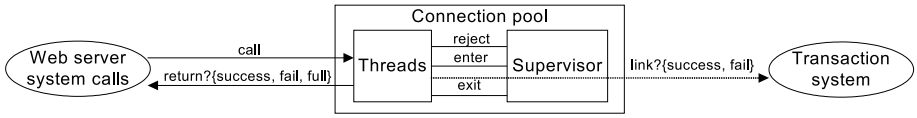

Figure 2.5: Connection pool design

Channel *enter* models an identified thread acquiring granted access for a connection from the pool, and it represents a system call from the web-server entering the connection pool for processing by the transaction system. Conversely, channel *exit* models an identified thread releasing grants from a previously held connection back to the pool. Finally, channel *reject* models an identified thread rejecting web-server requests due to either an overload of connections already in use, or too many threads waiting to be selected.

$$\textbf{channel}\ enter, exit, reject : ID$$

The channel set *Internals* defines the channels used by the supervisor for monitoring threads, whereas the channel set *Shared* defines the synchronisation interface between the threads and the supervisor.

$$
\begin{array}{lll}
\textbf{chanset}\ \mathit{Internals} & \widehat{=} & \{\!|\ enter, exit, reject\ |\!\} \\
\textbf{chanset}\ \mathit{Shared} & \widehat{=} & \mathit{Internals} \cup \{\!|\ link\ |\!\}
\end{array}
$$

The connection pool design is specified by independent threads being supervised

through the shared interface via parallel composition, where internal channels are hidden. Therefore, the only visible channels of the pool design are *call*, *return*, and *link*, as in the abstract specification.

The pool design process

The connection pool design is defined by process *PoolDesign*, which is divided in two parts: one defining the supervisor, and another defining each thread. The architecture of the design is given in Figure 2.5; each component is defined in the sequel.

$$\textbf{process}\ PoolDesign \bullet \textbf{begin}$$

The process state is formed by the finite set of active threads with granted access to connections, and the list of queued threads waiting to be served. The invariant guarantees that: (i) an active thread is not queued; (ii) the number of active and queued threads does not exceed the maximum number of connections, and the maximum queue size allowed; and (iii) if there are queued threads, then the number of active ones must be at the maximum allowed.

$$\begin{array}{|l} State \\ \hline active : \mathbb{F}\ ID \\ queue : \mathrm{seq}\ ID \\ \hline active \cap \mathrm{ran}\ queue = \varnothing \\ \#active \leq maxconn \land \#queue \leq queuesize \\ queue \neq \langle\rangle \Rightarrow \#active = maxconn \\ \hline \end{array}$$

Action *Init* specifies that initially each state component is empty.

$$Init \ \widehat{=}\ [\, State' \mid active' = \varnothing \land queue' = \langle\rangle \,]$$

In what follows, we define the behaviour of the monitoring supervisor and the concurrent threads.

Supervisor

The action *Supervisor* is defined using recursion: it indefinitely offers a choice to the environment between handling web-server system calls, or linking with the transaction system with the connection pool.

$$Supervisor \ \widehat{=}\ \mu\, X \bullet (SystemCall \,\Box\, TransactionSystem)\,;\, X$$

The action *SystemCall* models threads requesting access grant for connections. These requests can be either accepted, queued, or rejected.

$$SystemCall \ \widehat{=}\ EnterActive \,\Box\, QueueOrReject$$

A request is accepted and the calling thread becomes active provided that the active set has not reached its maximum size, as defined by action *EnterActive*. The guard compares the number of active threads with respect to the maximum number of connections allowed.

$$EnterActive \ \widehat{=}\ (\#active < maxconn)\,\&\, enter?t \rightarrow AddToActive$$

The state is updated according to the schema *AddToActive* in order to reflect the

fact that the thread identified by t is now active. This is captured by adding the chosen thread to the *active* set; the queue does not change. Note that in *Circus*, as the implicit variable t from the input on channel *enter* gets into the scope of the following action *AddToActive*, it is bound to the declaring (input) variable $t?$ from *AddToActive* with the same name.

$$AddToActive \mathrel{\widehat=} [\Delta State;\ t? : ID \mid active' = active \cup \{ t? \} \land queue' = queue]$$

The behaviour when the request is either queued or rejected is defined by action *QueueOrReject*. It is enabled whenever the number of active connections has reached the maximum number of connections allowed.

$$QueueOrReject \mathrel{\widehat=} (\#active = maxconn) \,\&\, (EnterQueue \,\Box\, Reject)$$

Provided that the active set is full and the queue has not reached its maximum value, the request is queued and the calling thread suspended, as defined by the next action.

$$EnterQueue \mathrel{\widehat=} (\#queue \neq queuesize) \,\&\, enter?t \rightarrow AddToQueue$$

The state is updated according to the schema *AddToQueue*, in order to reflect the fact that the thread t is now queued, and the active set does not change.

$$AddToQueue \mathrel{\widehat=} [\Delta State;\ t? : ID \mid active' = active \land queue' = queue \frown \langle t? \rangle]$$

If both the active set and the queue are full, the request is rejected and any calling thread t is notified through channel *reject*. This is defined by action *Reject*, which communicates the rejection for t under these conditions and successfully terminates.

$$Reject \mathrel{\widehat=} (\#queue = queuesize) \,\&\, reject?t \rightarrow Skip$$

The action *TargetSystem* models the supervisor attempt to link with the underlying transaction system and carrying our management tasks. These are the release of active connections, and the activation of queued threads.

$$TransactionSystem \mathrel{\widehat=} Linking \,\Box\, Managing$$

Links with the transaction system are established through the next action. It allows any active thread from *active* to perform a link operation and successfully terminate. Since only active threads are allowed to link, and active threads are those with granted access to connections, the response *full* meaning that the pool cannot handle the request is not allowed, and is ruled out via a predicate restricting the prefixing.

$$Linking \mathrel{\widehat=} link?t : (t \in active)?r : (r \neq full) \rightarrow Skip$$

Action *Managing* defines the management of the queue when an active thread releases its connection via the *exit* channel.

$$Managing \mathrel{\widehat=} exit?t : (t \in active) \rightarrow (UpdateActive \lor UpdateAll)$$

When the queue is empty, action *UpdateActive* removes the selected thread from the *active* set, and the queue remains empty.

UpdateActive

$$\Delta State;\ t? : ID$$

$$queue = \langle\rangle$$
$$active' = active \setminus \{ t? \} \land queue' = \langle\rangle$$

On the other hand, if the queue is not empty, action *UpdateAll* not only removes the

selected thread from the *active* set, but also includes the head of the queue in the active set and updates the queue accordingly.

$$
\begin{array}{|l}
\textit{UpdateAll} \\
\Delta\, \textit{State};\ t? : \textit{ID} \\
\hline
\textit{queue} \neq \langle\rangle \\
\textit{active}' = \textit{active} \setminus \{\, t? \,\} \cup \{\, \textit{head queue} \,\} \land \textit{queue}' = \textit{tail queue} \\
\hline
\end{array}
$$

As the last two schema expression preconditions are complementary, their disjunction represent a total operation always available for execution. This finishes the specification of the supervisor.

Threads

Action *Threads* models the behaviour of independent threads running concurrently. They are competing for access to connections in order to establish a link with the transaction system and fulfill the web-server request (see Figure 2.5). Action *Threads* is defined as the interleaving of *Thread* actions instantiated with valid thread *ID*s. Since threads do not modify the state, the partitions on the interleave operator are empty. Also, as *ValidIDs* is finite, the replicated interleaving is well-formed.

$$\textit{Threads} \ \widehat{=}\ |||\ \textit{tid} : \textit{ValidIDs} \bullet \textit{Thread}\,(\textit{tid})$$

Each *Thread* action defines the behaviour of a single thread identified by a given identifier as a formal parameter.

$$\textit{Thread} \ \widehat{=}\ (t : \textit{ID}) \bullet \textit{call}.t \rightarrow (\ \textit{Accept}\,(t) \,\Box\, \textit{TReject}\,(t)\ )\ ;\ \textit{Thread}\,(t)$$

Firstly, it accepts a system call, then a choice between granting access to establish the connection and rejecting the connection is given to the environment. The actual decision is taken by the supervisor, which runs in parallel with *Threads* when the pool is assembled below. The action *Accept* defines the behaviour when access is granted.

$$\begin{aligned}\textit{Accept} \ \widehat{=}\ (t : \textit{ID}) \bullet\ & \textit{enter}.t \rightarrow \textit{link}.t?r : (r \neq \textit{full}) \rightarrow \textit{exit}.t \rightarrow \\ & \textit{return}.t.r \rightarrow \textit{Skip}\end{aligned}$$

To request a link to the transaction system, threads must use the *enter* channel. Once access has been granted, the link to the underlying transaction system can take place via channel *link*. Whenever the link is established, the connection is released to the pool via channel *exit*. The action successfully terminates after the result of the link operation is sent to the web-server through channel *return*.

This design ensures that no concurrent access to connections can be granted at the same time to different threads. Linkage can only happen when a thread is allowed to enter the set of *active* connections via channel *enter*, as defined earlier on by the *Supervisor* action. Furthermore, this model allows other (possibly waiting) threads to have access to a connection before the current thread responds back to the web-server. The next action specifies how a thread behaves whenever a request must be rejected.

$$\textit{TReject} \ \widehat{=}\ (t : \textit{ID}) \bullet \textit{reject}.t \rightarrow \textit{return}.t.\textit{full} \rightarrow \textit{Skip}$$

The pool informs the thread that it cannot handle further requests via channel *reject*. After that, the thread informs to the web-server through channel *return* that the system call request cannot be accepted because the pool is *full*, and terminate.

Assembling the design

Action *PoolAssembly* defines the assembly of the connection pool as the parallel composition of *Threads* begin monitored by action *Supervisor*.

$$PoolAssembly \mathrel{\widehat=} \left(\begin{array}{c} Threads \\ [\![\{\,\} \mid Shared \mid \{\, active, queue \,\}]\!] \\ Supervisor \end{array} \right) \setminus Internals$$

These actions must synchronise on the shared interface defined according to channel set *Shared*. It contains all the internal channels between *Threads* and *Supervisor* plus the channel *link*. The internal channels are defined by the channel set *Internals* as *enter*, *exit*, and *reject* (see Figure 2.5). As the *Threads* action does not modify the state, which is the role of the *Supervisor* action, the partitions are defined accordingly.

Finally, the main action defines the overall behaviour of the *PoolDesign* process as the sequential composition of the action *Init* that initialises the process state, followed by the assembled pool defined by action *PoolAssembly*.

$$\bullet\ (Init \,;\, PoolAssembly)$$

end

As occurred with the abstract specification, we need to make sure the design also behaves as intended in the informal requirements by checking it for deadlock and divergence freedom.

2.3.4 Refinement

A stronger check often desirable is to prove that the design is a refinement of the abstract specification. A possible refinement to prove from the web-server point of view is that, since it does not know about the presence of a connection pool, it just makes system requests through the *call* channel and receives the transaction system response through the *return* channel. Therefore, the refinement we want to check is that the abstract specification is refined by the design, provided the *link* channel has been hidden in both processes.

$$(PoolSpec \setminus \{\!|\ link\ |\!\}) \sqsubseteq (PoolDesign \setminus \{\!|\ link\ |\!\})$$

The channel *link* must be hidden otherwise the refusal of the *Supervisor* action to engage on *link* could block another thread waiting to be queued. Moreover, hiding *link* also makes the monitoring of the *Supervisor* completely internal so that the choice between *reject* and *enter* seems nondeterministic from the web-server point of view, as it is in *PoolSpec*. In this case, the unsupervised threads can return a nondeterministic response after a sequence of *call-return* pairs.

The proof of this refinement check ensures the validity of our design with respect to informal requirements initially specified on the abstract system. In Chapter 3 we present the operational semantics explaining how these processes are translated into automata. Chapter 4 describes the algorithm to model check this example to either prove refinement or find a counter-example. Finally, in Chapter 5 we present how the previous definitions of Chapters 3 and 4 are assembled together in our *Circus* model checker prototype implemented in Java.

2.4 Summary

In this chapter, we briefly give an introduction to UTP and its role as the semantic framework for *Circus*. We also present an example of a *Circus* specification together with a design. Our subset of *Circus* is a concurrent language for refinement capable of specifying important aspects of reactive systems. For instance, it enables the definition of nondeterminism, communication, synchronisation, loops, external choice from the environment, abstraction via event concealment, and so on.

Circus is particularly useful for the design of the connection pool, as it clearly requires the specification of both behaviour and state changes. The model using FDR [Law04] applies some CSP patterns and makes use of the available functional language as an indirect way of achieving a state-rich description. We believe the *Circus* model is more intuitive, since the language has explicit support for data descriptions using Z constructs, as well as for abstraction through loosely defined components.

In the next chapter, we present an operational semantics for *Circus* which unfolds the process algebra into automata. As we shall see later in Chapter 4, the theory behind our model checker uses symbolic reasoning to perform exhaustive search for witnesses over the automata generated by the operational semantics. The compromise is (possibly) to demand theorem proving while performing model checking to discharge verification conditions generated by state operations or parallel composition.

The original example [Law04] provides a further refinement towards a real implementation in Java of the multi-threaded connection pool. We plan as future work to specify it in *Circus* and later prove that the implementation is a refinement of the design using our tool.

Chapter 3

Operational semantics

"A gramophone record, the musical idea, the written notes, and the sound waves all stand to one another in the same relation of depicting that holds between language and the world".
Ludwig Wittgenstein [Wit21, p.23]

In this chapter we present an operational semantics for *Circus* in order to define how to precisely (and formally) represent its specifications as automata. The semantics is heavily based on ideas present in the semantics for CSP_M [Sca98], the machine readable version of CSP implemented by FDR, adapted and extended as appropriate to accommodate state-rich aspects of *Circus*. Moreover, we have followed the style adopted by the denotational semantics [WC01a] and other characterisations of *Circus* [OCW05], which use Z as a metalanguage.

A Z idiom extending the Z Standard defined for the theorem prover ***Z/Eves*** is used as a metalanguage to describe data types and semantic functions for the subset of *Circus* presented in Figure 2.2. It is also used to describe the underlying automata theory (see [Fre04b]) and its properties of interest needed to implement the operational semantics of *Circus* in Java, described later in Chapter 5. Furthermore, the definitions given in this chapter have been typechecked, and checked for consistency (***Z/Eves*** domain checks) and applicability (precondition calculation) in ***Z/Eves*** [MS97].

It is important to mention that we are giving semantics to a well-formed (typechecked) *Circus* specification without name clashes and other similar problems related to scoping rules and contextual analysis. A parser for *Circus* has already been developed [BC02], as well as a typechecker [Xav06].

In the next section, we briefly present how one can link the operational semantics with other semantic models for *Circus* in UTP, hence establishing soundness between different semantic models. In Section 3.2, we present how the operational semantics is mapped to the underlying theory of automata that implements it. Next, from Sections 3.4 to 3.13 we present the operational semantics for basic actions, schema expressions, all forms of communication via prefixing, some commands such as variable declaration and assignment, guarded action, action call, simple recursion, sequential composition, internal and external choice, parallelism, event concealment through hiding, and finally, evaluation of local environments. After that, Section 3.14 presents a strategy for animating the operational semantics in ***Z/Eves***. Section 3.15 presents a comparison with FDR, and points out differences and similarities. Finally, Section 3.16 summarises typechecking information assumed from contextual analysis as well as some final considerations. Moreover, the definitions and related proofs for this chapter can be direct loaded into ***Z/Eves***, as shown in [Fre05d].

3.1 Linking theories

In this section we briefly present how the operational semantics can be linked with the semantic model of *Circus* in UTP. This, for instance, allows a proof of soundness of the operational semantics with respect to the denotational semantics of *Circus*.

An operational semantics is useful to define how programs are executed by computer. It is given as a relation establishing the execution steps of the program: a step relation between configurations. In the UTP, a configuration is a pair of program texts (s, P), where s represents the data state as a total assignment of values to names: it represents the current state of an action. P is a program text representing the remaining program to be executed in the state characterised by s. The transition relation between program texts (s, P) can be characterised in terms of the notion of refinement $(\sqsubseteq)$ in UTP as

$$(\mathsf{s}, \mathsf{P}) \xrightarrow{\varnothing} (\mathsf{t}, \mathsf{Q}) \ \widehat{=} \ (s \,;\, P) \ \sqsubseteq \ (t \,;\, Q)$$

for unlabelled transitions representing internal progress, and

$$(\mathsf{s}, \mathsf{P}) \xrightarrow{a} (\mathsf{t}, \mathsf{Q}) \ \widehat{=} \ (s \,;\, P) \ \sqsubseteq \ (s \,;\, ((\Box\, e : a \bullet e \rightarrow t \,;\, Q) \,\Box\, P))$$

for transitions labelled with the set of events representing visible communication. If an operational semantics is sound, the existence of a particular transition implies that the corresponding refinement should hold according to the denotational semantics in UTP. Although this guarantees soundness, it does not talk about completeness. That is, there might be some refinements in the denotational semantics that we do not have a transition rule defined for it. For example, it is not useful to operationally represent reflexive refinements such as

$$(s \,;\, P) \ \sqsubseteq \ (s \,;\, P)$$

since this would introduce too many spurious transitions. Considering this, one might argue that we still need an argument for sufficiency of the transition rules given in this Chapter. With such an argument we could claim that the rules available are enough to operationally represent *Circus*. This argument appears as soon as a proof of the correctness between the operational and denotational semantics has been completed. As the denotational semantics of *Circus* [Oli06] is well-advanced but still under development, we consider this proof as future work.

In $(\mathsf{s}, \mathsf{P}) \xrightarrow{\varnothing} (\mathsf{t}, \mathsf{Q})$, t is an intermediate state of P reached through internal progress $(\varnothing)$ that has started on state s and has action Q still to be executed, where P is the corresponding UTP predicate for the program text P. The semi-colon $(\,;\,)$ means sequential composition as defined in the UTP [HJ98, Section 2.2]. An important point is that the transition relation is defined between program texts, rather than their meanings. For $(\mathsf{s}, \mathsf{P}) \xrightarrow{a} (\mathsf{t}, \mathsf{Q})$, where we have visible communication, either the program makes internal progress without state changes, or some event e from the set a takes place and the configuration is updated accordingly.

Therefore, the transition relation characterises the stepwise execution of a program, where each transition represents (possibly) different behaviour observations as defined earlier. This way of presentation is useful because it enables one to link the operational semantics with respect to either algebraic [HJ98, Section 5.8], or denotational semantics [HJ98, Section 10.4] also described using the UTP. Thus, we avoid extensibility problems related to induction proofs of transition rules while reasoning about the operational semantics, as pointed out in [HJ98, Section 10.5, pp.277]. In

other words, by defining the step relation ($\_ \xrightarrow{lbl!} \_$) of the operational semantics in terms of the refinement relation ($\_ \sqsubseteq \_$) used in other semantic models, we expect to simplify the proof of correctness of the operational semantics with respect to these other semantic models.

3.2 Configuration and transition relation

In this section we define the data types used to build automata that represent *Circus* programs. We encode infinite state through symbols, and allow predicates as labels on the edges. For that, we have developed a specialised theory of automata mechanically formalised using ***Z/Eves*** (see [Fre04b]). It is the basis for the formal material presented here and in the next chapter, and it resembles standard automata theory [HMU01] with the extensions we need [CH93b].

We introduce given sets for names, and well-formed Z expressions.

$$[Name, ZExpr]$$

Types are defined using Z expressions.

$$Type == \mathbb{P}\ ZExpr$$

We also define simple Z predicates with a free-type.

$$ZPred ::= t \mid f \mid val\langle\langle\mathbb{Z}\rangle\rangle \mid expr\langle\langle ZExpr\rangle\rangle \mid not\langle\langle ZPred\rangle\rangle \mid and\langle\langle ZPred \times ZPred\rangle\rangle$$

These elements appear throughout our definitions. They are abstractly defined here as other tools can already deal with Z predicates and expressions [MU05].

Arcs

An automaton arc represents a set of events available for communication as defined by the channel types. The events are defined by the abbreviation Σ as pairs containing a channel name and the value being communicated as a *ZExpr*

$$\Sigma == Name \times ZExpr$$

where an *Arc* is a set of these pairs.

$$Arc == \mathbb{P}\ \Sigma$$

The expression Σ represents can contain unevaluated or loosely defined symbols. As we will see later, this allows us to represent finitely some sorts of infinite (or unbounded) systems through symbolic reasoning over the properties of their data types, rather than their actual values. For example, for a channel c such as

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

a valid value of Σ can be

$$\{ v : \mathbb{N} \bullet (c, v) \}$$

or simply

$$\{ (c, v) \}$$

Moreover, an empty arc ($\varnothing$) represents an unlabelled (silent) transition meaning internal activity. This is similar to FDR's approach, which represents internal progress with a special event not available in the language, named tau (τ).

Configuration

An automaton node is represented as a pair with a configuration and an environment containing declarations of channels, variables, and actions. A configuration is also a pair, containing the actual state of the process together with an action representing the remaining action to be executed. Including the syntactic program in a configuration is part of the strategy to permit lazy evaluation of the operational semantics, so that the transition system can be constructed on-the-fly, hence saving space and time.

User state

The state present in configurations is defined next by schema *USt*. It contains the user state definitions, and it is identified by a schema expression preceded by the keyword **state** as presented in Chapter 2. As we shall see later, some operators, such as variable declaration, allow the extension of the user state (see Section 3.6.1). Therefore, in order to allow the use of ***Z/Eves*** for reasoning about our semantics, we need to define the user state like an environment of names and types, where a series of properties hold between these components.

USt

$$
\begin{array}{l}
uvars : \mathrm{iseq}\ Name \\
utypes : \mathrm{seq}\ Type \\
indexOf : Name \pfun \mathbb{N}_1 \\
typeOf : Name \pfun Type \\
valueOf : Name \pfun ZExpr \\
\hline
\# uvars = \# utypes \\
\mathrm{ran}\ uvars = \mathrm{dom}\ indexOf = \mathrm{dom}\ typeOf = \mathrm{dom}\ valueOf \\
indexOf = uvars^{\sim} \\
typeOf = indexOf \fatsemi utypes \\
\forall\ n : \mathrm{dom}\ valueOf \bullet valueOf\ n \in typeOf\ n
\end{array}
$$

We use sequences *uvars* and *utypes* to refer to names and types, respectively, of the user state schema; the sequence of names is injective because there can be no components with duplicated names. We also provide three projection functions allowing the retrieval of information about each user state component. The invariant ensures that the projection functions are defined only for given names from *uvars*, hence guaranteeing that one can use only names defined in the projection functions for declared user state components. For a given name, the function *indexOf* returns the index in *uvars* for that given name; function *typeOf* retrieves the type of the given component name; and function *valueOf* gives the value of each component. The invariant ensures that all values assigned for a given name are in conformance to its declared type, which is important for keeping type-consistency during state updates.

Program text

The second element of the configuration pair is the program representing the remaining action. Actions are defined as a free-type according to the BNF syntax given in Figure 2.2. Variable declarations are formed by a name and a type.

$$VarDecl \ == \ Name \times Type$$

A schema text is a list of variable declarations and a list of predicates. This simpler

representation (that does not accept schema inclusion for example) is used for the sake of ***Z/Eves*** automation and is not present in the actual implementation.

$$SchText == \text{seq}\ VarDecl \times \text{seq}\ ZPred$$

Channel and name set expressions were simplified to a (possibly empty) sequence of *Name*. Extensions to include binary set operations, or user defined functions over these kind of expressions are straightforward and omitted here for simplicity.

$$CSExpr ::= cempty \mid clist\langle\langle \text{seq}\ Name \rangle\rangle$$

$$NSExpr ::= nempty \mid nlist\langle\langle \text{seq}\ Name \rangle\rangle$$

Prefixing contains a sequence of communication fields defining a pattern that includes both input or output, where synchronisation is represented with the empty sequence of communication fields.

$$Comm ::= in\langle\langle Name \times ZPred \rangle\rangle \mid out\langle\langle ZExpr \rangle\rangle$$

Input is formed by a variable name and a restricting predicate as in

$$\mathsf{c?x} : \mathsf{P} \equiv in\,(\mathsf{x}, \mathsf{P})$$

whereas output contains the (unevaluated) expression being output as in

$$\mathsf{c!}(x+1) \equiv out\,(x+1)$$

where loosely defined symbols are allowed. Possible actions are defined as follows.

$$\begin{aligned} Action ::= {} & \mathsf{Skip} \mid \mathsf{Stop} \mid \mathsf{Chaos} \mid sexpr\langle\langle SchText \rangle\rangle \\ \mid {} & prefixing\langle\langle (Name \times \text{seq}\ Comm) \times Action \rangle\rangle \\ \mid {} & circvar\langle\langle VarDecl \times Action \rangle\rangle \mid assign\langle\langle Name \times ZExpr \rangle\rangle \\ \mid {} & guard\langle\langle ZPred \times Action \rangle\rangle \mid call\langle\langle Name \rangle\rangle \mid rec\langle\langle Name \times Action \rangle\rangle \\ \mid {} & seqcomp\langle\langle Action \times Action \rangle\rangle \mid int\langle\langle Action \times Action \rangle\rangle \\ \mid {} & ext\langle\langle Action \times Action \rangle\rangle \\ \mid {} & par\langle\langle ((NSExpr \times Action) \times (NSExpr \times Action)) \times CSExpr \rangle\rangle \\ \mid {} & hide\langle\langle Action \times CSExpr \rangle\rangle \mid letvar\langle\langle (VarDecl \times ZExpr) \times Action \rangle\rangle \\ \mid {} & letmu\langle\langle (Name \times Action) \times Action \rangle\rangle \end{aligned}$$

Final configurations are those without defined transitions, such as Skip, Stop, guarded action with guard ***false***, and so on. We define special syntax, not available to the user, for local environments of (explicitly and implicitly) declared variables, and recursive actions. In a local environment, each tuple contains information relevant only to the scope of local declarations related to the corresponding action (see Section 3.13). Finally, a configuration is a pair formed by the user state together with the program text to be executed.

$$Config == USt \times Action$$

As a minor detail, note that we have always used binary tuples by including additional parentheses. We keep this style of specification because most operations in the Z toolkit (and indeed ***Z/Eves*** automation rules) are defined (and are more efficient) for binary tuples.

Environment

The environment is defined by a schema named *Env*, and it contains the set of declared names for channels, variables, and actions, as well as a set of fresh names. These sets form a partition on the given set of available names. The schema also contains partial functions recording information about these names.

$$
\begin{array}{|l}
\textit{Env} \\
\hline
chs, vars, acts, fresh : \mathbb{P}\ Name \\
cType, vType : Name \pfun Type \\
aCtx : Name \pfun Action \\
\hline
\langle chs, vars, acts, fresh \rangle\ \mathsf{partition}\ Name \\
\mathrm{dom}\ cType = chs \land \mathrm{dom}\ vType = vars \land \mathrm{dom}\ aCtx = acts \\
\hline
\end{array}
$$

The declared type of a channel or variable name is recorded by the functions *cType* and *vType* respectively. The function *aCtx* gives the text that is associated with an action name. The domains of these projection functions correspond to the related sets of names. Therefore, all declared channel or variable names have a corresponding type, and all declared action names have a corresponding syntax to which they refer.

Nodes and step relation

Automata nodes are given as a pair containing a configuration and an environment.

$$Node == Config \times Env$$

The transition system is defined as an infix relation between a source node configuration, a (possibly empty) set of enabling events, and a target node configuration. Moreover, the **syntax** keyword is used in ***Z/Eves*** for the definition of infix operators, in this case an infix binary relation one can refer to as `\steprel` in LaTeX.

syntax $\xrightarrow{lbl!}$ *inrel* `\_ \steprel \_`

$$\_ \xrightarrow{lbl!} \_ : Node \times Arc \leftrightarrow Node$$

As we use the schema calculus in the description of our semantics, we define the signature of a node using the next schema, where the invariant guarantees that variable names and types declared on both the user state and the node the environment are the same, hence ensuring consistency between them.

$$
\begin{array}{|l}
\textit{Sig} \\
\hline
S : USt;\ P : Action;\ E : Env \\
\hline
\mathrm{ran}\ S.uvars = E.vars \land \mathrm{ran}\ S.utypes = \mathrm{ran}\ E.vType \\
\hline
\end{array}
$$

In defining what we call a signature of a node, we are defining the configuration of our transition relation. We also capture in the definition of a signature the conditions that are required for a transition for such a configuration to be well-defined. Basically, these conditions are related to the well-formedness of the program, and that is why the environment is needed. In other words, the signature defines a restriction on the domain of the transition relation. Most of these restrictions are enforced by the typechecker. The signature for an action A, captures the restrictions of the domain of the transition relation for the configuration in which the action is A. Moreover, as

compound binary tuples on schemas can lead to longer proofs in ***Z/Eves***, we adopt a simplified idiom in *Sig*, where the (compound tuple) definition of *Node* and *Config* are explicitly expanded.

We use this schema definition to simplify the characterisation of configurations in the specification of the semantics. We need to record only the current state in configurations. That is, the after-state of a current configuration is the before-state of one of its successors, whereas the state in configurations without successors is already final. The schema *Step* defines the signature of the relation $(\_ \xrightarrow{lbl!} \_)$: all observations of before and after-states as specified by *Sig*, as well as a label for the set of events related to a transition.

$$Step \mathrel{\widehat=} [\Delta Sig;\ lbl! : Arc]$$

Next we restrict the possible steps for frequent cases. A silent step is an unlabelled transition, where the arc is empty.

$$SilentStep \mathrel{\widehat=} [Step \mid lbl! = \varnothing]$$

A read-only step represents a transition strictly related to behavioural aspects of *Circus* rather than state operations, as neither the state nor environment changes.

$$ReadOnlyStep \mathrel{\widehat=} [Step \mid S' = S \land E' = E]$$

These schemas are used throughout our description of the semantics.

Semantic functions

While defining a transition system for a *Circus* program, one could have calculated all possible maplets on the relation explicitly. In *Circus*, the set of possible configurations from an initial node are related to the immediately enabled arcs on the automaton representing the transition system. The general definition of the transition relation is given below as

$$\forall Step \bullet ((S, P), E) \xrightarrow{lbl!} ((S', P'), E')$$

That is, a defined step from action P on a before-state S, environment E, and an enabling arc $lbl!$, leads to action P' on an after-state S' and environment E', where the invariant of *Step* on the corresponding structure between the state and node environment holds.

Nevertheless, this might lead to possible space constraints in the case of large systems. Instead of performing these calculation directly, we use semantic functions to define the transition system pointwise as a set-valued function. This is the same approach taken by FDR; it is described for CSP_M in [Sca98, p.120].

Thus, like in FDR, the semantics is based on two functions, which are defined pointwise for nodes (signatures) for each kind of action. The function *enabled* defines the set of all arcs immediately available from a given node, whereas function *arcStep* defines the set of all nodes reached from the given node through some enabling arc. These functions abstractly define a general theory of automata in the style of Hopcroft [HMU01] with extensions [CH93b], where the edges are sets of events (arcs) and the nodes are pairs of a configuration and an environment.

$$enabled : Node \to \mathbb{P}\ Arc$$
$$arcStep : Node \times Arc \to \mathbb{P}\ Node$$

For simplicity, throughout proofs involving these functions, we have made them total,

where invalid configurations or enabled arcs simply lead nowhere: *enabled* returns an empty set, meaning that there are no arcs available, whereas *arcStep* returns an empty set, meaning that there are no possible next configurations. Final configurations without transitions return a similar result. Therefore, the operational semantics of *Circus* is defined by specifying these semantic functions for each operator in the BNF syntax, as defined by the *Action* free-type. This lazy approach is similar to the operational semantics of CSP_M [Sca98, Chapters 4 and 8] and its implementations [Gol00, For00], where the definitions of CSP_M processes are given in terms of similar semantic functions named *inits* and *after*. Moreover, the function *enabled* returns a set of sets as it needs to record information about individual arcs, and related properties of the underlying data types of communicated values.

New definitions in ***Z/Eves*** usually generate consistency (or domain) check conjectures that, once discharged, guarantee the consistency of the given specification. To improve the automation levels while discharging these proofs, we define additional (rather obvious or redundant) lemmas that pattern match most ***Z/Eves*** domain checks [MS97]. For instance, the next conjecture ensures that *enabled* is indeed total.

theorem rule rEnabledIsTotal
$\forall S : USt;\ E : Env;\ A : Action \bullet ((S, A), E) \in \mathrm{dom}\ enabled$

Labels such as rEnabledIsTotal introduce a named equation that can be used later in the proof of theorems; we use these labels throughout our descriptions. We use a convention on label names where prefix "`d`" stands for axiomatic descriptions; prefix "`t`" is used for conjectures (or theorems); and prefixes "`g`", "`r`", and "`f`" stand for ***Z/Eves*** automation theorems as assumption (`g`) rules, rewriting (`r`) rules, or forward (`f`) rules [Saa99b, Section 4.2]. The inscription "`rule`" tells the prover that such axiomatic definition can be used as an automatic rewriting rule, hence increasing the levels of automation. Such conjectures are proved and used throughout the chapter without further explanation. As we have used ***Z/Eves*** to check the text of this chapter, these conjectures need to be present in the order of use.

There is a relationship between the two functions *enabled* and *arcStep*, as defined by Theorem 3.1 below. The domain of *arcStep* is a relation (a set of pairs), which may be lifted to a set-valued function using relational image. This function is almost exactly *enabled*: we have to remove all pairs that *arcStep* would have mapped to the empty set, since these pairs can have no enabled arcs.

Theorem 3.1 (Relationship Between *enabled* and *arcStep*)

$$\forall\ n : Node \bullet enabled\ n = (\mathrm{dom}\,(arcStep \rhd \{\varnothing\}))(\!|\ \{n\}\ |\!)$$

The following well-formedness Theorem 3.2 is proved as a consequence of this relationship, and each definition of *enabled* and *arcStep* given below are proved to respect it. For this and other theorems, we defined yet another ***Z/Eves*** automation rule.

theorem rule rArcStepIsTotal
$\forall S : USt;\ E : Env;\ A : Action;\ a : Arc \bullet (((S, A), E), a) \in \mathrm{dom}\ arcStep$

Theorem 3.2 (Well-formedness Theorem)

$$\forall\ n : Node;\ a : Arc \bullet a \in enabled\ n \Leftrightarrow arcStep\,(n, a) \neq \varnothing$$

In words, an arc a is enabled in node n, exactly when it is possible to reach at least one target node through n via a.

Summary

The operational semantics of *Circus* is given in terms of the *enabled* and *arcStep* functions, which are presented for each available construct defined as an *Action*. In the following sections the definition of these semantic functions for the subset of *Circus* shown in Figure 2.2 is given. We start with declarations of channel, actions, and variables. Basic actions Skip, Stop, Chaos, and schema expressions come next. After that, we define two commands: variable declaration and assignment. All forms of prefixing, including synchronisation, input, output, and multi-part patterns including both input and output, are also defined. Afterwards, we define guarded actions, action call, and simple recursion. Sequential composition, internal and external choice, parallelism, and hiding are defined next. Finally, we define the local environments for variable declaration and simple recursion.

We have used ***Z/Eves*** to typecheck the formal terms of the semantics given in this chapter. We have also used ***Z/Eves*** and Jaza [Utt05] to animate a simplified version of this semantics for validation. This aid given by ***Z/Eves*** gave us a good argument for the precision of our implementation. A thorough proof of correctness with respect to other semantic models is still pending and is left as future work. This check for soundness between semantics requires a mature version of a UTP theory embedded in a theorem prover. Research on this front is already well-advanced [Oli06, OCW05, Nuk05]. Among other issues, it embeds the alphabetised relational calculus of UTP as a theory for the theorem prover ProofPowerZ [Lem03]. Eventually, this will enable us to mechanically prove the correctness of our operational semantics with respect to the denotational semantics of *Circus*.

3.3 Declarations

In the subset of *Circus* that we consider, it is possible to declare channels, actions, and local variables, and these declarations update the environment of nodes (*Env*); however, only variable declarations are part of an *Action*. Therefore, only variable declarations need to generate transitions, and so be included in the definition of functions *enabled* and *arcStep*.

Declarations of channels and actions need to be evaluated differently: they only update the node environment and do not generate transitions. One can think of these declarations as being evaluated during the contextual analysis that builds the initial node environment prior to compilation. This task is integrated with the *Circus* typechecker [Xav06] right after the typechecking analysis. We define the syntax for declaration of channels and actions using the free-type *Decl*, together with an additional ***Z/Eves*** rule about its applicability.

$$\begin{aligned} Decl \;::=\; & circchannel\langle\langle Name \times Type\rangle\rangle \mid synchchannel\langle\langle Name\rangle\rangle \\ \mid\; & circaction\langle\langle Name \times Action\rangle\rangle \end{aligned}$$

theorem rule rCircChannelIsTotal
$\forall x : Name;\ T : Type \bullet (x, T) \in \mathrm{dom}\ circchannel$

We define a special expression used as the value communicated via (untyped) synchronisation channels; it must be distinctive from any expression the user could define.

$$\mid\ Synch : ZExpr$$

This allows a homogeneous treatment of communication as a set of events from Σ: a

pair of a channel name and a corresponding expression being communicated.

Next, function *declare* is defined: it is used to update the environment with a given declaration of either a channel or an action. It is partial, since some declarations might not be well-formed, for instance, if they have already been previously defined. To include a channel in a given environment, one needs a *Name* and a *Type*, whereas to include an action one needs a *Name* and an *Action*. The function *declare* gives as result the environment obtained by updating the appropriate components of the environment taken as argument; the update is specified using relational overriding, set union, and set difference.

$$
\begin{array}{l}
declare : Env \times Decl \pfun Env \\
\hline
\dom\ declare = \\
\quad \{ E : Env;\ N : Name;\ T : Type \mid N \in E.fresh \bullet (E, circchannel\,(N, T)) \} \\
\quad \cup \{ E : Env;\ N : Name \mid N \in E.fresh \bullet (E, synchchannel\ N) \} \cup \\
\quad \{ E : Env;\ N : Name;\ A : Action \mid N \in E.fresh \bullet (E, circaction\,(N, A)) \} \\
\\
\forall\ E : Env;\ N : Name;\ T : Type \mid N \in E.fresh \bullet \\
\quad declare\,(E, circchannel\,(N, T)) = \\
\qquad \theta Env[cType := (E.cType \oplus \{ N \mapsto T \}), vType := E.vType, \\
\qquad aCtx := E.aCtx, chs := (E.chs \cup \{ N \}), vars := E.vars, \\
\qquad acts := E.acts, fresh := (E.fresh \setminus \{ N \})] \\
\\
\forall\ E : Env;\ N : Name \mid N \in E.fresh \bullet \\
\quad declare\,(E, synchchannel\ N) = \\
\qquad \theta Env[cType := (E.cType \oplus \{ N \mapsto \{ Synch \} \}), vType := E.vType, \\
\qquad aCtx := E.aCtx, chs := (E.chs \cup \{ N \}), vars := E.vars, \\
\qquad acts := E.acts, fresh := (E.fresh \setminus \{ N \})] \\
\\
\forall\ E : Env;\ N : Name;\ A : Action \mid N \in E.fresh \bullet \\
\quad declare\,(E, circaction\,(N, A)) = \\
\qquad \theta Env[cType := E.cType, vType := E.vType, \\
\qquad aCtx := (E.aCtx \oplus \{ N \mapsto A \}), chs := E.chs, \\
\qquad vars := E.vars, acts := (E.acts \cup \{ N \}), \\
\qquad fresh := (E.fresh \setminus \{ N \})]
\end{array}
$$

Quantified schema names are predicates where the schema declaration is quantified, and the schema predicates are filters. The notation $\theta SchName$ denotes the characteristic tuple of the given schema name [WD96, Sections 11.3 and 11.4]. For example, assuming the original environment (θEnv) having only one channel c of type T, its characteristic tuple is given as

$$
\begin{array}{lcl}
\theta Env & = & \langle chs \rightsquigarrow \{ \mathsf{c} \}, vars \rightsquigarrow \varnothing, acts \rightsquigarrow \varnothing, fresh \rightsquigarrow Name \setminus \{ \mathsf{c} \}, \\
 & & \quad cType \rightsquigarrow \{ \mathsf{c} \mapsto \mathsf{T} \}, vType \rightsquigarrow \varnothing, aCtx \rightsquigarrow \varnothing \rangle
\end{array}
$$

The result of declaring a new action A named X in this environment is characterised using the function *declare* as follows.

$$
\begin{array}{lcl}
declare(\theta Env, circaction\,(\mathsf{X}, \mathsf{A})) & = & \langle chs \rightsquigarrow \{ \mathsf{c} \}, vars \rightsquigarrow \varnothing, acts \rightsquigarrow \{ \mathsf{X} \}, \\
 & & \quad fresh \rightsquigarrow Name \setminus \{ \mathsf{c}, \mathsf{X} \}, \\
 & & \quad cType \rightsquigarrow \{ \mathsf{c} \mapsto \mathsf{T} \}, vType \rightsquigarrow \varnothing, \\
 & & \quad aCtx \rightsquigarrow \{ \mathsf{X} \mapsto \mathsf{A} \} \rangle
\end{array}
$$

Furthermore, ***Z/Eves***'s syntax extends the Z Standard, allowing substitution of expressions for variables; it is denoted by ":=". Thus, for this example, in the environment defined by the function *declare* exactly three components are updated: $aCtx$,

acts, and *fresh*. More generally, we give the semantics of a theta expression $\theta S[x := e]$ within a predicate P using existential quantification and standard renaming, provided y is fresh and e has the same type as x.

$$P\,(\theta S[x := e]) \equiv \exists\, y : \{\, e \,\} \bullet P(\theta S[y/x])$$

This is the interpretation ***Z/Eves*** takes for this Z extension (see [Fre04c, Section 7] for more details about ***Z/Eves*** idioms).

3.4 Basic actions

Circus has two actions that result in terminal configurations with no further transitions: Skip (the action that terminates immediately), and Stop (the action that deadlocks). It also includes unpredictable behaviour (or divergence) as action Chaos.

As terminal configurations, termination and deadlock have no after configuration, so there are no defined transitions for these actions, hence they have no state changes. Divergence represents unpredictable behaviour, where only the trace and the user state invariant is maintained. We also define transitions for schema expressions representing operations over the user state defined via the schema calculus.

3.4.1 Skip

The signature for Skip is given by the schema *SkipSig*, which defines Skip as the program of a general configuration characterised by *Sig*.

$$SkipSig \mathrel{\widehat=} [\, Sig \mid P = \mathsf{Skip} \,]$$

As it is a final configuration, there are neither arcs immediately enabled, nor transitions defined to further configurations.

$$\forall\, SkipSig \bullet enabled\,((S, P), E) = \varnothing$$

$$\forall\, SkipSig;\ lbl! : Arc \bullet arcStep\,(((S, P), E), lbl!) = \varnothing$$

Therefore, both *enabled* and *arcStep* return the empty set. This style of specification, consisting of a schema that specifies a signature with preconditions for the definition of the step transition of each action, is used for all other actions.

3.4.2 Stop

The action Stop represents deadlock and is defined similarly to Skip.

$$StopSig \mathrel{\widehat=} [\, Sig \mid P = \mathsf{Stop} \,]$$

$$\forall\, StopSig \bullet enabled((S, P), E) = \varnothing$$

$$\forall\, StopSig;\ lbl! : Arc \bullet arcStep\,(((S, P), E), lbl!) = \varnothing$$

The only difference is in the signature whose program is Stop.

3.4.3 Chaos

The action Chaos represents the possibility of unpredictable (or divergent) behaviour. We collect all actions in our *Circus* subset that could possibly introduce divergence in set *ChaoticActions* below. First, a couple of ***Z/Eves*** automation rules are needed.

theorem rule rSExprType
$\forall\, st : SchText \bullet sexpr\ st \in Action$

They provide to the prover information about maximal types and applicability. Although somewhat redundant information, these rules improve the levels of automation in a great extent [Fre04c].

theorem rule rSExprIsTotal
$\forall\, st : SchText \bullet st \in \mathrm{dom}\ sexpr$

$$SExprAct == \{ st : SchText \bullet sexpr\ st \}$$

This set of chaotic actions is formed by either Chaos, schema expressions, or event concealment via hiding, where Chaos is totally unpredictable, schema expressions diverge when executed outside their preconditions, and hiding diverges when too much is being concealed so that visible communication is no longer observed and yet the action keeps performing an unbounded sequence of silent (or internal) events.

$$ChaoticActions == \{ \mathsf{Chaos} \} \cup SExprAct \cup \{ A : Action;\ cs : CSExpr \bullet hide\,(A, cs) \}$$

For schema expressions, we prefer to use a characteristic set rather than an simpler expression such as

$$SExprAct == \mathrm{ran}\ sexpr$$

because the former is a Z idiom that is more convenient for automation with ***Z/Eves*** (see [Fre04c, Section 7]). The signature for such possibly divergent actions is given next as an extension to the general signature where the initial program belongs to the set *ChaoticActions*.

$$ChaoticSig \mathrel{\widehat=} [\, Sig \mid P \in ChaoticActions \,]$$

As divergence is treated as chaotic, we are not interested in any behaviour following its occurrence, and the only guarantee given is that the trace and the user state invariant always holds. Therefore, we define *ChaoticStep* to allow any possible after-state in which the program is still chaotic; this corresponds to a loop in the transition relation via a silent transition. This characterisation of divergence as a loop of silent transitions is similar to FDR's characterisation via τ-loops.

$$ChaoticStep \mathrel{\widehat=} [\, ChaoticSig;\ SilentStep \mid P = P' \,]$$

These definitions are reused later for the specification of other divergent cases.

Let us now define the action Chaos, which allows divergent behaviour through any arc *enabled* from a *ChaoticSig* starting from Chaos.

$$ChaosSig \mathrel{\widehat=} [\, ChaoticSig \mid P = \mathsf{Chaos} \,]$$

$$\forall\ ChaosSig \bullet enabled\,((S, P), E) = Arc$$

For the sake of implementation on a computer, we need to choose a finite representation for the enabling events of Chaos. This defines how the operational semantics is finally encoded into automata. The transition for Chaos is defined next as the chaotic

step where the original program is Chaos, as defined by schema *ChaosSig*.

$$ChaosStep \mathrel{\widehat=} ChaosSig \land ChaoticStep$$

$$\forall\ ChaosStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

Finally, we prove an applicability conjecture which says that we can always perform a *ChaosStep* for program Chaos as defined in *ChaosSig*.

theorem tChaosStepAppl
$\forall\ ChaosSig \bullet \text{pre}\ ChaosStep$

These applicability theorems are useful to point out the necessary (pre)conditions for each step, and we use them throughout our description for other actions to ensure the transitions are covering all cases. At first, while trying to prove conjecture, the prover pointed out the minimum necessary (or weakest) precondition to add as *ChaosSig* as

$$\forall\ Sig \bullet \text{pre}\ ChaosStep$$

Furthermore, encoding the pre/post conditions through schemas in this way is also helpful for automatic translation tools such `z2jml` [MU05][1], which could transform Z specifications into Java code annotations for extended static checking and documentation [DLNS98, Hui01, ECGN01], where checks such as exception freedom and partial program correctness can be performed. For more details on the use of JML in our Java implementation of the model checker see Section 5.2.3. Note that *Circus* Chaos is like Hoare's [Hoa85] *CHAOS* (or Roscoe's **div**) rather than FDR's *CHAOS*. These and other differences are collected and explained in Section 3.15.

3.4.4 Schema expression—SExpr $\mathrel{\widehat=}$ [*Decls* | *Preds*]

For schema expressions and other actions, we abstractly define additional functions for evaluating Z predicates and expressions in the current user state. These functions are implemented in the prototype using a layered architecture which enables integration with external tools such as theorem provers and SAT solvers (see Section 5.3.3).

$$evalP : USt \times ZPred \to ZPred$$
$$evalE : USt \times ZExpr \to ZExpr$$

We also abstractly define functions representing operations over a schema expression, such as precondition calculation, and update of the components of *USt* and *Env* from the text of a given schema expression action.

$$pre : SExprAct \to ZPred$$
$$updUSt : USt \times SExprAct \to USt$$
$$updEnv : Env \times SExprAct \to Env$$
$$ioExt : USt \times SExprAct \to USt$$

The function *pre* returns the corresponding Z predicate (expression) for a given schema action text. The functions *updUSt*, *ioExt*, and *updEnv* modify the user state and the environment according to the definition present in the given schema action text, respectively. A naive implementation for the effect of *updUSt* over *USt* can be simply

[1] This tool is still a prototype; a similar tool that translates Z to B already exists.

schema composition. In the prototype, we resort to a more clever approach using accumulated modifications inspired by *Concurrent Versioning System* (CVS) utilities [BF04]. For the functions *ioExt* and *updEnv* the effect on the user state and environment is just to include variable declarations; however, the function *ioExt* is particularly tailored for inclusion of input and output variables from schemas. Furthermore, in order to calculate the precondition of a schema expression, both input and output variables must be put in the context of the given user state. This task is performed using the *ioExt* function. As before, we include some additional rules about these functions to improve ***Z/Eves*** automation.

theorem rule rEvalPIsTotal
$\forall S : USt;\ P : ZPred \bullet (S, P) \in \text{dom } evalP$

theorem rule rEvalEIsTotal
$\forall S : USt;\ E : ZExpr \bullet (S, E) \in \text{dom } evalE$

theorem rule rIOExtResult
$\forall S : USt;\ A : SExprAct \bullet ioExt\,(S, A) \in USt$

theorem rule rIOExtIsTotal
$\forall S : USt;\ A : SExprAct \bullet (S, A) \in \text{dom } ioExt$

theorem rule rPreResult
$\forall A : SExprAct \bullet pre\ A \in ZPred$

theorem rule rPreIsTotal
$\forall A : SExprAct \bullet A \in \text{dom } pre$

theorem rule rUpdUStIsTotal
$\forall S : USt;\ A : SExprAct \bullet (S, A) \in \text{dom } updUSt$

theorem rule rUpdEnvIsTotal
$\forall E : Env;\ A : SExprAct \bullet (E, A) \in \text{dom } updEnv$

The signature for schema expressions is defined below; the initial program P belongs to the set *SExprAct* of well-formed schema expressions, where we have two cases to consider depending on whether the precondition is ***true*** or ***false***. In the case where the precondition holds, it is also required that the result of the update functions keep the invariant of *Sig* related to the correspondence between variable names in the state and in the environment.

SExprTrueSig
Sig

$P \in SExprAct$
$evalP\,(ioExt\,(S, P), pre\ P) = t$
$\text{ran}\,((updUSt\,(ioExt\,(S, P), P)).uvars) = (updEnv\,(E, P)).vars$
$\text{ran}\,((updUSt\,(ioExt\,(S, P), P)).utypes) = \text{ran}\,((updEnv\,(E, P)).vType)$

These conditions came through the (failed) proof of the applicability theorem for this

case mentioned below. For the first case, the evaluation of the precondition generates a verification condition to be discharged in order for this transition on the automata to be enabled, namely, the precondition calculation through the application of function *evalP* on a state extended with input and output variables through function *ioExt*.

$$
\begin{array}{|l}
\textit{SExprFalseSig} \\
\textit{ChaoticSig} \\
\hline
P \in \textit{SExprAct} \land \textit{evalP}\,(\textit{ioExt}\,(S, P), \textit{pre}\,P) = f \\
\hline
\end{array}
$$

Schema expressions where the precondition is ***true*** have only a silent transition via an empty arc.

$$\forall\ \textit{SExprTrueSig} \bullet \textit{enabled}\,((S, P), E) = \{\, \varnothing \,\}$$

Otherwise, if the precondition evaluates to ***false*** in the current state, a schema expression diverges enabling all arcs, just like Chaos.

$$\forall\ \textit{SExprFalseSig} \bullet \textit{enabled}\,((S, P), E) = \textit{Arc}$$

Provided the precondition evaluates to ***true*** in the current state, a schema expression successfully terminates leading to Skip via a silent arc. This silent transition is important as it captures the update of the user state and node environment. Such update is specified via the functions defined above.

$$
\begin{array}{|l}
\textit{SExprStepPre} \\
\textit{SExprTrueSig};\ \textit{SilentStep} \\
\hline
S' = \textit{updUSt}\,(\textit{ioExt}\,(S, P), P) \\
E' = \textit{updEnv}\,(E, P) \\
P' = \mathsf{Skip} \\
\hline
\end{array}
$$

The update on the state is defined by *updUSt* after extension of the user state to (possibly) include declaring input and output variables. Similarly, the environment is updated via the function *updEnv*. On the other hand, when the precondition evaluates to ***false***, the transition step for schema expressions creates a loop on a silent transition, similarly to what was defined for Chaos via the transition defined by schema *ChaoticStep*.

$$\textit{SExprStepNotPre} \mathrel{\widehat=} \textit{SExprFalseSig} \land \textit{ChaoticStep}$$

The complete definition for SExpr is given next as disjunction of each case.

$$\textit{SExprSig} \mathrel{\widehat=} \textit{SExprTrueSig} \lor \textit{SExprFalseSig}$$

$$\textit{SExprStep} \mathrel{\widehat=} \textit{SExprStepPre} \lor \textit{SExprStepNotPre}$$

$$\forall\ \textit{SExprStep} \bullet \textit{arcStep}\,(((S, P), E), \textit{lbl}!) = \{\, ((S', P'), E') \,\}$$

For the proof of the applicability conjecture, we need further ***Z/Eves*** rules below. Forward rules (frule) are used for exposing (sub)types of schema components. Assumption rules (grule) are used by all forms of rewriting tactics available. They are useful for discharging subgoals related to maximal types of involved expressions, as

they often appear in proofs. Rewriting rules (rule) are similar to assumption rules, but apply for a smaller set of tactics, hence allows a higher degree of control of the prover when applying different proof tactics.

theorem frule fSExprSigAType
$SExprSig \Rightarrow P \in SExprAct$

theorem grule gSigmaMaxType
$\Sigma \in \mathbb{P}\,(Name \times ZExpr)$

theorem rule rUpdUStResult
$\forall\, S : USt;\ A : SExprAct \bullet updUSt\,(S, A) \in USt$

theorem rule rUpdEnvResult
$\forall\, E : Env;\ A : SExprAct \bullet updEnv\,(E, A) \in Env$

theorem rule rUpdUStMaxResult
$\forall\, S : USt;\ A : SExprAct \bullet updUSt\,(S, A) \in$
$\quad \langle\!|\ indexOf : \mathbb{P}\,(Name \times \mathbb{Z});\ typeOf : \mathbb{P}\,(Name \times \mathbb{P}\ ZExpr);$
$\quad utypes : \mathbb{P}\,(\mathbb{Z} \times \mathbb{P}\ ZExpr);\ uvars : \mathbb{P}\,(\mathbb{Z} \times Name);$
$\quad valueOf : \mathbb{P}\,(Name \times ZExpr)\ |\!\rangle$

theorem rule rUpdEnvMaxResult
$\forall\, E : Env;\ A : SExprAct \bullet updEnv\,(E, A) \in$
$\quad \langle\!|\ aCtx : \mathbb{P}\,(Name \times Action);\ acts : \mathbb{P}\ Name;$
$\quad cType : \mathbb{P}\,(Name \times \mathbb{P}\ ZExpr);\ chs : \mathbb{P}\ Name;\ fresh : \mathbb{P}\ Name;$
$\quad vType : \mathbb{P}\,(Name \times \mathbb{P}\ ZExpr);\ vars : \mathbb{P}\ Name\ |\!\rangle$

theorem tSExprStepAppl
$\forall\ SExprSig \bullet \text{pre}\ SExprStep$

Details about proofs for these theorems can be found in [Fre05d]. As we analysed the source of this chapter with ***Z/Eves***, they must appear in the order they are needed.

3.5 Prefixing

In this section, we define all sorts of communication via prefixing. One can view communication via prefixing as a two-stage operation. Firstly, the communication takes place without user state changes leading to the following action remaining to be executed. Finally, this action is evaluated at a later stage to describe the entire behaviour of the prefixing. Operationally, prefixing can be seen as its equivalent sequential version in UTP [HJ98, Definition 8.2.5, pp.209] algebraically defined as

$$\mathsf{a} \rightarrow \mathsf{P} \ \widehat{=}\ (\mathsf{a} \rightarrow \mathsf{Skip})\,;\,\mathsf{P}$$

An important aspect on this style of evaluation is to allow lazy implementation of the external choice (see Section 3.10.2) and parallel composition (see Section 3.11)

operators. Firstly, we define the signature *UnSig* for all unary actions containing a following action A.

$$UnSig \mathrel{\widehat=} [\, Sig;\ A : Action \,]$$

The prefixing signature is defined next as a unary action extended with a valid channel (present in the environment) for the communication, where the current program is a prefixing action.

$$PrefixSig \mathrel{\widehat=} [\, UnSig;\ \mathsf{c} : Name \mid \mathsf{c} \in E.chs \land P \in \operatorname{ran} prefixing \,]$$

The way in which the channel c is used in the prefixing is defined later on, for each kind of prefixing. The additional type and scope checking information about channel c is needed for the proof of theorems in ***Z/Eves***. These checks are implemented directly by the *Circus* typechecker [Xav06], and are present in the Java implementation described in Chapter 5 as JML annotations only, rather than in the actual code. This exercise with ***Z/Eves*** is a good example on how our tools (not just the model checker) can benefit from mechanised specification and verification.

For all prefixing transitions, neither user state changes nor silent transitions are allowed, as defined by the schema *PrefixStep*. Moreover, prefixing with empty labels are those which are waiting due to the need to synchronise, or have a too restrictive input communication predicate.

$$PrefixStep \mathrel{\widehat=} [\, PrefixSig;\ ReadOnlyStep \mid lbl! \neq \varnothing \,]$$

For satisfying the predicate $(lbl! \neq \varnothing)$, we need to include additional verification conditions in the prefixing signatures presented below.

theorem frule fPrefixSigOSCType
$PrefixSig \Rightarrow \mathsf{c} \in \operatorname{dom} E.cType$

theorem frule fPrefixSigCTypeType
$PrefixSig \Rightarrow E.cType \in Name \pfun \mathbb{P}\, ZExpr$

We also include additional ***Z/Eves*** rules exposing the maximal types of some environment components needed in proofs involving the prefixing signature.

3.5.1 Synchronisation—$(\mathsf{c} \rightarrow \mathsf{A})$

The signature for synchronisation insists that the channel type is the set containing the synchronisation expression only, and the prefixing is has no communication fields. As there is no value being communicated, typechecking on c is enough for satisfying the restriction of *PrefixStep* that $(lbl! \neq \varnothing)$, and hence no additional verification condition is needed.

SynchPrefixSig
$PrefixSig$

$E.cType\ \mathsf{c} = \{\, Synch \,\} \land P = prefixing\ ((\mathsf{c}, \langle\rangle), A)$

Synchronisation performs a communication on the given channel without extra information exchange. Since events are defined as a tuple containing the channel name

and the value being communicated, synchronisation is initially offering only an arc with a single event $(c, Synch)$.

$$\forall\ SynchPrefixSig \bullet enabled\ ((S, P), E) = \{ \{ (\mathsf{c}, Synch) \} \}$$

SynchPrefixStep

$$SynchPrefixSig;\ PrefixStep$$

$$P' = A \land lbl! = \{ (\mathsf{c}, Synch) \}$$

$$\forall\ SynchPrefixStep \bullet arcStep\ (((S, P), E), lbl!) = \{ ((S', P'), E') \}$$

theorem tSynchPrefixStepAppl
$\forall\ SynchPrefixSig \bullet \text{pre}\ SynchPrefixStep$

On this occasion, the applicability theorem was helpful to indicate the restriction on the type of channel c in the environment.

3.5.2 Output prefixing—$(\mathsf{c!e} \rightarrow \mathsf{A})$ or $(\mathsf{c.e} \rightarrow \mathsf{A})$

Output prefixing is defined similarly to synchronisation; the difference is on the communication fields now containing the expression e as an output field.

$$OutPrefixSig \mathrel{\widehat=} [\, PrefixSig;\ \mathsf{e} : ZExpr \mid P = prefixing\ ((\mathsf{c}, \langle out\ \mathsf{e} \rangle), A) \,]$$

Moreover, we need an additional verification condition about the possible value of e still satisfying the type of c in the environment.

$$OutPrefixComm \mathrel{\widehat=} [\, OutPrefixSig \mid \mathsf{e} \in E.cType\ \mathsf{c} \,]$$

This "dynamic" typecheck is important to handle specifications such as

channel $c : \{0, 1, 2\}$
$\ldots$
$Action \mathrel{\widehat=} c!3 \rightarrow Skip$

The transition is defined similarly with the appropriate arc, where expression e is evaluated under the current user state.

$$\forall\ OutPrefixComm \bullet enabled\ ((S, P), E) = \{ \{ (\mathsf{c}, evalE\ (S, \mathsf{e})) \} \}$$

OutPrefixStep

$$OutPrefixComm;\ PrefixStep$$

$$P' = A \land lbl! = \{ (\mathsf{c}, evalE\ (S, \mathsf{e})) \}$$

$$\forall\ OutPrefixStep \bullet arcStep\ (((S, P), E), lbl!) = \{ ((S', P'), E') \}$$

The applicability theorem for the step case depends on the output prefixing commu-

nication taking place.

theorem tOutPrefixStepAppl
$\forall$ *OutPrefixComm* • pre *OutPrefixStep*

On the other hand, if the verification condition does not hold, just like Stop, the output prefixing deadlocks, since there is no enabled event satisfying the signature for output prefixing.

$$OutPrefixStop \mathrel{\widehat=} [\, OutPrefixSig \mid \neg OutPrefixComm \,]$$

$$\forall\ OutPrefixStop \bullet enabled\ ((S, P), E) = \varnothing$$

$$\forall\ OutPrefixStop;\ lbl! : Arc \bullet arcStep\ (((S, P), E), lbl!) = \varnothing$$

An interesting feature of expression (and predicate) evaluation is that we allow loosely defined variables to be present after evaluation. This encodes the properties related to the loose variable within the acceptable arcs and transitions, hence generating verification conditions modulo such properties. For example, if after evaluation of an expression such as

$$\mathsf{e} = \mathsf{x} - 1\,,\ \text{where}\ \ \mathsf{x} \in S.uvars \land S.typeOf\ \mathsf{x} \in \mathbb{N}$$

we still leave x loosely defined, this generates a verification condition, such as in the calculation of the semantics of a prefixing $c!(x-1) \rightarrow Skip$.

$$x - 1 \in \mathbb{N} \ \equiv\ x - 1 \in \{\, n : \mathbb{Z} \mid n \geq 0 \,\} \ \equiv\ x - 1 \geq 0 \ \equiv\ x \geq 1$$

We use a theorem prover module in the model checker to support the evaluation of expressions for the calculation of the automata, as well as the verifications of refinement (see Figures 5.1 and 5.5 in Chapter 5).

3.5.3 Input prefixing—(c?x : P → A)

The enabled arcs of input prefixing contains events formed by the channel name, and all values allowed by the declared channel type filtered by predicate P. As a new variable (x ∈ *cType* c) is implicitly declared to allow A to have access to the value communicated on x, its name must be fresh in the environment and not present in the user state.

InPrefixSig
PrefixSig; x : *Name*; P : *ZPred*

$\mathsf{x} \in E.fresh \land \mathsf{x} \notin \text{ran}\ S.uvars$
$P = prefixing\ ((\mathsf{c}, \langle in\ (\mathsf{x}, \mathsf{P})\rangle), A)$

Moreover, the restricting predicate P must not evaluate to ***false***, otherwise this would mean an empty set of possible events enabled, hence a deadlocked prefixing refusing to communicate anything. The last restriction that the type of the declared channel

is also not empty is needed for similar reasons: if there is no event enabled in the type of c, the prefixing is deadlocked.

$$
\begin{array}{|l}
\textit{InPrefixComm} \\
\textit{InPrefixSig} \\
\hline
\textit{evalP}\,(S, \mathsf{P}) \neq f \\
E.\textit{cType}\,\mathsf{c} \neq \varnothing \\
\hline
\end{array}
$$

This is another example of additional information included due to mechanical formalisation. The enabled arc is defined as all values e within the type of c, where the evaluation of predicate P is not too restrictive. As we do not expand input prefixing, and allow it to be defined by set comprehension, it is possible to encode infinite data types in a finite automaton via symbolic constants.

$$
\begin{array}{l}
\forall\ \textit{InPrefixComm} \bullet \textit{enabled}\,((S, P), E) = \\
\quad \{\, \{\, \mathsf{v} : E.\textit{cType}\,\mathsf{c} \mid \textit{evalP}\,(S, \mathsf{P}) \neq f \bullet (\mathsf{c}, \textit{evalE}\,(S, \mathsf{v}))\,\}\,\}
\end{array}
$$

For *arcStep*, the label is the same singleton element given in *enabled*.

$$
\textit{lbl}! = \{\, \mathsf{v} : E.\textit{cType}\,\mathsf{c} \mid \textit{evalP}\,(S, \mathsf{P}) \neq f \bullet (\mathsf{c}, \textit{evalE}\,(S, \mathsf{v}))\,\}
$$

Furthermore, the actual input value communicated from *lbl*! must be available during evaluation of the following action A. To implement scope for such local variables, we need a special action representing a local environment with a tuple containing the variable name x, its type T, and its actual expression e. It augments the current environment during the evaluation of A (see Section 3.13.1). This choice of having special syntax for local environment is standard for representing evaluation of scope and local variables in structured operational semantics [Mos04a, Mos04b]. This approach has two useful consequences in our case: it allows an intuitive description of sequential composition that is closer to the denotational semantics; and it makes implicit declaration of variables in input prefixing easier to handle. The transition ensures that the next program has a local environment for the variable in the communication, and that the available events are in accordance with the ones returned by *enabled*. This incurs in the implicit definition of a locally defined variable x within the range of possible expressions from the available arc

$$
P' = \textit{letvar}\,(((\mathsf{x}, \operatorname{ran}\ \textit{lbl}!), \textit{second}\,(\textit{elem}\,(\textit{lbl}!))\,), A)
$$

where the type of x contains the available value, which is chosen as the appropriate projection over any selected element from *lbl*!.

$$
\textit{lbl}! \in \mathbb{P}\ \Sigma \ \Rightarrow\ \textit{lbl}! \in \mathbb{P}\,(\textit{Name} \times \textit{ZExpr}) \ \Rightarrow\ \textit{second}\,(\textit{elem}\,(\textit{lbl}!)) \in \textit{ZExpr}
$$

To choose such an element from *lbl*!, we define the next function, which nondeterministically chooses an element from a given set of any type.

$$
\begin{array}{|l}
[X] \\
\hline
\textit{elem} : \mathbb{P}\ X \to X \\
\hline
\forall\, S : \mathbb{P}\ X \bullet \exists\, x : X \mid x \in S \bullet \textit{elem}\ S = x \\
\hline
\end{array}
$$

As we use Java as our target implementation language, element selection from a set has already been implemented for the case of enumerable sets. For sets with unevaluated symbols or defined by comprehension, theorem proving might be necessary. For

the consistency and applicability theorem proofs involving input prefixing, we need additional ***Z/Eves*** rules about the returned types of the projection functions.

theorem frule fPrefixStepRanLblType
$PrefixStep \Rightarrow \mathrm{ran}\ lbl! \in Type$

theorem frule fPrefixStepProjtLvlElemType
$PrefixStep \Rightarrow second\,(elem\ lbl!) \in ZExpr$

theorem rule rLetVarType
$\forall \mathsf{x} : Name;\ \mathsf{T} : Type;\ \mathsf{v} : ZExpr;\ A : Action \bullet letvar\,(((\mathsf{x}, \mathsf{T}), \mathsf{v}), A) \in Action$

theorem rule rLetVarIsTotal
$\forall \mathsf{x} : Name;\ \mathsf{T} : Type;\ \mathsf{v} : ZExpr;\ A : Action \bullet (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A) \in \mathrm{dom}\ letvar$

InPrefixStep
$InPrefixComm;\ PrefixStep$

$P' = letvar\,(((\mathsf{x}, \mathrm{ran}\ lbl!), second\,(elem\,(lbl!))), A)$
$lbl! = \{\, \mathsf{v} : E.cType\,\mathsf{c} \mid evalP\,(S, \mathsf{P}) \neq f \bullet (\mathsf{c}, evalE\,(S, \mathsf{v}))\,\}$

$\forall\ InPrefixStep \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E')\,\}$

Further ***Z/Eves*** rules about the type of expressions involved are needed for the applicability theorem proof that are given below.

theorem rule rValidValueType
$\forall E : Env;\ \mathsf{c} : Name \mid \mathsf{c} \in E.chs \land \mathsf{v} \in E.cType\,\mathsf{c} \bullet \mathsf{v} \in ZExpr$

theorem rule rInPrefixSigLblType
$\forall S : USt;\ E : Env;\ \mathsf{c} : Name \mid \mathsf{c} \in E.chs \bullet$
$\{\, \mathsf{v} : E.cType\,\mathsf{c} \mid true \bullet (\mathsf{c}, evalE\,(S, \mathsf{v}))\,\} \in (Name \leftrightarrow ZExpr)$

theorem tInPrefixStepAppl
$\forall InPrefixComm \bullet \mathrm{pre}\ InPrefixStep$

On the other hand, if the verification condition does not hold, the prefixing deadlocks since there is no enabled event available.

$InPrefixStop\ \widehat{=}\ [\, InPrefixSig \mid \neg InPrefixComm\,]$

$\forall\ InPrefixStop \bullet enabled\,((S, P), E) = \varnothing$

$\forall\ InPrefixStop;\ lbl! : Arc \bullet arcStep\,(((S, P), E), lbl!) = \varnothing$

For more details on local environments for implicitly declared variables are defined in Section 3.13.1.

3.5.4 Multi-part prefixing – *e.g.*, $(\mathsf{c?x : P!y \rightarrow A}), (\mathsf{c!x!y?z : P \rightarrow A})$

Multi-part prefixing is the most general pattern allowed for communication fields in prefixing actions. It allows input and output fields to be intermixed in a style similar to Roscoe's CSP (and FDR's CSP_M) [Ros97, p.27]. Firstly, we define the set of valid communication patterns as those within the *ValidComm* set below.

$$\begin{aligned} ValidComm \;==\; & \{\, S : USt;\; E : Env;\; c : Name;\; \mathsf{v} : ZExpr \mid c \in E.chs \land \\ & \quad \mathsf{v} \in E.cType\ c \bullet (S, ((E, c), out\ \mathsf{v}))\,\} \cup \\ & \{\, S : USt;\; E : Env;\; c, \mathsf{x} : Name;\; \mathsf{P} : ZPred \mid c \in E.chs \land \\ & \quad \mathsf{x} \in E.fresh \land \mathsf{x} \notin \operatorname{ran} S.uvars \bullet (S, ((E, c), in\ (\mathsf{x}, \mathsf{P})))\,\} \end{aligned}$$

It restricts input and output communication fields according to the preconditions related to declarations from the environment and user state, similarly as defined by schemas *OutPrefixSig* and *InPrefixSig*. More restrictive information can be added in a similar way if needed, for instance, to restrict the allowed patterns of interest for multi-part prefixing. The set of valid mixed communication fields are all those whose communication fields belong to the set of valid communications, provided the given channel name has been declared, similarly as defined by the *PrefixSig* schema.

$$\begin{aligned} ValidMixedComm \;==\; & \{\, S : USt;\; E : Env;\; c : Name;\; commS : \operatorname{seq} Comm \mid \\ & \quad c \in E.chs \land \\ & \quad (\forall\, m : \operatorname{ran} commS \bullet (S, ((E, c), m)) \in ValidComm) \\ & \quad \bullet (S, ((E, c), commS))\,\} \end{aligned}$$

Next, we abstractly define a function that creates the arc representing the pattern of a multi-part prefixing, giving the set of valid mixed communication fields under the current user state and node environment. Its complete definition is lengthy but straightforward: induction on the length of the sequence of communication fields. This style of specification using type restriction to define a total function, rather than a partial function to any action, is preferred to avoid difficulties while proving theorems in ***Z/Eves*** about partial functions [Fre04c, Section 7.6].

$$mkMultiPrefixArc : ValidMixedComm \rightarrow Arc$$

For example, suppose the following channel has been declared

$$\textbf{channel}\ \mathsf{c} : (\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$$

and we want to define a pattern that inputs the first two values from the product type of c with additional restrictive predicates, and also outputs the third value. This communication is encoded in a communication pattern such as

$$\mathsf{c?x : P_1?y : P_2!z \rightarrow A}$$

where the following arc would be generated by function *mkMultiPrefixArc*

$$\begin{aligned} & mkMultiPrefixArc\ (S, ((E, \mathsf{c}), \langle in\ (\mathsf{x}, \mathsf{P}_1), in\ (\mathsf{y}, \mathsf{P}_2), out\ \mathsf{z}\rangle)) \;= \\ & \quad \{\, \mathsf{v} : E.cType\ \mathsf{c} \mid evalP\ (S, \mathsf{P}_1) \neq f \land evalP\ (S, \mathsf{P}_2) \neq f \land \\ & \qquad \mathsf{z}\ proj\ \mathsf{v} \bullet (\mathsf{c}, evalE\ (S, \mathsf{v}))\,\} \\ & \quad \text{where in this case, } \mathsf{z}\ proj\ \mathsf{v} \Leftrightarrow \mathsf{z} = second\ (first\ \mathsf{v}) \end{aligned}$$

It follows directly from the appropriate combination between input and output prefixing definitions together with an appropriate projection of output expressions on the

value to be communicated. We define an additional ***Z/Eves*** rule about the applicability of *mkMultiPrefixArc* below.

theorem rule rMkMultiPrefixArcIsTotal
$\forall S : USt;\ E : Env;\ \mathsf{c} : Name;\ commS : \text{seq}\ Comm \mid$
$(S, ((E, \mathsf{c}), commS)) \in ValidMixedComm \bullet$
$(S, ((E, \mathsf{c}), commS)) \in \text{dom}\ mkMultiPrefixArc$

The signature for multi-part prefixing is similar to that for input prefixing and is defined by the next schema.

MultiPrefixSig
$PrefixSig;\ commS : \text{seq}\ Comm$

$P = prefixing\ ((\mathsf{c}, commS), A)$

As before, we need the result of the function that creates arcs not to be empty, rather than $E.cType\ \mathsf{c} \neq \varnothing$ as in the *InPrefixSig* schema, and to restrict the type of the value of output expressions as in the *OutPrefixSig* schema; these restrictions are encoded in the *ValidMixedComm* set.

MultiPrefixComm
MultiPrefixSig

$(S, ((E, \mathsf{c}), commS)) \in ValidMixedComm$
$mkMultiPrefixArc\ (S, ((E, \mathsf{c}), commS)) \neq \varnothing$

The *enabled* arc is defined as the result of the application of *mkMultiPrefixArc*.

$\forall\ MultiPrefixComm \bullet enabled\ ((S, P), E) =$
$\{\ mkMultiPrefixArc\ (S, ((E, \mathsf{c}), commS))\ \}$

The remaining action for execution after communication takes place depends on whether there are only output communication fields or not in the communication pattern. The former leads to the following action A directly, whereas the latter demands the declaration of (possibly nested) local environments, depending on the number of input communication fields that are present. The next set *OutCommOnly* characterises communication patterns consisting of output fields only.

$$OutCommOnly\ ==\ \{\ commS : \text{seq}\ Comm \mid (\forall\ c : \text{ran}\ commS \bullet c \in \text{ran}\ out)\ \}$$

The set of actions representing all local variable environments is given by the set *LetVarAct*. It is given explicitly, similarly to *SExprAct*, as it works better for ***Z/Eves***.

$LetVarAct\ ==\ \{\mathsf{x} : Name;\ \mathsf{T} : Type;\ \mathsf{v} : ZExpr;\ A : Action \bullet$
$letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A)\ \}$

It is used in the abstract definition of the next function, which creates the local variable environment for the given sequence of communication fields, arc, and following action.

$$mkMultiPrefixNext : (\text{seq}\ Comm \times Arc) \times Action \rightarrow LetVarAct$$

After a couple of additional ***Z/Eves*** rules, the transition step is defined.

theorem grule gMkMultiPrefixNextMaxType
$mkMultiPrefixNext \in$
$(\mathbb{P}(\mathbb{Z} \times Comm) \times \mathbb{P}(Name \times ZExpr)) \times Action \leftrightarrow Action$

Although the function *mkMultiPrefixNext* is defined as total, the prover does not know

information about its domain applicability. The "totality" rules are particularly useful for higher level of automation in ***Z/Eves*** consistency (domain check) proofs.

theorem rule rMkMultiPrefixNextIsTotal
$\forall\, commS : \text{seq}\ Comm;\ a : Arc;\ A : Action \bullet$
$((commS, a), A) \in \text{dom}\ mkMultiPrefixNext$

The arc is given according to function *mkMultiPrefixArc* applied over the sequence of communication fields *commS* under the given user state and node environment. If the elements in *commS* are for output communication only, as in $\mathsf{c!x!y} \rightarrow A$, then the next program is just A. Otherwise, if *commS* contains both input and output communication, as in $(\mathsf{c?x} : \mathsf{P}_1\mathsf{?y} : \mathsf{P}_2\mathsf{!y} \rightarrow A)$ from our example, then the next program is created through the *mkMultiPrefixNext* function.

MultiPrefixStep
$MultiPrefixComm;\ PrefixStep$

$lbl! = mkMultiPrefixArc\,(S, ((E, \mathsf{c}), commS))$
$commS \in OutCommOnly \Rightarrow P' = A$
$commS \notin OutCommOnly \Rightarrow$
$\quad P' = mkMultiPrefixNext\,((commS, lbl!), A)$

Function *mkMultiPrefixNext* simply assembles the local variable environment, in a similar way to what is done in the definition given for input prefixing. For our example $(\mathsf{c?x} : \mathsf{P}_1\mathsf{?y} : \mathsf{P}_2\mathsf{!y} \rightarrow A)$ the type of *lbl*! is

$$lbl! \in \mathbb{P}\,(Name \times ZExpr) \ \equiv\ lbl! \in \mathbb{P}\,(Name \times ((\mathbb{N} \times \mathbb{N}) \times \mathbb{N}))$$

In this case, function *mkMultiPrefixNext* builds a *letvar* action with appropriate projections over *lbl*!:

$$\begin{array}{l}
\text{ran}\ lbl! \in E.cType\,\mathsf{c} \ \Rightarrow\ \text{ran}\ lbl! \in (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \ \Rightarrow \\
\quad mkMultiPrefixNext\,((\langle in\,(\mathsf{x}, \mathsf{P}_1), in\,(\mathsf{y}, \mathsf{P}_2), out\,\mathsf{y}\rangle, lbl!), A) = \\
\qquad letvar\,(((\mathsf{x}, \text{dom}\,(\text{dom}\,(\text{ran}\ lbl!))), first\,(first\,(second\,(elem\,(lbl!))))), \\
\qquad\quad letvar\,(((\mathsf{x}, \text{ran}\,(\text{dom}\,(\text{ran}\ lbl!))), \\
\qquad\qquad second\,(first\,(second\,(elem\,(lbl!))))), A))
\end{array}$$

These patterns are difficult to read, but straightforward to define based on simple projections over product types.

$$\forall\ MultiPrefixStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

theorem rule rMkMultiPrefixArcMaxResult
$\forall\, S : USt;\ E : Env;\ \mathsf{c} : Name;\ commS : \text{seq}\ Comm \mid$
$(S, ((E, \mathsf{c}), commS)) \in ValidMixedComm \bullet$
$mkMultiPrefixArc\,(S, ((E, \mathsf{c}), commS)) \in \mathbb{P}\,(Name \times ZExpr)$

theorem rule rMkMultiPrefixNextMaxResult
$\forall\, commS : \text{seq}\ Comm;\ a : Arc;\ A : Action \bullet$
$mkMultiPrefixNext\,((commS, a), A) \in Action$

theorem tMultiPrefixStepAppl
$\forall\, MultiPrefixComm \bullet \text{pre}\ MultiPrefixStep$

The step function and applicability theorem are finally defined, after a couple of more

Z/Eves rules about maximal types related to these two functions. As before, if the verification condition does not hold, the prefixing deadlocks since there is no enabled arc available to satisfy the multi-part prefixing signature.

$$MultiPrefixStop \mathrel{\widehat=} [\, MultiPrefixSig \mid \neg MultiPrefixComm \,]$$

$$\forall\ MultiPrefixStop \bullet enabled\ ((S, P), E) = \varnothing$$

$$\forall\ MultiPrefixStop;\ lbl! : Arc \bullet arcStep\ (((S, P), E), lbl!) = \varnothing$$

As multi-part prefixing is a generalisation of the input and output cases, the prototype only needs to implemented the synchronisation and multi-part prefixing cases.

3.6 Commands

We define commands for variable declaration, which extends the user state and node environment, as well as variable assignment, which changes some value of the state to that of an expression.

3.6.1 Variable declaration—(var$\mathsf{x} : \mathsf{T} \bullet \mathsf{A}$)

Variables need to be recorded in the node environment in the same way as channels: their names and types need to be included. Nevertheless, as variable declaration is an *Action* (see Figure 2.2), it does generate transitions. For the signature for variable declaration, the following ***Z/Eves*** rules about *Action* are needed.

theorem rule rCircVarType
$\forall \mathsf{x} : Name;\ \mathsf{T} : Type;\ A : Action \bullet circvar\ ((\mathsf{x}, \mathsf{T}), A) \in Action$

theorem rule rCircVarIsTotal
$\forall \mathsf{x} : Name;\ \mathsf{T} : Type;\ A : Action \bullet ((\mathsf{x}, \mathsf{T}), A) \in \mathrm{dom}\ circvar$

The signature for variable declaration extends a unary definition with the local environment components; the invariant is also extended to guarantee that the additional type and scope rules needed for environment extension are enforced: the variable x must be new, and the value v is within its type T.

VarDeclSig

$UnSig;\ \mathsf{x} : Name;\ \mathsf{T} : Type;\ \mathsf{v} : ZExpr$

$\mathsf{x} \in E.fresh \land \mathsf{v} \in \mathsf{T}$
$P = circvar\ ((\mathsf{x}, \mathsf{T}), A)$
$(\exists\ Sig';\ a : Arc \bullet a \in enabled\ ((S, letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A)), E) \land$
$\quad ((S', P'), E') \in arcStep\ (((S, letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A)), E), a)\ )$

Moreover, the definition of a configuration for variable declaration needs to refer to the evaluation of one of its parts. For a transition for a variable block to be well-defined, there must exist a transition definition for the evaluation of the local environment for the following action A. This shows how we define programs in terms of other programs

in our semantics. The *enabled* function is then the result of applying *enabled* to the local environment for A.

$$\forall\ VarDeclSig \bullet enabled\ ((S, P), E) = enabled\ ((S, letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A)), E)$$

The step function is defined similarly, where the arc and the after configuration comes from the evaluation of the local environment.

VarDeclStep

$$VarDeclSig;\ Step$$

$$lbl! \in enabled\ ((S, letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A)), E)$$
$$((S', P'), E') \in arcStep\ (((S, letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{v}), A)\ ), E), lbl!)$$

$$\forall\ VarDeclStep \bullet arcStep\ (((S, P), E), lbl!) = \{\ ((S', P'\ ), E')\ \}$$

We could have defined *VarDeclSig* in terms of *VarDeclStep* for this transition, but we preferred to leave them separate.

theorem tVarDeclStepAppl
$\forall\ VarDeclSig \bullet$ pre $VarDeclStep$

This is another example where the applicability theorems were useful to point out a stricter signature precondition; in this case, that there must exist a definition of the semantic functions for the local environment in order for the local variable declaration to be well-defined.

3.6.2 Assignment—(x := e)

The last basic action is variable assignment whose signature is given below, right after another ***Z/Eves*** rule.

theorem frule fUStTypeOfMaxType
$USt \Rightarrow typeOf \in \mathbb{P}\,(Name \times \mathbb{P}\ ZExpr)$

AssignSig

$$Sig;\ \mathsf{x} : Name;\ \mathsf{e} : ZExpr$$

$$\mathsf{x} \in \mathrm{ran}\ S.uvars$$
$$evalE\ (S, \mathsf{e}) \in S.typeOf\ \mathsf{x}$$
$$P = assign\ (\mathsf{x}, \mathsf{e})$$

It extends the basic signature with the variable x and the expression e of the assignment, ensures that x has already been declared, and it also ensures that the evaluation of e in the before-state lies within the declared type of x. This last precondition is important for keeping the invariant after the state has been updated, and was detected through the (failed) proof of the applicability conjecture. Like schema expressions, assignment can perform only a silent transition that updates the state.

$$\forall\ AssignSig \bullet enabled\ ((S, P), E) = \{\varnothing\}$$

The transition for assignment leads to Skip with the appropriate update of the value of x, which now is mapped to the evaluation of expression e in the current state. As

in the definition of output prefixing, the definition of assignment also allows loosely defined symbols in the expression.

AssignStep

$$AssignSig;\ SilentStep$$

$$P' = \mathsf{Skip}$$
$$E' = E$$
$$S'.uvars = S.uvars$$
$$S'.valueOf = S.valueOf \oplus \{ \mathsf{x} \mapsto evalE\,(S, \mathsf{e}) \}$$

$$\forall\ AssignStep \bullet arcStep\,(((S, P), E), lbl!) = \{ ((S', P'), E') \}$$

Finally, we define two ***Z/Eves*** rules followed by three additional lemmas for the proof of the conjecture involving the applicability of variable assignment.

theorem frule fUStUTypesMaxType
$$USt \Rightarrow utypes \in \mathbb{P}\,(\mathbb{Z} \times \mathbb{P}\ ZExpr)$$

theorem rule rEvalEResult
$$\forall S : USt;\ \mathsf{e} : ZExpr \bullet evalE\,(S, \mathsf{e}) \in ZExpr$$

theorem disabled rule tInterDiffLemma $[X]$
$$\forall S : \mathbb{P}\ X;\ x : X \mid x \in S \bullet \{ x \} \cup (S \setminus \{x\}) = S$$

theorem disabled rule tAssignTypeOfUpdate
$$\forall USt;\ \mathsf{x} : Name;\ \mathsf{e} : ZExpr \mid \mathsf{x} \in \operatorname{ran} uvars \land \mathsf{e} \in typeOf\,\mathsf{x} \bullet$$
$$\operatorname{dom} typeOf = \operatorname{dom}\,(valueOf \oplus \{ \mathsf{x} \mapsto \mathsf{e} \})$$

theorem disabled rule tAssignUStUpdate
$$\forall USt;\ \mathsf{x} : Name;\ \mathsf{e} : ZExpr \mid \mathsf{x} \in \operatorname{ran} uvars \land evalE(\theta\, USt, \mathsf{e}) \in typeOf\mathsf{x} \bullet$$
$$USt[valueOf := valueOf \oplus \{(\mathsf{x}, evalE(\theta\, USt, \mathsf{e}))\}]$$

theorem tAssignStepAppl
$$\forall\ AssignSig \bullet \operatorname{pre}\ AssignStep$$

As variable assignment mentions particular components of *USt*, it leads to a more complex proof of applicability and demands extra lemmas not related to ***Z/Eves*** automation, such as *tInterDiffLemma*, *tAssignUpdate*, and *tAssignUStUpdate*, which ensure that the invariant of *USt* holds after the state update caused by the assignment takes place.

This shows that model checking will require checking that the proof of commands in a program maintain the state invariant. This kind of proof obligation is part of a fully formal development approach, in which the introduction of an assignment, for example, requires the proof that it maintains the invariant. Checking a *Circus* program that has been formally developed, so that the state invariant can be eliminated, will lead to less proof obligations, and, therefore, a higher degree of automation. On the other hand, if such a degree of formality has not been employed in development, model checking will enforce these important consistency constraints.

3.7 Guarded action—(g & A)

The signature for guards is defined next by extending the unary signature with a guard g given as a Z predicate.

$$GuardSig \mathrel{\widehat=} [\, UnSig;\ \mathsf{g} : ZPred \mid P = guard\,(\mathsf{g}, A)\,]$$

When the evaluation of the predicate in the guard in the current state is ***true***, we just evaluate the guarded action A.

$$\begin{array}{|l} TrueGuardSig \\ GuardSig \\ \hline evalP\,(S, \mathsf{g}) = t \\ (\exists\, Sig';\ a : Arc \bullet a \in enabled\,((S, A), E) \land \\ \quad ((S', P'), E') \in arcStep\,(((S, A), E), a)\,) \\ \hline \end{array}$$

$$\forall\ TrueGuardSig \bullet enabled\,((S, P), E) = enabled\,((S, A), E)$$

The transition leading to the following program is defined in terms of the transition for the guarded action.

$$\begin{array}{|l} TrueGuardStep \\ TrueGuardSig;\ Step \\ \hline lbl! \in enabled\,((S, A), E) \\ ((S', P'), E') \in arcStep\,(((S, A), E), lbl!) \\ \hline \end{array}$$

$$\forall\ TrueGuardStep \bullet arcStep\,(((S, P), E), lbl!) = arcStep\,(((S, A), E), lbl!)$$

theorem tTrueGuardStepAppl
$\forall\ TrueGuardSig \bullet \mathrm{pre}\ TrueGuardStep$

When the evaluation of the predicate is ***false***, the action is deadlocked and behaves similarly to Stop, where neither arcs nor configurations are available.

$$FalseGuardSig \mathrel{\widehat=} [\, GuardSig \mid evalP\,(S, \mathsf{g}) = f\,]$$

$$\forall\ TrueGuardSig \bullet enabled\,((S, P), E) = \varnothing$$

$$\forall\ FalseGuardSig;\ lbl! : Arc \bullet arcStep\,(((S, P), E), lbl!) = \varnothing$$

As the complete case is just disjunction, it does not affect the applicability theorem, because disjunction distributes through pre. Since the definition for guards is simple, the proof is very straightforward, and we omit the complete applicability case here for simplicity.

3.8 Call and recursion

Next, we define actions related to recursive programs.

3.8.1 Action call—N

A call to an action A fetches its body from the node environment using the $aCtx$ function, provided it has been previously declared ($\mathsf{N} \in acts$), and there exists a transition available for the retrieved action.

CallSig

$$UnSig;\ \mathsf{N} : Name$$

$$\mathsf{N} \in E.acts \land A = E.aCtx\,\mathsf{N} \land P = call\,\mathsf{N}$$
$$(\exists\, Sig';\ a : Arc \bullet a \in enabled\,((S, A), E) \land$$
$$((S', P'), E') \in arcStep\,(((S, A), E), a))$$

The transition are those of the fetched action.

$$\forall\ CallSig \bullet enabled\,((S, P), E) = enabled\,((S, A), E)$$

CallStep

$$CallSig;\ Step$$

$$lbl! \in enabled\,((S, A), E)$$
$$((S', P'), E') \in arcStep\,(((S, A), E), lbl!)$$

$$\forall\ CallStep \bullet arcStep\,(((S, P), E), lbl!) = arcStep\,(((S, A), E), lbl!)$$

theorem tCallStepAppl
$\forall\ CallSig \bullet$ pre $CallStep$

Together with action declaration, one can use action call to create mutual recursion between actions.

3.8.2 Recursion—(μ X • A)

We represent simple recursion as $rec\,(\mathsf{X}, A)$, where X is the recursive variable, and A is an action that uses X to call itself. Operationally, this corresponds to executing A in an environment in which X is bound to A itself. For this transition to be well-defined, X should not have been previously used and there should exist a transition defining the evaluation of the local environment for actions.

MuSig

$$UnSig;\ \mathsf{X} : Name$$

$$\mathsf{X} \in E.fresh \land P = rec\,(\mathsf{X}, A)$$
$$(\exists\, Sig';\ a : Arc \bullet a \in enabled\,((S, letmu\,((\mathsf{X}, A), A)), E) \land$$
$$((S', P'), E') \in arcStep\,(((S, letmu\,((\mathsf{X}, A), A)), E), a))$$

The *enabled* arcs are those of A within the local environment

$$\forall\ MuSig \bullet enabled\,((S, P), E) = enabled\,((S, letmu\,((\mathsf{X}, A), A)), E)$$

The step transitionis defined in terms of the local environment for recursive actions,

provided the arc comes from the enabled arcs of the local environment as well.

$$
\begin{array}{|l}
\textit{MuStep} \\
\textit{MuSig};\ \textit{Step} \\
\hline
\textit{lbl}! \in \textit{enabled}\,((S, \textit{letmu}\,((\mathsf{X}, A), A)), E) \\
((S', P'), E') \in \textit{arcStep}\,(((S, \textit{letmu}\,((\mathsf{X}, A), A)), E), \textit{lbl}!) \\
\hline
\end{array}
$$

$$\forall\ \textit{MuStep} \bullet \textit{arcStep}\,(((S, P), E), \textit{lbl}!) = \textit{arcStep}\,(((S, A), E), \textit{lbl}!)$$

theorem tMuStepAppl
$\forall\ \textit{MuSig} \bullet \text{pre}\ \textit{MuStep}$

The definition of local environment for actions as given by *letmu* is presented later in Section 3.13.2.

3.9 Sequential composition—(A ; B)

Before defining the transition for sequential composition, let us define a signature *BinSig* for binary actions.

$$\textit{BinSig} \mathrel{\widehat=} [\textit{Sig};\ A, B : \textit{Action}]$$

The signature for sequential composition is just the binary signature where the initial program is *seqcomp*.

$$\textit{SeqSig} \mathrel{\widehat=} [\textit{BinSig} \mid P = \textit{seqcomp}\,(A, B)]$$

We have two cases to consider: termination and progress of the left hand side. Action (A ; B) sequences to B, whenever A is Skip.

$$\textit{SeqSkipSig} \mathrel{\widehat=} [\textit{SeqSig} \mid A = \mathsf{Skip}]$$

This leads to the next action in sequence via a silent transition without state changes.

$$\forall\ \textit{SeqSkipSig} \bullet \textit{enabled}\,((S, P), E) = \{\varnothing\}$$

$$\textit{SeqSkipStep} \mathrel{\widehat=} [\textit{SeqSkipSig};\ \textit{ReadOnlyStep};\ \textit{SilentStep} \mid P' = B]$$

$$\forall\ \textit{SeqSkipStep} \bullet \textit{arcStep}\,(((S, P), E), \textit{lbl}!) = \{((S', P'), E')\}$$

theorem tSeqSkipStepAppl
$\forall\ \textit{SeqSkipSig} \bullet \text{pre}\ \textit{SeqSkipStep}$

The next case where (A ; B) makes progress to (A′ ; B) is defined in terms of transitions for A that lead to A'. If such transition exists A is not terminating, as Skip has no defined transitions. The *enabled* arcs are just those of A whereas the arc and next

program are defined in terms of the semantic functions for A.

$$
\begin{array}{|l}
\hline
SeqProgSig \\
SeqSig \\
\hline
(\exists\, Sig'[A'/P'];\ a : Arc \bullet a \in enabled\,((S, A), E) \land \\
\quad ((S', A'), E') \in arcStep\,(((S, A), E), a)\,) \\
\hline
\end{array}
$$

$$\forall\ SeqProgSig \bullet enabled\,((S, P), E) = enabled\,((S, A), E)$$

$$
\begin{array}{|l}
\hline
SeqProgStep \\
SeqProgSig;\ Step \\
A' : Action \\
\hline
lbl! \in enabled\,((S, A), E) \\
P' = seqcomp\,(A', B) \\
((S', A'), E') \in arcStep\,(((S, A), E), lbl!) \\
\hline
\end{array}
$$

$$\forall\ SeqProgStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

theorem tSeqProgStepAppl
$\forall\ SeqProgSig \bullet \text{pre}\ SeqProgStep$

The complete definition is just disjunction of each case.

$$SeqAllSig\ \mathrel{\widehat=}\ SeqSkipSig \lor SeqProgSig$$

$$SeqAllStep\ \mathrel{\widehat=}\ SeqSkipStep \lor SeqProgStep$$

$$\forall\ SeqAllStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

theorem tSeqAllStepAppl
$\forall\ SeqAllSig \bullet \text{pre}\ SeqAllStep$

Since pre distributes through disjunction, the related proofs are similar.

3.10 Choice

Like in CSP, *Circus* offers two forms of choice: internal or nondeterministic, and external. The former occurs without control of the external environment, whereas the latter demands its co-operation to engage in visible communication. Regardless of the case, we define a signature used for both actions.

$$ChOptsSig\ \mathrel{\widehat=}\ [\,BinSig;\ side : Action \mid side \in \{\,A, B\,\}\,]$$

It records the available options for execution in the component *side*.

3.10.1 Internal choice—(A ⊓ B)

Internal choice makes a nondeterministic choice through a silent transition, hence the empty set is the only arc enabled. It represents the internal activity of making the nondeterministic decision to execute either action A or action B.

$$IntChSig \mathrel{\widehat=} [\, ChOptsSig \mid P = int\,(A, B)\,]$$

$$\forall\ IntChSig \bullet enabled\,((S, P), E) = \{\, \varnothing \,\}$$

The transition selects arbitrarily the actions for execution, which becomes the program in the next node. The transition has empty arc and no state changes occur.

$$IntChStep \mathrel{\widehat=} [\, IntChSig;\ ReadOnlyStep;\ SilentStep \mid P' = side\,]$$

$$\forall\ IntChStep \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E')\,\}$$

theorem tIntChStepAppl
$\forall\ IntChSig \bullet$ pre $IntChStep$

As in UTP, this encoding of internal choice is simply disjunction.

3.10.2 External choice—(A □ B)

External choice allows choice by the external environment based on the availability of communication. As the actions involved in the choice may engage in silent transitions due to either internal activity or termination, external choice is not always deterministic. The general signature for external choice is given next.

$$ExtChSig \mathrel{\widehat=} [\, ChOptsSig \mid P = ext\,(A, B)\,]$$

We divide the definition of external choice in three cases: termination, visible communication, and internal activity via silent transitions of A or B. An external choice may terminate whenever any side of the choice is terminating.

$$ExtChSkipSig \mathrel{\widehat=} [\, ExtChSig \mid side = \mathsf{Skip}\,]$$

$$\forall\ ExtChSkipSig \bullet enabled\,((S, P), E) = \{\, \varnothing \,\}$$

In this case only a silent transition is *enabled* leading to Skip without state changes.

$$ExtChSkipStep \mathrel{\widehat=} [\, ExtChSkipSig;\ ReadOnlyStep;\ SilentStep \mid P' = \mathsf{Skip}\,]$$

$$\forall\ ExtChSkipStep \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E')\,\}$$

theorem tExtChSkipStepAppl
$\forall\ ExtChSkipSig \bullet$ pre $ExtChSkipStep$

This definition is compatible with Roscoe's interpretation of (Skip □ A), which is that termination cannot be refused by the environment and might happen outside its control [Ros97, Section 6.3].

The next case is the actual external choice being resolved due to visible communication *enabled* for any action recorded as an option in *side*. The signature is defined in terms of the existence of a transition for whichever possible choice of action.

$$
\begin{array}{|l}
\hline
ExtChProgSig \\
ExtChSig \\
\hline
(\exists\, ChOptsSig';\ a : Arc \bullet ReadOnlyStep[a/lbl!] \land \\
\quad a \in enabled\,((S, side), E) \land a \neq \varnothing \land \\
\quad ((S', P'), E') \in arcStep\,(((S, side), E), a)\,) \\
\hline
\end{array}
$$

In this case, the enabled events are those enabled by any of the actions that are available for choice.

$$\forall\ ExtChProgSig \bullet enabled\,((S, P), E) = enabled\,((S, A), E) \cup enabled\,((S, B), E)$$

The transition is defined to take into account progress on the execution of any of the actions. As visible communication comes only through prefixing, which does not alter the state, choice on visible progress does not lead to any state change as well.

$$
\begin{array}{|l}
\hline
ExtChProgStep \\
ExtChProgSig;\ ReadOnlyStep \\
\hline
lbl! \in enabled\,((S, side), E) \land lbl! \neq \varnothing \\
((S', P'), E') \in arcStep\,(((S, side), E), lbl!) \\
\hline
\end{array}
$$

So, whenever visible communication happens, the choice is resolved to the following action that arises from either A or B.

$$\forall\ ExtChProgStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

theorem tExtChProgStepAppl
$\forall\, ExtChProgSig \bullet \text{pre}\ ExtChProgStep$

Event selection happens through visible communication on all forms of prefixing. As prefixing is evaluated in two stages, the communication occurs first and then the prefixed action is evaluated next. As the signature of external choice depends upon the existence of a transition for each side in order for the external choices to take place, this strategy of evaluating prefixing enables us to implement external choice lazily. That is, in order to decide which action on an external choice should be selected, we do not need to fully evaluate the prefixing.

Finally, we consider the case of internal progress which does not resolve an external choice. Internal (silent) progress happens on the resolution of an internal choice, evaluation of variable declaration, action call, evaluation of schema expressions, and so forth. As we insist on the existence of a transition, this implicitly implies that *side* is not Skip. For the transition step, we define for each side a silent transition leading to either $(\mathsf{A}' \,\Box\, \mathsf{B})$ or $(\mathsf{A} \,\Box\, \mathsf{B}')$, where the complete internal progress case is just disjunction. This means that state updates do not resolve a choice and are

accumulated until a visible event takes place.

$$
\begin{array}{|l}
\textit{ExtChIntLSig} \\
\textit{ExtChSig} \\
\hline
side = A \\
(\exists\, Sig'[A'/P'] \bullet ((S', A'), E') \in arcStep\,(((S, A), E), \varnothing)\,)
\end{array}
$$

$$
\begin{array}{|l}
\textit{ExtChIntRSig} \\
\textit{ExtChSig} \\
\hline
side = B \\
(\exists\, Sig'[B'/P'] \bullet ((S', B'), E') \in arcStep\,(((S, B), E), \varnothing)\,)
\end{array}
$$

$$\textit{ExtChIntSig} \mathrel{\widehat=} \textit{ExtChIntLSig} \lor \textit{ExtChIntRSig}$$

$$\forall\, \textit{ExtChIntSig} \bullet enabled\,((S, P), E) = \{\, \varnothing \,\}$$

$$
\begin{array}{|l}
\textit{ExtChIntLStep} \\
\textit{ExtChIntLSig};\ \textit{BinSig}';\ \textit{SilentStep} \\
\hline
((S', A'), E') \in arcStep\,(((S, A), E), lbl!) \\
P' = ext\,(A', B)
\end{array}
$$

$$
\begin{array}{|l}
\textit{ExtChIntRStep} \\
\textit{ExtChIntRSig};\ \textit{BinSig}';\ \textit{Step};\ \textit{SilentStep} \\
\hline
((S', B'), E') \in arcStep\,(((S, B), E), lbl!) \\
P' = ext\,(A, B')
\end{array}
$$

$$\textit{ExtChIntStep} \mathrel{\widehat=} \textit{ExtChIntLStep} \lor \textit{ExtChIntRStep}$$

$$\forall\, \textit{ExtChIntStep} \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E') \,\}$$

theorem tExtChIntStepAppl
$\forall\, \textit{ExtChIntSig} \bullet \text{pre}\ \textit{ExtChIntStep}$

As before, the complete definition is defined as the disjunction of each individual case.

$$\textit{ExtAllSig} \mathrel{\widehat=} \textit{ExtChSkipSig} \lor \textit{ExtChProgSig} \lor \textit{ExtChIntSig}$$

$$\textit{ExtAllStep} \mathrel{\widehat=} \textit{ExtChSkipStep} \lor \textit{ExtChProgStep} \lor \textit{ExtChIntStep}$$

The shape of the automaton for external choice points out the possibility of a trivial optimisation in the definition of the terminating case of sequential composition. Like what we have done for action call, for $(\mathsf{Skip}\,;B)$, instead of generating a silent transition to B, we could have just evaluated the semantic functions for B directly.

3.11 Parallelism

Parallel composition is defined next. Firstly, we need to define additional functions to handle synchronisation sets defined by channel set expressions, as well as state partitions defined by name set expressions. The former is used for synchronisation on visible communications, whereas the latter restricts how the user state can be modified by each side of the parallelism in order to rule out concurrent write access.

3.11.1 Channel set expression evaluation—csext (E, cs)

Channel set expressions were defined earlier by the free type *CSExpr* (see Section 3.2) as a (possibly empty) sequence of *Name*. The events it represents according to the current environment are characterised by the *csext* function, which has a similar role as the functions *extensions* and *productions* defined for FDR [Gol00, p.61]. It is defined inductively on the structure of channel set expressions, provided all channel names have been previously declared in the node environment (ran SN $\subseteq$ $E.chs$). For this definition, we also need a ***Z/Eves*** forward rule exposing the maximal type of the related component from the environment.

theorem frule fEnvChsType
$\forall E : Env \bullet E.chs \in \mathbb{P}\ Name$

$$
\begin{array}{|l}
csext : Env \times CSExpr \pfun Arc \\
\hline
\dom csext = \{ E : Env \bullet (E, cempty) \} \cup \\
\qquad \{ E : Env; \ \mathsf{SN} : \seq Name \mid \ran \mathsf{SN} \subseteq E.chs \bullet (E, clist\,(\mathsf{SN})) \} \\
\forall E : Env \bullet csext\,(E, cempty) = \varnothing \\
\forall E : Env; \ \mathsf{c} : Name \mid \mathsf{c} \in E.chs \bullet \\
\qquad csext\,(E, clist\,(\langle \mathsf{c} \rangle)) = \{ \mathsf{v} : E.cType\,\mathsf{c} \bullet (\mathsf{c}, \mathsf{v}) \} \\
\forall E : Env; \ \mathsf{SN}_1, \mathsf{SN}_2 : \seq Name \mid \ran \mathsf{SN}_1 \subseteq E.chs \land \ran \mathsf{SN}_2 \subseteq E.chs \bullet \\
\qquad csext\,(E, clist\,(\mathsf{SN}_1 \frown \mathsf{SN}_2)) = \\
\qquad\qquad csext\,(E, clist\,(\mathsf{SN}_1)) \cup csext\,(E, clist\,(\mathsf{SN}_2))
\end{array}
$$

The domain of *csext* guarantees that only names of already declared channels can be used. For the base case on the empty channel set, the result is the empty arc. The unit case yields all pairs (c, v), where v belongs to the type of the declared channel c. The inductive case is defined as the set union of the resulting extensions for each given sequence. This function can be totalised by augmenting the typechecker to deal with the case where the given names did not refer to previously declared channels.

3.11.2 Name set expression evaluation—nsext (S, ns)

Name set expressions are defined similarly, where the names now must come from the user state rather than the environment. Augmenting the evaluation of both channel and name set expressions to include definition of set operations or user defined functions is straightforward. Set operations, such as union, intersection and difference, are implemented directly with corresponding Z toolkit operators [Saa99a], whereas user defined functions (including operator templates) can be defined appropriately by the introduction of the corresponding axiomatic (or generic) definition for the user function. These extensions are available at the prototype implementation, and they

are omitted here for simplicity.

$$
\begin{array}{l}
nsext : USt \times NSExpr \pfun \mathbb{P}\ Name \\
\hline
\mathrm{dom}\ nsext = \{ S : USt \bullet (S, nempty) \} \cup \\
\qquad \{ S : USt;\ \mathsf{SN} : \mathrm{seq}\ Name \mid \mathrm{ran}\ \mathsf{SN} \subseteq \mathrm{ran}\ S.uvars \bullet \\
\qquad\quad (S, nlist\ (\mathsf{SN})) \} \\
\forall\ S : USt \bullet nsext\ (S, nempty) = \varnothing \\
\forall\ S : USt;\ n : Name \mid n \in \mathrm{ran}\ S.uvars \bullet nsext\ (S, nlist\ (\langle n \rangle)) = \{ n \} \\
\forall\ S : USt;\ \mathsf{SN}_1, \mathsf{SN}_2 : \mathrm{seq}\ Name \mid \mathrm{ran}\ \mathsf{SN}_1 \subseteq \mathrm{ran}\ S.uvars \land \\
\qquad \mathrm{ran}\ \mathsf{SN}_2 \subseteq \mathrm{ran}\ S.uvars \bullet nsext\ (S, nlist\ (\mathsf{SN}_1 \frown \mathsf{SN}_2)) = \\
\qquad\quad nsext\ (S, nlist\ (\mathsf{SN}_1)) \cup nsext\ (S, nlist\ (\mathsf{SN}_2))
\end{array}
$$

Moreover, contextual analysis must also enforce that no dashed variables are present in name set expressions.

3.11.3 Parallel operator—(A ⟦ ns0 | cs | ns1 ⟧ B)

The parallel operator of *Circus* usually synchronises either on a set of events or on termination. Firstly, let us define the signature for parallel composition.

$$
\begin{array}{l}
ParSig \\
\hline
BinSig;\ cs : CSExpr;\ ns0, ns1 : NSExpr \\
\hline
(E, cs) \in \mathrm{dom}\ csext \land (S, ns0) \in \mathrm{dom}\ nsext \land (S, ns1) \in \mathrm{dom}\ nsext \\
\langle nsext\ (S, ns0), nsext\ (S, ns1) \rangle\ \mathsf{partition}\ (\mathrm{ran}\ S.uvars) \\
P = par\ (((ns0, A), (ns1, B)), cs)
\end{array}
$$

It extends the binary signature with a channel set expression representing the events that require synchronisation, as well as the name set expressions representing the user state components partition. This guarantees that the channel and name set expressions are well-formed.

We need to define four cases: distributed termination, internal progress on silent transitions from either parallel action, synchronisation on visible communication, and deadlock (see Figure 3.1). Before defining the cases for parallelism, we need to de-

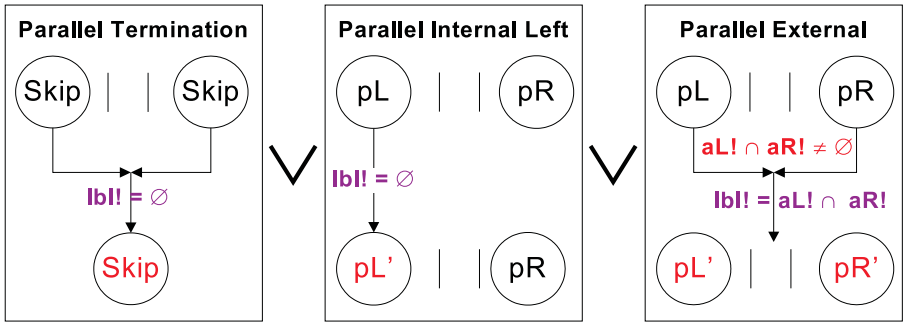

Figure 3.1: Transitions for parallelism

scribe how the after-state of the parallel composition is obtained after termination by merging the after-states of each side, in such a way that the update restrictions imposed by the partitions are used to avoid concurrent writing access to the user

state. Prior to execution, all the before-states on each parallel action and the parallel operator are the same. The merge signature is defined by the schema *ParMergeSig*. The input variable *changed*? represents the names allowed to change from the original state S. Additionally, the schema *UStEnvInv* ensures that the merge operation keeps the correspondence in structure between the user state and the node environment as defined by schema *Sig*.

$$UStEnvInv \mathrel{\widehat=} Sig \setminus (P)$$

ParMergeSig

$$S : USt;\ E : Env;\ changed? : \mathbb{P}\ Name$$

$$UStEnvInv$$
$$changed? \subseteq \operatorname{ran} S.uvars$$

Before defining the actual merge, we need additional definitions for ***Z/Eves***. We need ***Z/Eves*** rules exposing maximal types of *USt*, as well as an additional lemma for the relationship between declared variables and their values.

theorem frule fUStUVarsType
$$\forall S : USt \bullet S.uvars \in \operatorname{seq} Name$$

theorem frule fUStValueOfType
$$\forall S : USt \bullet S.valueOf \in Name \pfun ZExpr$$

theorem rule tUStUVarsNameInValueOfDom
$$\forall S : USt \mid name \in \operatorname{ran} S.uvars \bullet name \in \operatorname{dom} S.valueOf$$

As we must restrict the user states that can and cannot change, we also need a special sequence operator not available in the Z Standard toolkit, namely, sequence anti-filtering. It is defined below as an infix function $(\_ \lfloor \_)$ with precedence 3.
syntax $\lfloor$ *infun*3 `\_ \nfilter \_`

$[X]$

$$\_ \lfloor \_ : \operatorname{seq} X \times \mathbb{P}\ X \to \operatorname{seq} X$$

$$\forall\ s : \operatorname{seq} X;\ F : \mathbb{P}\ X \bullet (s \lfloor F) = s \upharpoonright (X \setminus F)$$

This operator defines the concealment of elements of set F from sequence s. We also include an additional lemma about sequence anti-filtering saying how it distributes over relational range.

theorem rule rRanNFilter $[X]$
$$\forall\ s : \operatorname{seq} X;\ F : \mathbb{P}\ X \bullet \operatorname{ran}(s \lfloor F) = (\operatorname{ran} s) \setminus F$$

Originally, we used a more intuitive definition for the operator:

$$\forall\ s : \operatorname{seq} X;\ F : \mathbb{P}\ X \bullet (s \lfloor F) = squash\,(s \rhd F)$$

Nevertheless, proofs involving *squash* require induction over natural numbers and are usually harder. Therefore, we preferred the description in terms of available operators, which also increases the level of automation via reuse of previously defined ***Z/Eves***

rules. The user state merge is defined with the schema *ParMergeStep* below. We have the merge signature together with two copies of the user state components: the after-state of the parallel composition (S'), and the after-state of one side (N') with the new values to be restricted accordingly. Although the execution of each side might extend the user state, it must not remove previous components, nor compromise the corresponding structure between user state and the node environment.

ParMergeStep

$$
\begin{array}{l}
ParMergeSig;\ S', N' : USt;\ E' : Env \\
\hline
UStEnvInv' \land UStEnvInv'[N'/S'] \\
\mathrm{ran}\ S.uvars \subseteq \mathrm{ran}\ S'.uvars \\
\mathrm{ran}\ S.uvars \subseteq \mathrm{ran}\ N'.uvars \\
\mathrm{ran}\ N'.uvars \subseteq \mathrm{ran}\ S'.uvars \\
\forall\ chg : \mathrm{ran}\,(S.uvars \upharpoonright changed?) \bullet S'.valueOf\ chg = N'.valueOf\ chg \\
\forall\ keep : \mathrm{ran}\,(S.uvars \downharpoonright changed?) \bullet S'.valueOf\ keep = S.valueOf\ keep
\end{array}
$$

In the first quantifier, variable *chg* represents all user state components filtered via the set *changed*? of variables allowed to change their values. To define the state merge, the dashed value of *chg* from the after-state of the parallel composition must be the same as the value of the after-state of the side being executed.

$$S'.valueOf\ chg = N'.valueOf\ chg$$

In a complementary fashion, the variable *keep* in the second quantifier represents all user state components that are not allowed to change their values, and hence must remain the same as the original before-state.

$$S'.valueOf\ keep = S.valueOf\ keep$$

For instance, assuming the user state component names are given as

$$uvars = \langle \mathsf{x}, \mathsf{y}, \mathsf{z} \rangle$$

and the input variables representing the state partition are given as

$$ns0 = \{ \mathsf{x} \} \land ns1 = \{ \mathsf{y}, \mathsf{z} \}$$

If the input variable is

$$changed? = nsext\,(S, ns0)$$

one would have the predicate representing the state merge restrictions as

$$
\begin{array}{l}
S'.valueOf\ \mathsf{x} = N'.valueOf\ \mathsf{x} \land S'.valueOf\ \mathsf{y} = S.valueOf\ \mathsf{y} \\
\land\ S'.valueOf\ \mathsf{z} = S.valueOf\ \mathsf{z}
\end{array}
$$

The after-state of the parallelism can change according to each partition, but now we have to consider two partitions as both sides terminate. Firstly, we define the signature for the merge on each side, where the *changed*? set is renamed accordingly

with new names as *changedL* for action A and *changedR* for action B.

$$ParSkipMergeSigA \mathrel{\widehat=} ParMergeSig[changedL/changed?]$$

$$ParSkipMergeSigB \mathrel{\widehat=} ParMergeSig[changedR/changed?]$$

The merge step for synchronisation must also consider updates from both sides.

$$ParSkipMergeStep \mathrel{\widehat=} ParSig \land ParMergeStep$$

Next, we rename the new after-state (N') of each parallel action accordingly.

$$ParSkipMergeStepA \mathrel{\widehat=} ParSkipMergeStep[changedL/changed?, SL'/N']$$

$$ParSkipMergeStepB \mathrel{\widehat=} ParSkipMergeStep[changedR/changed?, SR'/N']$$

For the schema *ParSkipMergeStepA*, this leave us with the following components.

$$S, S', SL' : USt;\ P, A, B : Action;\ E : Env;\ cs : CSExpr;$$
$$ns0, ns1 : NSExpr;\ changedL : \mathbb{P}\ Name$$

The interesting elements of the parallelism are: the same before-state (S) for P, A, and B; and the distinct after-states for P as S', for A as SL', and for B as SR'.

Going back to the definition of parallelism, the case of distributed termination occurs whenever both sides have terminated. It is defined as a silent transition leading to Skip. That is the time where the state merge occurs. The next schema defines the signature for termination. It defines the appropriate restriction on state partitions by limiting what each side can change.

ParSkipSig

$$ParSig;\ ParSkipMergeSigA;\ ParSkipMergeSigB$$

$$A = B = \mathsf{Skip}$$
$$changedL = nsext\,(S, ns0) \land changedR = nsext\,(S, ns1)$$
$$(\exists\, Sig';\ SL', SR' : USt \bullet$$
$$\quad ParSkipMergeStepA \land ParSkipMergeStepB \land$$
$$\quad P' = \mathsf{Skip}\,)$$

$$\forall\ ParSkipSig \bullet enabled\,((S, P), E) = \{\,\varnothing\,\}$$

Next, for the silent step transition leading to termination, we rename the after-state (S'), and the arc representing the chosen path for each side of the parallel action.

$$SilentStepA \mathrel{\widehat=} SilentStep[SL'/S', aL!/lbl!]$$

$$SilentStepB \mathrel{\widehat=} SilentStep[SR'/S', aR!/lbl!]$$

With these definitions, we can now introduce the transition step for termination of parallelism where the state merge takes place. From a silent transition on each side under the conditions of the schema *ParSkipSig*, both sides lead to Skip with after-states SL' for A and SR' for B, and the state merge is restricted according to the

definition of the schema *ParSkipMergeStep* for each side.

ParSkipStep

$$ParSkipSig;\ SilentStep;\ SilentStepA;\ SilentStepB$$

$$ParSkipMergeStepA \land ParSkipMergeStepB$$
$$P' = \mathsf{Skip}$$

$$\forall\ ParSkipStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

theorem tParSkipStepAppl
$\forall\, ParSkipSig \bullet \text{pre}\ ParSkipStep$

The next case where synchronisation is not necessary happens through independent progress due to either silent transitions or visible events outside the synchronisation set. Internal progress occurs on the resolution of internal choice, evaluation of variable declaration, action call, assignment, schema expression evaluation, and so forth. Visible communication occurs through the execution of prefixing. As occurred with external choice, the way prefixing is implemented allows lazy evaluation of parallelism because only the communication part of the prefixing is processed. For all these cases, provided the label is not within the synchronisation set or is the empty arc for internal progress, only one side of the parallelism progresses according to whatever is enabled for it.

ParIndBase

$$ParSig;\ Step$$
$$side? : Action;\ lbl! : Arc$$

$$side? \in \{\,A, B\,\}$$
$$lbl! \cap csext\,(E, cs) = \varnothing$$
$$lbl! \in enabled\,((S, side?), E)$$

The next two schemas define the signature where synchronisation is not necessary. It is defined provided that there exists a transition on either side respecting the appropriate merge step restrictions, as defined by schema *ParIndBase*.

ParIndLSig

$$ParSig$$

$$(\exists\, BinSig';\ a : Arc \bullet ParIndBase[A/side?, a/lbl!] \land$$
$$((S', A'), E') \in arcStep(((S, A), E), a)\,)$$

ParIndRSig

$$ParSig$$

$$(\exists\, BinSig';\ a : Arc \bullet ParIndBase[B/side?, a/lbl!] \land$$
$$((S', B'), E') \in arcStep(((S, B), E), a)\,)$$

For the step function, we have the restrictions on the signature according to the

schema *ParIndBase* for each side, and that the action progresses to a new parallel composition where progress has been made on the corresponding side. That is, action $(\mathsf{A} \,[\![\, \mathsf{ns0} \mid \mathsf{cs} \mid \mathsf{ns1} \,]\!]\, \mathsf{B})$ leads to action $(\mathsf{A}' \,[\![\, \mathsf{ns0} \mid \mathsf{cs} \mid \mathsf{ns1} \,]\!]\, \mathsf{B})$ (or $(\mathsf{A} \,[\![\, \mathsf{ns0} \mid \mathsf{cs} \mid \mathsf{ns1} \,]\!]\, \mathsf{B}')$).

ParIndLStep

$$ParIndLSig;\ BinSig';\ ParIndBase[A/side?]$$

$$((S', A'), E') \in arcStep\,(((S, A), E), lbl!)$$
$$P' = par\,(((ns0, A'), (ns1, B)), cs)$$

ParIndRStep

$$ParIndRSig;\ BinSig';\ ParIndBase[B/side?]$$

$$((S', B'), E') \in arcStep\,(((S, B), E), lbl!)$$
$$P' = par\,(((ns0, A), (ns1, B')), cs)$$

The complete definition for independent progress is just disjunction for each side.

$$ParIndSig \mathrel{\widehat=} ParIndLSig \lor ParIndRSig$$

$$ParIndStep \mathrel{\widehat=} ParIndLStep \lor ParIndRStep$$

theorem tParIndStepAppl
$\forall\, ParIndSig \bullet \text{pre}\ ParIndStep$

As parallelism is rather complex, the applicability theorem was useful to provide the minimum information necessary for it to be well-defined. We define the semantic functions *enabled* and *arcStep* as the complete case for progress later, which involves both progress without synchronisation (either internal or visible), and synchronised communication on visible events.

Next, before we define the case for synchronisation, we need to include the condition under which synchronisation takes place. That is, we need to add the chosen arcs of each side: they must be contained within the synchronisation set, and they must also agree on at least a particular event. This restriction is a verification condition for the synchronisation transition of parallelism to exist, and we explicitly defined it via the schema *ParSynchVC*.

$$ParVCSig \mathrel{\widehat=} [\, ParSig;\ aL!, aR! : Arc \,]$$

ParSynchVC

$$ParVCSig$$

$$aL! \in enabled((S, A), E)$$
$$aR! \in enabled((S, B), E)$$
$$aL! \cap aR! \cap csext\,(E, cs) \neq \varnothing$$

Nevertheless, whenever this verification condition does not hold, the parallel composition is deadlocked. That is, each side wishes to perform different events that must synchronise, but no agreeable event for synchronisation exists.

$$ParStop \mathrel{\widehat=} [\, ParVCSig \mid \neg\, ParSynchVC \,]$$

As expected, for this case where the parallelism deadlocks, the definition of the se-

mantic functions are similar to those for Stop.

$$\forall\ ParStop \bullet enabled\,((S, P), E) = \varnothing$$

$$\forall\ ParStop;\ lbl! : Arc \bullet arcStep\,(((S, P), E), lbl!) = \varnothing$$

With the definition of the verification condition for synchronisation, we can now specify the last case of synchronised behaviour. The signature for this case is defined provided that there exists a transition for both actions that satisfies the verification condition and the restrictions on possible state and environment extensions.

ParSynchSig

$$ParSig$$

$$\begin{aligned}
&(\exists\ BinSig';\ SL', SR' : USt;\ aL!, aR! : Arc \bullet \\
&\quad ParSynchVC \land UStEnvInv'[SL'/S'] \land UStEnvInv'[SR'/S'] \land \\
&\quad ((SL', A'), E') \in arcStep\,(((S, A), E), aL!) \land \\
&\quad ((SR', B'), E') \in arcStep\,(((S, B), E), aR!) \land \\
&\quad P' = par\,(((ns0, A'), (ns1, B')), cs)\ )
\end{aligned}$$

For the transition step we also need new copies of the after-state and chosen arcs, as defined by the next two schemas.

$$StepA\ \mathrel{\widehat{=}}\ Step[SL'/S', aL!/lbl!]$$

$$StepB\ \mathrel{\widehat{=}}\ Step[SR'/S', aR!/lbl!]$$

Provided there is agreement on synchronisation events, it leads to an action P' as shown in Figure 3.1.

ParSynchStep

$$ParSynchSig;\ BinSig';\ Step;\ StepA;\ StepB$$

$$\begin{aligned}
&ParSynchVC \\
&lbl! = aL! \cap aR! \\
&((SL', A'), E') \in arcStep\,(((S, A), E), aL!) \\
&((SR', B'), E') \in arcStep\,(((S, B), E), aR!) \\
&P' = par\,(((ns0, A'), (ns1, B')), cs)
\end{aligned}$$

theorem tParSynchStepAppl
$\forall\ ParSynchSig \bullet$ pre $ParSynchStep$

The complete definition for progress on parallelism is given as the disjunction between the independent and the synchronised progress cases.

$$ParProgSig\ \mathrel{\widehat{=}}\ ParIndSig \lor ParSynchSig$$

$$ParProgStep\ \mathrel{\widehat{=}}\ ParIndStep \lor ParSynchStep$$

$$\forall\ ParProgStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

Finally, we define the enabled arcs for visible communication or internal progress. Since synchronisation insists on shared communication, it might affect the events an

action accepts. Acceptable events are calculated according to whether they are synchronisation events or not. A synchronisation event can be accepted only when both actions are accepting it. An event that does not need synchronisation or internal progress can be accepted if either action is accepting it. As the *enabled* function represents acceptances sets in our semantics, it is responsible for encoding the expected availability of visible communication according to the synchronised behaviour of parallelism. Apart from the case where the parallelism diverges, there are two possible outcomes from the events of enabled arcs on each side: (i) they must synchronise when contained in *cs*; or (ii) they can occur independently otherwise, and this includes internal progress.

To properly define *enabled* arcs for these two cases, we need to define two infix operators. The functions $(\_ \searrow \_)$ and $(\_ \diamond \_)$ distribute set difference and set intersection over power set respectively.

syntax $\searrow$ *infun3* `\_\searrow\_`
syntax $\diamond$ *infun3* `\_ \diamond \_`

$$
\begin{array}{|l}
[X] \\
\_ \searrow \_ : \mathbb{P}(\mathbb{P}\ X) \times \mathbb{P}\ X \to \mathbb{P}(\mathbb{P}\ X) \\
\_ \diamond \_ : \mathbb{P}(\mathbb{P}\ X) \times \mathbb{P}\ X \to \mathbb{P}(\mathbb{P}\ X) \\
\hline
\forall\ P : \mathbb{P}(\mathbb{P}\ X);\ p : \mathbb{P}\ X \bullet P \searrow p = \{\ q : \mathbb{P}\ X \mid q \in P \bullet q \setminus p\ \} \\
\forall\ P : \mathbb{P}(\mathbb{P}\ X);\ p : \mathbb{P}\ X \bullet P \diamond p\ = \{\ q : \mathbb{P}\ X \mid q \in P \bullet q \cap p\ \}
\end{array}
$$

The distribution over set operators is used to avoid flattening each individual arc enabled, hence loosing information about their individual properties. That is, as *enabled* returns a set of arcs (or a set of set of events), flattening this set of sets would result in loss of information about what had been accepted (or refused) by each individual arc. Thus, the flattening would compromise information about the failures of an action, while calculating acceptances sets of actions during normalisation algorithm (see Section 4.5).

The *enabled* arcs for progress in parallelism is defined next. In the first expression, arcs that are outside *cs* represent the events that can happen freely without synchronisation. Therefore, we apply set difference distributed over the union of the set of *enabled* arcs from each action, hence removing the synchronisation events determined by *cs* from these enabled arcs. The second expression represents the enabled arcs that must agree with the synchronisation events from *cs*. For that we apply set intersection distributed over the intersection of the set of *enabled* arcs from each action, hence enforcing the synchronisation on the events from *cs*. These restrictions on *enabled* arcs are defining what each action of the parallelism is accepting to do because of the need to synchronise.

$$
\begin{array}{l}
\forall\ ParProgSig \bullet enabled\ ((S, P), E) = \\
\quad (\ (enabled\ ((S, A), E) \cup enabled\ ((S, B), E)) \searrow csext\ (E, cs)\ ) \cup \\
\quad (\ (enabled\ ((S, A), E) \cap enabled\ ((S, B), E)) \diamond csext\ (E, cs)\ )
\end{array}
$$

Nevertheless, these restrictions enforced by the distributed set difference and intersection can also introduce empty arcs in a situation where only visible communication would be available. We cannot remove these arcs as they could have been present on either action due to internal progress. For the case where internal events do exist, the *ParIndStep* is satisfied and can handle them. On the other hand, for the scenario where empty arcs originally unavailable might be introduced, the restrictions

on *ParIndStep* enforce that there must exist an enabled transition from A to A' via some arc with visible communication outside cs in order for the progress to take place. Therefore, schema *ParIndStep* would not be defined for the introduced silent transition. Thus, this silent transition is then ruled out as there is no condition under which schema *ParProgSig* can be satisfied. Let us show how this works in more detail. For example, a pantomime horse process [Ros97, Chapter 2] without state defined by parallel composition of both the front and back parts is given as:

$$\begin{array}{lcl} F & \widehat{=} & (forward \rightarrow F) \,\Box\, (back \rightarrow F) \,\Box\, (nod \rightarrow F) \,\Box \\ & & (weigh \rightarrow F) \,\Box\, (sthelse \rightarrow F) \\ B & \widehat{=} & (forward \rightarrow B) \,\Box\, (back \rightarrow B) \,\Box\, (wag \rightarrow B) \,\Box \\ & & (kick \rightarrow B) \,\Box\, (sthelse \rightarrow B) \\ H & \widehat{=} & F \,[\![\, \{\,\} \mid \{\!|forward, back|\!\} \mid \{\,\} \,]\!]\, B \end{array}$$

Both the front (F) and the back (B) of the horse can do events, some of which they must agree on. The enabled events would be calculated as follows

$$\begin{array}{lcl} enabled\,((S, F), E) & = & \{\,\{\,forward\,\}, \{\,back\,\}, \{\,nod\,\}, \{\,weigh\,\}, \{\,sthelse\,\}\,\} \\ enabled\,((S, B), E) & = & \{\,\{\,forward\,\}, \{\,back\,\}, \{\,wag\,\}, \{\,kick\,\}, \{\,sthelse\,\}\,\} \\ enabled\,((S, H), E) & = & \\ & & \left(\begin{array}{l} \{\,\{\,forward\,\}, \{\,back\,\}, \{\,nod\,\}, \{\,weigh\,\}, \\ \{\,wag\,\}, \{\,kick\,\}, \{\,sthelse\,\}\,\} \end{array}\right) \searrow \{\,forward, back\,\} \Big) \cup \\ & & \Big(\, \{\,\{\,forward\,\}, \{\,back\,\}, \{\,sthelse\,\}\,\} \diamond \{\,forward, back\,\} \,\Big) \\ & = & \{\,\varnothing, \varnothing, \{\,nod\,\}, \{\,weigh\,\}, \{\,wag\,\}, \{\,kick\,\}, \{\,sthelse\,\}\,\} \cup \\ & & \{\,\{\,forward\,\}, \{\,back\,\}\,\} \\ & = & \{\,\varnothing, \{\,nod\,\}, \{\,weigh\,\}, \{\,wag\,\}, \{\,kick\,\}, \\ & & \{\,sthelse\,\}, \{\,forward\,\}, \{\,back\,\}\,\} \end{array}$$

The enabled events which do not need synchronisation are

$$\{\,\{\,nod\,\}, \{\,weigh\,\}, \{\,wag\,\}, \{\,kick\,\}, \{\,sthelse\,\}\,\}$$

They can occur through *ParIndStep* because they satisfy

$$\begin{array}{lcl} lbl! \cap csext\,(E, cs) = \varnothing & \Rightarrow & lbl! \cap \{\,forward, back\,\} = \varnothing \\ lbl! \in enabled\,((S, side?), E) & \Rightarrow & \left(\begin{array}{c} lbl! \in enabled\,((S, A), E) \vee \\ lbl! \in enabled\,((S, A), E) \end{array}\right) \end{array}$$

The events that must synchronise are

$$\{\,\{\,forward\,\}, \{\,back\,\}\,\}$$

They can occur through *ParSynchStep* because they satisfy

$$\begin{array}{l} aL! \in enabled\,((S, A), E) \wedge aR! \in enabled\,((S, B), E) \\ \wedge\ aL! \cap aR! \neq \varnothing \wedge lbl! = aL! \cap aR! \end{array}$$

The case of concern is the originally unavailable silent transition now introduced through the calculation of enabled. The only case through which it could take place is *ParIndStep*, since *ParSynchStep* explicitly requires that ($lbl! \neq \varnothing$). So, from *ParIndStep* we have that

$$\begin{array}{lcl} \varnothing \cap csext\,(E, cs) = \varnothing & \Rightarrow & \varnothing \cap \{\,forward, back\,\} = \varnothing \\ \varnothing \in enabled\,((S, side?), E) & \Rightarrow & \left(\begin{array}{c} \varnothing \in enabled\,((S, A), E) \vee \\ \varnothing \in enabled\,((S, A), E) \end{array}\right) \end{array}$$

The second equation having the empty set enabled can never be satisfied for either

case. Therefore, this transition is ruled out since it does not meet any condition of the schema *ParProgStep*.

$$ParAllSig \mathrel{\widehat=} ParProgSig \lor ParSkipSig \lor ParStop$$

$$ParAllStep \mathrel{\widehat=} ParProgStep \lor ParSkipStep$$

The complete definition for parallelism is just disjunction of the progress cases, the termination case, and deadlock. As pre distributes through disjunction, the complete applicability theorem is omitted here for simplicity.

3.12 Hiding—(A \ hs)

Event concealment via hiding is used to abstract communication from specifications. The signature for hiding extends the unary signature with a channel expression for the hiding set, and also ensures that events from hs being hidden come from channels previously defined in the environment.

HideSig

$$UnSig;\ hs : CSExpr$$

$$(E, hs) \in \mathrm{dom}\ csext$$
$$P = hide\,(A, hs)$$

We have three cases to consider: (i) where the hiding terminates; (ii) where internal progress occurs due to either silent transitions, or concealed events; and (iii) where a visible communication not being hidden occurs. The signature for the case of termination insists on A being Skip; the step leads to Skip via a silent transition without state changes: it evaluates $hide\,(\mathsf{Skip}, hs)$ to Skip.

$$HideSkipSig \mathrel{\widehat=} [\, HideSig \mid A = \mathsf{Skip} \,]$$

$$\forall\ HideSkipSig \bullet enabled\,((S, P), E) = \{\, \varnothing \,\}$$

$$HideSkipStep \mathrel{\widehat=} [\, HideSig;\ ReadOnlyStep;\ SilentStep \mid P' = \mathsf{Skip} \,]$$

$$\forall\ HideSkipStep \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E') \,\}$$

theorem tHideSkipStepAppl
$\forall\ HideSkipSig \bullet \mathrm{pre}\ HideSkipStep$

The case of events being hidden is next; it is characterised by the existence of some arc available from A, which is contained in the set of events being hidden.

HideIntVC

$$HideSig;\ a! : Arc$$

$$a! \in enabled\,((S, A), E)$$
$$a! \subseteq csext\,(E, hs)$$

This covers both internal progress when $(a! = \varnothing)$, as$(\varnothing \subseteq X)$ for any X, and

concealed communication when $(a! \subseteq csext\,(E, hs))$. The signature insists that there exists a transition for A in such circumstances.

HideIntSig

$$HideSig$$

$$(\exists\, Sig'[A'/P'];\ a! : Arc \bullet HideIntVC \land ((S', A'), E') \in arcStep\,(((S, A), E), a!)\,)$$

The step transition progresses to $hide\,(A', hs)$, where A' comes from the transition defined for A through $a!$ because of either internal progress or concealed events.

HideIntStep

$$HideIntSig;\ UnSig';\ SilentStep;\ HideIntVC$$

$$((S', A'), E') \in arcStep\,(((S, A), E), a!)$$
$$P' = hide\,(A', hs)$$

$$\forall\, HideIntStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$$

theorem tHideIntStepAppl
$\forall\, HideIntSig \bullet$ pre $HideIntStep$

Finally, the last case of visible communication not being hidden is characterised by the existence of some arc available from A, which is neither internal, nor being concealed.

HideExtVC

$$HideSig;\ a! : Arc$$

$$a! \in enabled\,((S, A), E)$$
$$a! \neq \varnothing$$
$$a! \cap csext\,(E, hs) = \varnothing$$

The signature for the case where events are not being hidden insists that there exists a transition defined for A in such circumstances.

HideExtSig

$$HideSig$$

$$(\exists\, Sig'[A'/P'];\ a! : Arc \bullet HideExtVC \land ((S', A'), E') \in arcStep\,(((S, A), E), a!)\,)$$

Similarly, the step transition progresses to $hide\,(A', hs)$, where A' comes from the transition defined for A through $a!$. The arc for this transition is defined by removing the concealed events from every arc enabled from A via the difference operator $(\_ \searrow \_)$ which distributes set difference over power set. Because of this application of *enabled* and $(\_ \searrow \_)$, we need two more ***Z/Eves*** rules about maximal types.

theorem rule rEnabledMaxType
$\forall\, S : USt;\ A : Action;\ E : Env \bullet$
$enabled((S, A), E) \in \mathbb{P}\,(\mathbb{P}\,(Name \times ZExpr))$

The applicability theorems for hiding were helpful to determine the weakest verifica-

tion condition needed for the application of hiding on each case.

theorem rule rCSExtResult
$\forall\ E : Env;\ hs : CSExpr \mid (E, hs) \in \mathrm{dom}\ csext \bullet$
$csext\,(E, hs) \in \mathbb{P}\,(Name \times ZExpr)$

HideExtStep
$HideExtSig;\ UnSig';\ Step;\ HideExtVC$

$lbl! \in enabled\,((S, A), E) \searrow csext\,(E, hs)$
$((S', A'), E') \in arcStep\,(((S, A), E), a!)$
$P' = hide\,(A', hs)$

$\forall\ HideExtStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E')\,\}$

theorem tHideExtStepAppl
$\forall\, HideExtSig \bullet \mathrm{pre}\ HideExtStep$

In CSP, the intuition for the refusals set of hide is that the events of hs being concealed from A will represent internal progress for $(\mathsf{A} \setminus \mathsf{hs})$ when A becomes stable [Ros97, p.199]. Therefore, these events must also be part of the refusals set of $(\mathsf{A} \setminus \mathsf{hs})$, as the UTP definition for hiding also hints [HJ98, p.213, Defintion 8.2.14]. Nevertheless, as *enabled* mentions what is available, we need to consider acceptances rather than refusals. Although (minimal) acceptances sets are not complementary to (maximal) refusals sets, they can be used for the purposes of model checking and encoding of the operational semantics, as exemplified by FDR's implementation usage of acceptances sets [Gol01]. Moreover, maximal (or minimal) sets are used as they are a more economical representation without compromising the original information. An example where the relationship between the refusals set and the acceptances set is not straightforward is given below. Let us define a CSP process P as

$$P \mathrel{\widehat{=}} (a \rightarrow P \,\Box\, b \rightarrow P) \setminus \{\!|\, b \,|\!\}$$

If we apply the hide-step law [Ros97, p.81] we get the equivalent process

$$Q \mathrel{\widehat{=}} (a \rightarrow \setminus \{\!|\, b \,|\!\}) \sqcap (a \rightarrow Q \setminus \{\!|\, b \,|\!\} \,\Box\, Q \setminus \{\!|\, b \,|\!\})$$

To show that acceptances set are not always complementary to refusals sets, one just needs to calculate the initial events of the automata representing these processes. For P, the initial events are $\{\, a \,\}$, the union of those from each without b, whereas for Q' the initial events are $\{\, \tau \,\}$. The theoretical motivation for the use of refusals or acceptances sets in the CSP models is discussed in [Ros97, Section 11.4, pp.278]. Practical motivations for the use of acceptances sets are emphasised in [Ros94b, Gol01]. Our reason for using acceptances sets are similarly practical: they tend to be smaller and easier to calculate than refusals sets. Like in parallelism, to avoid losing information about refusals while concealing events of hs from A, we must not flatten the events from *enabled* arcs. For example, suppose that the maximal refusals of A are given as

$$A.ref' \in \mathbb{P}\ Arc \land A.ref' = \{\,\{\, d, e, f \,\}, \{\, b, c, d, e \,\}, \{\, a, b, c, e, f \,\}\,\}$$

where let us assume all events available from Σ, and the following events from set hs

being hidden as defined below

$$\Sigma = \{ a, b, c, d, e, f \}$$
$$hs = clist\,(a, d) \;\Rightarrow\; csext\,(E, hs) = \{ a, d \}$$

So, the maximal refusals of $(A \setminus hs)$ must now contain hs on every arc

$$(A \setminus hs).ref' = \{ \{ a, d, e, f \}, \{ a, b, c, d, e \}, \Sigma \}$$

As we are dealing with minimal acceptances rather than maximal refusals sets because of empirical advantages [RSR$^+$01, Chapter 4], the corresponding acceptances sets are

$$\begin{array}{lcl} A.accs' & = & \{ \{ a, b, c \}, \{ a, f \}, \{ d \} \} \\ (A \setminus hs).accs' & = & \{ \{ b, c \}, \{ f \}, \varnothing \} \end{array}$$

The complete definition for progress on hiding is defined next as the disjunction of the last two cases. We use set difference for acceptances wherever we had set union for refusals, distributed over the set of arcs enabled for A using function $(\_ \searrow \_)$.

$$HideProgSig \;\widehat{=}\; HideIntSig \lor HideExtSig$$

$$HideProgStep \;\widehat{=}\; HideIntStep \lor HideExtStep$$

$$\forall\, HideProgSig \bullet enabled\,((S, P), E) = enabled\,((S, A), E) \searrow csext\,(E, hs)$$

$$\forall\, HideProgStep \bullet arcStep\,(((S, P), E), lbl!) = \{ ((S', P'), E') \}$$

$$HideAllSig \;\widehat{=}\; HideSkipSig \lor HideIntSig \lor HideExtSig$$

$$HideAllStep \;\widehat{=}\; HideSkipStep \lor HideIntStep \lor HideExtStep$$

The complete definition for hiding is just the disjunction of the progressing case, and the termination case. Again, the applicability theorem is omitted as it holds directly because pre distributes through disjunction.

3.13 Local environments

Local environments are used for extensions of the user state and environment: they appear during the evaluation of other actions, and they are not available in the *Circus* syntax (see Figure 2.2 in p. 13). We define two local environments: one for (implicit or explicit) local variable declarations, and another for local actions declarations, required in the semantics of recursion.

They consist of a tuple containing local information and an action A where the information is visible. The information in the tuple is considered in the evaluation of A, but it is not included in the state or the environment resulting from the evaluation of the local environment itself. In this way, the environment is kept local.

3.13.1 Implicit variable declaration—letvar $(((\mathsf{x}, \mathsf{T}), \mathsf{e}), \mathsf{A})$

A local environment for variables is written with the *letvar* action.It appears as the result of the step transition for (explicit) variable declaration via action *circvar* (see Section 3.6.1), and as the implicit variable declaration for input prefixing (see Section 3.5.3). Its signature extends a unary signature with a variable name and its type, together with a valid value when evaluated in the current state.

LetVarSig

$$UnSig;\ \mathsf{x} : Name;\ \mathsf{T} : Type;\ \mathsf{e} : ZExpr$$

$$evalE\ (S, \mathsf{e}) \in \mathsf{T}$$
$$P = letvar\ (((\mathsf{x}, \mathsf{T}), \mathsf{e}), A)$$

We have two cases to consider: termination, and progress of A. The local environment terminates whenever its local action is Skip.

$$LetVarSkipSig \mathrel{\widehat=} [\, LetVarSig \mid A = \mathsf{Skip} \,]$$

The effect of evaluating the environment upon termination leads to Skip through a silent transition where no state change is needed.

$$\forall\ LetVarSkipSig \bullet enabled\ ((S, P), E) = \{ \varnothing \}$$

$$LetVarSkipStep \mathrel{\widehat=} [\, LetVarSkipSig;\ ReadOnlyStep;\ SilentStep \mid P' = \mathsf{Skip} \,]$$

$$\forall\ LetVarSkipStep \bullet arcStep\ (((S, P), E), lbl!) = \{ ((S', P'), E) \}$$

theorem tLetVarSkipStepAppl
$\forall\ LetVarSkipSig \bullet \text{pre}\ LetVarSkipStep$

The definition of the case in which A makes progress requires extension of the user state and node environment, followed by evaluation of A, and finally, removal of such information from the node environment. For this, we define signatures for extension and removal first. In an extension signature, the local variable x has not been previously declared in neither the environment nor the user state, as defined by the next schema.

LetVarSigExtend

$$LetVarSig$$

$$\mathsf{x} \in E.fresh \land \mathsf{x} \notin \text{ran}\ S.uvars$$

Conversely, the signature for the transition in which the information about x is removed requires that x is known.

LetVarSigRemove

$$LetVarSig$$

$$\mathsf{x} \notin E.fresh \land \mathsf{x} \in \text{ran}\ S.uvars \land S.utypes \neq \langle\rangle$$

Some known facts we need to tell ***Z/Eves*** for the next proofs are added using addi-

tional theorems.

theorem rule tUStUVarsNameHasValue
$\forall S : USt;\ \mathsf{x} : Name \mid \mathsf{x} \in \operatorname{ran} S.uvars \bullet \mathsf{x} \in \operatorname{dom} S.valueOf$

theorem rule tUStUVarsNameIsFromNonEmptyUVars
$\forall S : USt;\ \mathsf{x} : Name \mid \mathsf{x} \in \operatorname{ran} S.uvars \bullet \neg S.uvars = \langle\rangle$

For the intermediate step where the local action executes with the extended components, we define the next signature *LetEnvLocalSig*: it sets the minimum requirements for schema composition. That is, although it is a signature, because there is a hidden intermediate step, it does need to mention the after-state components of this additional step.

$$LetEnvLocalSig \mathrel{\widehat=} Step \land UnSig'$$

The signature for *letvar* is defined next: it does not allow changes on channel and action components of the environment.

LetVarLocalSig

$LetVarSig;\ LetEnvLocalSig$

$E'.cType = E.cType$
$E'.aCtx = E.aCtx$

Before defining the local operations, we need a some more ***Z/Eves*** rules.

theorem frule fUStUTypesSeqMaxType
$\forall S : USt \bullet S.utypes \in \operatorname{seq}(\mathbb{P}\ ZExpr)$

theorem frule fLetVarLocalSigTMaxType
$LetVarLocalSig \Rightarrow \mathsf{T} \in \mathbb{P}\ ZExpr$

Next, we define the local extension to the node required by evaluation of *letvar*. The user state needs to include the declaration x having type T, where its initial value is the evaluation of e in the current state. This is achieved via sequence concatenation for name and type, and overriding for the evaluated value. Similarly, the environment also needs to be extended with the declaration of x mapped to T through overriding.

LetVarLocalExtend

$LetVarSigExtend;\ LetVarLocalSig$

$E'.vType = E.vType \oplus \{\mathsf{x} \mapsto \mathsf{T}\}$

$S'.uvars = S.uvars \frown \langle \mathsf{x} \rangle$
$S'.utypes = S.utypes \frown \langle \mathsf{T} \rangle$
$S'.valueOf = S.valueOf \oplus \{\mathsf{x} \mapsto evalE\,(S, \mathsf{e})\}$

Conversely, we remove the local information using domain anti-restriction, and the

front function for sequences.

LetVarLocalRemove

LetVarSigRemove; *LetVarLocalSig*

$$E'.vType = \{ \mathsf{x} \} \lhd E.vType$$

$$S'.uvars = front\,(S.uvars)$$
$$S'.utypes = front\,(S.utypes)$$
$$S'.valueOf = \{ \mathsf{x} \} \lhd S.valueOf$$

This is enough for the definition of *enabled* for *letvar*: it is the *enabled* arcs of the local action A in the locally extended user state and node environment.

$$\forall\ LetVarLocalExtend \bullet enabled\,((S, P), E) = enabled\,((S', A), E')$$

The local step transition is defined next. We assume that it already contains extended information on its initial state, therefore, we use the signature for removal as defined by *LetVarSigRemove*. If the local configuration $((S, A), E)$, where S and E have already been extended, leads to $((S', A'), E')$, the next program is the local environment within A', where the (possibly) new value for e comes from S', provided S' remained with the same structure ($S.uvars = S'.uvars$).

LetVarLocalStep

LetVarSigRemove; *LetEnvLocalSig*; *Step*

$$lbl! \in enabled\,((S, A), E)$$
$$S.uvars = S'.uvars$$
$$((S', A'), E') \in arcStep\,(((S, A), E), lbl!)$$
$$P' = letvar\,(((\mathsf{x}, \mathsf{T}), (S'.valueOf\ \mathsf{x})), A')$$

theorem frule fEnvFreshType
$\forall\ E : Env \bullet E.fresh \in \mathbb{P}\ Name$

The complete step for the case of progress of the local action is defined via schema composition. Firstly, the original state is extended with local information via the schema *LetVarLocalExtend*. Secondly, this extended state is now the before-state where the actual step transition is defined via *LetVarLocalStep*. Finally, the local information is removed from the after-state of *LetVarLocalStep* via schema *LetVarLocalRemove*.

$$LetVarProgStep \mathrel{\widehat=} LetVarLocalExtend \fatsemi LetVarLocalStep \fatsemi LetVarLocalRemove$$

$$\forall\ LetVarProgStep \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E) \,\}$$

The complete signature for progress is defined next by schema *LetVarProgSig*.

LetVarProgSig

LetVarSig

$$(\exists\ UnSig';\ a : Arc \bullet LetVarProgStep[a/lbl!])$$

From the original signature, there must exist an after-state according to the operations

performed via schema *LetVarProgStep*.

theorem tLetVarProgStepAppl
$\forall$ *LetVarProgSig* • pre *LetVarProgStep*

$$LetVarAllSig \mathrel{\widehat=} LetVarSkipSig \lor LetVarProgSig$$

$$LetVarAllStep \mathrel{\widehat=} LetVarSkipStep \lor LetVarProgStep$$

The complete definition is the disjunction of the termination case and the progress case, and the applicability theorem holds directly.

3.13.2 Implicit action declaration—$\mathsf{letmu}\,((\mathsf{X}, \mathsf{act}), \mathsf{A})$

Local environment for implicitly declared actions is written with the *letmu* action. It appears as the result of the step transition for recursion (see Section 3.8.2). Its signature extends a unary signature with a name and a local action.

LetMuSig

$$UnSig;\ \mathsf{X} : Name;\ act : Action$$

$$P = letmu\,((\mathsf{X}, act), A)$$

As before, we have two cases to consider: termination and progress. The local environment terminates whenever its local action is Skip, and the definitions are similar to those of *letvar*.

$$LetMuSkipSig \mathrel{\widehat=} [\, LetMuSig \mid A = \mathsf{Skip} \,]$$

$$\forall\ LetMuSkipSig \bullet enabled\,((S, P), E) = \{\, \varnothing \,\}$$

$$LetMuSkipStep \mathrel{\widehat=} [\, LetMuSig;\ ReadOnlyStep;\ SilentStep \mid P' = \mathsf{Skip} \,]$$

$$\forall\ LetMuSkipStep \bullet arcStep\,(((S, P), E), lbl!) = \{\, ((S', P'), E) \,\}$$

theorem tLetMuSkipStepAppl
$\forall$ *LetMuSkipSig* • pre *LetMuSkipStep*

For *letmu*, only the environment needs extension, since no information about declared actions is present in the user state. The signature requires that X has not been previously declared in the environment, as defined by the next schema.

LetMuSigExtend

$$LetMuSig$$

$$\mathsf{X} \in E.fresh$$

Conversely, for removing the additional information as defined by the next schema,

we require that X is known ($\mathsf{X} \notin E.fresh$).

LetMuSigRemove

$$LetMuSig$$

$$\mathsf{X} \notin E.fresh$$

Once more, ***Z/Eves*** requires an additional fact for the next proofs.

theorem rule tEnvFreshNameIsNotInActs
$\forall E : Env; \mathsf{X} : Name \mid \mathsf{X} \in E.fresh \bullet \neg \mathsf{X} \in E.acts$

For the intermediate step we reuse the definition of *LetEnvLocalSig*. The signature for *letmu* is defined next: it does not allow changes in the state, and in the channel and the variable components of *Env*.

LetMuLocalSig

$$LetMuSig;\ LetEnvLocalSig$$

$$S' = S$$
$$E'.cType = E.cType$$
$$E'.vType = E.vType$$

The local extension by *letmu* is also similar: we need to update action context ($E.aCtx$) in the environment.

LetMuLocalSigExtend

$$LetMuSigExtend;\ LetMuLocalSig$$

$$E'.aCtx = E.aCtx \oplus \{\mathsf{X} \mapsto act\}$$

Similarly, removal uses domain restriction again.

LetMuLocalSigRemove

$$LetMuSigRemove;\ LetMuLocalSig$$

$$E'.aCtx = \{\mathsf{X}\} \lhd E.aCtx$$

The following definitions are just like those for *letvar*, but with the necessary adjustments. The enabled arcs are the same as those of A in an extended environment defined according to schema *LetMuLocalSigExtend*.

$$\forall\ LetMuLocalSigExtend \bullet enabled\ ((S, P), E) = enabled\ ((S', A), E')$$

The evaluation of the local environment containing the name X for the recursive call, and the related action *act* is given next.

LetMuLocalStep

$$LetMuSigRemove;\ LetEnvLocalSig;\ Step$$

$$lbl! \in enabled\ ((S, A), E)$$
$$((S', A'), E') \in arcStep\ (((S, A), E), lbl!)$$
$$P' = letmu\ ((\mathsf{X}, act), A')$$

As before, it is defined using schema composition by firstly extending the signature

with the local environment information, performing the transition step within the extended environment, and finally, restoring the environment.

$$LetMuProgStep \mathrel{\widehat=} LetMuLocalSigExtend \fatsemi LetMuLocalStep \fatsemi LetMuLocalSigRemove$$

$$\forall\ LetMuProgStep \bullet arcStep\,(((S, P), E), lbl!) = \{\,((S', P'), E)\,\}$$

$$\begin{array}{|l} LetMuProgSig \\ \hline LetMuSig \\ \hline (\exists\ UnSig';\ a : Arc \bullet LetMuProgStep[a/lbl!]\,) \\ \hline \end{array}$$

theorem tLetMuProgStepAppl
$\forall\ LetMuProgSig \bullet \text{pre}\ LetMuProgStep$

The complete definition is the disjunction of the termination and the progress cases.

$$LetMuAllSig \mathrel{\widehat=} LetMuSkipSig \lor LetMuProgSig$$

$$LetMuAllStep \mathrel{\widehat=} LetMuSkipStep \lor LetMuProgStep$$

This completes the definition of local environment for implicity action declaration, as well as the operational semantics.

3.14 Animating the operational semantics

The ***Z/Eves*** manual suggests that one can use the tool to prove conjectures about schemas as a means to animate Z specifications [Saa99b, Section 3.2.5]. As we have defined the operational semantics using schemas, it is possible to animate *Circus* programs in ***Z/Eves*** by combining the definitions of the step transitions of the operators of a program via schema composition. For example, the failed attempt at proving such conjectures leads to the predicate representing the values of configurations reached after the execution of the composed schema.

In [Fre05b], we have defined a simplified version of the operational semantics to enable animation in both ***Z/Eves*** and Jaza, where a single channel and restricted (yet expressive) operations were used. A simplified semantics has also been used for animation with Jaza and [WCF05a]. Such exercise was useful for understanding and validation of the semantics, and uncovered subtleties that might have been kept hidden otherwise. Details of this simplified version and animation scripts for ***Z/Eves*** and Jaza are available in the appendix.

3.15 Comparison with the semantics of CSP_M

This chapter presents an operational semantics for model checking a subset of *Circus*, and it is based on the operational semantics of CSP_M [Sca98]. Nevertheless, due to the presence of infinite inscriptions in a finite automata, the operational semantics of *Circus* presents some important differences. These arise due to the necessity of

representing schema expressions, assignments, and other characteristics of a state-rich language such as *Circus*.

The main difference from the operational semantics of CSP_M is the representation of arcs as sets of events with (possibly unevaluated) symbols instead of a single event. The use of unevaluated symbols allows us to represent common features of Z that are not covered by FDR, such as uninitialised state, loosely defined components, infinite data types, and so on. The arcs with sets of events also allow us to include UTP observational variables, as well as other semantic constraints as extra condition through set comprehension notation.

In FDR, there are two special events: ✓ (tick), and τ (tau). The former represents successful termination, whereas the latter represents internal progress. Deadlock is represented as lack of immediately available events (*i.e.,* everything is being refused), whereas divergence is represented as an infinite sequence of τ's due to a τ-loop in the automaton. In *Circus* we have no special events. Internal progress is represented by an unlabelled transition via an empty arc such as recursion unfolding and internal choice, whereas termination is represented by the terminal configuration with action Skip. As in FDR, deadlock is represented via lack of immediately visible events (*i.e.,* complete refusals), which happens whenever there is no arc *enabled* immediately ($enabled = \varnothing$). Note that this is different from only internal progress immediately available ($enabled = \{ \varnothing \}$).

Divergence is represented by a loop of unlabelled transitions (a silent loop). In FDR, these τ-loops representing divergence are recognised via a transitive closure on a transition system restricted to τ events only on the automaton edges [Ros94b], where the standard implementation is depth first search (DFS). Research on a parallel version of FDR using graph pruning to detect divergence is under development in [MH00]. In Circus, divergence is recognised similarly as a loop of unlabelled transitions with empty arcs instead of τ's.

Nevertheless, in the prototype we include information about $okay'$ on the nodes of explicitly divergent actions, such as schema expressions outside their preconditions. Divergence check in FDR is a known bottleneck due the inefficiencies of DFS. Our experiments on using $okay'$ to characterise early (or explicit) divergence might give an important performance improvement for the implementation of divergence detection because it might be possible to avoid DFS. Instead, a more efficient search such as parallel variations of breath first search (BFS) could be used [OV96, Jon83]. The outcome of this investigation and the parallel implementation of other model checking algorithms are left as future work.

Surprisingly, FDR does not define a primitive divergent process explicitly, such as **div** from [Ros97]. Instead, one needs to introduce it explicitly via hidden events as in

```
channel c
A = c -> A
div = A \ {|c|}
```

The primitive process *CHAOS* in FDR is defined in CSP_M as the generalised nondeterministic (internal) choice over all events from the set A or deadlock.

```
CHAOS(A) = |~| x:A @ x -> CHAOS(A) |~| STOP
```

In FDR, *CHAOS* is the most nondeterministic divergence-free process. In *Circus*, that is different: Chaos is like Hoare's *CHAOS* [Hoa85] or Roscoe's **div**, the bottom element in the lattice of the failures-divergences model.

Another important consequence of having automaton arcs with sets of events is the ability to deal with infinite data types in finite automata. This mainly appears in the nodes representing input prefixing containing infinite data types. In FDR, each event initially available for an input prefixing is expanded into an external choice of each possible event. For example, a CSP_M process such as

```
P = c?x: {y | y <- {0..3}} -> P
```

is expanded in FDR to the equivalent process

```
P = (c.0 -> P) [] (c.1 -> P) [] (c.2 -> P) [] (c.3 -> P)
```

Such expansion is unsuitable for infinite (or unbounded) types, or expressions with loosely defined components commonly present in Z specifications. Our encoding of sets in the arcs enables an implementation of input prefixing that avoids this (possibly infinite) expansion to external choice, hence tackling directly the state explosion problem in model checking state-rich specifications via symbolic reasoning for these cases. The onus to pay is the possibility of theorem proving when unevaluated symbols have complex properties involving their types or possible values. For simple (possibly infinite) types, however, automatic model checking might still be possible.

Like in Roscoe's CSP and FDR, in our approach to model checking, termination cannot be refused. Therefore, an external choice such as $(\mathsf{Skip} \Box \mathsf{P})$ is in fact an internal choice, as the environment has no control over the possibility of termination being chosen (or being refused). Different views are adopted in different versions of CSP. For example, Hoare's CSP forbids the choice of termination in an external choice [Hoa85, Chapter 5], whereas Schneider's CSP requires co-operation with the external environment when termination is offered in an external choice [Sch00, p.76]. The motivation in Hoare's CSP is simplicity, whereas in Schneider's CSP it is due to extensions with real-time. For *Circus* and Roscoe's CSP, termination is considered internal as the external environment has no control upon its occurrence.

In terms of syntax, output prefixing in *Circus* such as $\mathsf{c!v}$ and $\mathsf{c.v}$ can be used interchangeably; however, in FDR $\mathsf{c.v}$ can only be used for literal values v, whereas $\mathsf{c!v}$ also allows expressions such as $\mathsf{c!(x + y)}$, which are evaluated at compilation time. There might be some unpublished technicality behind this decision of which we are not aware. A possible explanation is that "!" is used to override bindings from multiple identifiers involving "?" and ".", as pointed out in [Ros97, p.27]. For instance, for a channel c with cross type $(T \times T \times T \times T)$ where $T == \{0, 1\}$, the communication $c?w.x!y.z$ is equivalent to $c?w?x!y!z$. That is, dots between a question (or exclamation) mark are transformed accordingly by the parser. Moreover, we perform a simple static check of output prefix values with respect to the channel type declared in the environment. This is different from FDR that performs dynamic typechecking for the value of output prefixing. For example, if the type of channel c is T and one has a process in FDR such as

```
A(x) = c!x -> SKIP
A(4)
```

FDR finds the problem on the output communication $c!x$ while performing the refinement search by issuing an error message (*"Value outside the channel protocol"*).

Another syntactic distinction is in the parallel operator of *Circus*. It differs from the one in FDR due to the presence of state partitions needed to avoid concurrent writing on the state from different actions. Semantically, the difference lies: (i) in the way we calculate the immediate accepted (or refused) arcs due to the presence of set of events

in the arcs of the automaton; and (ii) in the calculation of the merge from the after-state of the involved actions into the after-state of the parallelism, which is not needed in FDR. In FDR, the events from different arcs are flattened via distributed union. This flattening of acceptable arcs on enabling arcs is not a problem in FDR because arcs on the automaton represent single event instead of a set of events. Moreover, we define a function (*csext*) that defines the elements of channel set expressions which is similar to functions *extensions* and *productions* defined in FDR [Gol00, p.61].

3.16 Summary

In this chapter we present an operational semantics for model checking *Circus*. It includes the definition of a transition system for the subset of *Circus* defined in the BNF syntax given in Figure 2.2. We use a style of presentation based on the schema calculus such that pre and post conditions for each transition are separated as a signature and a step schema. The ***Z/Eves*** database for the operational semantics of this chapter can be found in [Fre05d].

Apart from clarity and animation, this is also useful for application of automatic tools which translate Z to JML annotations. This enables precise documentation of the Java prototype of Chapter 5. Furthermore, we prove an applicability theorem calculating the precondition of each step with respect to the corresponding signature. Often the description of operators is separated in cases, such as termination, internal activity, and visible progress of the operands. These cases were put together using schema disjunction. The whole operational semantics is also the disjunction of each definition of signature and step schemas for each defined action.

Although the operational semantics presented here has been typechecked and analysed for consistency and applicability with ***Z/Eves***, a correctness proof with respect to the denotational semantics is still pending. This is left as future research, as there is no mature version of UTP theories in a theorem prover yet, but research in this direction is already well-advanced [Oli06, OCW05, Nuk05]. Eventually, these theories will enable us to provide a complete mechanical proof of correctness of the operational semantics with respect to the denotational semantics, following the approach for linking theories in UTP briefly presented in Section 3.1, and fully detailed in [HJ98, Chapter 10].

At various points of our description, we expect a contextual analysis of the *Circus* programs to guarantee several properties. Wherever names for declaration of variables, channels, and actions are needed, we expect contextual analysis to guarantee a pool of fresh names from which we can choose. The declaration of channels, channel sets, name sets, and actions are expected to come from contextual analysis as the initial environment. Since we need to apply projection functions over values communicated for multi-part prefixing, we expect the types of these values to be well-formed. Name sets used in parallel operators are expected to be partitions of the user state without dashed components. Finally, channels from channel sets used for parallelism and hiding must be previously declared in the environment in order to be used. All this can be enforced by a *Circus* typechecker [Xav06].

Chapter 4

Model checking *Circus*

"Mathematical proof is like a shining diamond, the brightest and hardest piece of evidence available".
Bertrand Russell [Rus57]

This chapter presents a model checking strategy for *Circus* programs and its formal description, which is mechanised using ***Z/Eves***. It is concerned with key and distinctive aspects of state-rich languages. The strategy is explained in different layers of abstraction in a top-down fashion.

In the next section, we give a detailed discussion about refinement model checking in *Circus*. Next, Section 4.2 introduces the main ideas and problems related to the task of model checking state-rich languages, such as state explosion, theorem proving, symbolic automata, and so forth, and describes our approach to solve these problems. After that, Section 4.3 gives an informal presentation of our model checking strategy architectural components. The next four sections present the formalisation of our model checking strategy as abstract *Circus* specifications: the normalisation of the specification automaton from the refinement relation, the witness search, and the production of debugging information. The formalisation of these stages of the *Circus* model checking strategy has been typechecked and checked for consistency and applicability with ***Z/Eves***. Section 4.8, presents a sequential algorithm calculated using *Circus* refinement laws [CSW02, Cav97, Mor94] from the witness search abstract specification. The witness search was chosen for applying refinement calculation as it plays a central role in the model checking task. It establishes the correctness criterion for whether the refinement order holds or not, through an exhaustive search over both automata. The proof obligations generated by the application of refinement laws have also been discharged with ***Z/Eves***. Finally, Section 4.9 presents a summary and some final considerations.

4.1 Refinement model checking

In the literature, the term "model checking" is almost always related to temporal logic model checking [CGP00, Kur94, CGL90, McM93a]. In [Ros94b], the term is interpreted in a different way: it means establishing refinement based on language containment between two different automata. As in [Ros94b], for us model checking is the establishment of properties of a design given by a specification. As *Circus* is strongly based on the notion of refinement, our approach to model checking is also to establish refinement. Our model checker follows the approach of the theory of

refinement for model checking CSP [Ros94b, Ros97, BHR84, BR85] and its model checker tool architecture [Sca98, Gol01].

In *Circus*, the notion of refinement is expressed as a relation of improvement between two programs with respect to their levels of nondeterminism. A specification I is a refinement of a specification S, denoted by $S \sqsubseteq I$, if and only if, for all possible observations of I, there exists a corresponding observation in S. As usual, refinement establishes a partial order between S and I.

A partial order is a binary relation that is reflexive, antisymmetric, and transitive. This last property establishes an important argument for the selection of model checking through refinement instead of other available approaches [CGP00, McM93a, CW96]. That is, the transitivity of refinement allows *Circus* to be used throughout the entire software development process.

Model checking through refinement aims at establishing a refinement relation between two *Circus* models given as finite automata describing all observable behaviours. These automata are built from the operational semantics that represents the behaviour of *Circus* programs, as presented in Chapter 3. Refinement is established via an exhaustive search over the behaviour of these models, provided they can be represented with finite automata.

Refinement model checking supports stepwise development of systems, as in the connection pool example given in Section 2.3. Moreover, refinement can also be used to reason about system properties, which is the usual goal of model checking techniques. For example, if a property P of a program S given as predicate or a satisfiability formula can be represented as a *Circus* program, and if S refines P, that means the system satisfies the property. This is known in the literature as model checking property oriented specifications [RSR$^+$01, Sch98, Low96].

In order to enable refinement model checking with our exploration strategy, it is important to have at least one automaton operationally deterministic in order to save space in the refinement search structures [Gol01, p.133]. In this transformation process, to avoid loosing information about nondeterministic choices, we must make an accurate judgment about what is being accepted prior this transformation. Thus, one automaton in the refinement relation is transformed into a semantically equivalent normal form in order to become deterministic. Since after normalisation one of the automata is deterministic, any nondeterministic choice is certainly from the non-normalised one, hence accurate judgment on nondeterministic choices follows directly. This is important for the efficiency of debugging whilst recreating the trace of a counter-example on the automata, because the found trace is certain to be unique. Although this approach is not imperative and skipped in other refinement checking tools [PY96b], we believe normalisation is an important step during model checking. It usually improves performance considerably, as empirical experiences have shown over the years [Ros94b, Gol01].

4.2 Our approach to model checking

This section introduces the key points of our approach to model checking *Circus* programs. It includes the problems involved, and the proposed solutions.

4.2.1 Symbolic automata

The data structure used to represent *Circus* programs is a *Predicate Transition System* (PTS) generated by the operational semantics defined in Chapter 3. A PTS is a vari-

ation of a labelled transition system (LTS) [HMU01], where the arcs are represented by sets of events possibly defined using symbolic constant, and the value of state components are given by expressions also involving symbolic constants. This kind of symbolic automaton underlies the theory of automata we use for model checking (see [Fre04b, Chapter 4]). It is distinctive, as it enables the representation of the semantics of *Circus* programs, with complex statements, such as Z schemas or predicate calculus, through set comprehension notation.

The search strategy employed is a variation of *Breadth First Search* (BFS) that enables efficient parallelisation. BFS is used because if a witness is found, then it is always the cheapest possible in terms of necessary time and work effort for performing the search [Ros94b, Gol01]. The search is carried out by exploring mutually reachable arcs as node pairs from the automata on both sides of the refinement relation. It is divided in two stages: (i) compatibility checking between node pairs with respect to the refinement ordering criteria; and (ii) finding successors for compatible node pairs. The way these stages relate enables an efficient parallelisation known to improve the overall task to a great extent [MH00].

The search algorithm looks for incompatible node pairs that are reachable via a given trace from each automaton of the refinement relation. A trace is a sequence of events, or the language that these automata recognise. Despite the fact that each arc is labelled with sets, the language recognised by these automata is still a sequence of single events; however, as we allow symbols to appear in the trace, this might represent the set of allowed values of the type of the symbol. Moreover, since after normalisation nondeterminism is present in only one of the automata, each trace characterises a unique path that both automata are able to perform: an interesting advantage of having a normal form.

We use *Circus* itself to specify important components of the *Circus* model checker. Furthermore, the refinement calculus of *Circus* [CSW02] (and Z [Cav97]) is used to calculate a design for one of the most important components of the architecture: the witness search algorithm. The main motivation behind this path of development is not just correctness: we also illustrate the expressiveness of *Circus*.

4.2.2 State explosion problem on state-rich specifications

Our approach to check refinement between specifications or properties as automata follows from the same idea implemented by FDR. Nevertheless, the main problem while performing model checking for *Circus* through explicit state enumeration is the state explosion caused by the possibly infinite number of distinct states, for instance, from operations defined using schema expressions.

State explosion is a well-known problem in the model checking community [CGP00, CW96, Ros94b, Mot01]. It is tackled through an efficient search over an appropriate data structure that represents the specifications being analysed. An appropriate data structure is one that enables mechanical transformation to a human-readable form used for debugging purposes consuming the least amount of time and memory possible. We use symbolic automata that are able to represent in a finite state space, infinite data types. In our context, the main problem to be solved is how to represent and efficiently search such an automaton, which can contain operations over (possibly) infinite or unbounded finite data types.

4.2.3 Symbolic refinement model checking

Due to the possible presence of complex predicates on the arcs of the automaton representing a *Circus* program, a compromise between automation and expressiveness needs to be made, as it might not be possible to automatically decide the reachable configurations while finding successor pairs. Therefore, some sort of decision-making strategy must be employed.

We envisage three levels of automation our strategy can provide: (i) automatic decision procedures; (ii) interactive theorem proving; or (iii) user interaction. They establish the extent of desired automation versus expressiveness (see Table 4.1). The automatic approach provides an alternative to the use of a theorem prover to check refinement via explicit state enumeration. It works for *Circus* programs involving Z schemas that specify operations and commands that are restricted to finite data types involving decidable logic.

Automation	**Technique**	**Expressiveness**
Automatic	Explicit state enumeration	Restricted predicate calculus and simple finite types.
Automated	Proof obligations	Full predicate calculus and infinite types.
Interactive	Simple questions	Decidable predicate calculus and complex finite types.

Table 4.1: Automation levels for model checking

The second automated approach generates correctness conditions for refinements between more complex *Circus* programs, where infinite data types might be used. The proof obligations generated must be discharged later with a theorem prover, such as ***Z/Eves*** [Saa99b] and ProofPowerZ [Lem03], where additional theories about *Circus* and UTP are being developed [Oli06, Nuk05].

The final interactive approach is a simplified version of the automated approach. In this case tactics from a mature application-oriented theory might simplify proof obligations generated by schema operations to simple questions about set containment or set equality tests, for instance.

The prototype related of Chapter 5 implements these approaches by providing a pluggable theorem proving module based on oracles: entities that are queried for an answer about a formula whenever necessary. One can argue that this concern about automation is rather unusual for traditional model checking. Nevertheless, usual model checking techniques are mainly focused on behavioural aspects of systems, whereas in *Circus* we are addressing both behavioural and data aspects of a model.

For the definition of a symbolic automaton abstract data type, as well as correct transformation operations needed throughout the model checking task, we provide an automata theory inspired in Hopcroft [HMU01]. This theory has been formally mechanised using ***Z/Eves*** and is available in [Fre04b]. It includes data structures such as nondeterministic and deterministic predicate transition systems, with operations for querying immediately available arcs, stepping functions, transformation between different transition systems, and so forth. Moreover, a series of correctness properties for these data types are given as theorems. For instance, it includes language containment theorems, structure preserving theorems about transformations between different transition systems needed throughout the refinement search, and so on. This theory enables us to concretely represent the *Circus* operational semantics in a finite

state symbolic automaton.

By using a symbolic automaton to represent *Circus* programs where arcs have sets of events, we could also enable the embedding of UTP semantics through annotation in the nodes or restrictions in the arcs via set comprehension notation. This is an interesting aspect of our architecture, because it might enable the opportunity of extension that UTP offers (see Section 3.1 on linking theories). Whenever other paradigms such as mobility [Tan05] become available in UTP, we might be able to include them in the model checker by extending the *PTS* theory with appropriate annotations, as well as providing new refinement violation conditions if necessary. Therefore, we hope that by using this data structure, model checking *Circus* is as extensible as its theoretical background in UTP.

4.3 Architecture for a *Circus* model checker

Our model checker architecture has four components: (i) parser, (ii) typechecker, (iii) compiler, and (vi) refinement checker, as shown in Figure 4.1. The first two components are not addressed in this thesis, and we just present a brief overview of them. The third component is related to the operational semantics presented in Chapter 3; it is the implementation of a compiler able to construct a predicate transition system representing a *Circus* specification. Although the operational semantics of *Circus* is new, we consider the last component as the major contribution of this thesis. We give the description in *Circus* of the most important aspects of this component, which analyses the data structure provided by the compiler. That includes the preparation of the input *PTS* automata, the witness search algorithm, and the human-readable presentation of debugging information.

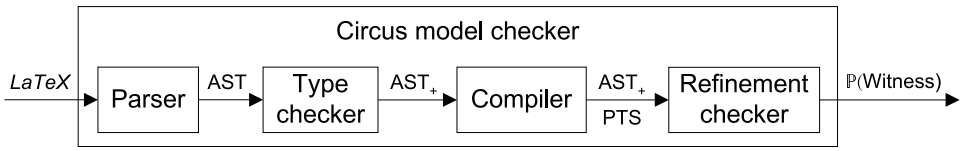

Figure 4.1: *Circus* model checker architecture

In what follows, we give a brief explanation of each component and its importance in the model checker tool. Later, we focus on the model checking component.

4.3.1 Parser

The parser transforms *Circus* programs given in LaTeX markup [BC02] into an abstract syntax tree (AST) [WB00] representing *Circus* programs as actions and Z Standard [Pan00] paragraphs; it is based on an extension to the *Community Z Tools* (CZT) project [MU05, MFMU05]. The *Circus* parser is an extension of the *CZT* parser for the Z Standard that also recognises the action language of *Circus*.

4.3.2 Typechecker

The typechecker receives an AST from the parser and annotates it with additional environment information for declarations of channels, channel sets, processes, actions, name sets, and local variables.

The theory and practice behind this component is under development in [Xav06]: it implements the type-system of *Circus*. The resulting AST_+ represents a well-formed *Circus* program where typechecking and scoping rules have already been resolved.

4.3.3 Compiler

The annotated AST_+ is given to the compiler that implements the operational semantics defined in Chapter 3. The compiler generates the transition system representing well-formed *Circus* programs that is necessary to perform the model checking algorithms. The construction and transformation process on the normal form automaton might generate verification conditions, as well as the need to evaluate expressions and predicates, hence theorem proving integration is demanded.

Since the operational semantics is calculated on-the-fly, further compilation still occurs while performing the exhaustive search related to the search algorithm.

4.3.4 Refinement checker

This component takes two compiled automata representing the specification and the implementation side of the refinement order. As we consider normalisation, divergence checking, witness search, and debugging the core of our architecture, their descriptions are given in *Circus* itself. Theorem proving is a module related to an external tool. An closer view of the refinement checker is presented in Figure 4.2.

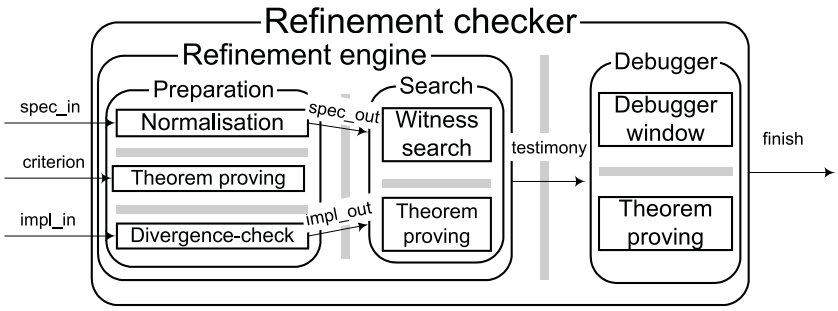

Figure 4.2: *Circus* refinement checker

The refinement checker process is defined in *Circus* by the parallel composition of two other processes, namely, the refinement engine and the debugger. The refinement engine process is composed of two other parts: one that prepares the received automata, and another that performs witness search. The debugger is divided in a series of debug windows representing the information gathered by the witness search. Whenever some sort of decision becomes necessary, these components can request support from an external theorem prover or the user as an environment running in parallel with the refinement checker.

The refinement engine receives the two automata from the compiler through channels *spec_in* and *impl_in* respectively. The channel *criterion* defines the level of detail (or the properties of interest) the witness search should look for. During the preparation, the normalisation process is responsible for transforming the specification automaton in order to make the accurate judgment on nondeterministic choices mentioned earlier. This is achieved by applying a semantics-preserving transformation on the specification automaton in order to make it deterministic, and without

silent transitions. The result is a normal form that is passed to the search engine (see the arrow labelled *spec_out* in Figure 4.2).

The divergence checking process is responsible for finding divergence in the implementation automaton; it distinguishes between ordinary, and divergent nodes. As occurred with the normalised specification, this divergence checked automaton is passed to the search engine (see the arrow labelled *impl_out* in Figure 4.2). This is achieved by finding loops on silent transitions over the automaton via DFS. Explicit divergence, such as Chaos or schema expressions outside their precondition, are detected directly by marking the relevant nodes during compilation.

After the preparation of the two automata, the witness search can be carried out. At this point, the exhaustive search on the state space is performed in order to prove refinement. When the exhaustive search is complete, if an incompatible pair is found, then a counter-example witness is generated showing the trace of the problem from the failing pair upwards to the root of the search in the transition system. Otherwise, if all node pairs are compatible, then a successful report is generated.

In Section 4.4, we present the necessary data structures, channels and the processes involved. Furthermore, a witness search sequential algorithm, calculated from the abstract specification using *Circus* refinement laws, is also presented. By using the refinement calculus to reach the code from the abstract specification, we ensure that the algorithm is correct by construction, and important properties such as loop invariants are precisely documented. Additionally, we also explain some important properties and proofs guaranteeing that transformations made on the involved automata by the preparation component are semantics-preserving.

Firstly, the formalisation of the normalisation process is important in order to ensure that the applied transformations do not incur semantic loss, hence not compromising the refinement proof established via the witness search. We are taking our own medicine to formally specify the key aspects of the model checker architecture in order to enhance the integrity of the prototype tool with respect to the formal abstract model. Secondly, formalising an abstract version of the witness search process is important not only for establishing the correctness of the refinement proof, but also for opening ground for the proof of correctness of possible implementations. For instance, we apply refinement laws to calculate a sequential algorithm. The derivation of a parallel version based on [MH00] is set as future research. Furthermore, our theory of automata, and the abstract specification of witness search can also be used to formally calculate the witness search algorithm of FDR. The properties exposed by the presented sequential version for *Circus* also explain properties of FDR's algorithm itself. Finally, the formalisation of the debugger strengthens the reliability on the information presented to the environment for the precise interpretation of failures.

Normalisation of the specification

The approach taken for normalisation is standard: the automaton from the specification side of the refinement relation is made deterministic with distinction of divergent states [Ros94b]. This solution follows from the previous successful experience of FDR, on which our architecture is based. One component of the refinement relation is transformed to become deterministic, with no silent transitions left and divergent states specially labelled. This transformed automaton, said to be in normal form, is then checked against the other, possibly nondeterministic automaton. Since one of the automata is deterministic, free of internal events, and with divergent states explicitly distinguished, visiting all possible combined traces of these automata implies that

the entire state space has been visited. Furthermore, every possible arc visited is now unique. This is crucial for efficiently rebuilding the witness of a failure from debugging information. The approach works correctly, provided that the transformed specification remains operationally deterministic, while preserving the information about its nondeterminism. Moreover, it must also preserve the semantic properties held by the original nondeterministic version with respect to every compatibility criterion under consideration. For example, the language recognised by the normal form and information about nondeterministic choices must be equivalent to the original automaton.

Although the normalisation usually shrinks the original automaton's size, it can increase the number of distinct states to the power set of the original number of states in the worst case. For instance, if the original automaton has 10 states, its deterministic version can have up to 2^{10} states. Fortunately, from empirical experience, this situation is unlikely to happen in real examples, because the sort of process that would be represented by an automaton with this problem is rather unusual. For instance, Roscoe's approach to model checking CSP illustrates such situation in [Ros94b, p.316].

An important decision is to select the specification side of the proposed refinement relation to normalise. The abstract specification is selected because it is likely to benefit more from nondeterminism elimination than the detailed implementation. Therefore, its number of distinct states is likely to be smaller. Another possible situation where the normal form could shrink the size of the original automaton happens when divergence is detected at earlier stages. In this case, no further investigation is needed and their acceptances sets are empty, hence space and time are saved.

An alternative to the use of normalisation would be to maintain both automata nondeterministic, and keep within every pair of nodes being checked the trace where this pair came from. This approach is taken by an experimental refinement model checker for CSP called ARC [PY96b, PY96a]. The results from their experiments show that the approach scaled well when compared with FDR (version 1). In the comparison with the improved version of FDR (version 2), this advantage no longer holds. From our own experiments so far, normalisation has shown to decrease the actual number of states and transitions necessary to represent and model check the specification against the implementation.

The normal form potentially reduces the time and space complexities for the witness search algorithm as well. First of all, as already said, since the normal form is deterministic, every trace through it is unique. Therefore, during the witness search, there is no need to keep track of the trace leading to the pair. Moreover, if the complexity of the search for finding successor node pairs is $\mathcal{O}(s * i^2)$ in the normal form, it would be $\mathcal{O}(s^2 * i^2)$ in the nondeterministic version, where s and i are the number of nodes from the specification and the implementation automata, respectively. That means the memory savings of the normal form grow exponentially as the number of states of the automaton grows, and the depth of the search increases. The significance here lies in the fact that usually an enormous amount of these pairs must be kept in memory while being checked.

Moreover, the determinism of the normal form is also useful for debugging purposes. As mention later in the refinement search algorithm (see Section 4.8), a trace containing node pairs, where one node comes from a deterministic transition system, enables one to rebuild the graph of an invalid trace from the faulty node up to the root of the search. Therefore, having a normal form usually lead to less memory needed to check for refinement.

Both alternatives have their advantages and compromises. In practice, the normalisation approach seems to work more efficiently. Throughout the remaining for-

malisation, we point out particular scenarios where the normalisation is effective. A benchmark comparing both approaches with and without normalisation is given in [PY96b]. Further detailed discussion on this topic in a more general context can also be found in [CH93b, HMU01, ASU86].

Divergence checking of the implementation

In CSP and FDR, divergence is represented as a infinite sequence of internal communications. Internal communication is represented with an internal event labelled τ. Thus, FDR calculates divergence information for the implementation via a depth-first search (DFS) looking for τ-loops. DFS is used as it is the standard implementation for finding loops in graphs [HMU01]. The DFS performed on the implementation is known as a performance bottleneck in FDR [Gol01, ST04, MH00].

The divergence checking of the implementation automaton is not interesting in itself, and standard search algorithms, such as efficient variations of DFS [DN93], can be employed. The main point is the possible presence of UTP observational variable $okay'$ (see Section 2.1) representing the possibility of divergence in the automaton as an aid for our search. It accounts for a possible performance improvement of our tool with respect to FDR. We expect to perform experiments and benchmarks in this direction as future research.

The formal definition for this process is straightforward. It defines a transitive closure over the implementation automata where the transition relation is restricted to silent arcs, in a way much similar to the one defined for detecting divergence for the normal form in Section 4.5. More details about the divergence checking formal specification can be found in [Fre04d].

Witness search

The witness search process is responsible for proving whether the proposed refinement ordering holds or not. If it does, a successful report is generated. If it does not hold, the process provides sufficient information that can be used to produce a suitable human-readable account of the failure. As it represents the most important aspect of our strategy, the entire process has been formally described in *Circus*, and a derivation of a sequential algorithm code has been carried out using refinement laws. More details are given in Sections 4.6 and 4.8.

Debugging

Appropriate casting of the output of the witness search process is vital in order to make the outcomes from the model checker useful. This is provided by a debugger process that directly interacts with the search engine in order to retrieve human-readable information from witnesses found by the search algorithm.

The debugging process uses information, such as (minimal) acceptances sets, trace information, and so on, in order to enable the user to make an appropriate and accurate judgement about failures. In FDR, for instance, this is given with a tree-view of the automata on the failed traces together with additional information about the observed and offending failures, such as acceptances and refusals sets.

In our abstract description, each debug session contains a debug window with the minimum information necessary for creating more detailed debugging facilities for a particular snapshot of the transition system. The minimum requirements to characterise a failure are: the witness to where it came from; the trace on the automata

from the given witness; the observed acceptances from the specification at the point of failure; and the offending acceptances from the implementation at the point of failure. Further information, such as acceptances on various points chosen by the user, is also possible, but not yet formalised. Moreover, our prototype is not yet concerned with visual presentations of the debugging facilities and simple textual output is given instead. The results are given as appropriate data structures, similarly to the approach taken by FDR's debugger object model, which have been explained in detail in [Fre03].

Theorem proving

Theorem proving is important in order to discharge generated proof obligations during compilation, normalisation, divergence checking, witness search, and debugging (see Figure 4.2). Our concern is to provide an interface between the results generated by the refinement checker, and any necessary theorem prover as an external tool.

We have an interface between our model checker tool and available theorem provers such as ***Z/Eves***, which are suitable to discharge generated proof obligations and verification conditions. It is based on a pluggable architecture divided in three layers concerned with: (i) the available set operations; (ii) expression and predicate evaluation; and (iii) an oracle responsible for the final answer if necessary (see Section 5.3.3). Other theorem provers, such as Isabelle [NPW03] and PVS [Sha], can also be used, but such considerations are left as future research.

4.4 Automata theory

In this section we present a minimum subset of the theory of automata necessary for the description of the components of our model checker; it is an extension of traditional automata theory [HMU01] tailored for refinement model checking [CH93b, RGG+95]. The whole theory of automata, with additional ***Z/Eves*** rules and theorems about properties of the data type and of the transformations, is presented in [Fre04b]. The data structures given here are a sketched version of the complete theory that is also in accordance with the operational semantics given in Chapter 3 and implementation given in Chapter 5.

4.4.1 Generic transition system

As in the operational semantics (see Section 3.2), the possible events for communication are defined by Σ, here abstracted in its structure to a given set.

$$[\Sigma]$$

As before, an arc is represented as a set of possible events from Σ.

$$Arc == \mathbb{P}\,\Sigma$$

The transition system (TS) is defined below: it is a generic definition with respect to the types of nodes. Having arcs as sets is important for including semantic properties of state-rich processes in an automaton, such as loosely defined components, as well as to characterise infinite data types in a finite data structure, both via symbolic reasoning. The generic parameter N allows us to reuse this definition for both nondeterministic and deterministic nodes defined later on, as well as other unexplored

instantiations. For instance, FDR's transition system can be represented by this data type, provided we impose the restriction that arcs must have cardinality 1.

$$TS[N] == N \times Arc \leftrightarrow N$$

The transition system establishes the relationship between nodes connected through arcs acting as firing conditions: from a source node n_s and an enabling arc a, we have an available path (n_s, a) leading to possible target nodes n_t, written as

$$n_s \xrightarrow{a} n_t$$

In the theory presented in [Fre04b], we have instantiated N for nondeterministic and deterministic nodes. Moreover, we are interested in transition systems that are finite (FTS).

$$FTS[N] == \mathbb{F}((N \times Arc) \times N)$$

Nevertheless, as $(\mathbb{P}\ \_)$ is more suitable for ***Z/Eves*** automation than $(\mathbb{F}\ \_)$, we delay the inclusion of the finiteness restriction as much as possible.

Next, we present some projection functions over $TS[N]$. The functions *nodes* and *arcs* return all the nodes and arcs from the transition system.

$$\begin{array}{|l} [N] \\ \hline nodes : TS[N] \rightarrow \mathbb{P}\ N \\ arcs : TS[N] \rightarrow \mathbb{P}\ Arc \\ \hline \forall\, ts : TS[N] \bullet nodes\ ts = \mathrm{dom}\,(\mathrm{dom}\ ts) \cup \mathrm{ran}\ ts \\ \forall\, ts : TS[N] \bullet arcs\ ts = \mathrm{ran}\,(\mathrm{dom}\ ts) \\ \hline \end{array}$$

For the model checking algorithms, we define abstract versions of the main semantic functions of our theory of automata that are compatible with the operational semantics. The *enabled* function returns the set of arcs of a transition system from a particular node N. The *arcStep* function returns the set of nodes of a transition system reached through a given node and arc restricted via relational image.

$$\begin{array}{|l} [N] \\ \hline enabled : TS[N] \times N \rightarrow \mathbb{P}\ Arc \\ arcStep : TS[N] \times N \times Arc \rightarrow \mathbb{F}\ N \\ \hline \forall\, ts : TS[N];\ n : N \bullet enabled\,(ts, n) = \{\, a : Arc \mid (n, a) \in \mathrm{dom}\ ts \,\} \\ \forall\, ts : TS[N];\ n : N;\ a : Arc \bullet arcStep\,(ts, n, a) = ts\,(\!|\, \{\, (n, a) \,\} \,|\!) \\ \hline \end{array}$$

A predicate transition system (PTS) is an automaton composed by a generic transition system, an initial node, and the set of nodes to be marked as divergent. Divergent nodes must be nodes of the transition system, and whenever the transition system is not empty, the initial node must have at least one outgoing transition enabled, as our graph is always connected. In other words, if the transition system

is not empty, from the definition of *enabled*, the initial node (*init*) must be a source node in the transition system *ts*.

$$
\begin{array}{|l}
\hline
PTS[N] \\
init : N \\
ts : TS[N] \\
div : \mathbb{P}\ N \\
\hline
div \subseteq (\{\ init\ \} \cup nodes\ ts) \\
\neg ts = \{\ \} \Rightarrow (\ \exists\ a : Arc \bullet a \in enabled\ (ts, init)\ ) \\
\hline
\end{array}
$$

The nodes representing distinct states are defined in Chapter 3. Again we abstract its structure by defining it as a given set.

$$[Node]$$

A nondeterministic predicate transition system (*IPTS*) is defined next as an finite *PTS* on *Node*.

$$IPTS \ == \ \{\ pts : PTS[Node] \mid pts.ts \in FTS\ \}$$

We calculate a deterministic transition system (*DTS*) by the transformation of a nondeterministic automaton via a special kind of subset construction, where the node is a non-empty power set of original nodes.

$$DNode \ == \ \mathbb{P}_1\ Node$$

A deterministic node represents the possible set of nodes that a nondeterministic choice through an event could lead up to. That is, from the same source node, one can reach more than one target node through the same event. A deterministic node must not be empty because it is formed by at least one allowed target node from the original nondeterministic transition system. Since the normal form considers only visible communication, silent transitions, which are represented by the empty arc, are also not allowed. Thus, *DArc*, the type of the arcs of a deterministic transition system (*DTS*), is the non-empty power set of Σ.

$$DArc \ == \ \mathbb{P}_1\ \Sigma$$

A deterministic transition system (*DTS*) is defined as a partial function from an arc to another target node of type *DNode*. In spite of the fact that our theory of automata follows Hopcroft's style, his version of a *DTS* is given as a total function, where invalid arcs lead to an empty node. Initially, our design took the same approach; however, due to arcs being sets of events, this totalisation is not suitable. Although we tried to avoid partial functions in ***Z/Eves*** due to the increased complexity in related proofs, this time it was not possible. The *DTS* function needs to be partial for two reasons: (i) deadlock is represented as nodes without outgoing arcs; and (ii) each pair of arcs $a, b \in DArc$ on a *DTS* must be disjoint. Therefore, it is not possible to take into account all possible nodes and arcs. A more restricted model is necessary, and is given by the next abbreviation.

$$
\begin{aligned}
DTS \ == \ \{\ & f : DNode \times DArc \pfun DNode \mid \forall\ p, q : DNode \times DArc \mid \\
& \neg\ p = q \land p \in \mathrm{dom}\ f \land q \in \mathrm{dom}\ f \land p.1 = q.1 \bullet p.2 \cap q.2 = \{\ \}\ \}
\end{aligned}
$$

For all partial functions f, wherever any two different elements p and q on its domain have a node in common, then the possible outgoing arcs on f from that node must

be disjoint. This preserves the determinacy with respect to a single event going out from the same node on f. This is an interesting example of the aid of using a theorem prover whilst developing our theory. This fact passed unnoticed for quite a while until proof obligations related to consistency of automata transformations pointed out the problem, which lead to the revised correct version.

4.5 Normalisation process specification

As already mentioned, normalisation is a procedure to transform an automaton to make it deterministic and free from internal transitions. It is known in automata theory as subset construction [HMU01, Chapter 2]. In order to avoid semantic loss during the normalisation, additional information regarding nondeterminism and divergence is also recorded.

4.5.1 Normal form automaton

The normal form is a *PTS* instantiated to *DNode* with a deterministic and finite transition system, where arcs must not represent silent transitions. Since the normal form is created by subset construction from a valid nondeterministic automaton, every deterministic node is not to be empty, and it contains original nodes from the nondeterministic automaton being transformed.

While transforming the automaton, we need to record the information about the explicit nondeterminism on the nodes being conjoined, in order to avoid semantic loss on possible arcs originally enabled for every nondeterministic arc being removed. Since the deterministic automata is on the power space of the original, the information recorded is the set of enabled arcs that leaves each original node $n \in Node$ contained in the new deterministic node $dn \in DNode$, where $n \in dn$. This set defines what the original nodes must accept eventually, and its called the node acceptances set; it is defined by the next abbreviation.

$$Acceptances == \mathbb{P}\ DArc$$

We use acceptances sets in order to reuse available definitions for immediately enabled communication, as defined by the operational semantics, hence simplifying the description of related abstract data types. The refusals of a node can be calculated as the complement of the acceptances set with respect to the universe of available arcs drawn from Σ. Furthermore, (minimal) acceptances are preferred instead of (maximal) refusals mainly for efficiency purposes, as suggested by empirical experiences from [RSR$^+$01, Chapter 4] and [Ros94b].

For refinement check purposes, we want to record the minimal set of possible acceptances in the normal form, as it allows a more economical representation without information loss. So, let us define a function used to calculate minimal sets. It defines the set of incomparable sets under proper subset ordering.

$$[X]$$

$$minimal : \mathbb{P}(\mathbb{P}\ X) \rightarrow \mathbb{P}(\mathbb{P}\ X)$$

$$\forall SS : \mathbb{P}(\mathbb{P}\ X) \bullet minimal\ SS = \{ S : \mathbb{P}\ X \mid S \in SS \land \neg (\exists S' : \mathbb{P}\ X \mid S' \in SS \bullet S' \subset S) \}$$

We also need another function used to calculate divergence information about a normal node by chasing silent loops. It is defined as a transitive closure of the transition

system restricted to silent arcs only; this is the standard way of characterising loops in a graph [HMU01, Chapter 2]. The function that calculates this closure on silent transitions receives a transition system, and the node to check for silent loops. If the node is present on the transitive closure result, then it is divergent.

$$
\begin{array}{|l}
silentClosure : TS[Node] \times Node \to \mathbb{P}\, Node \\
\hline
\forall\, nts : TS[Node];\ n : Node \bullet \\
\quad (\exists\, R : Node \leftrightarrow Node \mid R = \{\, n' : Node \mid (n, \{\,\}) \mapsto n' \in nts \bullet n \mapsto n' \,\} \bullet \\
\qquad silentClosure\,(nts, n) = (R^{+})(\!|\, \{\, n \,\} \,|\!)\ )
\end{array}
$$

The closure operation is applied on a restricted version of the transition system containing silent arcs only, as suggested in [Ros94b, pp.362]. This restriction is recorded by the homogeneous relation R. Over this restricted relation, we can now apply the transitive closure (postfix) operator. As we are interested in divergence information for n only, it is retrieved through relational image on the transitively closed relation R. The presence of the existentially quantified variable R forbids us to declare the definition as a rewriting rule in ***Z/Eves***. The variable is needed because the prover does not allow the application of any rule for a set parameter declared on-the-fly [Saa99b, Chapter 4]. Next, we describe an infix relation including all divergent nodes from a nondeterministic transition system, provided the node forms a silent loop.
syntax *isDivIn inrel*

$$
\begin{array}{|l}
\_\, isDivIn\, \_ : Node \leftrightarrow TS[Node] \\
\hline
\forall\, nts : TS[Node];\ n : Node \bullet n\ isDivIn\ nts \Leftrightarrow n \in silentClosure\,(nts, n)
\end{array}
$$

This relation is also generalised for deterministic nodes of the normal form.
syntax *isDIVIn inrel*

$$
\begin{array}{|l}
\_\, isDIVIn\, \_ : DNode \leftrightarrow TS[Node] \\
\hline
\forall\, nts : TS[Node];\ dn : DNode \bullet \\
\quad dn\ isDIVIn\ nts \Leftrightarrow (\exists\, n : Node \mid n \in dn \bullet n\ isDivIn\ nts)
\end{array}
$$

A node of a normalised automaton is divergent if any of its nodes is divergent. The data type of the normal form is defined by *NFPTS* as a deterministic *PTS* (*DPTS*) with finite transition system (*FTS*). In order to avoid semantic loss during normalisation, additional information regarding nondeterminism and divergence is recorded.

$$
\begin{array}{|l}
\textit{DPTS} \\
PTS[DNode] \\
accs : DNode \nrightarrow Acceptances \\
\hline
ts \in DTS \\
ts = \{\} \Rightarrow accs = \{\} \\
\neg ts = \{\} \Rightarrow \mathrm{dom}\ accs = (\{init\} \cup nodes\ ts) \\
\mathrm{dom}\ accs = (\{\, init \,\} \cup nodes\ ts) \\
\bigcup(\mathrm{ran}\ accs) \subseteq arcs\ ts \\
div \subseteq (\{\, init \,\} \cup nodes\ ts) \\
\forall\, dn : DNode \mid dn \in div \wedge dn \in \mathrm{dom}\ accs \bullet accs\ dn = \{\,\} \\
\forall\, dn : DNode \mid dn \in \mathrm{dom}\ accs \bullet \\
\quad \forall\, e : \Sigma \mid e \in \bigcup(accs\ dn) \bullet e \in \bigcup(enabled\,(ts, dn))
\end{array}
$$

A *PTS* instantiated to *DNode* forms the basis of the normal form, where the transition

system is both deterministic (DTS) and finite (FTS).

$$NFPTS == \{ pts : DPTS \mid pts.ts \in FTS \}$$

The *accs* function records the set of acceptances sets for every deterministic node dn of the transition system ts. It guarantees that information about nondeterminism is not lost, since for every original node $n \in dn$, such that $dn \in \mathrm{dom}\ accs$, there exists a set containing the acceptances set of each n individually calculated by the *enabled* function. The function domain guarantees that it contains information about every node in ts, in order to avoid any semantic loss during normalisation. The range also ensures that events in acceptances sets are consistent with respect to available arcs. The set *div* contains the deterministic nodes marked as divergent. It records divergence information from the original automaton, since silent transitions have been eliminated while making the transition system deterministic via subset construction briefly explained below. While normalising the specification, since divergence is considered catastrophic, once it has been found, any further original behaviour about acceptances can be ignored. This is another interesting feature that justifies normalisation: it can significantly shrink the size of the normal form when divergence is detected, as further information need be neither calculated nor stored. In this catastrophic treatment, for a divergent node dn, $accs\ dn$ is the empty set ($\varnothing$). Deadlocked nodes, however, have an empty set element ($\{ \varnothing \}$) as the result of the application of *accs* to dn.

4.5.2 Declared channels

In *Circus* we have only the failures-divergences model. Nevertheless, we define possible violation conditions layered by different criteria, as defined by the next free-type *Criterion*. This modularisation is useful for the simplification of proof obligations during the witness search algorithm derivation (see Section 4.8), as well as to allow room for extension. Once other criteria become available, they can be included by extending the this free-type. More information about compatibility criteria is presented in Section 4.6.1.

$$Criterion ::= tr \mid sfl \mid fldv$$

Next, we declare three channels.

$$\textbf{channel}\ select\_criterion : Criterion$$
$$\textbf{channel}\ spec\_in : IPTS$$
$$\textbf{channel}\ spec\_out : NFPTS$$

They receive the selected granularity of the criterion for refinement, the original automaton input as a PTS of *Node* with a finite transition system ($IPTS$), and the normal form output as an $NFPTS$, respectively (see Figure 4.2).

4.5.3 Normal form process state

The normalisation process is sequential.

$$\textbf{process}\ Normalisation\ \widehat{=}\ \textbf{begin}$$

The process state contains the selected model, the original automaton as an $IPTS$, and the normal form produced as a $NFPTS$. The invariant guarantees that divergence information is relevant only for the failures-divergences criterion, and also that every

divergent node must be related to a silent loop in the original automaton, as defined by the $(\_\ isDIVIn\ \_)$ relation.

state

$$
\begin{array}{|l}
\textit{NFState} \\
\hline
c : Criterion;\ oa : IPTS;\ nf : NFPTS \\
\hline
c \neq fldv \Rightarrow nf.div = \{\} \\
\forall\, dn : DNode \mid dn \in nf.div \bullet dn\ isDIVIn\ oa.ts \\
\forall\, dn : DNode \mid dn \in \mathrm{dom}\ nf.accs \bullet nf.accs\ dn = \\
\quad (\mathbf{if}_Z\ (dn \in nf.div)\ \mathbf{then}\ \{\}\ \ \mathbf{else} \\
\quad\ minimal\,(\{\, n : Node \mid n \in dn \bullet \bigcup (enabled\,(oa.ts, n) \setminus \{\})\,\})\,) \\
\hline
\end{array}
$$

Information about acceptances needs to be calculated for all deterministic node in the normal form; they are either empty if $(nf.ts = \varnothing)$, or those from $nf.ts$ together with the initial normal node $(nf.init)$ otherwise. Divergent nodes $(dn \in nf.div)$ have an empty acceptances set $(nf.accs\ dn = \{\})$. Non-divergent nodes have acceptances contained in the minimal set of enabled arcs without silent transitions in the original automaton, regardless of the arc they come from, as these arcs are flattened via generalised set union.

State initialisation

The components of the state are initialised with the schema *InitNFState*. The selected criterion and original automaton are given via the input variables *spec*? and *mdl*?. Since no calculations have yet been made, the normal form is left empty, as given by the ***Z/Eves*** $\theta DPTS$ expression with appropriate renaming (see [Fre04c] for details on this ***Z/Eves*** idiom).

$$
\begin{array}{|l}
\textit{InitNFState} \\
\hline
NFState';\ spec? : IPTS;\ mdl? : Criterion \\
\hline
m' = mdl? \land oa' = spec? \\
nf' = \theta\, DPTS[ts := \{\}, init := \{\, spec?.init\,\}, accs := \{\}, div := \{\}] \\
\hline
\end{array}
$$

An empty normal form nf' is relevant for the proof of the initialisation theorem for the normal form process given below.

theorem tInitNFStatePRE
$\forall\, mdl? : Criterion;\ spec? : IPTS \bullet \exists\ NFState' \bullet InitNFState$

The next action initialises the state after receiving the original specification automaton and selected criterion through channels *spec_in* and *select_criterion*, respectively.

$$
Initialise \mathrel{\widehat=} spec\_in?spec \rightarrow select\_criterion?mdl : (mdl = fldv) \rightarrow InitNFState
$$

As *Circus* only has the failures-divergences model at the moment, we restrict the possible criteria appropriately in the communication. Whenever other models are available, this restriction can be relaxed.

Normalising

After initialisation, only the normal form can be modified and all other state compo-

nents remain the same, as defined by schema *NFOps*.

$$NFOps \mathrel{\widehat=} [\Delta NFState \mid c' = c \land oa' = oa]$$

The normalisation operation calculates the automaton in normal form and is defined by the schema *Normalise* below. The initial node of the normal form is retrieved from the transitive closure over silent transitions from the original automaton. The transition system is built according to the *silentSubset* function to calculate the deterministic transition system from the original automaton via a specialised form of subset construction that takes the set of events from arcs into account.

$$silentSubset : TS[Node] \to DTS$$
$$silentClose : TS[Node] \times Node \to DNode$$

The definitions of *silentClose* and *silentSubset* are a variation of the standard definition given by Hopcroft in [HMU01, Chapter 2], and are rather long. Its peculiarity lies in making arcs deterministic and is presented in the explanation of the *regions* function; uninteresting details are omitted here for simplicity and are fully present in [Fre04b, Axiomatic Definition 3.6]. As we calculate the transition system on-the-fly, we do not have the entire transition system, and the definition is given pointwise for each node of the original automaton. In this way, subset construction for *DTS* resembles grammar productions, as an automaton to represent the language of the built *DTS* is not completely known up front; instead, we know the appropriate rules on how to produce the automaton.

In words, for each node of the deterministic transition system, the target node is defined with respect to an appropriate step of original nondeterministic versions via enabled arcs. This is achieved by the description of the domain of the deterministic transition system being built. In our case, however, the determinacy of arcs is rather unusual and needs to be addressed properly. This is the subtle issue which differs from the standard definition, and we include its details below.

In order to guarantee the invariant property of a *DTS*, we need to build disjoint arcs appropriately. The corresponding deterministic transition system in the normal form is built by calculating the possible individual regions of events related to the combination of enabled arcs, such that the invariant of *DTS* holds. The *regions* function calculates the disjoint sets of a nonempty set of sets. It is generically defined as the generalised disjointness of all possible subsets of the available arcs obeying subset ordering. We call generalised disjointness the difference between the generalised intersection and the generalised union of corresponding sets.

$$[X]$$
$$regions : \mathbb{P}(\mathbb{P}\ X) \to \mathbb{P}(\mathbb{P}\ X)$$
$$\forall S : \mathbb{P}(\mathbb{P}\ X) \bullet regions\ S = \{ P : \mathbb{P}_1(\mathbb{P}\ X) \mid P \subseteq S \bullet (\bigcap P) \setminus (\bigcup (S \setminus P)) \} \setminus \{\{\}\}$$

Taking the generalised disjointness under subset ordering gives rise to the set of all possible distinct regions of a non-empty set of sets without elements in common, for our particular case, enabled arcs. For example, the regions of a set S containing three enabled arcs A, B, and C in Σ as shown in Figure 4.3. The *regions* function obeys three interesting properties. Firstly, the maximum number of regions is in the power

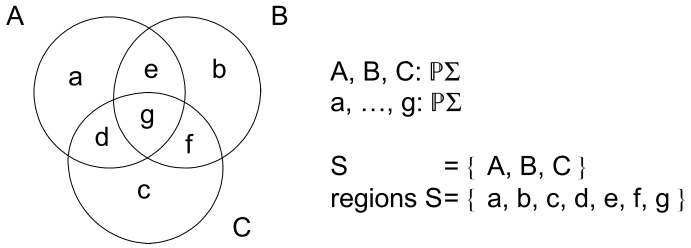

Figure 4.3: Regions of a nonempty set of sets

space of the original set because of the power set constructor, given as

$$\#(regions\ S) = (2^{\#S} - 1)$$

Secondly, the regions of S are disjoint.

$$a0 \in regions\ S \land a1 \in regions\ S \land a0 \neq a1 \Rightarrow a0 \cap a1 = \{\}$$

This means that *regions* S is divided according to the amount of event sharing happening between the original sets. In this way, event sharing is arranged such that they form a *Pascal Triangle*. For instance, the events in $((A \cap B) \setminus C)$ are preserved in e, and the same happens for all other intersections. One nice consequence of this is that it becomes easier to implement using bit-masking. This comes directly from the use of power sets under subset ordering. These properties are particularly useful for modelling parallelism between processes with different amounts of event sharing. For example, the set of arcs related to interleaving could be given as

$$interleaving(S) \ \equiv \ regions\ S$$

since all other regions would be empty and could be eliminated. Because *regions* S builds a *Pascal Triangle*, it respects the level of event sharing among each set. Thus, nondeterministically selected parallel processes can become deterministic while performing normalisation. This turns out to be very useful in the normal form when dealing with parallelism. Nevertheless, for automata with complex expressions on their arcs, this might lead to the evaluation of expressions related to regions, hence theorem proving while building the normal form. Finally, the third property guarantees that *regions* S preserves the original elements of S as they have the same elements, if the boundaries created to make arcs deterministic were flattened ($\bigcup S = \bigcup (regions\ S)$). At last, we present schema *Normalise*, which calculates the normal form automaton.

Normalise

NFOps

$$nf'.init = silentClose\,(oa.ts, oa.init)$$
$$nf'.ts = silentSubset\,(oa.ts)$$
$$nf'.accs = \{\, dn : DNode \mid dn \in (\{\, nf'.init \,\} \cup nodes\,(nf'.ts)) \bullet dn \mapsto$$
$$\quad (\textbf{if}_Z\ (dn \in nf'.div)\ \textbf{then}\ \{\}\ \ \textbf{else}$$
$$\qquad minimal\,(\{\, n : Node \mid n \in dn \bullet \bigcup (enabled\,(oa.ts, n) \setminus \{\})\})) \,\}$$
$$m \neq fldv \Rightarrow nf'.div = \{\}$$
$$m = fldv \Rightarrow nf'.div = \{dn : DNode \mid$$
$$\quad dn \in (\{\, nf'.init \,\} \cup nodes\,(nf'.ts)) \land dn\ isDIVIn\ oa.ts \,\}$$

The specialised subset construction functions are used to build the normal form au-

tomaton from the original transition system. Like in the state invariant, divergent nodes have an empty acceptances set, whereas non-divergent nodes have minimal acceptances according to immediate available events without outgoing transitions.

The action *GiveResult* outputs the produced normal form through the *spec_out* channel and terminates.

$$GiveResult \mathrel{\widehat=} spec\_out!nf \rightarrow Skip$$

The main action uses recursion and sequential composition to initialise the state, perform normalisation, output the resultant normal form.

$$\bullet\ (\mu\, X \bullet Initialise \,;\, Normalise \,;\, GiveResult \,;\, X)$$

end

Apart from building the *regions* of sets of sets to make the original transition system deterministic, the normalisation process is standard [HMU01] (*i.e.*, subset construction on nondeterministic automata), and further details are omitted here for simplicity.

4.6 Witness search specification

Witness search is responsible for finding whether all the behaviours of I are allowable by at least one behaviour of S, such that they have a trace in common. The behaviours of interest of the search process depend on the selected criterion to establish refinement, which in turn has a specific violation condition.

Let (s, i) be a node pair representing a configuration from automata S and I, where we want to establish the refinement $S \sqsubseteq I$. In this context, a common trace is a sequence of visible arcs used to bring the cartesian product of pairs (s, i) to possibly reach a pair which has incompatible behaviours. An incompatible behaviour is characterised by the violation of at least one of the refinement criterion on the selected refinement model.

At first, the search is seeded with a pair of initial nodes from each automaton. The strategy performs an exhaustive search between every mutually reachable pairs (s, i) in order to check compatibility between each pair. A pair is mutually reachable in the search if, and only if, they follow from the selection of a compatible transition on both automata with respect to the chosen arc. If an incompatible pair is found, then the proposed refinement does not hold and debugging information is available. Otherwise, if the process completes the exhaustive search without finding such a pair, then the proposed refinement does hold.

4.6.1 Compatibility checks—granularity of refinement criteria

Although *Circus* has only one model for refinement checking, the failures-divergences model, we can factor its compatibility criteria into smaller and compositional definitions. This not only simplifies the proof obligations related to the algorithm derivation, but also allows a modular explanation (and implementation) of each aspect of compatibility checking that is open for extension. For each factored case, we carry different information over the state space characterising an important aspect of reactive systems that we want to understand and analyse. We factor the failures-divergences model as: (i) traces criterion related to language inclusion between automata, which represents safety properties; (ii) stable-failures criterion related to acceptances sets, which represent nondeterminism; and (iii) failures-divergences criterion related to

characterisation of divergent behaviour. In any case, the refinement claim is that the implementation is more detailed, and less nondeterministic than the specification, but still fulfilling the same set of conditions. In what follows, we present these criteria with schemas representing their corresponding violation condition predicate.

Node pair invariant

Regardless of the compatibility criteria to be tested, the node pair under consideration must be valid. A node pair (sN, iN) is valid if, and only if, $sN \in DNode$ belongs to the nodes of the normal form automaton $nf \in NFPTS$, and $iN \in Node$ belongs to the nodes of the implementation automaton $ip \in IPTS$ as defined by the next schema.

$$
\begin{array}{|l}
\hline
NodePairInv \\
nf : NFPTS;\ ip : IPTS;\ sN : DNode;\ iN : Node \\
\hline
sN \in (\{\, nf.init \,\} \cup nodes\,(nf.ts)) \land iN \in (\{\, ip.init \,\} \cup nodes\,(ip.ts)) \\
\hline
\end{array}
$$

This condition, defined for the general theory of automata, is used to prove properties in ***Z/Eves*** only. In practice, the operational semantics generates only valid nodes.

Traces criterion

The set of traces of a specification S is the set of all finite sequences of events representing visible communications. It defines the language the automaton of S recognises. The traces criterion represents what S can do, without requiring it to do anything. It restricts the behaviour of S instead of requiring S to do something useful.

The traces criterion is formally defined as, for all traces $ti \in \text{seq}\ \Sigma$ of the implementation, there exists at least one trace $tn \in \text{seq}\ \Sigma$ allowed by the normal form, such that ti prefix tn. In other words, the language of the implementation must be contained within the language of the normal form. Thus, the violation condition on traces is given by the calculation of the language of both automata from the node pair being checked. This is guaranteed by ensuring that immediately available events from the implementation node iN are contained in the immediately available events of the normal form node sN, for all valid node pairs (sN, iN). This violation of the traces criterion is defined by the schema $TrVl$.

$$TrVl \mathrel{\widehat=} [\, NodePairInv \mid \neg \bigcup (enabled\,(ip.ts, iN)) \subseteq \bigcup (enabled\,(nf.ts, sN)) \,]$$

The immediately available events from each node come from flattening all enabled arcs available from each node. This means that silent (or internal) communication on the implementation does not violate refinement with respect to traces.

Stable-failures criterion

This criterion specifies what might be accepted, and it is related to nondeterministic properties. The criterion is called stable because divergence is not relevant and its occurrence is treated optimistically. That is, we assume divergence does not occur, hence ignore its possible presence in the automata.

The stable-failure criterion of a specification S is a pair represented by a trace tr and a minimum acceptances set $accs$. The acceptances sets are all the arcs containing events from individual arcs that the automaton of S accepts after the trace recorded in tr. The stable-failures of S is the set of all failure pairs $\mathcal{F}$ formed by $(tr, accs)$. It does not just limit what S can do, but also defines what S accepts after tr.

This criterion specifies what can happen, hence it can capture nondeterminism on internal choices. An internal choice can only reduce the range of possible behaviours observable from S by eliminating other behaviours that would have remained if some controllable form of choice was provided. Therefore, the effect of a nondeterministic choice made by S is to constrain the ability of S to perform these other eliminated arcs. In this way, this criterion puts a limit on what S can fail to do, so it requires S to accept at least a certain range of behaviours on the same trace. The violation condition on the stable-failures model is given by the schema $SFlVl$ for valid nodes.

$$
\begin{array}{|l}
\hline
SFlVl \\
NodePairInv \\
\hline
\neg\,(\,\varnothing \notin enabled\,(ip.ts, iN)\,) \Rightarrow \\
\quad (\exists\, as : \mathbb{P}_1\, \Sigma \mid as \in nf.accs\, sN \bullet as \subseteq \bigcup (enabled\,(ip.ts, iN)))\,) \\
\hline
\end{array}
$$

On the one hand, since divergence is not relevant, its occurrence is treated optimistically. As the normal form does not mention internal communication, silent transitions are assumed to come from nodes of the implementation automaton. On the other hand, for a implementation node that does not have silent communication immediately available (*i.e.*, a stable node), there must exist a minimum acceptance as in the normal form from sN, such that language inclusion still holds with respect to iN. The immediate communications of the implementation must contain at least one of the acceptances sets available at the specification.

Failures-divergences criterion

A divergence is a situation where every subset of the alphabet can be refused, and every available trace is a possible behaviour, an anomaly to be avoided. Nevertheless, the stable-failures criterion treats divergence optimistically by assuming it does not happen, and no information about divergences is included. Therefore, a richer criterion to differentiate unpredictable from stable behaviour is necessary.

This criterion adds divergence information for refinement checking. Thus, a divergence is a pair $(\mathcal{F}, \mathcal{D})$, where the first element is a stable-failure with acceptances sets, and the second element is divergence that has been detected after a trace containing a loop of internal transitions. The violation condition related to divergences is given by the schema $DvVl$.

$$DvVl \;\widehat{=}\; [\,NodePairInv \mid \neg\,(\,sN \in nf.div \vee \neg\, iN \in ip.div\,)\,]$$

As in the failures-divergences criterion divergence is treated pessimistically, any state after divergence is considered unstable (or catastrophic) and its behaviours are ignored. Therefore, if a specification node sN is marked as divergent in the normal form, then the refinement already holds. If the normal form allows catastrophic behaviour to happen, then any behaviour from the implementation is valid. Otherwise, if sN is not divergent, then the implementation node iN is not allowed to behave catastrophically and must not be divergent as well.

Complete refinement criteria

The next schema gives the general violation condition as the combination of previous

violation definitions.

$$
\begin{array}{|l}
\textit{GenVl} \\
\hline
\textit{NodePairInv} \\
\hline
\textit{TrVl} \lor (\neg\, m = tr \land (\textit{SFlVl} \lor (m = \textit{fldv} \land \textit{DvVl})))
\end{array}
$$

Regardless of the criterion, every pair must be valid and checked for traces violation. For the traces criterion, this is enough. Other different criteria must satisfy the stable-failures check. Finally, the divergence violations are checked only for the failures-divergences criterion.

4.6.2 Witness definition

In what follows, we define the data type for witnesses. It is the key for finding successors of a compatible node pair correctly during the witness search process.

Data type

The new data type necessary for representing witnesses of a refinement failure comes next. They record the joint trace on both automata until a node pair that violates the general refinement check criteria ($GenVl$) has been found. Moreover, since the normal form is deterministic, there exists only one trace through which the sequence of node pairs can be reached in both automata. This gives rise to an interesting efficiency advantage from having a normal form: there is no need to record for all pairs being checked the trace that led to them.

In what follows, we present alternatives we explored together with some discussion about their unsuitability, and the actual choice for the witness data type. Like in the definition of DTS, this is another interesting exercise that motivates (and once more justifies) the use of theorem proving. Initially, witnesses could have been defined as a sequence of node pairs where the first element comes from the normal form, and the second from the implementation.

$$\textit{Witness} \;==\; \mathrm{seq}(\textit{DNode} \times \textit{Node})$$

Unfortunately, this simple definition is not enough for a thorough and efficient witness search with debugging capabilities, and a more complex data type is needed. This occurs because, as we do not include the whole transition system on the witness but just nodes, we need to add extra information to enable us to rebuild the transition system backwards from the failing pair back to the root of the search.

Firstly, let us define a node pair with the next abbreviation. It contains a deterministic node from the normal form, and a node from the implementation.

$$\textit{NodePair} \;==\; \textit{DNode} \times \textit{Node}$$

Since all joint arcs of node pairs are unique due to the deterministic property of the normal form, the sequences that define witnesses are injective. Therefore, one could suggest the following abbreviation to define as the witnesses data type.

$$\textit{Witness} \;==\; \mathrm{iseq}\, \textit{NodePair}$$

Nevertheless, this is still not suitable for the debugger to provide the environment with useful (readable) information. For that, the debugger needs to traverse the transition

system of both automata backwards to generate the entire witness from the root down to the point of failure.

At this stage, a strict and tight relation between the two phases of witness search and debugging arises. The former must provide additional information on the witnesses to allow the latter to perform its job properly. One possible suggestion would be to have the parent nodes of each node pair stored in the sequence. Unfortunately, due to the state explosion problem, we cannot afford the amount of space necessary for this approach, as it would double the memory necessary to store a single witness.

The approach taken to provide an efficient data-type tailored for the search and debugger processes within our "memory budget" is well-known, simple, cheap, and efficient. It is well-known because the witness search algorithm is a variation of breath-first-search (BFS). It is simple because BFS grants us the property that the witness found is minimal [Gol01, Ros94b]. BFS is characterised by rounds that search a layer of the data structure. So, the approach is cheap because it needs the space of a single natural number per node pair as additional information for the layer index. It is also efficient because it is well-known that BFS can be implemented in parallel [MH00, DPP00, OV96]. One can think of the layers of the search being descending levels on the automata. Again, due to the uniqueness property provided by the determinism of the normal form, pairs on higher levels must be related to a single pair on the level above. Therefore, the layer number will allow the debugger to regenerate the transition system backwards from the failed pair up to the root node with the addition of a natural number per node pair. Consequently, the witness data type could be

$$Witness \;==\; \mathrm{iseq}\,(NodePair \times \mathbb{N})$$

Nevertheless, a tricky problem pointed out by ***Z/Eves*** still remains: the injectivity of the sequence must be only on the first element of the pair, since distinct pairs can be at the same layer of the search.

In order to solve this problem elegantly, one can opt for a top-down approach using sequential composition and an auxiliary projection function as a particular definition of injectivity. Alternatively, a bottom-up approach more suitable for mechanical analysis in ***Z/Eves*** uses a pair of sequences.

$$Witness \;==\; \mathrm{iseq}\; NodePair \times \mathrm{seq}\; \mathbb{N}$$

The first sequence in the pair is an injective sequence of $NodePair$. The second sequence is of natural numbers representing the layers of the search being checked. This solution of selective injectivity works provided that both sequences have the same length: the unique node pair at index i in the first injective sequence is linked with the layer of the search in the second sequence at the same index. This idea of a joint path is given below

$$JointPath \;==\; \{\, SNP : \mathrm{iseq}\; NodePair;\; SCL : \mathrm{seq}\; \mathbb{N} \mid \#\, SCL = \#\, SNP \,\}$$

The first element is a sequence of node pairs (SNP), whereas the second is the sequence of checking layers (SCL) that must have the same size. Because a witness is the result of a search that found at least one incompatible node pair, it cannot be empty.

$$Witness \;==\; JointPath \setminus \{\, (\langle\rangle, \langle\rangle) \,\}$$

This abbreviation defines the data type used for a witness of a refinement failure.

Joint trace retrieval

We retrieve the joint trace from a witness through the *wtsTrace* function below. It is defined in terms of trace retrieval functions for both the normal form and the implementation automata. A sequence s is a trace from a witness w, provided that it is a trace in the normal form and in the implementation with respect to the node pair of failure recorded as the last node of a witness.

$$
\begin{array}{l}
NFWtsTr : NFPTS \times DNode \rightarrow \text{seq}\ \Sigma \\
IPWtsTr : IPTS \times Node \rightarrow \text{seq}\ \Sigma \\
wtsTrace : NFPTS \times IPTS \times Witness \rightarrow \text{seq}\ \Sigma \\
\hline
\forall\, nf : NFPTS;\ ip : IPTS;\ w : Witness;\ s : \text{seq}\ \Sigma \bullet \\
\quad (wtsTrace\,(nf, ip, w) = s) \Leftrightarrow \\
\quad\quad ((IPWtsTr(ip, (last\,(w.1)).2) = s) \land \\
\quad\quad\ (NFWtsTr(nf, (last\,(w.1)).1) = s))
\end{array}
$$

The functions on each automaton are defined inductively on the length of the trace and are omitted here.

Invariant

A valid witness must obey four criteria: (i) the last element of the node pair sequence must be valid, but no information about their compatibility is known, since it is the current pair being checked; (ii) there must exist a trace in both automata corresponding to the node pair sequence of a witness; (iii) the level of each node pair recorded on a witness strictly increases with respect to the sequence index; and (iv) every node pair in the front of a witness must be valid and compatible, not violating the general compatibility criteria.

$$
\begin{array}{l}
\textit{WitnessInv} \\
\hline
m : Criterion;\ nf : NFPTS;\ ip : IPTS;\ w : Witness;\ NodePairInv \\
\hline
(sN, iN) = last\,(w.1) \\
\exists\, T : \text{seq}\ \Sigma \bullet T = wtsTrace\,(nf, ip, w) \\
\forall\, i : 1 \mathinner{..} (\#\, w.2 - 1) \bullet w.2\,(i) \leq w.2\,(i+1) \\
\forall\, dn : DNode;\ n : Node \mid (dn, n) \in \text{ran}\,(front\ w.1) \bullet \\
\quad NodePairInv[dn/sN, n/iN] \land \neg GenVl[dn/sN, n/iN]
\end{array}
$$

The first predicate establishes that the last node pair in the witness is the one currently being checked. It must be valid, but nothing regarding its compatibility is known yet. The second predicate enforces that the search for new node pairs on the given witness candidate w is consistent with respect to both automata by having a trace in common. It establishes nodes that are mutually reachable, while searching for new successor pairs. If one can create a valid non-empty sequence of pairs over the two automata, then it must be possible to retrieve the unique trace related to such sequence. The trace is unique because of the deterministic property of the normal form. The third predicate ensures that the level index recorded in the witness strictly increases as the check is performed. Thus, lower level nodes must appear before higher levels node. This consistency on the levels information is important for the debugger to provide accurate information while rebuilding the transition system from the failed pair up to the root of the search. The last predicate establishes that all pairs of nodes in the witness sequence, except only of the last, are valid and do not violate the general

refinement checking criteria. The last pairs of nodes may or may not be compatible. That is, while an incompatible pair has not yet been found, keep searching until one is found, or else we are finished. Furthermore, every node on the *front* of the witness sequence must be formed by valid and compatible node pairs. This ensures the meaning of a witness with respect to the mutually reachable node pairs concept: a sequence of node pairs without repetition, where all pairs are valid and compatible except the last one.

4.6.3 Witness search parameters

Next, let us define the types for parameters used during witness search. The algorithm can be used either for proving correctness via exhaustive search, or testing failures via selective search.

$$EMode ::= chk \mid tst$$

According to the restrictions made on the Z part, there are three levels of automation: decidable, restricted, and unrestricted predicate calculus. This defines the level of interaction mentioned in Section 4.2.3 as: automatic, automated, or interactive (see Table 4.1). These options can be used to fine-tune the theorem proving module.

$$ALevel ::= dec \mid res \mid full$$

The next free-type defines the possible queries available about a witness search. The environment can ask either for a simple query if the required refinement holds or not, or for a verbose query including debugging structures.

$$RefQuery ::= rqrReport \mid rqDebug$$

For a simple query, the next free-type defines if the refinement report was successful, or if a failure has been found.

$$RefRep ::= rrSuccess \mid rrFailure$$

During witness search, we allow the environment to choose the number of witnesses to search for. The next global constant loosely defines the maximum number of witness that can be requested on a search as a strictly positive number left undefined.

$$maxWts : \mathbb{N}_1$$

The refinement parameters comes next, after the definition of a boolean free-type.

$$Boolean ::= f \mid t$$

$$\begin{array}{|l} RSParams \\ \hline m : Criterion;\ em : EMode;\ al : ALevel;\ nf : NFPTS; \\ ip : IPTS;\ wr : 1 \mathinner{..} maxWts;\ sd : Boolean \\ \hline \end{array}$$

They respectively represent the refinement model (m), the execution mode (em), the automation level (al), the two sides of the refinement relation as the normal form (nf) and the implementation (ip) automata, the number of witnesses requested by the external environment (wr) in case of failure, and a boolean flag ("*search done*") used to perform the search only once for the same configuration of parameters (sd).

4.6.4 Declared channels and process state

For the witness search process, six new channels are declared. The first three are used to collect the corresponding process parameters according to schema *RSParams* above, and the last three are used to allow the environment to query for the process results. The other components for *RSParams* come from the result of the normalisation and divergence checking processes (see Figure 4.2).

$$\textbf{channel}\ mode : EMode$$
$$\textbf{channel}\ autoLevel : ALevel$$
$$\textbf{channel}\ witnesses\_no : 1 \,..\, maxWts$$
$$\textbf{channel}\ query : RefQuery$$
$$\textbf{channel}\ report : RefRep$$
$$\textbf{channel}\ testimony : \mathbb{P}\ Witness$$

The channel *query* defines the environment choice for simple or extended search information. The channel *report* returns the result from a simple query about refinement checks being successful or not. The *testimony* channel returns debugging information contained in a set of witnesses found during a search. For debug queries on successful refinement checks, this set is empty. This set represents the main output of the model checker debugger, as shown in Figures 4.1 and 4.2.

The abstract version of the *Circus* specification for the witness search comes next. This process plays a central role on the model checking task: it establishes whether the refinement relation ($S \sqsubseteq I$), between a normal form S and an implementation I, holds or not.

$$\textbf{process}\ WitnessSearch\ \widehat{=}\ \textbf{begin}$$

The process state contains the witness search parameters plus the set of witnesses found during a search. If this set is empty after the search, then it means that the refinement holds. The refinement fails to hold otherwise.

state

RSState

$$RSParams;\ wts : \mathbb{P}\ Witness$$

$$wts \in \mathbb{F}\ Witness \land \#wts \leq wr$$
$$\forall\, sN : DNode;\ iN : Node;\ a : \mathbb{P}_1\ \Sigma \mid NodePairInv \land \neg\, GenVl \land a \in enabled\,(ip.ts, iN) \bullet \neg \bigcup (enabled\,(nf.ts, sN)) \cap a = \{\}$$
$$\forall\, w : Witness \mid w \in wts \bullet \exists\, sN : DNode;\ iN : Node \bullet WitnessInv \land GenVl$$

The state invariant guarantees that the number of witnesses searched does not exceed the amount requested. The requirement that *wts* is finite is given as a predicate instead of a declaration, since this is a better Z idiom for ***Z/Eves***. To ensure that the *NFPTS* automaton given in *RSParams* has been properly normalised, for every valid node pair (sN, iN) not violating any refinement condition, such that there is a visible arc ($a \in \mathbb{P}_1\ \Sigma$) available from iN, at least one event from sN must also be available in the normal form. Otherwise, the compatibility criteria would not be catching refinement violations. Finally, every witness in *wts* must satisfy the witness invariant whenever there is a compatibility check violation.

The action *InitRSState* initialises the search parameters according to the given input variables. Furthermore, the search done (sd) flag is reset and the set of witnesses

is empty since no check has been performed yet.

$$
\begin{array}{|l}
\textit{InitRSState} \\
\hline
RSState'; \ spec? : NFPTS; \ impl? : IPTS \\
mdl? : Criterion; \ emd? : EMode; \ alv? : ALevel; \ nwts? : 1 \mathinner{..} maxWts \\
\hline
nf' = spec? \land ip' = impl? \land m' = mdl? \land em' = emd? \\
al' = alv? \land wr' = nwts? \land wts' = \{\} \land sd' = f \\
\hline
\end{array}
$$

A proof that there exists an initialisation for this process is stated with the next applicability conjecture discharged in ***Z/Eves***.

theorem tInitRSStatePRE
$\forall\, mdl? : Criterion; \ emd? : EMode; \ alv? : ALevel; \ spec? : NFPTS;$
$\quad impl? : IPTS; \ nwts? : 1 \mathinner{..} maxWts \bullet \exists\, RSState' \bullet InitRSState$

The action *RefCheckParams* gathers the witness search parameters through the corresponding channels and initialises the state. The normal form is received through channel *spec_out* in the local variable *spec*?. This communication comes from the normalisation process defined earlier.

$$
RefCheckParams \mathrel{\widehat=} \left(\begin{array}{l}
spec\_out?spec \rightarrow select\_criterion?mdl \rightarrow \\
mode?emd \rightarrow alevel?alv \rightarrow witnesses\_no?nwts \rightarrow \\
impl\_out?impl \rightarrow InitRSState
\end{array} \right)
$$

Next, the values of the selected model, the execution mode, the automation level, the number of witnesses requested, and the implementation automaton are input by the environment through the appropriate channels. Finally, the action updates the state according to schema action *InitRSState*.

4.6.5 Searching for witnesses

After initialisation, the search parameters may not change. The set of witnesses may increase and previously found witnesses are not lost.

$$
\begin{array}{|l}
\textit{RSOps} \\
\hline
\Xi\, RSParams; \ \Delta\, RSState \\
\hline
wts \subseteq wts' \\
\hline
\end{array}
$$

As before in the compatibility criteria, we factor the searching for witness with respect to each available compatibility criterion. This allows a modular combination of refinement criteria. Thus, for each available criterion, an operation is defined to search for witnesses. We start by the traces criterion.

$$
\begin{array}{|l}
\textit{TrWtsSearch} \\
\hline
RSOps \\
\hline
m = tr \land wts' \subseteq \{\, w : Witness; \ sN : DNode; \ iN : Node \mid \\
\qquad\qquad WitnessInv \land TrVl \bullet w \,\} \\
\hline
\end{array}
$$

The set of witnesses found must be a subset of the set containing all valid witnesses according to the witness invariant (see Section 4.6.2) and related violation condition

(see Section 4.6.1). For instance, for the traces criteria, the witnesses found must be contained in the set of witnesses satisfying the witness invariant that violate the traces criterion, as defined by schema *TrWtsSearch*. Since we are only interested in the number of witnesses requested (wr), the value of wts' is a subset of, rather than equal to, the entire amount of valid witnesses. Similarly, for the stable-failures criteria, the individual violation conditions are disjoined in schema *SFlWtsSearch*.

SFlWtsSearch

$$RSOps$$

$$m = sfl \land wts' \subseteq \{ w : Witness;\ sN : DNode;\ iN : Node \mid WitnessInv \land (TrVl \lor SFlVl) \bullet w \}$$

As expected, the most detailed failures-divergences criteria contains the disjunction of all violation conditions.

FlDvWtsSearch

$$RSOps$$

$$m = fldv \land wts' \subseteq \{ w : Witness;\ sN : DNode;\ iN : Node \mid WitnessInv \land (TrVl \lor SFlVl \lor DvVl) \bullet w \}$$

Finally, we define the total operation for finding witnesses, regardless of the criteria, as the disjunction of the witness search operations available. A similar approach can be taken if one wants to extend the refinement criteria, such as with specialised failures [BL04]. This modular approach gives room for future extensions by providing a new violation criterion.

$$FindWitnesses \mathrel{\widehat=} (TrWtsSearch \lor SFlWtsSearch \lor FlDvWtsSearch)$$

The proof that this operation is total is given by the theorem

theorem tFindWitnessesAppl
$\forall RSState \bullet \text{pre } FindWitnesses$

The mechanical proof of this and other theorems can be found in [Fre04d].

The *FindWitnesses* action encapsulates the most important aspect of the witness search: the compatibility check of node pairs, and finding successors for an already compatible pair. The former is established by the violation conditions for each criterion, whereas the latter is guaranteed by the witness invariant. This is detailed through the calculation of the sequential algorithm given in Section 4.8, which is based on this abstract version of witness search.

4.6.6 Refinement queries

A successful refinement report is characterised by a search that could not find a witness. This is output through the variable $rr!$ given by the next action, where no change to the state occurs.

$$ReportSuccess \mathrel{\widehat=} [\Xi RSState;\ rr! : RefRep \mid wts = \{\} \land rr! = rrSuccess]$$

$$ReportFailure \mathrel{\widehat=} [\Xi RSState;\ rr! : RefRep \mid \neg\, wts = \{\} \land rr! = rrFailure]$$

Conversely, the refinement relation fails to hold when the search is able to find a wit-

ness, and *rrFailure* is output through *rr*! in the *ReportFailure* action. The *ReportInfo* schema is a total operation for the report query as the disjunction of each case.

$$ReportInfo \mathrel{\widehat=} (ReportSuccess \lor ReportFailure)$$

theorem tReportInfoAppl
$\forall RSState \bullet \text{pre } ReportInfo$

The action *Report* enables the environment to query for a refinement report through the *query* channel, successfully terminating afterwards.

$$Report \mathrel{\widehat=} \textbf{var } rr : RefRep \bullet (\, query!rqReport \rightarrow ReportInfo \,) \; ; \; (\, report!rr \rightarrow Skip \,)$$

When a query report is selected by the environment (*rqReport*), the action calculates the information as specified by the *ReportInfo* operation. This value is then output via the *report* channel. Debugging information can be required by communicating the value *rqDebug* through the *query* channel as defined by the *Debug* action.

$$Debug \mathrel{\widehat=} query!rqDebug \rightarrow testimony!wts \rightarrow Skip$$

In that case, any witnesses found are output via channel *testimony*. The next action offers to the environment the possibility for a search report or debugging information.

$$Query \mathrel{\widehat=} Report \,\Box\, Debug$$

The communication from action *Debug* is of interest for the debugger process defined in Section 4.7 in order to provide human-readable information to the environment.

4.6.7 Refinement checking action

The action *Reset* signals the termination of a witness search for a particular configuration of parameters. It restarts the process for the next witness search round as given by the refinement search process main action defined below.

$$Reset \mathrel{\widehat=} reset \rightarrow Skip$$

The next action gathers together the expected flow of control for the top-level operations available after the initialisation of the state. These operations are either a search for refinement, or one of the two available queries for reports or debugging information. Variable *sd* is used in a guard to ensure that, for every (re-)initialisation of the state, witness search is performed only once. The other query operations can be performed any time after the search has been done.

$$RefCheck \mathrel{\widehat=} \mu X \bullet \left(\begin{array}{l} (sd = t) \,\&\, Query \,\Box \\ (sd = f) \,\&\, (\; FindWitnesses \,;\, sd := t \;) \end{array} \right) ; X$$

$$\bullet \; (\mu X \bullet RefCheckParams \,;\, ((Reset \,\Box\, RefCheck) \,;\, X))$$

end

The main action of the process is defined recursively. It enables the environment to either reset or perform the available top-level operations after gathering the refinement search parameters, and then recurse.

4.7 Debugger specification

Together with high levels of automation, the capacity to provide human-readable counter-examples from a refinement failure is the most user-effective aspect of model checking. That means it is pointless to find a problem that cannot be understood by the user, or is cluttered with enormous (usually unreadable) amounts of data.

In practice, the debugger is just an information caster. It retrieves data from the witness search process and casts it into a human-readable format. For example, in FDR a debugger window provides a tree-view with the process graph from the root of the search down to the point of failure, together with information about traces and acceptances at each node. FDR's debugger can be studied in detail due to an algorithm that investigates its object model [Fre03].

We provide the specification of a debugger similar to FDR with debugging windows representing witnesses returned by the search algorithm with additional information about traces, acceptances, and divergence. The abstract specification of the debugger process in *Circus* comes next.

4.7.1 Channels and data types

Firstly, let us declare a given set representing a unique debugging window identifier. It is used to establish a debugging session with the user.

$$[DbgWndID]$$

The information retrieved by the debugger from the witness search process is a set of witnesses. Inspired by FDR, together with trace counter-examples, we also want to provide at each node pair in a witness information about its acceptances sets with respect to the trace performed up to that point. This defines what is needed as the local information contained in a debug window: the witness to build the graph of failure; the trace of failure; and the observed and offending acceptances at the incompatible node pair.

$$DbgWnd \mathrel{\widehat=} [\, wts : Witness;\ trf : \mathrm{seq}\ \Sigma;\ obs : Acceptances;\ off : Acceptances \,]$$

The casting of acceptances enables sophisticated exploration of the failure by the environment. Two new channels are declared to allow the environment to select one debug window to work with, and to return to the environment the calculated information of a debug window: the type of *inform* is the schema $DbgWnd$.

$$\textbf{channel}\ select\_window : DbgWndID$$
$$\textbf{channel}\ inform : DbgWnd$$

The debugger is a sequential process that interacts with the witness search process (see Figure 4.2).

$$\textbf{process}\ Debugger \mathrel{\widehat=} \textbf{begin}$$

The next step is to retrieve acceptances information from the witness received from the search process. Function *wtsAccs* returns the set of acceptances pairs from a witnesses related to the given normal form and implementation. The function returns a pair of acceptances sets representing those observed by the specification, and those offended by the implementation. The acceptances sets of a witness are calculated through information present in the witness invariant, the acceptances function of the normal

form, and the transition system of the implementation. The observed acceptances come from the normal form *accs* function, whereas the offending acceptances comes from the *enabled* arcs of the implementation at the point of failure.

$$wtsAccs : NFPTS \times IPTS \times Witness \rightarrow \mathbb{P}\,(Acceptances \times Acceptances)$$

$$\begin{aligned}
&\forall\, nf : NFPTS;\ ip : IPTS;\ w : Witness \bullet \\
&\quad wtsAccs\,(nf, ip, w) = \{\, obs, off : Acceptances;\ m : Criterion; \\
&\qquad sN : DNode;\ iN : Node \mid (sN, iN) = last\,(w.1) \land WitnessInv \land \\
&\qquad GenVl \land obs = nf.accs\,(sN) \land off = enabled\,(ip.ts, iN) \\
&\qquad\quad \land \neg\,(off \subseteq obs) \bullet (obs, off)\,\}
\end{aligned}$$

Obviously, the offending acceptances must not be contained in any of the observed ones. This trivially holds since a valid witness has an incompatible last node pair, as guaranteed by the witness invariant and the generic violation criteria.

At a later stage we should consider including a node pair from the given witness as an additional parameter. It would allow one to select the acceptances information at points different from the one of failure. The definition of this additional debugging facility is left as future work.

4.7.2 State definition and initialisation

The process state is defined via promotion [WD96, Chapter 13], where debugging windows represent the local state. It contains the global debugging parameters, such as the normal form and the implementation automata, and a function for the promotion of open debug windows.

state

DbgState

$$c : Criterion;\ nf : NFPTS;\ ip : IPTS;\ dbgi : DbgWndID \pfun DbgWnd$$

The initialisation of the state components is defined by the schema *InitDbgState*. These values are initialised according to the input parameters, and as there are no open debug windows initially, the *dbgi* function is empty.

InitDbgState

$$DbgState';\ mdl? : Criterion;\ spec? : NFPTS;\ impl? : IPTS$$

$$c' = mdl? \land nf' = spec? \land ip' = impl? \land dbgi' = \{\,\}$$

The next action receives the initialisation parameters from the environment and initialises the process state.

$$Initialise \mathrel{\widehat=} select\_criterion?mdl \rightarrow spec\_out?spec \rightarrow impl\_out?impl \rightarrow InitDbgState$$

The applicability theorem of the state initialisation schema is trivially ***true*** as we assign (type-correct) values to all initial values of the debugger state.

4.7.3 Debugging sessions—starting debugging windows

After initialisation, only the function about open debug windows can be altered, as defined by the next schema *DbgOps*.

$$DbgOps \mathrel{\widehat=} [\,\Delta DbgState \mid m' = m \land nf' = nf \land ip' = ip\,]$$

A failure occurs whenever the process retrieves a witness from the witness search

process through the *wtsSet* input variable. This witness is then set as the working witness of a fresh debug window.

$$DbgFailure \;\widehat{=}\; [\,\Delta DbgWnd;\; wtsSet? : \mathbb{P}\; Witness \mid wts' \in wtsSet?\,]$$

The promotion to an operation on the process state includes the fresh debug window, provided it is possible to calculate the debug window information from the process state at this stage. The trace is calculated via function *wtsTrace* for the given witness of the local debugging window and related automata. This trace always exists, as guaranteed by the witness invariant. Details of the *wtsTrace* function are trivial and omitted here (see [Fre04a, Axiomatic Definition 3.5, and Schema 4.7] for details).

$$\begin{array}{|l}
\hline
DbgWndPromote \\
DbgOps;\; \Delta DbgWnd;\; wndID? : DbgWndID \\
\hline
wndID? \in \mathrm{dom}\; dbgi \land \theta\, DbgWnd = dbgi\; wndID? \\
trf' = wtsTrace\,(nf, ip, wts') \\
(obs', off') \in wtsAccs\,(nf, ip, wts') \\
dbgi' = dbgi \oplus \{\, wndID? \mapsto \theta\, DbgWnd' \,\} \\
\hline
\end{array}$$

The *wtsAccs* function is used to choose one of the offending acceptances from available node pairs to include into the debug window. Finally, the fresh debug window is inserted into the process state. Unfortunately, because the calculation of the debug window depends on both automata, this is not a free-promotion.

A debug session is formed by the existence of a local debug window representing a failure promoted into the process state. The *DbgFailure* schema guarantees that the freshly promoted debug window is indeed from a valid witness from *wtsSet*?.

$$DbgSession \;\widehat{=}\; (\exists\, DbgWnd \bullet DbgFailure \land DbgWndPromote)$$

Finally, the action *StartDbgSession* synchronises with the witness search process on the *testimony* channel to retrieve the set of witnesses found.

$$StartDbgSession \;\widehat{=}\; testimony?\, wtsSet \rightarrow \begin{pmatrix} (\,(wtsSet = \varnothing)\;\&\; Skip) \;\Box \\ (\,(wtsSet \neq \varnothing)\;\&\; DbgSession) \end{pmatrix}$$

Whenever a witness has been found, the set *wtsSet* is not empty and a new debug session is started via the operation defined by the schema *DbgSession*. Otherwise, if there are no witnesses available, the action just silently terminates.

4.7.4 Debugging session selection

Once the debugging windows have been defined, the environment is allowed to chose between them. Firstly, we name the set of available debug window identifiers in function *dbgi* from the process state.

$$PossibleDbgWndID \;==\; \{\, x : DbgWndID;\; DbgState \mid x \in \mathrm{dom}\; dbgi \bullet x \,\}$$

Next, action *ChooseDbgWnd* allows the selection of one of the available debug windows from the process state via channel *select_window*, where the chosen identifier comes from the set defined above.

$$\begin{aligned} ChooseDbgWnd \;\widehat{=}\;\; & select\_window?\, dwID : (dwID \in PossibleDbgWndID) \rightarrow \\ & inform!(\; dbgi\; dwID\;) \rightarrow Skip \end{aligned}$$

The debug window information chosen is then output to the environment via channel

inform as a binding calculated with the *dbgi* function.

$$Inform \mathrel{\widehat=} ((dbgi = \varnothing) \,\&\, Skip \,\Box\, (dbgi \neq \varnothing) \,\&\, ChooseDbgWnd)$$

Finally, the action *Inform* enables the environment to choose from available debug windows, or successfully terminate whenever there are no windows open.

4.7.5 Debugger main behaviour

Like the witness search process, we allow the environment to reset the process parameters and start debugging for new automata. Action *Reset* is defined similarly as before for the witness search process.

$$Reset \mathrel{\widehat=} reset \rightarrow Skip$$

The top-level debug operation is defined by action *Debug* via recursion. It allows new debugging sessions to be started followed by the provision of information about its results in sequence.

$$Debug \mathrel{\widehat=} \mu\, X \bullet (StartDbgSession \,;\, Inform \,;\, X)$$

The main action, initialises the state and offers the environment the external choice to either reset with new parameters via action *Reset*, or retrieve debug information via action *Debug*.

$$\bullet\ (\mu\, X \bullet Initialise \,;\, ((Reset \,\Box\, Debug) \,;\, X))$$

end

This completes the abstract specification of the selected components from the model checker strategy shown in Figure 4.2.

4.8 Witness search sequential implementation

After presenting the abstract *Circus* specifications for each part of the model checking component (see Figure 4.2), we describe in this section the sequential implementation of the witness search algorithm. It is derived through *Circus* refinement laws from the abstract witness search specification presented in Section 4.6.

4.8.1 Process state

The sequential implementation is given next by the *SeqWtsSearch* process. Proof obligations generated by this calculation were discharged using ***Z/Eves*** and can be found in [Fre04d].

process $SeqWtsSearch \mathrel{\widehat=}$ **begin**

The sequential implementation needs some additional state components. For the sake of modularity, these components are described separately by the next schema *SeqRSStCmp*. Apart from the witnesses found, the sequential state also includes three sequences used during the (specialised BFS) witnesses search sequential algorithm. The set *swts* records the finite witnesses found during the search. The sequences *ck* and *pd* records the pairs of nodes that have already been checked to be compatible, and the pairs pending to be checked, respectively. In a traditional BFS implementation,

members of *pd* are usually marked as *unknown*, and members of *ck* as *visited*. These sequences are injective since node pairs are processed only once. The sequence *lvl* records the current level (or layer) of the search. Together with *ck* and an offending node it forms possible witnesses. Finally, *wnp* and *wl* record the current (working) node pair and level of the search.

$$
\begin{array}{|l}
\hline
\textit{SeqRSStCmp} \\
\textit{swts} : \mathbb{P}\ \textit{Witness};\ \textit{ck}, \textit{pd} : \mathrm{iseq}\ \textit{NodePair};\ \textit{lvl} : \mathrm{seq}\ \mathbb{N} \\
\textit{wnp} : \textit{NodePair};\ \textit{wsN} : \textit{DNode};\ \textit{wiN} : \textit{Node};\ \textit{wl} : \mathbb{N} \\
\hline
\textit{swts} \in \mathbb{F}\ \textit{Witness} \land \textit{wnp} = (\textit{wsN}, \textit{wiN}) \\
\hline
\end{array}
$$

Each component of the working node pair is also given explicitly due to some technicalities related to a better level of automation while mechanising tuples in ***Z/Eves***.

We decided to have these additional state components instead of local variables during the witness search, because they are strictly related to the nature of the process being developed. Having these state components as a modular schema also makes refinement proofs in ***Z/Eves*** easier. Moreover, because the state has a rather complex invariant, it is introduced in a compositional and stepwise fashion, going from a basic setup to the actual version assembled via schema conjunction. Semantically, local variables of the main action and state components are equivalent.

Let us introduce the complete state components via the next schema. They are formed by the abstract refinement parameters together with the additional sequential algorithm components used by the sequential implementation.

$$\textit{SeqRSStateComponents} \mathrel{\widehat{=}} \textit{RSParams} \land \textit{SeqRSStCmp}$$

The basic invariant is directly related to the abstract version of the state defined by *RSState*, and defined here by schema *SeqRSStateBasic*.

$$
\begin{array}{|l}
\hline
\textit{SeqRSStateBasic} \\
\textit{SeqRSStateComponents} \\
\hline
\#\textit{swts} \leq \textit{wr} \\
\forall\, a : \textit{Arc} \mid \neg\, \textit{GenVl}[\textit{wsN}/\textit{sN}, \textit{wiN}/\textit{iN}] \land \neg a = \{\} \land \\
\qquad a \in \textit{enabled}\,(\textit{ip.ts}, \textit{wiN}) \bullet \neg \bigcup (\textit{enabled}\,(\textit{nf.ts}, \textit{wsN})) \cap a = \{\} \\
\hline
\end{array}
$$

It contains the restriction on the number of witnesses to be searched, as well as the normal form consistency predicate. Differently from the abstract version (see the schema *RSState* in Section 4.6.4 on page 108) that universally quantifies node pairs to consider for normal form consistency, now we are using the working node pair instead. The next part of the invariant includes the restriction on the sequences used to form new witnesses. Since witnesses found are formed by the checking and level sequences, these sequences must have the same size in order to fulfill the joint path and witness invariants defined earlier (see Section 4.6.2). This condition is defined by the *SeqRSStateInvWts* schema.

$$\textit{SeqRSStateInvWts} \mathrel{\widehat{=}} [\, \textit{SeqRSStateComponents} \mid \#\textit{ck} = \#\textit{lvl} \,]$$

In what follows, predicates related to the invariant of particular state component elements are given. This separation of the invariant addresses the complexity of refinement proofs in ***Z/Eves*** by providing a modular structure adequate for the

theorem prover during the discharge of proof obligations generated by the refinement calculation. Let us define some additional functions over automata first. The function PA defines the maximum state space to be searched as the cross product between all the nodes from the specification and implementation automata respectively.

$$PA : NFPTS \times IPTS \to \mathbb{P}\ NodePair$$

$$\forall\, nf : NFPTS;\ ip : IPTS \bullet PA\,(nf, ip) = (\{\, nf.init \,\} \cup nodesnf.ts \times \{\, ip.init \,\} \cup nodesip.ts)$$

The function PS represents the actual size of this set. As each automaton has at least the initial node, the result must be a strictly positive natural number.

$$PS : NFPTS \times IPTS \to \mathbb{N}_1$$

$$\forall\, nf : NFPTS;\ ip : IPTS \bullet PS\,(nf, ip) = \#(PA\,(nf, ip))$$

The restrictions on the members of the working node pair, checking, and pending sequences are defined next by the schema $SeqRSStateInvWnpCkPd$.

$$\begin{array}{l} SeqRSStateInvWnpCkPd \\ \hline SeqRSStateComponents \\ \hline wnp \in PA\,(nf, ip) \land NodePairInv[wsN/sN, wiN/iN] \\ ck \in \mathrm{iseq}\,(PA\,(nf, ip)) \land pd \in \mathrm{iseq}\,(PA\,(nf, ip)) \\ \forall\, sNck : DNode;\ iNck : Node \mid (sNck, iNck) \in \mathrm{ran}\ ck \bullet \\ \quad NodePairInv[sNck/sN, iNck/iN] \land \neg GenVl[sNck/sN, iNck/iN] \\ \forall\, sNpd : DNode;\ iNpd : Node \mid (sNpd, iNpd) \in \mathrm{ran}\ pd \bullet \\ \quad NodePairInv[sNpd/sN, iNpd/iN] \end{array}$$

The working node belonging to each automaton is guaranteed by the first two (equivalent) predicates. They define that wnp is inside of the product of both automata, and also enforce that the node pair invariant holds. The redundancy is not strictly needed, but it is useful for the sake of proofs in ***Z/Eves***, where the individual elements of a node pair tuple must be mentioned. Similarly, the last four predicates ensure that members of ck and pd are valid. The predicate

$$ck \in \mathrm{iseq}\,(PA\,(nf, ip)) \land pd \in \mathrm{iseq}\,(PA\,(nf, ip))$$

guarantees that the node pairs of both sequences must be valid members from the product automata; it helps ***Z/Eves*** mechanisation. The next predicate strengthens the invariant on checked members, insisting they must not only be valid, but also compatible, as guaranteed by the predicate

$$\neg\, GenVl[sNck/sN, iNck/iN]$$

where the redundancy on the validity of elements through $NodePairInv$ is used once more. The last predicate says that the node pairs on pd are valid, and as they are pending, nothing about their compatibility is known yet. The invariant relating the members of the checked and pending sequences is given next by the schema $SeqRSStateInvRelCkPd$.

$$SeqRSStateInvRelCkPd \ \widehat{=}\ [\, SeqRSStateComponents \mid \mathrm{ran}\ pd \cap \mathrm{ran}\ ck = \{\,\} \,]$$

In order for the search to terminate, it is very important that checked nodes are not included as pending, and the number of pairs checked must not exceed the state space

of both automata as defined by the PS function above. The former is guaranteed by the disjointness of their elements with the predicate

$$\text{ran } pd \cap \text{ran } ck = \{\}$$

whereas the later is defined by the restriction on the size of ck being less than or equal to the size of the product automata ($\#ck \leq PS(nf, ip)$). This can be derived as a lemma using the predicate ($ck \in \text{iseq}(PA(nf, ip))$) given in schema *SeqRSStateInvWnpCkPd* above. The relationship is explicitly stated through the next lemma. It has been proved in ***Z/Eves*** and can be found in [Fre04a, Lemma 3.2].

theorem lISeqNFIPSizeBounded
$$\forall nf : NFPTS;\ ip : IPTS;\ s : \text{iseq } NodePair \mid s \in \text{iseq}(PA(nf, ip)) \bullet \#s \leq PS(nf, ip)$$

After that, we introduce the invariant related to the search level via the next schema.

SeqRSStateInvWlLvl
$$\begin{array}{l} SeqRSStateComponents \\ \hline \forall i : 1 \,..\, (\#lvl - 1) \bullet lvl(i) \leq lvl(i+1) \\ \forall j : 1 \,..\, \#lvl \bullet lvl(j) \leq wl \end{array}$$

The levels recorded by the lvl sequence must never decrease, as the sequence grows during the search. Consequently, because the working level must be the most recent one, every recorded level at index j must also be lower than the working level wl. The invariant relating found witnesses with pending members is given by the next schema. If any witness has already been found, then it must not be formed by any members pending to be checked. This must hold since the normal form guarantees uniqueness of pairs being searched.

SeqRSStateInvRelSwtsPd
$$\begin{array}{l} SeqRSStateComponents \\ \hline \forall w : Witness \mid w \in swts \bullet \text{ran } pd \cap \text{ran } w.1 = \{\} \end{array}$$

The invariant on found witnesses is given next by the schema *SeqRSStateInvSwtsElem*.

SeqRSStateInvSwtsElem
$$\begin{array}{l} SeqRSStateComponents \\ \hline \forall w : Witness \mid w \in swts \bullet WitnessInv[wsN/sN, wiN/iN] \\ \quad \wedge\ GenVl[wsN/sN, wiN/iN] \end{array}$$

Like the abstract version, every witness found must satisfy the witness invariant, where the last node pair is violating the general refinement compatibility criteria. The difference is that the sequential version fixes this last element on a witness to be the current working node pair, instead of any valid one.

state

$$\begin{array}{rcl} SeqRSState & \widehat{=} & SeqRSStateBasic \wedge SeqRSStateInvWts \wedge \\ & & SeqRSStateInvWnpCkPd \wedge SeqRSStateInvRelCkPd \wedge \\ & & SeqRSStateInvWlLvl \wedge SeqRSStateInvRelSwtsPd \wedge \\ & & SeqRSStateInvSwtsElem \end{array}$$

Finally, the complete process state is defined by the schema *SeqRSState* as the con-

junction of the previous schemas.

Perhaps one might argue that leaving all these variables in the state complicates some proof obligations (and the state itself). Instead, the complexity should be distributed through the introduction of variable blocks and sequential composition during the refinement calculation. We think that having the major aspects of the algorithm present in a single place makes it easier to understand the algorithm invariant. Moreover, using the refinement calculus in that way would yield a higher number of proof obligations to discharge. Having the predicates spread around means more *"easy-to-hard"* proofs everywhere. Keeping these key predicates together happened to generate a couple of *"difficult"* at early stages followed by *trivial-to-easy* remaining proof obligations. We believe our choice saved a considerable amount of proof effort in ***Z/Eves***.

4.8.2 Retrieve relation

We decided to keep the data refinement as simple as possible for the sake of conciseness of proof obligations. In our implementation it is just the identity relation linking appropriate components of different state spaces.

$$SeqRSRetrieve \mathrel{\widehat=} [\, RSState;\ SeqRSState \mid swts = wts \,]$$

Previously, we had given the found witnesses in the sequential implementation on the state component *swts* as an injective sequence. This does not give any insight, but increases the number of proofs to be discharged to a great extent because ***Z/Eves*** lacks extensive automation support for injections and injective sequences. Thus, we are assuming that the target implementation language provides some sort of set representation. This poses no problem as most programming languages provide such data structure. In particular, with our choice with Java, this follows directly.

4.8.3 State initialisation

The sequential version of the state initialisation is given by the next action. It receives the same state components in the same order as the abstract version, and initialises each state component with the read values, as well as appropriate initial values via an assignment command.

$$SeqRefCheckParams \mathrel{\widehat=} \left(\begin{array}{l} spec\_out?spec \rightarrow select\_criterion?mdl \rightarrow mode?emd \rightarrow \\ alevel?alv \rightarrow witnesses\_no?nwts \rightarrow impl\_out?impl \rightarrow \\ \quad m, em, al, nf, ip, wr, := mdl, emd, alv, spec, impl, nwts, \\ \quad swts, ck, lvl, pd, \quad \varnothing, \langle\rangle, \langle\rangle, \langle(spec.init, impl.init)\rangle, \\ \quad wnp, \quad (spec.init, impl.init), \\ \quad wsN, wiN, wl \quad spec.init, impl.init, 0 \end{array} \right)$$

The model, the search mode, the automation level, the normal form, the implementation and the number of witnesses are input by channels *spec_out*, *select_criterion*, *mode*, *alevel*, *witnesses_no*, and *impl_out* respectively. The set of witnesses, the checking sequence, and the level sequence are initially empty. The pending sequence has the working node pair as the root nodes from each automata, and the current working level is 0. During our calculation, we proved that this initialisation of parameters is a refinement of the abstract version by establishing the simulation below

(see [WC01a] and [WD96, Chapter 16]).

$$RefCheckParams \preccurlyeq SeqRefCheckParams$$

This proof uses an intermediate schema representing the initialisation. It is then transformed to the given assignment using the assignment introduction law (see [Cav97, Appendix D, Law assC]). The complete simulation including the applicability and correctness theorems proofs is given in [Fre04d].

4.8.4 Sequential witness search algorithm

The action *SeqWitnesses* is the most important, as it describes the sequential algorithm for witness search formally calculated through refinement laws from the abstract specification. It defines how node pairs are checked for compatibility, as well as how new pairs are found to progress the search. To give an overview of the algorithm we provide the entire derived code in Figure 4.4 written using guarded commands, which are part of the *Circus* notation; explanation of the algorithm is provided afterwards. This code has been derived by calculation using the refinement calculus for Z (ZRC) [Cav97], and action refinement laws for *Circus* [CSW02], where all proof obligations have been mechanically discharged using ***Z/Eves***. An explanation of each part is provided afterwards. In total, we applied around 68 laws, which generated 38 proof obligations. The following list illustrates the proof effort of mechanised theorem proving involved while dealing with these proofs.

- 14 trivial proofs (5-10 minutes)—proved directly by ***Z/Eves***.
- 12 easy proofs (30-45 minutes)—minor manipulations (possibly lengthy), but still straightforward.
- 8 hard proofs (2-7 days)—lengthy and complex proofs usually depending on a subset of ***Z/Eves*** rules and lemmas related to generic properties, as well as cases analysis.
- 4 difficult proofs (6-10 days)—these are the proofs that exposed most of the inconsistencies in the automata theory, and in the formal definitions presented.

Although the algorithm is similar to the algorithm of FDR presented in [MH00], the mechanised proof effort exposed the loop invariants, and a great amount of hidden information that is interesting for the understanding of the witness search problem for refinement model checking in general.

Now, let us give an overview of the algorithm. It is divided in two stages: (i) compatibility check; and (ii) successor node pairs search. For the compatibility check, we first assign the working node pair to the variable *wnp*, update the pending pairs, and incrementing the working level accordingly. If *wnp* violates the refinement condition, then it must be included as a new witness on *swts*. It is formed with the previous checked pairs together with the offending working node pair, and working level. Otherwise, if *wnp* is compatible, then the sequence of checked pairs and search level are updated likewise, and the next stage of the algorithm for finding successor pairs starts.

Because our data structures have sets on arcs instead of single events, the search is more complex than the one in FDR's algorithm. The arcs immediately available for communication in the implementation are retrieved through the *enabled* function. For each of those arcs, one needs to progress appropriately in both the normal form and

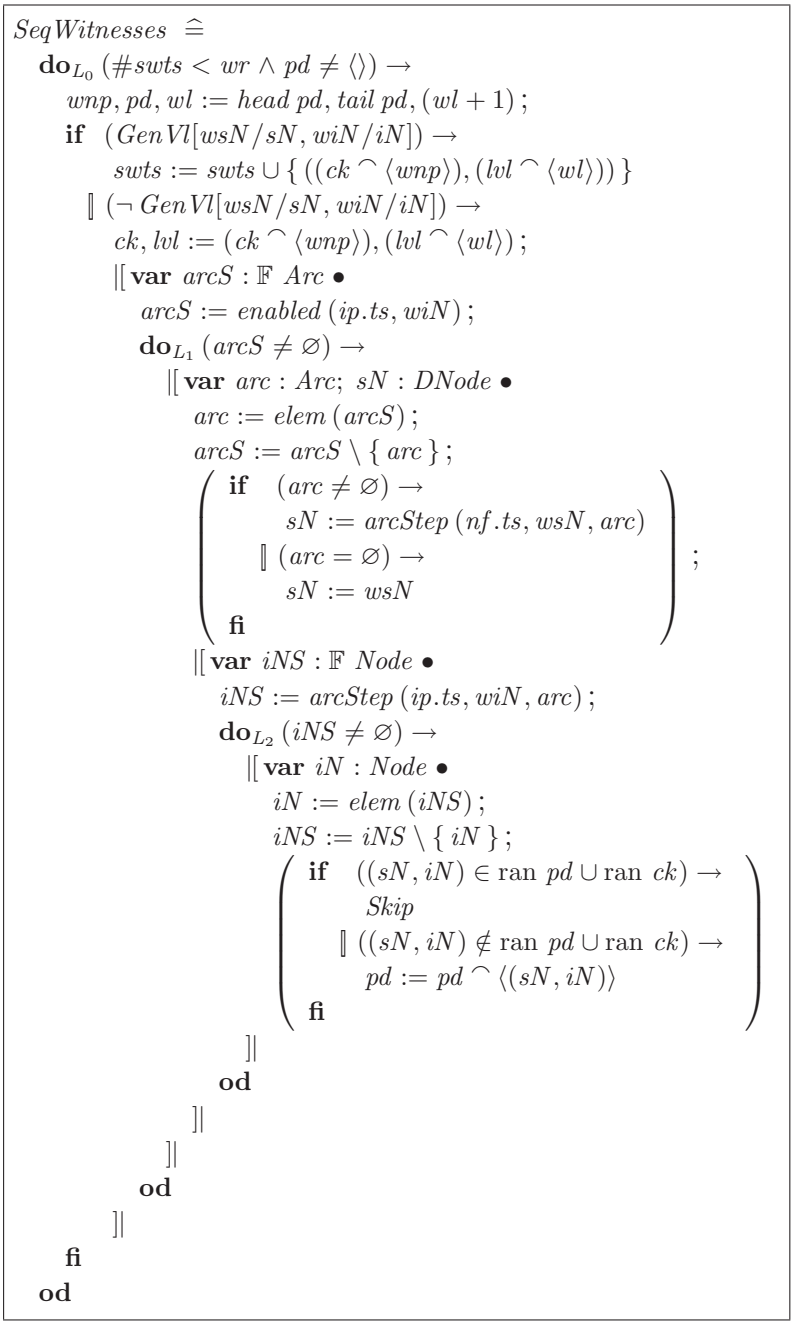

Figure 4.4: Sequential witness search algorithm

the implementation. Such progress is certain because the state invariant described via schema *SeqRSStateBasic* ensures that the normal form is well-formed with respect to the generic violation criteria defined by schema *GenVl* in Section 4.6.1. This is a requirement for mechanisation in ***Z/Eves***, which ensures that the given automata have been created using our algorithms. In order to exhaust all enabled implementation arcs ($arcS$), we choose an appropriate specification node successor (sN), and select all available implementation successor nodes ($iN \in iNS$) on the same *arc*. The appropriate specification successor node is defined according to the arc chosen. If it is a silent transition, it represents nondeterminism (from an internal choice, for instance) being resolved in the implementation. As the specification normal form is already deterministic and has no silent transitions left, there is no successor node for the specification in the case of a silent transition being chosen from the implementation. Otherwise, the selection of successors follows from the *arcStep* function. Moreover, depending on the results of *arcStep* and *enabled*, we could distinguish between checking via exhaustive search, and testing via selective search. That is, we could allow the user to conspicuously choose from the returning sets in the case of refinement testing. In this way, our tool would be more like ProBE [For00] for CSP_M, rather than FDR. Although this does not prove refinement, it still does help to find (sound) valuable counter-examples. Instrumenting the algorithm in this direction is left as future work.

In what follows, we present the proof that the abstract witness search represented by the schema *FindWitnesses* from Section 4.6.5 is refined by the sequential implementation given by the algorithm described by the action *SeqWitnesses* in Figure 4.4. As we have derived the sequential algorithm code from the abstract specification that defines containment over all valid and available witnesses, it is correct (and exhaustive) by construction. We present the most important parts of the derivation and their related proof obligations, which reveal interesting points of the algorithm. As some parts of the code are quite similar, we omit comments on repeated or straightforward proofs. A complete derivation with proof obligations mechanically discharged in ***Z/Eves*** can be found in [Fre04d]. In what follows, loop invariants, variants, and guards are highlighted with serif and slated font. Proof obligations generated by application of refinement laws are explained.

Individual witness update

The main loop looks for witnesses until the number of witnesses requested have been reached, or no more pending node pairs remain. This is better understood if analysed in three distinct stages: (i) compatibility checking, (ii) finding successors setup, and (iii) found successors update. The first stage results in either identifying a new witness, or updating the sequence of checked pairs, depending on the compatibility criterion check. The second stage is the largest and most complex. It exploits the transition system structure through the semantic functions *enabled* and *arcStep*, in order to properly setup the nodes to be searched from each automaton, such that an exhaustive search is indeed performed. Finally, the last component updates the pending node pairs with the gathered information from the second stage. In fact, following the guidelines for FDR presented in [MH00], these stages can be put in parallel in order to work faster and more efficiently. A formal derivation of a parallel algorithm is left as future work.

Now, let us present our strategy for a proof that the sequential algorithm calculated with refinement laws is a refinement of the abstract specification. We start the derivation by expanding and simplifying the action *FindWitnesses* to its equivalent

version that mentions individual witnesses, as given by the schema *WtsUpdate*. It is needed because in the refinement calculus for Z, loop invariants must mention each individual element affected. Since the algorithm is given as a loop, the transformation from the subset containment of schema *FindWitnesses* to the update on each individual witness defined using quantifiers is necessary.

WtsUpdate

$$RSOps$$

$$\forall\, w : Witness \mid w \in wts' \bullet (\exists\, sN : DNode;\ iN : Node \bullet WitnessInv \land GenVl)$$

The proof that

$$FindWitnesses \;\Leftrightarrow\; WtsUpdate$$

is lengthy but not illuminating. For both directions of the equivalence, it uses subset inclusion definitions followed by case analysis for each refinement criteria available. The next step is to include the initialisation context in order to allow us to establish the main loop invariant. Schema *WtsUpdateInitCtx* establishes the initialisation context, where the set of witnesses is empty.

$$WtsUpdateInitCtx \;\widehat{=}\; [\, WtsUpdate \mid wts = \{\,\} \,]$$

In the abstract specification, this initialisation is part of the initialisation of the whole process, which considers not only *wts*, but all the refinement parameters. Since we are concentrating on the refinement of *FindWitnesses*, we need to factor out this property of the process. Once we have proved that the individual witness update is equivalent to the original abstract specification, we should do the same for the sequential implementation and establish a simulation between both the abstract and the sequential versions. Because the implementation does not mention schemas, we need an intermediate representation. Firstly, let us define the sequential state restrictions corresponding to the schema *RSOps*. It ensures that the search parameters do not change, and the main variables of the algorithm do not loose accumulated knowledge.

SeqRSOps

$$\Xi\, RSParams;\ \Delta\, SeqRSState$$

$$swts \subseteq swts' \land ck \;\mathsf{prefix}\; ck' \land lvl \;\mathsf{prefix}\; lvl' \land wl \leq wl'$$

After that, we define the sequential individual witness search in the same way as *WtsUpdate*, but now mentioning the sequential state component $swts'$, instead of wts' from schema *SeqWtsUpd*. It also fixes the last incompatible element of a witness as the elements of the working node pair.

SeqWtsUpd

$$SeqRSOps$$

$$\forall\, w : Witness \mid w \in swts' \bullet WitnessInv[wsN'/sN, wiN'/iN] \land GenVl[wsN'/sN, wiN'/iN]$$

To prove simulation between the abstract individual witness update and the sequential implementation, we define schema *SeqWtsUpdPInitCtx* that includes the initialisation context as before. It is a partial initialisation context since there is not enough information regarding all the sequential state components yet. We choose this style

as it reduces the proof effort related to related proof obligations.

$$SeqWtsUpdPInitCtx \mathrel{\widehat=} [\, SeqWtsUpd \mid swts = \{\} \,]$$

We proved that the partial initialisation of the sequential individual witness update for any refinement criteria simulates the abstract individual witness update for any refinement criteria.

$$WtsUpdateInitCtx \preccurlyeq SeqWtsUpdPInitCtx$$

The applicability and correctness proofs are lengthy but simple. In fact, the simulation

$$WtsUpdate \preccurlyeq SeqWtsUpd$$

also holds because $wts \subseteq wts'$ is present in schema $RSOps$. The initialisation contexts were included earlier here for the sake of refinement proofs discharged in ***Z/Eves*** while establishing the main loop invariant in the proof of correctness for the sequential algorithm. The sequential individual witness update with a complete initialisation context is given next. It contains all the necessary information to establish the main loop invariant. This early style of presentation in the introduction of the initialisation context is needed once more in order to use ***Z/Eves*** for the proof obligations related to the application of Z refinement laws.

$$SeqWtsUpdInitCtx \mathrel{\widehat=} [\, SeqWtsUpdPInitCtx \mid ck = \langle\rangle \land wl = 0 \,]$$

Finally, after establishing the link between the abstract and the sequential individual witness update via simulation, we want to prove that the given algorithm is an action refinement of the latter

$$SeqWtsUpdInitCtx \sqsubseteq_{\mathcal{A}} SeqWitnesses$$

This is the most complex proof, which involves a series of refinement laws and related proof obligations.

The application of ZRC laws on this action refinement proof generated proof obligations discharged with ***Z/Eves***. They come from the specification statements [Mor94] and must be appropriately translated back to schemas in order to be analysed with ***Z/Eves***, before we can start discharging the proof obligations. This translation is not mechanised since there is no support for specification statements in ***Z/Eves***, but it can be justified by the laws of ZRC [Cav97]. Nevertheless, the use of schema calculus makes the proofs concise, elegant, and easier to follow than the alternative by hand.

Main loop

Let us describe the introduction of the main loop (labelled L_0) by explaining its invariant, guard, and variant respectively. With application of appropriate laws and further simplifications, the *main loop invariant is reduced to the sequential state invariant already presented in schema* $SeqRSState$. That is not surprising because we moved the algorithm main variables to the state on purpose at the beginning, in order to have the main loop invariant clear from the state invariant itself.

The main loop guard defines the termination condition:

$$\#swts < wr \land \neg pd = \langle\rangle$$

either enough witnesses have been found, or there are no more pending pairs to be checked. It has been previously introduced by strengthening the postcondition.

By definition [Mor94], a loop variant is an integer expression whose value is strictly decreased by the loop body. Nevertheless, *swts* is a set that only grows at every iteration. Furthermore, despite the fact the size of *pd* always shrinks by 1 because of the current node pair being checked and removed, it can grow in the same iteration when new pairs are found for a compatible working node pair. Due to its conditional variation, *pd* cannot be part of the variant; however, we can use *ck* as it is a sequence that only grows. There is a clear relation between *ck* and *pd* guaranteed by the loop invariant: their elements are both valid and disjoint.

$$ck \in \text{iseq}\,(PA\,(nf, ip)) \wedge pd \in \text{iseq}\,(PA\,(nf, ip)) \wedge \text{ran}\ pd \cap \text{ran}\ ck = \{\,\}$$

This property of validity and disjointness between the members of these sequences fills the gap between the invariant mentioning *ck* and *swts*, and the guard mentioning *swts* and *pd*. Since a node pair is always either compatible or not, either the value of *ck* or *swts* will increase, but not both. *The main loop variant is then defined as*

$$(PS\,(nf, ip) - \#ck) + (wr - \#swts)$$

The checking sequence is bound by the product size of both automata via function *PS* as we can never check more than what is available, whereas the set of witnesses is bound by the number requested on *wr*.

Working node pair

Right after the introduction of the main loop, we retrieve the first pending pair as the current working node pair (*wnp*) via the assignment introduction law.

$$wnp, pd, wl := head\ pd, tail\ pd, (wl + 1)$$

Moreover, the value of *pd* is updated to its tail, and the working level is incremented by one. So, we need to check *wnp* for compatibility in order to either include it as the last pair of a new witness, or to include it as a new checked pair that we need to search for successor pairs.

To discharged the proof obligation related to the introduction of this assignment, we need to establish the properties of *wnp* with respect to *pd*. Firstly, we remove the condition that *pd* is not empty from the guard, since its tail might be empty. Next, we introduce the properties of *wnp* that allow its update via assignment. It becomes either part of a witness if it is an incompatible node pair, or it is included in the checked sequence otherwise. Therefore, *wnp* is neither pending as a member of *pd*, nor checked as a member of *ck*. This is important in order to keep the injectivity property of *pd* and *ck*, whilst finding successor pairs or new witnesses later on.

$$\neg\, wnp \in \text{ran}\ pd \wedge \neg\, wnp \in \text{ran}\ ck \wedge$$
$$WitnessInv[w := (ck \frown \langle wnp \rangle, lvl \frown \langle wl \rangle), sN := wsN, iN := wiN]$$

Finally, the working node pairs must satisfy the witness invariant with the appropriate instantiations because it might become part of a new witness if incompatible. The proposed new witness will be formed by the concatenation of the checked sequence with *wnp*, and the concatenation of the level sequence with the working level, provided *wnp* is incompatible. The guard on the first alternative statement for compatibility check ensures the condition for a witness having a generic violation has happened.

After introducing the properties of *wnp* and *pd*, it is now possible to introduce the assignment to these variables. The proof obligation for this assignment to *wnp*, *pd*,

and wl is rather complex and divided in three parts: (i) the state invariant, (ii) the number of witnesses requested, and (iii) the update conditions for wnp given above. The state invariant is difficult to discharge, but possible due to a series of properties between injective sequences, and functions $head$ and $tail$. The second part is easy because it is already present on the precondition as the main loop guard. The update conditions of wnp are also a difficult and lengthy proof. Once more, properties between injective sequences and the involved functions are used to finish this subgoal. After the assignment, the value of wnp does not change until the next iteration of the search.

Compatibility check

The node pair is compatible depending on whether the violation condition holds for that node pair or not. This leads to a complementary (and deterministic) alternation with two branches: one for incompatible pairs to generate witnesses, and another for compatible pairs to search for new successors. The compatibility check predicate used as the alternation guard is given by the schema $GenVl$ with appropriate substitutions for the working node pair elements.

If the working node pair is not compatible, then it must form a new witness to be included in $swts$. This assignment finishes the incompatible branch of the alternative for incompatible node pairs.

$$swts := swts \cup (ck \frown \langle wnp \rangle, lvl \frown \langle wl \rangle)$$

The proof obligation it introduces is divided in three parts. Firstly, the assignment must keep the state invariant related to $swts$. This is possible because ck and pd members are disjoint, and wnp is not a member of both sequences. Secondly, the individual witness update must be preserved. For this, it is enough to show that the new witness candidate preserves the individual witness update. This is achieved because wnp is neither a compatible pair, nor a member of pd. Finally, the part of the main loop variant related to $swts$ ($wr - \# swts$) can now be discharged. This is possible with lengthy predicate calculus manipulations, set theory properties, and because the assignment to wnp establishes the property that the witness candidate with wnp does satisfy the witness invariant.

Otherwise, if the working node pair is compatible, then it must become a checked pair and further successors must be searched. Prior to this, we need to update the checked sequence with the working node pair already checked as compatible via concatenation. A similar concatenation must happen for lvl with wl in order to the invariant that requires that the size of these sequences are equal.

$$ck, lvl := ck \frown \langle wnp \rangle, lvl \frown \langle wl \rangle$$

To discharge the proof obligation for this assignment, it is necessary to first establish the properties relating these variables. The properties of wnp must change since it will be part of ck and still not part of pd.

$$wnp \in \text{ran } ck \land \neg wnp \in \text{ran } pd \land \\ (\forall w : Witness \mid w \in swts \bullet \neg wnp \in \text{ran } w.1)$$

We also remove the information relating wnp and the witness invariant, since witnesses are no longer under concern inside this part of the alternative. Since the working node pair was pending and could not have been already checked, it must not belong to any

witness previously found. The next predicate enforces that ck' and lvl' are no longer empty, and their values are explicitly given.

$$\neg ck' = \langle\rangle \land \neg lvl' = \langle\rangle \land ck' = ck \frown \langle wnp \rangle \land lvl' = lvl \frown \langle wl \rangle$$

These predicates are necessary in order to discharge the remaining part of the main loop variant related to ck $(PS\,(nf, ip))$.

The proof obligation for the assignment to ck and lvl is divided in five parts: (i) the maintenance of the state invariant; (ii) the new update properties for the working node pair; (iii) the alternative compatibility guard; (iv) the update predicates relating ck and lvl; and (v) the loop variant related to ck. The second, fourth, and fifth cases are lengthy but not illuminating. They are discharged by appropriate use of toolkit theorems for sets and sequences together with schema calculus. The third case is trivial because it is already present in the assumptions. The complex aspect of this proof is the first case related to the maintenance of the state invariant. It needs auxiliary lemmas guaranteeing that the assignment keeps ck injective, and that members of ck and pd are still disjoint. This is possible because originally ck and pd members are disjoint, and wnp was neither a member of pd nor of ck. Moreover a series of toolkit theorems for sets, sequences, and other manipulations are also necessary. The complete proof is given in [Fre04d].

Finding successor pairs

At this point the requirement related to the main loop variant has been completely discharged, and the state invariant is simplified with respect to predicates mentioning ck and lvl. This simplification removes from the original state invariant the predicate about the number of witnesses $(\#swts \leq wr)$, the predicate about the sizes of both sequences ck and lvl to form new witnesses $(\#ck = \#lvl)$, and the invariant on the members of both sequences ck and lvl. The final task is to look for successor pairs from the compatible working node pair. The new node pairs should be included as pending, provided they have not already been checked.

To find successor pairs of a compatible node, we need to look for all immediately enabled arcs with visible arcs of the implementation, and check where they lead to in the normal form. This is achieved with nested loops. For silent (or internal) communications, the implementation can progress freely and the normal form remains stationary. That is because nondeterministic choices on the normal form have already been resolved through subset construction. Firstly, for all the immediately enabled arcs of the implementation, we reach a possible node on the normal form and a set of possible nodes on the implementation. Secondly, from these implementation nodes $(iN \in iNS)$, we need to check whether all their possible combinations with the fixed specification node (sN) have already been checked or are still pending. Finally, nodes that are neither in pd nor in ck are included as pending for the next iteration of the search algorithm.

Enabled arcs Let us now detail the code related to enabled arcs. We introduce and initialise via assignment a new variable representing a finite set of arcs from the implementation that are immediately enabled.

$$\textbf{var}\ arcS : \mathbb{F}\ Arc \bullet arcS := enabled\,(ip.ts, wiN)$$

This set of arcs represents possible arcs from the implementation being searched for successors. As each arc is being explored, this set shrinks by one. For this assign-

ment, we need to establish the properties of $arcS$ with respect to the implementation transition system with predicate

$$arcS \subseteq enabled\,(ip.ts, wiN) \land \\ (\forall\, a : Arc \mid a \in arcS \bullet \neg arcStep\,(ip.ts, wiN, a) = \varnothing\,)$$

That is, subset containment with respect to *enabled* ensures that exploring new arcs from $arcS$ preserves the amount of arcs remaining to be searched. As we are implementing a loop searching for individual arcs, we also need to establish the properties for each arc $a \in arcS$. Since each arc a is *enabled*, due to the well-formedness theorem of the transition system (see Theorem 3.2 on page 32), this *enabled* arc must step somewhere in the implementation automaton. This forms part of the invariant of the second loop exploring the immediately available arcs at the implementation automaton; it is labelled L_1 in Figure 4.4.

The invariant of L_1 is given in four parts: (i) the guard from the if statement ensuring the working node pair is compatible; (ii) the simplified state invariant; (iii) the new properties of the working node pair after the update of ck; and (iv) the properties of $arcS$ just mentioned. Since we use the cardinality of $arcS$ as the variant, $arcS$ must be finite. The loop guard is given as $(arcS \neq \varnothing)$, and the variant is based on the size of $arcS$.

The next two assignments respectively chose and remove an arc from $arcS$. The proof obligations generated by their introduction are straightforward and not commented here. Set elements are chosen based on the generic definition of *elem* below, also used for the operational semantics.

$$[X]$$
$$elem : \mathbb{P}\ X \to X$$
$$\forall\, S : \mathbb{P}\ X \bullet \exists\, x : X \mid x \in S \bullet elem\ S = x$$

$$arc := elem\,(arcS)\,;\, arcS := arcS \setminus \{\, arc \,\}$$

Since we are shrinking the size of $arcS$ by one at each iteration of L_1 via these assignments, the proof obligation related to the loop variant can be discharged from the postcondition. For the proof obligation raised by the introduction of the first assignment, the additional predicate $(arc \in arcS)$ is necessary in order to establish the property that the chosen arc is indeed an element from the set $arcS$.

Set element selection and removal play a key role in the differentiation between refinement check and refinement test. It is related to the mutually reachable paths mentioned in witness invariant (see Section 4.6.2 on page 104). For refinement checking, these two operations are just normal element selection and removal as defined by the *elem* function and set difference operator. On the other hand, for refinement testing, appropriate decision procedures or user intervention can be used in order to implement the desired behaviour. For example, the option of user intervention during refinement test for *Circus* would be similar to the options given by ProBE [For00] for CSP_M. This is a topic for future research.

Mutually reachable node pairs Once an arc of the implementation has been chosen, we need to calculate the nodes it leads to in both automata. This is achieved via the result of the *arcStep* function through the selected arc from the elements of the working node pair. The normal form is deterministic and free from internal transitions. Since internal progress on the implementation does not need to match

progress on the normal form, silent transitions from the implementation must progress freely while the working node of the normal form remains the same.

$$\left(\begin{array}{l} \textbf{if} \quad (arc \neq \varnothing) \rightarrow \\ \qquad sN := arcStep\,(nf.ts, wsN, arc) \\ \quad [\!] \; (arc = \varnothing) \rightarrow \\ \qquad sN := wsN \\ \textbf{fi} \end{array}\right)$$

Firstly, we check in the normal form where the chosen arc leads to, provided it is not a silent transition. Otherwise, if the chosen arc is empty, the normal form node remains the same. To record this value, we declared the variable $sN \in DNode$. The properties of this variable are defined depending on whether the arc represents a silent transition or not. Since the working node is compatible, if the chosen arc contains visible communication, there must exist at least one arc in the normal form through the given arc, hence sN must be valid, as defined by the invariant from schema $SeqRSStateBasic$. This alternative on arcs is introduced using a *if* statement with complementary guards on the chosen arc being empty or not. If it is empty, the value of sN is wsN. Otherwise, the value of sN is the node returned through the application of the $arcStep$ function from the current working node on the normal form through the chosen arc. The proof obligation generated by the introduction of the assignments are straightforward. Finally, we need to select the nodes of the implementation where the given arc leads to through the working implementation node wiN. As the implementation is nondeterministic, there are possibly more than one reachable node and this leads to the last loop of the algorithm (labelled L_2). For this we declare a new variable as the set of nodes from the implementation $iNS \in \mathbb{F}\ Node$. The properties of this set are established by the next predicate

$$iNS \neq \varnothing \land iNS \subseteq arcStep\,(ip.ts, wiN, arc)$$

The elements of iNS come from the possible arcs of the implementation reached from wiN through arc via the $arcStep$ function. Because the working node is compatible, there must exist at least one arc in the implementation through the given arc, hence iNS cannot be empty. This allows us to initialise iNS with the following assignment.

$$iNS := arcStep\,(ip.ts, wiN, arc)$$

Its introduction generates a straightforward proof obligation.

New successor pairs The final part of the algorithm is the inclusion of new successor pairs as pending, provided they have not been checked yet. Since we already have a normal form node sN, and a set of reachable arcs iNS from the implementation, we need to iterate over iNS to form (possibly) new node pairs (sN, iN), where $iN \in iNS$.

Firstly, we set up the last loop L_2 over all elements of iNS. It has a structure similar to the two previous loops. That is, we remove elements from iNS one by one and check whether the node pair formed by sN and the element chosen from iNS is new or not. *The invariant of L_2 is the invariant of L_1 together with the property about iNS given below*

$$iNS \subseteq arcStep\,(ip.ts, wiN, arc)$$

The guard is $iNS \neq \varnothing$, and the variant is the size of iNS. Secondly, similar to the derivation for $arcS$, we need to retrieve and remove an element from iNS. It is

assigned to a new variable $iN \in Node$ representing the chosen implementation node that is removed from iNS.

$$iN := elem\,(iNS)\,;\, iNS := iNS \setminus \{\, iN \,\}$$

Since we are shrinking iNS by one at each iteration, it is not difficult to discharge the proof obligation related to the loop variant of L_2. The property about iN is similar to the previous loop (L_1) with $arcS$. Finally, we introduce the alternation checking whether the node pair found is new or old. If it is neither pending nor checked, then it must be new. Therefore, the guard for new pairs is given as

$$(sN, iN) \notin \mathrm{ran}\ pd \cup \mathrm{ran}\ ck$$

In this case, we need to include (sN, iN) at the end of pd via the assignment

$$pd := pd \frown \langle (sN, iN) \rangle$$

On the other hand, if the node pair found is either pending, or checked, then it must be old. In this case, nothing is needed and we simply Skip. This forms the complementary guards for new and old node pairs being searched.

The proof obligation generated by the introduction of Skip in the old pair case is straightforward because the postcondition is present in the precondition; however, the proof obligation generated by the new pair is lengthy and complex. We chose to explain this proof in deeper detail due to its importance for the correctness of the algorithm with respect to the search being exhaustive. This proof obligation is generated due to the introduction of the assignment to the pending sequence including the new node pair. The complexity is broken into five subgoals. The first subgoal is related to the type of pd remaining injective after the assignment.

$$pd \frown \langle (sN, iN) \rangle \in \mathrm{iseq}\ PA\,(nf, ip)$$

Since (sN, iN) comes from the $arcStep$ function, it belongs to the product automata $PA\,(nf, ip)$. Finally, because the alternative guard insists that (sN, iN) does not belong to the range of pd, to discharge this subgoal, we just need to prove that concatenation of a new element to injective sequences keeps the sequence injective. For this, we have an additional toolkit theorem stating that element concatenation on injective sequences gives an injective sequence, provided the element is not in the range of the sequence.

$$\forall\, A : \mathbb{P}\ X \bullet \forall\, s : \mathrm{iseq}\, A;\ x : A \mid \neg\, x \in \mathrm{ran}\ s \bullet s \frown \langle x \rangle \in \mathrm{iseq}\ A$$

The second subgoal is related to the property of wnp not being in the range of pd.

$$\neg wnp \in \mathrm{ran}\,(pd \frown \langle (sN, iN) \rangle)$$

From the Z toolkit we know that ran distributes over sequence concatenation. Thus, the subgoal can be rewritten as two smaller goals, after some predicate calculus and further simplifications on a unit sequence.

$$(\neg wnp \in \mathrm{ran}\ pd) \land (\neg wnp = (sN, iN))$$

The first conjunct is easily discharged because it is already available in the antecedent. The second part is easily discharged because the alternation guard, which is included in the antecedent of our proof obligation, mentions that $(\neg(sN, iN) \in \mathrm{ran}\ pd)$. The

remaining subgoals are related to properties on the state invariant. The third subgoal to prove is that the extension of pd still guarantees that it is disjoint from ck.

$$SeqRSStateInvRelCkPd[pd := pd \frown \langle (sN, iN) \rangle] \Rightarrow \\ \operatorname{ran}(pd \frown \langle (sN, iN) \rangle) \cap \operatorname{ran}\ ck = \{\}$$

This proof is lengthy but easy, because the alternation guard already ensures that (sN, iN) does not belong to both sequences. The forth subgoal to prove is that the concatenation on pd ensures that it must not belong to any witness previously found.

$$SeqRSStateInvRelSwtsPd[pd := pd \frown \langle (sN, iN) \rangle] \Rightarrow \\ \forall\, w : Witness \mid w \in swts \bullet \operatorname{ran}(pd \frown \langle (sN, iN) \rangle) \cap \operatorname{ran}\ w.1 = \{\}$$

This proof is again lengthy. It is discharged by the fact that (sN, iN) is both valid and new. That is, it belongs to the product automaton, and is neither in pd nor in ck. Further Z toolkit theorems involving sequence concatenation, range, and intersection distribution are also needed. Finally, the fifth subgoal is about the state invariant related to the members of pd being valid.

$$\forall\, sNpd : DNode;\ iNpd : Node \mid (sNpd, iNpd) \in \operatorname{ran}(pd \frown \langle (sN, iN) \rangle) \bullet \\ NodePairInv[sNpd/sN, iNpd/iN]$$

It is proved with the information in the antecedent that (sN, iN) comes from the *arcStep* function, and hence must be valid. This completes the entire proof and the explanation of the algorithm code from Figure 4.4.

4.8.5 Sequential witness search actions

To complete the description of the sequential implementation algorithm, let us define the remaining actions. It is straightforward to prove that they are related by simulation to the corresponding abstract action; the proofs are similar to that concerning *SeqRefCheckParams*. The *SeqReport* action is similar to the abstract version, but guarded commands are used instead of schemas.

$$SeqReport \mathrel{\widehat=} \mathbf{var}\ rr : RefRep \bullet query!rqReport \rightarrow \\ \left(\begin{array}{ll} \mathbf{if} & (swts = \varnothing) \rightarrow\ rr := rrSuccess \\ [\!] & (\neg swts = \varnothing) \rightarrow\ rr := rrFailure \\ \mathbf{fi} & \end{array} \right) ;\ report!rr \rightarrow Skip$$

The remaining actions are just like the abstract specification with appropriate implementation versions of the state.

$$\begin{array}{lcl} SeqDebug & \mathrel{\widehat=} & query!rqDebug \rightarrow testimony!(swts) \rightarrow Skip \\ SeqQuery & \mathrel{\widehat=} & SeqReport \,\Box\, SeqDebug \\ SeqReset & \mathrel{\widehat=} & reset \rightarrow Skip \\ SeqRefCheck & \mathrel{\widehat=} & \mu\, X \bullet \left(\begin{array}{l} (sd = t)\ \&\ SeqQuery \,\Box \\ (sd = f)\ \&\ (\ SeqWitnesses\,;\, sd := t\ ) \end{array} \right) ;\ X \\ SeqStart & \mathrel{\widehat=} & \mu\, X \bullet SeqRefCheckParams\,; \\ & & \quad (\ SeqReset \,\Box\, SeqRefCheck\ )\,;\ X \end{array}$$

$\bullet$ *SeqStart*

end

The refinement proofs for these actions are straightforward and omitted here.

4.9 Summary

The main goal of this chapter is to present a refinement model checking strategy for *Circus* in a top-down fashion. Overall, our main goal is to provide an efficient solution for the problem of refinement model checking state-rich specifications in *Circus*. This goal is achieved by the PTS data structure together with a search strategy over it that produce human-readable debugging information. Integration with theorem proving to discharge generated proof obligations for state operations is considered. Even though theorem proving might become necessary, previous implementation experience [Fre02] has shown that automatic decision procedures can be implemented for infinite data types in CSP with particular restrictions on their complexity and decidability. Having sets of events on the arcs allows us to avoid the expansion of input prefixes to external choices, as well as represent symbolically state variables via loosely defined components. Besides dealing with the state explosion problem, this also allows the inclusion of infinite data types in channel communications.

We also explain how we have compromise expressiveness versus automation, in order to integrate model checking with different decision making techniques, mainly theorem proving. Furthermore, during the derivation of the algorithm, we pointed out the aspects to be considered while implementing refinement checking or testing.

The modularisation of violation conditions for witnesses for different refinement criteria, allows the algorithm to be generalised for additional criteria, such as one including specialised failures [BL04] or mobility [Tan05]. The calculated algorithm and related data structures are ready for further extensions whenever they become available. The additional effort is restricted to a few proof obligations involving the generic violation conditions for witness search. This supports the extensibility of *Circus* available through the expressive power of UTP.

The derived sequential algorithm is similar to FDR's algorithm given in [MH00]. The details of the derivation for the *Circus* sequential algorithm are useful as it explains hidden aspects of the code, such as loop invariants. To the extent of our knowledge [Ros94b, RGG$^+$95, CH93b, Gol01, MH00], similar refinement model checking algorithms [MH00, PY96b] have not been formally specified and derived before. As the algorithm is derived directly from the abstract *Circus* specification using refinement laws, its correctness is guaranteed by construction. The search is indeed exhaustive and the number of witness found is bound to the number requested, provided the witness invariant has been satisfied and some violation condition has occurred. This accounts for the soundness of our approach.

Finally, as far as we know, the derivation of the witness search algorithm is the largest case study of the refinement calculus for Z [Cav97]. We expect the mechanisation in ***Z/Eves*** of a Z toolkit extension, the theory of automata, the operational semantics, the model checker abstract *Circus* specification, and the refinement derivation and proof obligations to be used as the starting point for other formal tools for formal languages based on the notion of refinement (see Appendix A for additional material). In the long term, we aim at providing an integrated development environment for *Circus* that is capable of performing iterated integrated analysis [Sha96, Sha02, Rus00] by taking our own medicine and applying formal modelling and verification throughout the development process of our tools. It should contain features similar to the ones available in FDR. For instance, GUI, structured debugging, scripting, and beyond. This is also an effort in building a tool that performs automated formal analysis for a unified theory.

Chapter 5

A prototype model checker

"Logic is the legislator of reason."
Immanuel Kant [Kan65]

This chapter describes the *Circus* model checker prototype implemented in Java. The tool implements the operational semantics of Chapter 3, the witness search algorithm and other data structures of Chapter 4, and assemble all components together according to the architecture explained in Section 4.3 (see Figure 4.1 and Figure 4.2). We aim with this prototype to prove refinement for *Circus* programs allowing experiments with integration of refinement model checking and theorem proving. We also use *Concurrent Versioning System* (CVS) [BF04] for maintaining different versions, and controlling (possibly distributed) modifications.

In the next section we present a short tutorial introducing the commands of the textual user interface of the *Circus* model checker. Next, Section 5.2 explains the most interesting software engineering considerations involved in the design of the tool, as well as the techniques applied. Section 5.3 presents some details of the architecture implementation and how to extend it, for instance, with further refinement models or new theorem prover plugins. Finally, in Section 5.4, we give a summary and some final considerations.

5.1 Using the model checker

Our aim is to provide a model checker tool for *Circus* that can be used similarly to FDR. Nevertheless, due to the possible complexity introduced through Z, some user interaction might be required in the form of interactive theorem proving. The main goal is to have a sound tool that exhaustively checks for refinement with the highest levels of automation possible, and which gives an accessible (human-readable) counter-example in the case of failure. In practice, the way one is expected to use the tool is depicted by the cyclic process presented in Figure 5.1. It is an instantiation of iterated integrated analysis [Sha96, Sha02, Rus00].

Firstly, a *Circus* specification in LaTeX[1] is given to the compiler, which produces a predicate transition system on-the-fly as the checking progresses through the witness search algorithm. It is done by fitting together the *Circus* parser [BC02], the typechecker [Xav06], and the operational semantics of Chapter 3. On-the-fly model

[1] As the model checker is based on the CZT AST, extensions for Unicode are available, and e-mail markup format is planned [MU05].

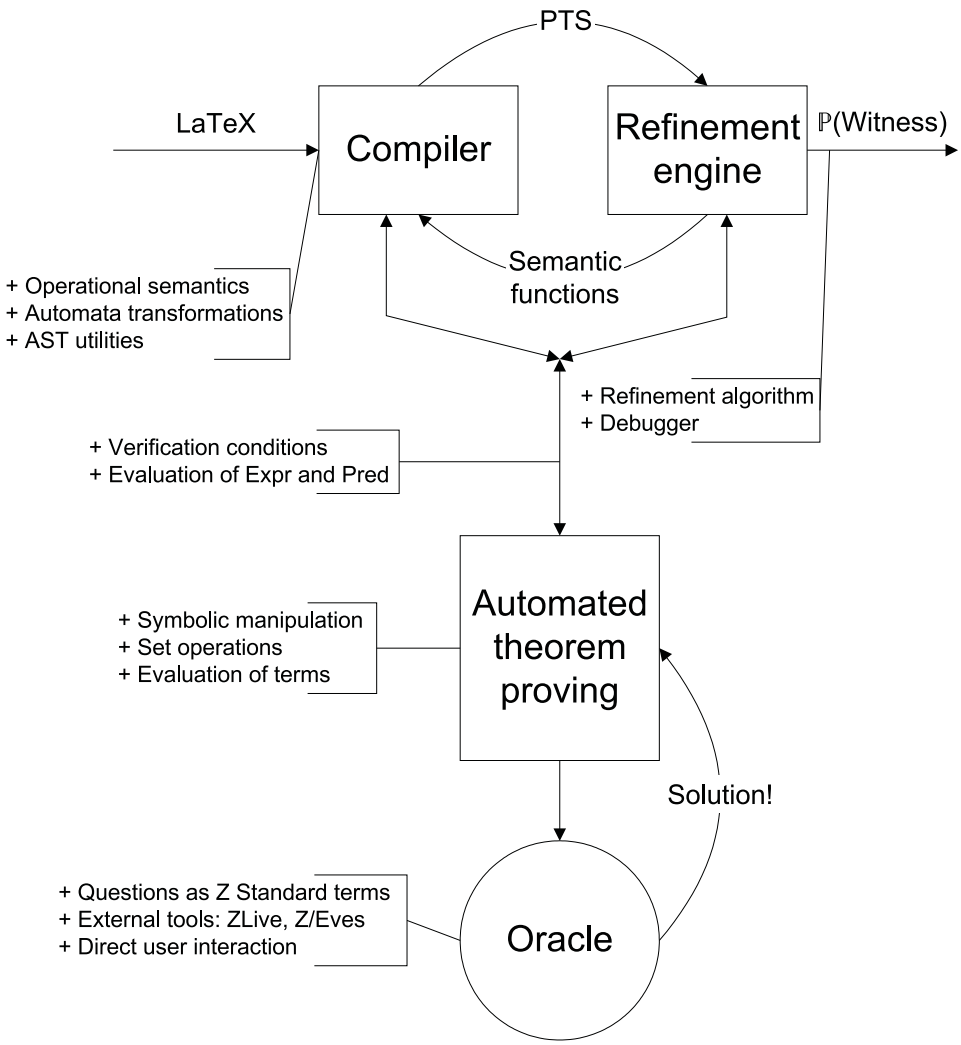

Figure 5.1: Model checker prototype usage

checking is a known technique [Pel94], here adopted to our scenario. The direct benefit is that, since model checking often finds problems early in the design, we do not need to calculate the whole transition system up front. Due to the state explosion problem that often arises, keeping track of the entire transition system is not efficient.

Secondly, right after the compilation process, the specification is normalised and the implementation nodes are checked for divergence. During these transformation algorithms, automated theorem proving might be necessary mainly because of the verification conditions of the operational semantics, and the normalisation task that makes the specification side of the refinement relation deterministic.

Thirdly, the refinement search is carried out by the refinement engine, which is a direct implementation of the code presented in Figure 4.4. It receives the partially compiled transition system and performs the main activities involved in witness search: (i) the validation of the refinement search criteria; (ii) the identification of new successor nodes to check; and (iii) generation of counter-examples if necessary. Refinement criteria validation demands evaluation of predicates and expressions, which in turn might require automated theorem proving. While searching for new nodes, the refinement algorithm feeds back to the compiler for further compilation through calls to the semantic functions, now for different programs represented as the available nodes after the refinement criteria check. If a counter-example is found during the search, the debugger comes into play by properly casting the refinement search information into a human-readable format presented as textual output.

Finally, the oracle component of the automated theorem proving part is responsible for discharging proofs that the previous stages could not handle automatically. They are passed to general purpose theorem provers or the user. At the moment, only a simple user interface is provided. As the AST is in conformance with the Z Standard, most theorem provers that support Z can be used.

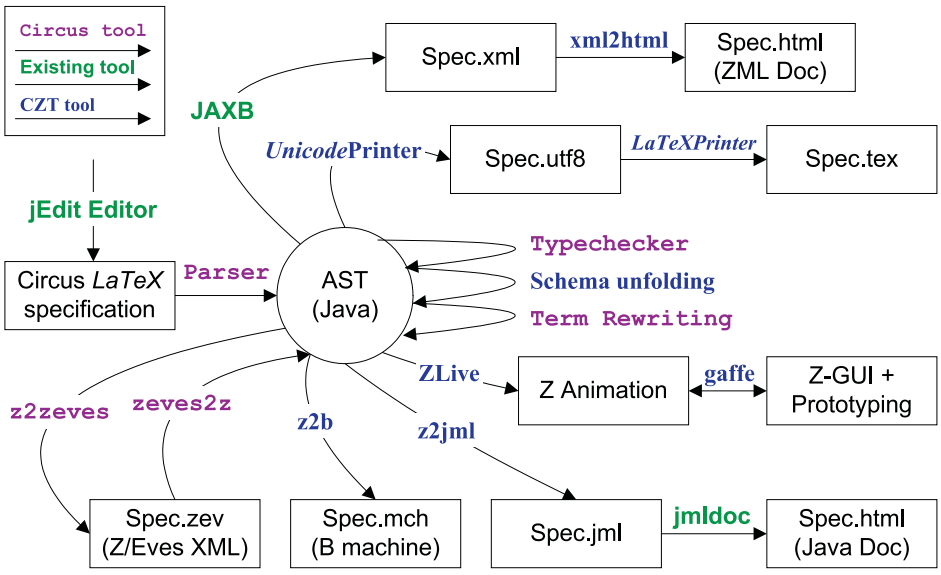

Figure 5.2: Available outputs for CZT AST

One could use available CZT tools we have augmented for *Circus* (see Figure 5.2) in order improve the usability of our prototype. Firstly, we have extended the CZT AST to handle *Circus* (see [Fre05c]). This allowed us to integrate with available tools for

this framework. From there, the original *Circus* parser [BC02] reviewed for integration then generates the CZT AST representing the *Circus* program. The extended CZT AST can them be translated to a series of output formats, such as ZML (Z XML markup for Z Standard extended to Circus), HTML, Unicode, or LaTeX. Tools such as a *Circus* typechecker [Xav06] and a simple term rewriting tool (as an integrated part of the theorem proving module), and the CZT schema unfolder can then be used for inclusion of additional annotations, and for performing semantics-preserving transformations. CZT also provides support for the *jEdit* text editor [JEd98] as a plugin, where one is able to type in Unicode or LaTeX specifications for Z; the *jEdit* extension for *Circus* is planned as future work. For Z specifications, it is possible to generate B machines [Sch02], JML annotations [LBR98], or ***Z/Eves*** server API commands in XML [SM05] from the AST as well.

With JML annotations, we have documented the Z aspects of the abstract *Circus* specification of our model checking strategy presented in Chapter 4 by translating the (Z part of the) LaTeX source into JML annotations added to the Java code of both the compiler and the refinement checker. (A usable tool to support the translation of Z specifications to JML is under development in CZT). With such annotations, one is capable to carry out extended static checking for partial code correctness or exception freedom using available JML tools [DLNS98, ECGN01, Hui01, BRL03]; such checks are planned as future work.

With the ***Z/Eves*** server API commands we implement as a CZT extension, one is able to drive the prover through a socket connection running anywhere in a network. We expect it to be of interest to other tool builders as well. This translation allows Java programs to interact with the ***Z/Eves*** prover directly, hence allowing automatic proof of simple theorems via specific proof commands. Other applications, such as schema unfolding or specialised proof tactics from basic ***Z/Eves*** proof commands, are also possible. In fact, this translation tool has been added as a theorem prover plugin for the *Circus* model checker theorem proving module. It did not only increase the levels of automation of the model checker, but also allowed the encoding of specific proof tactics not otherwise available as ***Z/Eves*** proof commands.

Since the operational semantics is described using Z, parts of it could be translated to JML as pre and postcondition annotations into the compiler code. The invariant and related predicates of the witness search algorithm are also embedded into the code in the same fashion. Although this does not guarantee code correctness, it does provide a stronger argument for the integrity between the specified architecture in *Circus* and its actual Java implementation. We aim with this effort to fill part of the gap between formal specification and its actual implementation.

5.1.1 Text UI

At the moment, we have a text based user interface, similar to a UNIX shell window, which allows one to use the tool through a set of specific commands. They are either: (i) top-level operations, (ii) execution options, or (iii) special (internal) operations mostly used for inspecting the internal behaviour of the tool. In total, there are around 19 commands in the current version (CVS Id 0.1.11). The most important top-level operations are the first three commands from Table 5.1. The `execute` command runs the tool in batch mode, where the given file must contain one command per line to be executed. These can be any of the commands listed using the `help` command. Comments are allowed by prefixing a line with the # character. The command `parse` parses and typechecks the given file, hence loading the necessary definitions for

Command	Arguments	Description
`execute`	filename	Executes commands in batch mode.
`parse`	filename	Parses and typechecks the given file.
`mc`	LHS RHS	Checks whether $LHS \sqsubseteq_{FD} RHS$ holds.
`show`	`all` | `chs` | `acts` | `vars`	Shows corresponding definitions.
`help`		Shows the complete list of commands.
`history`		Lists the history of executed commands.
`repeat`	i	Repeats i^{th} command from history.

Table 5.1: Top-level operations for the text UI

Boolean Flag	Description
`pretty`	Pretty-printer.
`rsinvcheck`	Run-time check for refinement search invariant.
`verbosewts`	Verbose information counter-examples.
`verboseerror`	Verbose error information.

Table 5.2: Additional operations and flags for the text UI

model checking. It is similar to the `Load` command in FDR's GUI menu. At this point one can issue the `show` command with the appropriate filter, where `all` is the default. It shows all the declared channels, actions, variables, and current statistics of execution time for each command. Finally, the `mc` command enables the user to model check two actions by giving their names. Furthermore, the tool also accepts some boolean flags, as well as other commands used to drive and inspect the model checker more closely. Some of the most useful are summarised in Table 5.2.

5.1.2 Known restrictions and usage details

The current parser still presents problems with some Z constructs, such as operator templates and infix functions. We extend the *Circus* typechecker to include the initial environment for the nodes of the operational semantics, as mentioned in Section 3.3.

As mentioned in Section 2.2.3, we suggest that the user specifies normalised schemas: those schemas where components are declared with their maximal types. This is particularly useful whenever evaluation of predicates and expressions from the operational semantics and witness search algorithm demand theorem proving.

We plan as future work to extend the parser to include syntax for stating refinement claims, hence allowing a mode of execution similar to one allowed by the FDR server. Further extensions, suggestions, and up-to-date information will be available on the tool web-site at `www.cs.york.ac.uk/circus/model-check`.

5.2 Software engineering considerations

In this section, we briefly present the most important software engineering issues considered during the model checker development process.

We have chosen Java as the target programming language for a series of reasons: available refinement calculus that allows program derivation from formal specification [Cor04]; widespread availability of tools related to our task; platform and operating system independence; active research and development worldwide [JCP98]; and

so on. In terms of efficiency, however, Java is not an obvious choice, but for a prototype tool, this is not a main issue.

5.2.1 Community Z tools

The Community Z Tools (CZT) project is an open-source Java framework for building formal methods tools for Z Standard and Z extensions [MU05]. It is an integrated framework being used to implement several formal methods, which adds object-orientation [Smi00], real-time features and process algebra extensions to Z [MD00], and now we have extended it to support *Circus* [Fre05c]. The major motivation for using the CZT is the fact that it has strong conformance with the Z Standard.

The use of this framework greatly reduced the effort required to implement *Circus* as a new Z extension. For instance, this framework provides the basic tools expected in a Z environment, such as conversion between LaTeX, Unicode, and XML formats for Z, as well as parsing, unparsing (*e.g.*, pretty printing), typechecking and animation tools, with a WYSIWYG Z editing environment integrated within the *jEdit* [JEd98] text editor. There are also more experimental tools under development, such as Z-to-B and Z-to-JML translators, a Z animator, and a GUI-builder for Z specifications.

In order to use these facilities, first of all we extend CZT with the *Circus* syntax. CZT has a common *Abstract Syntax Tree* (AST) for Z Standard that is to be used by every extension, and has been described as an XML markup for Z specifications (ZML) [UTS+03]. This is an interchange format that can be used to exchange parsed Z specifications between sessions and tools written in different languages. Our first task was to create an XML schema file with additional productions for *Circus*.

Next, we used the CZT GnAST tool to generate the Java AST interfaces and their implementation classes automatically from the *Circus* XML file. The generated code looks similar to the code produced by Java data binding tools, such as JAXB [JAX]. The main purpose of GnAST is to generate well-designed AST classes. For example, the AST classes generated by GnAST support an extensible variant of the visitor design pattern [MdC01, Mar97, Nor97] that is fundamental for the implementation of any operation that traverses the AST, such as typechecking, compilation, translation, and so forth. To define a new operation, all one needs is to implement a new visitor.

The automatic AST generation from XML files dramatically reduces the time required to develop a new Z extension, ensures a common style, and improves maintainability. For instance, the complete AST folder representing Z Standard contains around 420 Java files. In total, GnAST generates 773 Java files representing Z Standard and *Circus* from a single XML file. This provides a very convenient and consistent way to obtain AST interfaces and classes.

Finally, the current parser for *Circus* [BC02] has already been modified to use the CZT AST, and the *Circus* typechecker [Xav06] is following the same direction as an extension of the CZT Z typechecker. More details about CZT and its extensions can be found in [MFMU05].

5.2.2 Design patterns

The prototype implementation is strongly based on design patterns [GHJV95, Ris00] and framework modelling [Rie00]. This enabled us to have desirable software engineering benefits [Mey97], such as modularisation, encapsulation, reuse, extensibility, clarity, maintainability, and so on.

We have made extensive use of patterns, such as *Strategy*, *Decorator*, *Visitor*,

Iterator [GHJV95], and so on, in order to allow a pluggable architecture where one can adapt the current implementation for new user interfaces, new algorithms, or external tools, in a very pragmatic fashion. This is particularly important for the link with external theorem provers, while trying to evaluate complex predicates and expressions that appear in the components of the model checker architecture.

5.2.3 Implementation integrity

As we have used formal specification and verification throughout the software development process, we would like to transfer our results to the level of code, as much as possible. This establishes a strong link between the findings at the level of the model and the actual code. The benefits of such effort is to extend precision of the formal model into the executing code.

Nevertheless, due to the lack of tool support to refine the formal specification down to code, our code has been only partially verified. Even so, the effort has paid off as some implementation bugs were discovered (and indeed avoided) due to the strong link we created. The approach that we adopt is a precise documentation strategy. It can strengthen the argument for partial code correctness via extended static checking through the JML annotations already mentioned.

We use three different approaches in this direction, each serving a particular purpose. The choice is made mainly due to tool support availability, as well as level of active research.

Java assertions

The new version of Java (1.5) includes a built-in assertion facility that enables one to encode preconditions as assertions checked at run time by the virtual machine. Once the code has become stable, one can switch off this assertions, hence incurring no performance penalty.

We include Java assertions for method and constructor parameters in order to ensure consistency, according to the expected invariants defined in the formal model. Another use of assertions is for ensuring the type correctness of newly created ASTs. For example, in the automated theorem proving module, ASTs that represent Z expressions can be created, and in that case they must be in conformance with the original expression type.

JUnit test cases

JUnit is a Java testing framework enabling test automation by providing support for individual tests cases, as well as test suites, in an extensible fashion through class and interface inheritance [JUn05]. In practice, one needs to write a test case for each relevant class, where at least each public method of the class being tested ought to be included in the test case.

Previous experience [Fre02] has been reused here to enable the test of properties for part of the Z toolkit encoding in Java, set theory properties, and so on. They are used in the simple term rewriting part of the automated theorem proving component of the model checker. We also create test cases to represent Z conjectures from our formal model. Although a checked test does not guarantee a theorem, it does give stronger confidence on the running code.

JML annotations

The *Java Modelling Language* (JML) [BCC+03, LBR98, JP01, LPC+04] is a *Behavioural Interface Specification Language* (BISL) tailored to Java [Win87]. It enables one to augment Java code using special Java comment annotations for method pre and postconditions, class invariants, exceptional behaviour specifications, loop invariants, and so on. They are written as an extension of the Java syntax for expressions that also includes quantifiers, in a style similar to specification statements of the refinement calculus [Mor94].

The aim of using JML is to allow at least a precise documentation of the intended behaviour of a method or the invariant of a class. A strong aspect of JML is its freely available tool support for extended static analysis [DLNS98], loop and class invariant identification [Hui01, ECGN01], run time assertion checking, HTML documentation generation, and so forth. For instance, whenever an annotation about an invariant or precondition is breached, warnings are given exposing problems in the code; however, some of the JML tools used [DLNS98] are known to be neither sound nor complete. That means that they could both miss an error, or warn about nonexistent errors. This is a compromise deliberately taken by the JML tool designers for the sake of higher levels of automation.

For the simple purpose of code documentation, JML is used in our implementation. For instance, some of the invariants from the operational semantics, the *Circus* abstract specification of the architecture components, and the refinement search algorithm are included as JML annotations. As stated in the manual [LPC+04], the aim of JML is not to provide partial code correctness or exception freedom, but to be an aid for more robust and precise software development. In this sense, it has been very useful.

5.3 Implementation architecture

The *Circus* model checker architecture has been previously explained in Section 4.3 (see Figure 4.1 and Figure 4.2). In this section we present some implementation details of its architecture in terms of the design patterns used, solutions provided, and support for extensibility.

5.3.1 *Circus* compiler

The compiler implements the operational semantics given in Chapter 3 through a series of CZT visitor pattern implementations: one for each particular aspect of the semantics with respect to the parsed and typechecked AST, as shown in Figure 5.3.

In total there are five visitors implementing different aspects of the semantics: (i) loading of an initial environment containing declared channels, variables, and actions; (ii) calculation of information about divergence; (iii) and (iv) as the *enabled* and *arcStep* semantic functions, respectively; and (v) effects of state update operations over the user state.

Visitors

The first visitor is used once, right after typechecking, in order to load the initial environment throughout the whole AST structure. This is used at later compilation stages, in order to create the environments for the transition system nodes.

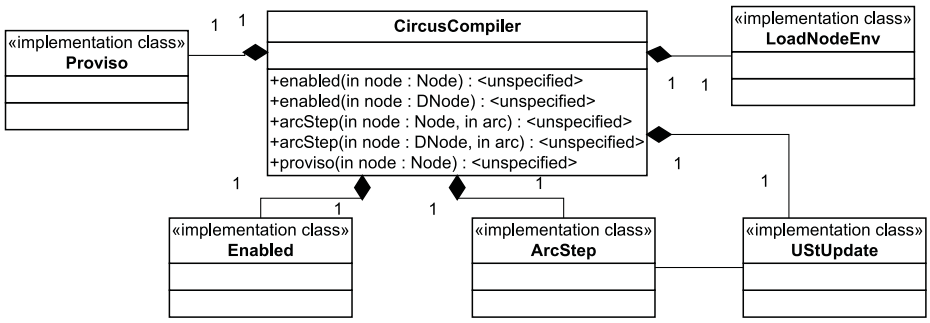

Figure 5.3: *Circus* compiler simplified class diagram

The next visitor calculates extra information about divergence whenever new nodes are created. These values are given as predicate annotations on the nodes that help the divergence check search performed at a later stages of the refinement search. For instance, schema expressions can be marked as divergent if their precondition is ***false***.

Finally, the last three visitors implement the semantic functions, as well as the user state updates during the on-the-fly compilation of each AST program. In this way, each rule defined for the two semantic functions *enabled* and *arcStep* are directly (and separately) implemented as a visiting method for each program construct represented by the AST. These are the core of the operational semantics implementation, and some additional detail is given below. For a more comprehensive understanding of implementation issues, one could consult the prototype code documentation that is given using JavaDOC following the style guidelines advised for framework construction [Rie00].

Visiting protocol

The CZT visitor design pattern controls the flow of evaluation of ASTs, and each visiting method should provide a homogenous result with respect to each AST program having a visiting method [MU05]. The `Enabled` visitor returns for each AST program a Z expression representing the set of arcs immediately available for communication. Arcs themselves are calculated at the communication part of each prefixing action, and are given as a Z expression representing the set of events from Σ they characterise. Events are tuples containing a channel name and an expression for the value being communicated, which can in turn be another tuple of expressions. For instance, the set of enabled arcs for an action such as

$$(c?x : P \rightarrow A) \Box (d!a \rightarrow B)$$

is given as

$$\{ \{ v : typeOf\,(\theta\, Env, c) \mid v = x \land P \bullet (c, v) \}, \{ (d, a) \} \}$$

Note that the values x and a are left unconstrained as a reference expression. That means the interpretation of such expression is to be all possible values acceptable by the type of the corresponding channel. Apart from parallelism and hiding, the rules for *enabled* are fairly simple. Thus, the resulting Z set expression is preferably given by extension rather than comprehension, if possible. This is useful for increasing the levels of automation, as enumerable sets can often be manipulated automatically.

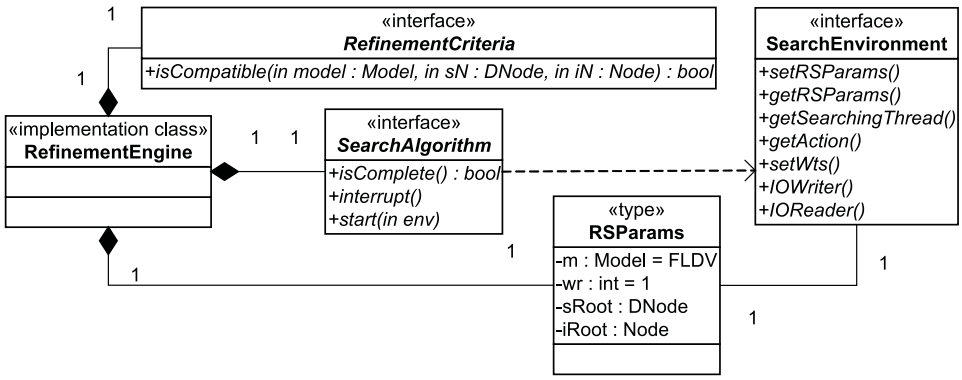

Figure 5.4: Refinement checker simplified class diagram

Although most operations performed by the compiler can be automatically discharged via trivial term rewriting, the simpler the set expression of arcs, the lower the amount of theorem proving that will be involved in the refinement checks, as well as the generation of the transition system during compilation.

Correspondingly, the `ArcStep` visitor returns for each AST program a set of *Node*s representing each possible target reached through a particularly chosen arc. This information is later used by the compiler to assemble the actual predicate transition system (PTS) currently loaded. Moreover, while executing the step function, possible user state updates are performed by the `UStUpdate` visitor accordingly. It returns for each state update case, a boolean value representing the success or failure of the underlying data structures modifications. These state update operations are more likely to demand theorem proving.

Normal form

The normal form is calculated right after compilation through the results of the operational semantics functions for each original node $n \in Node$ with a given deterministic node $dn \in DNode$. While retrieving the immediate available events for a normal form node dn, we calculate the minimal acceptances sets. The available transitions of dn are given as the special subset construction for PTS that takes *regions* of arcs into account, in order to make the normal form deterministic (see the definition of *regions* in Section 4.5.3). It also calculates information about divergent nodes via the information gathered through the `Proviso` visitor, as well as chasing silent transition loops using DFS. Due to the nature of expressions in arcs, the calculation of *regions* and minimal acceptances sets might raise the necessity of theorem proving.

5.3.2 Refinement checker

The refinement checker is built from three main components defined as Java interfaces, where each implements the *Strategy* pattern, as shown in Figure 5.4. The implementations of the `RefinementCriteria` interface encode the violation conditions as defined in Section 4.6.1. The implementation of interfaces `SearchAlgorithm` and `SearchEnvironment` encodes the *SeqWitnesses* and *SeqRefCheck* actions, respectively, as detailed in Section 4.8. The class `RSParams` directly implements the *RSParams* schema from Section 4.6.3.

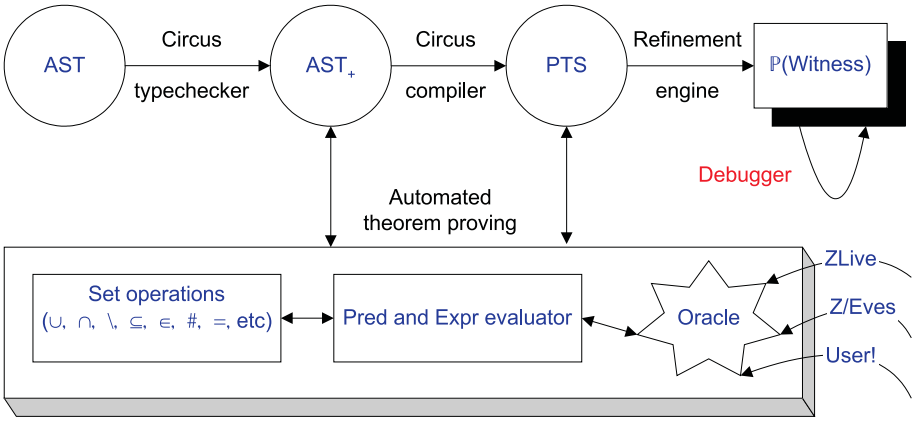

Figure 5.5: Refinement checker components

This architecture enables different implementations of refinement criteria, search algorithm, and search environment respectively. It works much like a plugin based architecture that can include different parts, even at run-time if necessary. The direct benefit is that other tool builders could extend the *Circus* model checker with minimum effort, and without interfering with the other components of the architecture, hence gaining in modularity with less risk of introducing inconsistencies.

To include new refinement models, one just needs to add new violation conditions by implementing the `RefinementCriteria` interface accordingly. For example, the recent work that has been done elsewhere in extending the stable failures model for CSP to include particular restrictions to refusals sets, such as singleton or bounded refusals [BL04], could be included via this interface. This enables one to check refinement with respect to a particular failure only, instead of the whole universe of possible failures. With the pluggable architecture based on design patterns, such an extension would require a single Java class of about 50 lines of code. A parallel implementation of the refinement search algorithm needs only to implement the `SearchAlgorithm` interface, and possibly extend parts of the theorem proving facilities, depending on the dependencies among the implementing threads. Similarly, a different search environment, over a distributed network for instance, could be set up through the `SearchEnvironment` interface. Finally, new refinement search parameters can also be added to the `RSParams` class accordingly.

5.3.3 Theorem proving

This module is structured in three stages as shown in Figure 5.5. It is responsible for discharging proof obligations and verification conditions generated. These tasks are mainly evaluation of expressions and predicates, such as arithmetic or set operations. The architecture works in three levels of complexity regarding the way in which the proof obligations are handled: (i) trivial modifications done automatically through term rewriting rules based on the Z toolkit theorems or set theoretical laws, such as $(\varnothing \setminus X = \varnothing)$; (ii) automatic theorem proving through animation tools and satisfiability solvers, where decidable logic is required; and (iii) interactive theorem proving or direct user intervention.

The first stage provides a basic set operations interface that has two implementations: (i) one for (possibly infinite or unbounded finite) symbolic sets defined by comprehension, such as $\{ x : \mathbb{Z} \mid x \geq 1 \}$; and (ii) another for enumerable sets, such

as $\{ x, y, z \}$. The operations available in this interface are set union, intersection, difference, subset, membership, cardinality, equality, and a few others.

The next stage is used for automated evaluation of expressions and predicates. It works as a satisfiability solver or automatic theorem prover; however, whenever interaction is needed, the last stage is used. Interaction is usually needed mainly due to subset containment or set membership tests over complex expressions. At this point, other animation tools or automatic theorem provers could also be plugged in, either to evaluate predicates or expressions, or to discharge generated proof obligations and verification conditions.

Finally, whenever automatic proof is not possible, the set evaluator interface can delegate the request to an oracle that is responsible for discharging the claim. At the moment, the only plugin available that is connected to the prototype is a fairly simple textual interface that sends queries to the user, as shown in Figure 5.5. The actual implementation currently supports two oracle interfaces: CZT ZLive, and ***Z/Eves*** server API commands interface. CZT ZLive [MU05] is a Z animation tool originally available in Haskell [Utt05] that is being ported to Java. The ***Z/Eves*** server interface, together with the translator from CZT AST to ***Z/Eves*** XML, enables the model checker to request ***Z/Eves*** to prove conjectures and collect the results.

This approach, using different tools in a pipeline, handling different levels of complexity, from automatic term rewriting until interactive theorem proving, or indeed user intervention, has been broadly used in other tools, such as ESC/Java2 [DLNS98] for JML and SPADE for Spark-Ada [Bar03].

A subtle point for future research is the analysis of possible dependencies between proofs generated through the use of parallel search algorithms. Depending on how the parallel search is implemented, this may incur in modifications or restrictions in the way the other modules can ask for evaluation of terms. For instance, one can put the compatibility criteria check and the finding successor stages of the witness search algorithm of Section 4.8.4 in parallel feeding each other. In this situation, requests for evaluation of terms from the criterion check must have priority over the finding successor requests, in order to avoid finding successors of incompatible pairs.

5.3.4 Debugger

As with the automated theorem proving oracle, the debugger is another opportunity for user interactivity. It allows the user to inspect counter-examples discovered during the refinement search with extended information about traces, failures (as acceptances sets), and divergence.

The interesting aspect of the debugger is its ability to browse through the execution path, hence enabling the exploration of the behaviour of the specification (or implementation) from the point of failure up to the root of the search. Once the point of inspection is chosen, the information available is cast into a human-readable form.

Although we already have the formal specification of the debugger, we are still experimenting with its implementation in the prototype. At the moment, only a fairly simple (non-interactive) textual output is given. We plan as future work to use a Java graph representation API, such as Graphviz [GN99], in a similar way as used by Spec# tools [BLS04, Spe05], in order to build up the failed path so that the user can usually track down the problem visually.

5.4 Summary

In this chapter we present a short introduction on how to use our model checker tool via its textual user interface. We also explain the most relevant software engineering considerations and techniques involved in the design. In particular, we present how the model checker integrates with the CZT project, and how proof engines can be used as plugins for the theorem prover module.

The architecture of the implementation is presented, where the possibility of extensions of the compiler and refinement checker are briefly explained. As the theorem proving module is independent, it could also be used separately by other tools, much like the ICS module [Gro05] is used as part of the PVS theorem prover [Sha02], for instance. An obvious extension plugin for the future is one which includes the UTP theories under development for *Circus* in ProofPowerZ [Oli06, OCW05].

In the next chapter, we conclude our work with a list of contributions, a summary overview of the thesis, as well as related and future work.

Chapter 6

Conclusions

"Where one cannot arrive flying, let one arrive limping"
Sigmund Freud [Fre55]

Currently, if formal verification of systems with complex state and behaviour is required, a considerable amount of time and effort is demanded for appropriate representation and analysis by computer [CW96]. This is mainly due to the state explosion problem that occurs in such scenarios [CGP00, RGG$^+$95], as well as due to the lack of adequate tool support for the task. In this context, the theory and tools behind our model checker for *Circus* aim at shedding light and providing guidance in bridging this gap between current demands, and actual tool support. As a tool, our model checker embraces and addresses the challenge of integrating refinement model checking with theorem proving support [Sha96, Sha02, CCo$^+$04] for a state-rich refinement language, such as *Circus* [WC01a, CSW02]. From the various (formal) verification techniques available, we intend to take the best out of the most successful, that is, model checking and theorem proving: high automation levels with as few compromises on expressiveness as possible. We expect the integration described in this thesis to set a roadmap, both in theory and practice, for tools aimed at formal verification of combined language aspects. So far, it has enabled formal verification through refinement of an expressible subset of *Circus*, hence enabling checks of both behavioural and data aspects of concurrent reactive systems that were not possible before.

Our aim is to push the state of the art of mechanisation by providing tool support for formal specification and verification of complex systems. In this direction, we advance the theories behind our tool, which are needed for enabling model checking of different language paradigms, such as behavioural aspects from CSP, and data aspects from Z and guarded commands present in *Circus*. Furthermore, we present how to finitely represent some kinds of infinite state systems through symbolic automata. Through the inclusion of symbols to represent state variables, and generation of some verification conditions, we enable the analysis of more complex systems by integration with theorem proving.

To faithfully represent *Circus* programs, we define an operational semantics that unfolds the process algebra into automata for analysis by computer. For that we also build a generic theory of automata based on [HMU01, CH93b], which is capable of representing different language aspects, including, but not limited to, data and behaviour, hence offering a wide range of possibilities. Although beyond the scope of this thesis, we believe this theory of automata might be expressible enough to enable inclusion of other language paradigms that are being added to *Circus*, such as mobility [Tan05], real-time [SJ02], or object orientation [CSW05, JLL02]. This is possible provided the feature under consideration is expressible through higher-order

logic or set theoretic models. As the semantics of *Circus* is based in UTP [HJ98], which in turn is based on alphabetised relational calculus, we expect one to be able to represent these additional paradigms in the future as well.

Our work is part of a bigger project that aims at providing theory and tool support for *Circus*. In this direction, a refinement calculus for *Circus*, as well as theorem proving support for UTP are being developed in [Oli06]. Another example is an implementation of a translation strategy from *Circus* to Java, which is also under development [Fre05a]. Moreover, the project has close cooperation with industry in both funding and information exchange [Qin04].

During the development process of the tool, we believe that formalisation plays an important role in increasing the integrity levels through a combination of techniques. In this direction, the model checker architecture itself is formalised through refinement. The abstract specification of each module of the refinement checker is presented as a *Circus* specification itself, and the refinement search algorithm based on [Gol01, Ros94b, RGG+95, MH00] is calculated from it using available refinement laws for *Circus* [Oli06, CSW02, Cav97]. The operational semantics and underlying automata theory is also formally defined, where Z is used as a metalanguage. Throughout the development process, ***Z/Eves*** [Saa99b, MS97] is used to discharged proof obligations from the algorithm derivation, animate the operational semantics, prove properties of the theory of automata, and so on. This effort enables not only the discovery of bugs at early stages of the development process, but also a higher degree of confidence in the achieved results. Moreover, to bridge the gap between formal specification and actual code, we also use JML [LBR98, LPC+04] annotations to document our findings at the level of the Java code. This is done mainly for the implementation of the operational semantics and refinement search algorithm. This exercise shows how one can go from an abstract formal specification to code mechanically, hence gathering the knowledge to step forward in on the roadmap for building formal verification tools formally.

The Java prototype of Chapter 5 implements the operational semantics of Chapter 3, the preparation stages of normalisation and divergence detection of Chapter 4, the calculated refinement search algorithm and the (simple) debugging facilities also presented in Chapter 4, and finally, a pluggable theorem proving component. The implementation also integrates with a series of external tools and frameworks (see Figure 5.2 in page 135). These included the integration with our own set of tools, such as a parser [BC02] and typechecker [Xav06], as well as external frameworks, such as the CZT project [MU05, MFMU05] for Z Standard compliance. We also implement three theorem prover plugins prototypes: (i) one using the ZLive[1] Z animator from CZT [UM03] for automatic animation of simple expressions and predicates; (ii) one using the ***Z/Eves*** theorem prover [Saa99b] for automatic theorem proving of more expressible terms through a programmable interface we have also developed; and (iii) one questioning the user directly for whenever the automated plugins are not enough. These plugins are separate tools, which are capable of interpreting the LaTeX output produced by the model checker, and then return results of possible transformations. For instance, suppose the state contains an element $x : \mathbb{Z}$, while checking for refinement, an expression such as

$$x \in \{ a : \mathbb{N} \mid a \bmod 2 = 0 \}$$

is given to the plugin, which in turn can either automatically return whether this predicate is ***true*** or ***false***, or ask the user for help in the case where the predicate is

[1] It is a Java prototype tool of another Z tool for Haskell [Utt05].

too complex for automatic evaluation. Other plugins are possible, such as one using PVS's [Sha] satisfiability solver [Gro05]. Although full-scale integration with a considerable case-study is yet to be done and outside the scope of this thesis, these plugins can be pipelined to increase the degree of automation. This trend is common to other formal verification tools, such as the Spark-Ada toolset (version 7.2) [Bar03], which combines different tools from automatic satisfiability solvers to automated theorem provers, or JML tools [DLNS98, BRL03], which combines automatic extended static checking with various plugins for different theorem provers. Throughout this development process we use JML [LBR98], Java assertions, and JUnit test cases [JUn05] trying to encode the findings from the formal specification and analysis into the actual code, as much as possible. In spite of the fact that it does not guarantee code correctness, it does increase our levels of confidence in the running code.

Finally, through these experiences, we produce some additional material available in the Appendix A as a CD-ROM. This material includes: (i) an advanced ***Z/Eves*** tutorial, which explains undocumented features, such as induction proofs, or proof planning for better performance; (ii) an extended ***Z/Eves*** toolkit providing better coverage of proof rules for certain definitions, such as finiteness, injections, sequences, and so on; (iii) a CZT AST to ***Z/Eves*** XML API translation tool, which allows one to (partially) convert Z Standard in CZT to ***Z/Eves*** Z-LaTeX; and (iv) a programmable interface prototype to ***Z/Eves*** that allows one to interact with the prover directly via Java programs, as well as create new proof tactics. We also contribute to the (partial) integration of *Circus* tools, such as the parser and the typechecker, into the CZT framework by providing an appropriate extension to the XML schema representing Z Standard to support *Circus*.

In the next section we summarise the work done in this thesis in some more detail. After that, Section 6.2 presents related work in different levels: from very closely related tools, such as FDR, to possible extensions and improvements in the proposed architecture drawn from other research areas in the model checking community. Finally, in Section 6.3 we present topics for future research.

6.1 Thesis summary

In this thesis, we present a formal specification of an architecture for a *Circus* model checker together with its implementation prototype in Java. It involves a series of phases responsible for transforming *Circus* programs input as LaTeX markup into structured debugging information about refinement check failures, or successful reports about a refinement proof.

The parser for *Circus* [BC02] is responsible for generating an abstract syntax tree (AST) [WB00] from the LaTeX input. This AST is implemented by the *Community Z Tools* (CZT) project [MU05] for the Z Standard [Pan00], and we extend it for *Circus* [MFMU05]. The validation of the AST checks that it represents a well-formed *Circus* specification free from type, scope, or naming problems by typechecking [Xav06], and annotating the AST with additional information about contextual-analysis. This annotated abstract syntax tree (AST_+) is passed to the compiler that implements the operational semantics of Chapter 3 defined using ***Z/Eves***, in order to generate a predicate transition system (PTS) to be analysed by the model checking algorithms defined in Chapter 4. The strategy and algorithms for model checking defined in Chapter 4 explain how the tool checks whether the stated refinement between the abstract specification and the design holds or not. For that, the tool might generate

proof obligations for state operations that need to be discharged. If it does not hold, a suitable human-readable counter-example must be provided and proof obligations can be discharged using a theorem proving plugin. Due to the state explosion problem, it is not an efficient solution to generate the whole transition system from the AST_+. Instead, the compiler is able to generate the *PTS* on-the-fly as needed by the refinement checker. These stages are summarised in Figure 4.1 on page 87. The prototype presented in Chapter 5 accounts for the necessary implementation details of the operational semantics from Chapter 3, as well as the model checking architecture, search strategy, and algorithms from Chapter 4.

The refinement checker for *Circus* is responsible for proving refinement between a specification S and its design or implementation I, denoted by $S \sqsubseteq I$. The compiler generates two well-formed *PTS*s representing each side of the refinement relation we want to analyse. Firstly, a semantics-preserving transformation on the specification side is performed, in order to make it deterministic and free from internal (silent) transitions. This operation is specified in *Circus* itself, and it generates a normal form from the original *PTS* (see Section 4.5). Normalisation is important because it eliminates nondeterministic choices present in the specification side of the refinement relation. The specification side is chosen for normalisation because it usually represents a more abstract view of the problem, hence it is more likely to gain from nondeterminism elimination. This elimination is achieved by using a specialised version of subset construction over automata, and usually results in the state space shrinking in size, although the size can grown to the power space of the original automaton in the worst (unusual) case [Ros94b]. In spite of this fact, empirical experiences show that it is usually smaller than the original transition system [Ros94b, Gol01]. Apart from (possibly) reducing the state space, the normal form is also useful for memory economy. Since it is deterministic, the information about the trace for a witness counter-example used by the debugger is reduced in a great extent, as mentioned in Section 4.6.

After these preparations on the automata, the model checker can perform the witness search to either establish the refinement order, or find a witness counter-example exposing the reasons of failure for the debugger to present in human-readable format. The witness search performs a variation of BFS in node pairs coming from the automata for the actions on each side of the refinement relation. The search looks for mutually reachable node pairs until all pairs have been checked, or an incompatible node pair has been found. A witness represents a sequence of node pairs with a trace in common in both automata, where the last node pair violates at least one compatibility criterion. The compatibility criterion is based on the properties one wants to express. Another advantage of having a normal form is that, since it is deterministic, this trace is unique. Whenever an incompatible pair is found, the search data structures contain enough information to enable the debugger process to answer the queries for witnesses from the environment. The debugger is able to build up the transition system backwards from the point of failure back to the root of the search. These two components of witness search and debugging are also formally specified as a *Circus* process in Sections 4.6 and 4.7, respectively.

The correctness of the witness search algorithm is important in order to guarantee the correctness of the refinement check. The abstract version of the witness search process defines a witness invariant, compatibility criteria for different properties, and criteria for finding successor node pairs mutually reachable from each automaton. The witness invariant is important in order to ensure that witnesses found are not only minimal, but also faithful with respect to the automata representing the *Circus* programs. As we search using BFS over a deterministic normal form, it is known that

any witness found is already minimal [Gol01]. Each refinement criterion has been formally defined using the schema calculus in a modular fashion. This enables further extension, as long as new criteria related to different properties raises interest, such as the extended failures mentioned in [BL04].

From the abstract *Circus* specification of the model checker, we calculate a sequential algorithm through the application of *Circus* refinement laws [CSW02, Cav97]. This algorithm resembles the search algorithm for FDR presented in [MH00]. The major argument for the refinement calculation of the algorithm is because this guarantees it to be correct by construction. Additionally, we have formal documentation (see [Fre04d]) of important properties, such as loop invariants, that have not yet been explained in depth in the available literature [Ros94b, RGG$^+$95, Gol01, MH00], as far as we know. In itself, to the extent of our knowledge, this derivation constitutes the largest case-study on the application of the Z refinement calculus (ZRC) [Cav97].

A precise description of the debugger is also important because it ensures that the filtering on the search data structures for presentation purposes neither introduces nor hides relevant information about a counter-example. The debugger is defined following similar ideas from FDR. It has debugging sessions with debugging windows of information about individual arcs related to search failures. This information consists of the witness causing the failure, the trace that lead to the problem, and additional information about observed and offending acceptances sets at the point of failure. Further information can be added in order to enable a tree-view of the failure, or observation of acceptances at different node pairs; they were not implemented and are beyond our scope.

The *PTS* generated by the compiler is the data structure that needs to address the two main problems for model checking mentioned in [CGP00]: state explosion, and searching strategy. The former is addressed by symbolic representation of values, whereas the latter is guaranteed by the formal derivation of a refinement search algorithm. Moreover, in the *PTS* generated by the compiler, implicit nondeterminism is represented by a silent transition with empty arcs, whereas divergence is characterised by a loop of unlabelled (or silent) transitions. An interesting feature is the representation of state variables through symbolic values, rather than actual ones. For instance, this allows the definition of loosely defined Z constants in *Circus* programs, which cannot be expressed using FDR. Another interesting feature of this data structure to handle the state explosion problem is that infinite data types on prefixing communications can be represented using set comprehension notation with (possible) symbolic references to state variables, hence allowing a finite representation of some kinds of infinite state systems.

These process descriptions and algorithm derivation are the heart of our strategy for model checking *Circus* specifications; they provide great insight and reliable precision, as we use ***Z/Eves*** in the description and proof of properties throughout the development process. We also use ***Z/Eves*** to typecheck, check for consistency (***Z/Eves*** domain checks [MS97, Appendix B]), check for applicability (precondition proofs), and animate [Saa99b, Section 3.2.5, and 3.2.6] the operational semantics presented in Chapter 3. The automata theory of predicate transition systems generated by the compiler and used by the refinement checker is also formally described in ***Z/Eves***, together with a series of theorems about automata properties (see [Fre04b]). For instance, the transformations applied on the automaton during normalisation have been proved to be semantics-preserving. The Z part of each *Circus* process specifying the refinement checker presented in Chapter 4 is typechecked, as well as the operational semantics of Chapter 3. The proof obligations generated by the application of

refinement laws during the derivation of the sequential algorithm is discharged using ***Z/Eves*** once more. We expect this mechanised proof effort to become the source for further development of formal tools related to refinement model checking of state-rich languages, such as *Circus*.

A prototype implementing these ideas is developed in Java, where some of the formal material is included as JML annotations [LBR98, BCC+03, LPC+04] (see Section 5.2.3). At least, JML provides a clearer (and more precise) code documentation. It also enables further analysis of the code with tools such as ESC/Java [DLNS98] for extended static checking, exceptional behaviour, or partial code correctness (*i.e.*, termination) with respect to pre and postcondition specifications. Even further, tools such as LOOP [Hui01] or Daikon [ECGN01] can be used for finding loop and class invariants, and tools such as JACK [BRL03], can be used for dealing with more complex pre-post condition analysis via integration with main stream theorem provers. This approach to automatic formal verification follows the trend set out by the *verifying compiler* grand challenge [BHW04]. Moreover, we use a series of design patterns for Java during the implementation [GHJV95, WB00, Lea00, SSRB00, CS95], as well as an informal framework design notation guideline has been used during the development of the object-oriented hierarchies [Rie00].

Our aim is to provide a model checker tool for *Circus* that will encourage academia and industry to use the language and its techniques in a similar way that FDR did for CSP, hence the richer setting of language paradigms and verification techniques that *Circus* offer [WC01a, CSW02, CSW05, SJ02, Woo02, Tan05] will be better exploited in the future.

6.2 Related work

In the literature [CW96], there are two main approaches to model checking: classical based on temporal logic [CGP00, McM93a, Kur94]; and refinement based on set-theoretic models [Ros94b, RGG+95]. They are mainly concerned with behavioural aspects of systems, where concurrency plays a central role. Classical model checking is a decision procedure involving an automaton that represents the system to be checked against specification properties defined as temporal logic formulae. To deal with the state explosion problem that arises from industrial-scale systems, such as hardware design, symbolic automata encoded as *Binary Decision Diagrams* (BDDs) [Bry86] are used in industrial-strength model checkers [CCO+05a]. Refinement model checking aims at establishing a refinement relation between two formal specifications, given as finite automata representing all observable behaviours, via an exhaustive search over these behaviours. The main difference between these approaches is that the latter enables not only the proof of specification properties, but also formal stepwise development through refinement of systems from early abstract specifications down to the actual code: our major motivation for choosing *Circus*.

There are few published works related to refinement model checking tools. To the extent of our knowledge, the available sources of refinement model checking are related to CSP [Gol01, CH93b, Gol00], which we discuss extensively throughout Chapters 3 and 4.

Another experimental refinement model checking tool that supports CSP is presented in [PY96b] with detailed documentation about implementation and strategies used; it is named ARC. Its refinement search algorithm is very similar to FDR and our own. The main difference is that ARC does not perform normalisation, hence re-

finement search and debugging structures need to carry additional information about the trace of possible nondeterministic paths chosen. Examples present in the given reference show that ARC performed better than FDR1; however, the improvements in FDR2 were better than ARC's.

The Alloy tool, which provides support for automatic formal verification of state-rich systems, is a proof-checker or model finder given in [Jac00, Jac02]. Our tool can also be used similarly with the aid of an appropriate theorem prover plugin, such as the ICS solver [Gro05] or the ***Z/Eves*** XML interface mentioned in Section 5.3.3.

CSP || B [ST02, ST04] orthogonally combines B machines with CSP_M specifications, so that B theorem provers and FDR can be used independently. Nevertheless, in this combination, restrictions in the way B can be combined with CSP are necessary in order not to compromise the refinement check results.

Finally, in [LB03], a tool that combines CSP with B for refinement model checking is presented. Like our approach, they combine CSP with a B theorem prover through a deep combination of the semantic models in a new tool built on the top of their B theorem prover. The tool is capable of animating B specifications either automatically or interactively with wide coverage of the languages, where integration with CSP is explained in [BL05]. Nevertheless, the refinement checking features are still limited and under development, as mentioned in [LBP05].

In what follows, we briefly summarise the available approaches that are related to model checking in general.

6.2.1 Automata theoretic methods

Automata theoretic methods represent both implementation and specification properties as an automaton. Refinement model checkers are an example of such methods. The language each automaton represents reflects the expressiveness of the formal notation. The verification to be performed is language containment between automata representing programs. Therefore, if the specification contains the language of the implementation, then the implementation satisfies the specification.

This procedure of language containment can be repeated up to a point where one find a satisfactory version close to the final implementation code, while still satisfying the original properties of the abstract specification automaton. This kind of stepwise development cycle is very attractive for industrial-scale development.

In [Mil80, Mil90], Milner takes an approach different from that adopted by FDR for CSP to modelling the CCS process algebra. Instead of collecting a set of sequences of traces like in CSP, the model captures external behaviours by means of a tree of observations. The notion of correctness is established by observational equivalence between a pair of processes on external communication only: an observer cannot distinguish between two processes by any experiment. Another form of equivalence is defined in order to capture internal actions: it is called behavioural equivalence. When one considers internal actions while establishing the equivalence relation between two processes, nondeterminism might be expressed and captured. These forms of equivalence are proved by establishing a bisimulation relation between the two processes.

6.2.2 Temporal logic

Clarke and Emerson [CGL90] coined the concept of model checking by proving the validity of a temporal logic specification against a specific finite-state model representing an implementation as an automaton. Temporal logics specify desired specification

properties by combining the usual propositional logic operators with tense operators: they are used to form statements about how conditions change in time [MP92, MP95]. This idea of temporal properties for verification caught the interest of the hardware design community mainly due to the availability of tool support.

The formal model of the design is normally represented by a Kripke structure, which must represent the implementation requirements of the system under consideration [CGP00, Chapter 2]. Whenever all states of the Kripke structure have been visited and checked, the algorithm stabilises by reaching a fixed point, hence establishing the truth of the formula under concern. The model checker *SPIN* is a successful example of using temporal logic for model checking [Hol97].

This approach using temporal logics is known as classical model checking. It is different from ours because it does not establish an ordering of improvement.

Temporal logic and refinement model checking are useful for different kinds of problems. In [LMC01], Leuschel discusses the difficulties and complexities of making such comparisons and transformations between temporal logics and refinement model checking, as well as some suggestions for enhancing the FDR refinement model checker, so that one can translate temporal formulae to CSP_M. The suggestions aim at achieving a greater degree of automation for the transformation process from classical temporal logic model checking into refinement model checking. This effort would enable one to prove temporal logic properties using refinement model checking. In [HJ98, Chapter 0], the possibility (or desire) of including temporal logic in UTP is mentioned. This would enable one to check temporal logic properties in *Circus*, as its semantic model is based in UTP.

Other algorithms have been proposed in order to deal with the state explosion problem in the context of temporal logic model checking. The most successful one is the use of *Ordered Binary Decision Diagrams* (OBDDs) as the data structure used to represent temporal logic formulae [Bry86]. The use of OBDDs in a model checker is known in the literature as *Symbolic Model Checking* [McM93a]. The *SMV* model checker tool is a successful example of the use of OBDDs in industrial-scale model checking [CCO$^+$05a, McM93b].

This form of symbolic model checking can be argued as a new algorithm for theorem proving—a proof tactic. Symbolic representation of temporal logic formulae using OBDDs has allowed industrial-scale model checking, as mentioned in [CGP00, Chapter 1]. Nevertheless, because of the structure of OBDDs being tailored to represent boolean valued functions, this symbolic approach is better suited for the validation of specific kinds of finite-state systems, such as digital hardware circuits or security protocols; they do not have a complex state. Although OBDDs are tailored to check very structured systems like in hardware design, software modelling is an entirely different issue. Software systems usually have a much more complex behaviour, which increases the possible number of states to explore, making OBDDs an inefficient data structure, as pointed out in [McM93a, Chapter 2].

In spite of the fact our data structures also involve symbolic algorithms, our approach for model checking is based on automata theoretic methods, language inclusion, acceptances sets and divergence information, in order to establish refinement, rather than checking temporal formulae.

6.3 Future work

In the long term, our main objective with our model checker is to provide efficient industrial-strength refinement model checking with integrated theorem proving support. For that we still need to investigate the criteria of efficiency and scalability, while using *PTS* for representing *Circus* programs, initially from our domain of high-integrity and safety-critical systems. For this, we indicate a series of activities for future work, which include: proof of soundness between the different semantic models; theorem proving automation support through application-oriented theories; improved refinement model checking algorithms; and industrial-scale applicability.

In this thesis, we are concerned with an expressive subset of the action language of *Circus*. This subset enables description of most important aspects of concurrent systems such as loops, nondeterminism, communication, synchronisation, deterministic choice, local variables, abstraction via concealment of events, and so forth. To achieve our scalability goal, firstly we need to cover the whole *Circus* language, including new operators and *Circus* processes.

The proof of correctness of the operational semantics presented in Chapter 3 with respect to the well-advanced denotational semantics in UTP [OCW05, Oli06] is yet to be done. As the transition relation can be defined using the refinement relation of UTP, as presented in Section 3.1, it is possible to link the denotational and operational semantics. This proof is important because it ensures soundness between different semantic models of *Circus*, hence guaranteeing the consistency of the language, its tools and techniques.

During the refinement search, tool support for discharging proof obligations generated is needed. For this, fine-tuned application-oriented theories are fundamental. Thus, we plan to provide those for most common constructs from our case-studies as extensions to the base theory of *Circus* in ProofPowerZ [OCW05]. Eventually, from what our experience with theorem provers shows, as the tools mature, a considerable improvement in automation levels is achieved. Theorem proving can also be used to find suitable abstractions from complex data types in order to allow better model checking performance and automation. The results of the model checker, if necessary, can then be further analysed with the debugger to draw the final conclusions. This modular approach allows a wide variety of systems to be analysed. The outcome is an environment that enables iterated integrated analysis comprising model checking and theorem proving, as mentioned in [Sha02, Rus00].

In order to enable efficient industrial-scale model checking, we see parallelisation of model checking algorithms as a major consideration. Parallel behaviour usually introduces unpredictable observations, a feature we cannot afford to have. Therefore, particular attention must be given to this important issue. From the abstract specification of the witness search, a parallel algorithm that is similar to the one presented for FDR in [MH00] can be derived, and others can be proposed and studied as well. We can apply *Circus* refinement laws again [CSW02] on the abstract model checking specification from Section 4.6, in order to produce a parallel version of the refinement search algorithm to be implemented in Java. As occurred during the derivation of the sequential version in the prototype, proof obligations can be discharged mechanically.

The parallel algorithm for FDR [MH00], for example, is similar in structure to the sequential algorithm given in Section 4.8. The difference is that the compatibility criteria check component is in parallel with the component that looks for successors node pairs. Thus, such derivation ought to take into account the issues related to dependencies between proof obligations generated, in order to allow proper theorem proving

integration in this parallel setting. Additionally, the modularisation of compatibility criteria check and the abstract witness search specification allows extension for the inclusion of other refinement models. In the context of [Tan05, CSW05], mobility and object-orientation can be considered as initial candidates. Extended failures models for CSP [BL04] can also be included, as mentioned in Section 5.3.2. This calculation of a parallel algorithm should explore some of the new refinement laws for the action language of *Circus* and its parallel operator [Oli06]. We believe that the theory of automata from [Fre04b], used as the basis of our witness search and operational semantics, can be used to formalise the algorithms of FDR as well.

In Section 4.2.3, we explain the idea of using different levels of automation while model checking, where expressiveness of specifications is the compromise. We plan to explore the suitability of this facility as we analyse more examples with the prototype by incorporating more powerful automatic tools as new theorem proving plugins, such as PVS's satisfiability solver [Gro05]. Allowing the user to act as an environment selecting enabled communication paths according to the expected behaviours of the system is also an interesting topic for future research. It would enable test of properties for complex systems, rather than refinement proofs, and would have an interface similar to ProBE [For00]. Whenever explicit model checking is not sufficient, and proof obligations need to be discharged, we can use application-oriented theories extensively. Experiences of our industry partner QinetiQ Malvern on the use of application-oriented theories to increase automation levels have been effective in automatically discharging proof obligations. Furthermore, if stand-alone parallelisation of model checking algorithms is not enough, we can resort to grid technology in order to enhance the possibilities of the parallel algorithm through distribution of search data via a grid network [OGP02, OGM04, Cir03, ACBR03, PCB03].

As we analyse more complex examples, we can approach the state explosion problem on two fronts: (i) specific compression techniques of the *PTS* automata generated by the *Circus* operational semantics; and (ii) generic techniques involving theorem proving, such as data abstraction and data independence. For the former, we can investigate successful compression techniques used in FDR's automata [RGG$^+$95], in order to envisage possible manipulations of our *PTS* representing *Circus* programs. Another interesting technique closely related to our data type is the stubborn set compression technique described in [Kok98]. For the latter, we can study data abstraction and independence for CSP from [Laz99], and for another combination of Z and CSP in [Mot01, Far03].

We can also investigate how data independence can be encoded into the *PTS* through annotations during compilation, such as early divergence detection, and simple forms of data independence. Early divergence detection, such as a schema expression executing outside its precondition, can be included as Z predicate annotations in the automata nodes. This would enable us to avoid the bottleneck caused by chasing silent transitions through DFS, the standard approach for divergence detection in refinement model checking tools [Ros94b, Gol01]. These annotations can also be useful for finitely representing some sorts of infinite state systems, hence providing a simple form of automatic data independence, similar to those defined in [Mot01, Far03].

From these investigations, we can improve efficiency by decreasing the time and space complexity necessary for both automatic explicit state enumeration model checking, or automated symbolic model checking with integrated theorem proving support in a parallel environment.

The FDR capabilities shown in [RGG$^+$95, RSR$^+$01] are our efficiency targets, since as far as we know, it is the best industrial-scale refinement model checking

tool available. Firstly, we plan to specify in *Circus* the *CyberCash* security protocol described in [RSR+01], where complex statements about data aspects in FDR's functional language will be abstracted by using the Z schema calculus. Finally, we will perform a benchmark analysis on the *Circus* model with respect to the original FDR/PVS model. The benchmark criteria could be in terms of the number of transitions and distinct states needed to describe the system, complexity of generated proof obligations, time necessary to perform the analysis, and readability or usefulness of debugging information presented. This will enable us to provide a clearer comparability between an integrated model checking and theorem proving solution, in comparison with the adaptation and simplification needed in order to apply industrial-scale and high-performance tools. We would like to emphasise that our aim with this exercise would be more to compare the scalability of our approach, rather than actual numbers, since both these FDR and PVS have decades of maturity and enhancement behind them.

With a stable, sound, and optimised tool with additional fine-tuned application-oriented theories support, we are enabled to analyse paramount industrial-scale examples. For instance, during the development of the *Mondex Purse*, which was the first project to produce a product with *ITSEC Level E6* certification, industry thought that the amount of accuracy via formal analysis required was science fiction [Hea97]. That project has proved that formal analysis can be a science fact, although proofs could not be dealt with by machine [SCW00]. Even though this was a milestone in the safety-critical software domain in using Z and its refinement calculus for formal specification and verification, no behavioural aspects such as order of execution or concurrency patterns were defined. With tool support for *Circus*, we can include the missing behaviour patterns with CSP operators and guarded commands. Using our model checker to prove properties about this system, as automatically as possible, would be the ultimate test.

Following these efforts, we intend to provide in the long term an integrated development environment for *Circus* with improved facilities such as: improved debugger; integration with other tools, such as theorem provers to analyse debugging information; inclusion of algorithms to aid the calculation of abstractions, such as the one presented in [Mot01, Far03]; and so on.

Appendix A

CD-ROM

This appendix contains pointers to extra material not included in the thesis document due to space constraints. The contents are organised as a CD-ROM that includes *Circus* programs in LaTeX, Z specifications in ***Z/Eves*** Z-LaTeX, Jaza LaTeX, compiled (*.pdf, *.ps) documents, ***Z/Eves*** database (*.zev, *.zcs) files, and so on.

The CD-ROM directory hierarchy is equivalent to the table of contents for the appendix. In this sense, files related to each section are present in the directory corresponding to the section number and name. Each relevant directory contains a Readme.txt giving further explanation of the directory contents.

A.1 Additional material

The extensive use of ***Z/Eves*** led to a series of additional material that might be useful elsewhere for formal specification and verification purposes.

A.1.1 *Z/Eves*

While widely using ***Z/Eves*** throughout the development process from the very beginning, interesting material were produced, as briefly detailed below. For opening the given examples, one needs either ***Z/Eves*** *v*.2.3 and Python *v*.2.3 for loading the ***Z/Eves*** database (*.zev) files in the GUI, or ***Z/Eves*** and emacs for loading the Z-LaTeX sources directly (see [Saa99b]).

A.1.1.1 Advanced tutorial

This tutorial contains extended documentation for: effective Z idiom, useful naming conventions, proof commands, proof planning, proof optimisation, complete induction proof example, and so on. The available definitions of the induction proof are also available as a ***Z/Eves*** database.

A.1.1.2 Toolkit extensions

The toolkit extensions document consists of two parts: extended operators and extended automation. The former includes new operators and their related properties, whereas the latter includes additional automation rules for the original ***Z/Eves*** Z toolkit [Saa99a].

A.1.1.3 Programmable interface

The ***Z/Eves*** programmable interface is now one subproject of the CZT project and can be accessed on-line at [UM03]; it is not in the CD-ROM.

A.2 CZT integration

The files related to this section contain the XML schemas extending Z Standard schema to include the constructs for *Circus*. These are mainly two files: one for the language BNF syntax, and another for special constructs needed for our tools. The former is also available on-line at [UM03].

A.3 *PTS* (automata) theory

The files related to this section contain the automata theory used to represent the operational semantics of Chapter 3; it is sketched in Section 4.4.

A.4 Operational semantics

In this section we provide the operational semantics presented in Chapter 3 as a ***Z/Eves*** database, as well as a simplified version used for animation and validation purposes ***Z/Eves*** and Jaza, respectively.

A.4.1 Full version in *Z/Eves*

A.4.2 Simplified version animation in *Z/Eves*

A.4.3 Simplified version animation in Jaza

A.5 Model checker processes in *Circus*

The files related to this section contain the complete formalisation of the model checker architecture presented in Chapter 4.

A.6 Refinement search algorithm derivation

The files related to this section contain the derivation of the refinement search algorithm presented in Section 4.8.

A.7 Thesis document sources

The files related to this section contain the original LaTeX sources of this thesis.

A.8 LaTeX related

This section contains additional LaTeX files, such as *.bib, *.sty, *.eps files, in order to enable compilation of the original sources provided.

Bibliography

[Abr96] Jean-Raymond Abrial. *The B Book—Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-52149619-5.

[ACBR03] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, and Paulo Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[AKW03] D. Atiya, S. King, and J. C. P. Woodcock. Ravenscar Protected Objects: a *Circus* Semantics. Technical Report 356, Deprtment of Computer Science, University of York, June 2003.

[ASM] Abstract State Machines web-site. www.eecs.umich.edu/gasm/.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[BA] B Archive web-site. www.afm.sbu.ac.uk/b/.

[Bar03] John Barnes. *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley, 2^{nd} edition, 2003.

[BBDS97] Eerke A. Boiten, Howard Bowman, John Derrick, and Maarten Steen. Viewpoint consistency in z and lotos: A case study. In *FME*, pages 644—664, 1997.

[BC02] André Barbosa and Ana Cavalcanti. A Parser for *Circus*. Graduation Research Project, 2002.

[BCC+03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll5. An Overview of JML Tools and Applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Electronic Notes in Theoretical Computer Science, pages 73—89. University of Nijmegen, Elsevier, March 2003.

[BF04] Moshe Bar and Karl Fogel. *Open Source Development with CVS*. Paraglyph Press, 3^{rd} edition, 2004.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of ACM*, 31(3):560—599, July 1984.

[BHW04] J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The Verified Software Repository: a Step Towards the Verifying Compiler. UK Grand Challenge for Computer Research, Steering Committee, 2004.

[BL04] Christie Bolton and Gavin Lowe. A Hierachy of Failures-Based Models: Theory and Application. *Theoretical Computer Science Journal*, June 2004.

[BL05] Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FME 2005: Formal Methods*, number 3582 in Lecture Notes in Computer Science, pages 221—236. Springer-Verlag, 2005.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Proceedins of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[BR85] S. D. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Process. *Lecture Notes in Computer Science*, 197:281—305, 1985.

[BRL03] L. Burdy, A. Requet, and J. L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *Proceedings of Formal Methods Europe, Pisa*, number 2805 in Lecture Notes in Computer Science, pages 422—439. FM, Springer-Verlag, 2003.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677—691, August 1986.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[BT] B Toolkit web-site. www.b-core.com/btoolkit.html.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Text in Computer Science. Springer-Verlag, 1998.

[Cav97] Ana Cavalcanti. *A Refinement Calculus for Z*. PhD thesis, Oxford University, 1997.

[CCo+04] Sagar Chaki, Edmund M. Clarke, Joel ouaknine, Natasha Sharygina, and Nishant Sinha. State/Event-Based Software Model Checking. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *4th International Conference in Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 128—147, 2004.

[CCO+05a] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Marco Pistore, and Marco Roveri. *NuSMV 2.2 User's Manual*. Carneige Mellon University - at Trento Italy, nusmv.irst.itc.it, 2005.

[CCO05b] Ana Cavalcanti, Phil Clayton, and Colin O'Halloran. Control Law Diagrams in *Circus*. In *Proceedings of Formal Methods Europe*, number 3582 in Lecture Notes in Computer Science, pages 253—268, 2005.

[CGL90] E. Clarke, O. Grumberg, and D. Long. Model Checking. Technical report, Carnegie Mellon and AT&T Bell Labs, April 1990.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, February 2000.

[CH93a] Rance Cleaveland and Matthew Hennessy, editors. *FACJ*, volume 5. Springer-Verlag, 1993.

[CH93b] Rance Cleaveland and Matthew Hennessy. *Testing Equivalence as a Bisimulation Equivalence in [CH93a]*, volume 5, chapter 1, pages 1—20. Springer-Verlag, 1993.

[Cir03] Walfredo Cirne. Grid Computing For Bag of Tasks Applications. In *Proceedings of the Third IFIP Conference on E-Commerce, E-Business and E-Goverment*, September 2003.

[Cor04] Marcio Cornelio. *Refactoring as Formal Refinements*. PhD thesis, Universidade Federal de Pernambuco, Brazil, 2004.

[CS95] James O. Coplein and Douglas Schmidt, editors. *Pattern Languages of Program Design*, volume 1 of *Software Pattern Series*. Addison Wesley, 1995.

[CSW02] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of Actions in *Circus*. In J. Derrick, E. Boiten, J. C. P. Woodcock, and J. Wright, editors, *Proceedings of REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[CSW05] 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.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods—State of the Art and Future Directions. *ACM Computer Surveys*, 28(4):626—643, December 1996.

[CW05] Ana Cavalcanti and Jim Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. Department of Computer Science, University of York, 2005.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, May 1976.

[DLNS98] David Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center (SRC), www.research.digital.com/SRC/, 1998.

[DN93] John Dillenburg and Peter C. Nelson. Improving the efficiency of depth-first search by cycle elimination. *Information Processing Letters*, 45:5—10, January 1993.

[DPP00] Agostino Dovier, Carla Piazza, and Alberto Policriti. A Fast Bissimulation Algorithm. Technical report, University di Verona and University Udine, November 2000. UDM/14/00/RR.

[ECGN01] M. D. Ernest, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):1—25, 2001.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with Two Levels of Semantics. Technical Report 83-01, Technische Universität Berlin, 1983.

[Far03] Adalberto Cajueiro Farias. Efficient and Mechanised Analysis of Infinite CSP-Z Processes. Master's thesis, Universidade Federal de Pernambuco, May 2003.

[Fis97a] Clemens Fischer. Combining CSP and Z. Technical report, University of Oldenburg, 1997.

[Fis97b] Clemens Fischer. Model Checking CSP and Z. Technical report, University of Oldenburg, 1997.

[Fis98] Clemens Fischer. How to Combine Z with Process Algebra. Technical report, University of Oldenburg, 1998.

[Fis00] Clemens Fischer. *Combination and Implementation of Process and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[FMA] Formal Methods Archive web-site. www.afm.sbu.ac.uk/.

[FME] Formal Methods Europe web-site. www.fmeurope.org/.

[For00] Formal Systems (Europe) Ltd. *ProBE User's Manual version 1.28*, May 2000.

[Fre55] Sigmund Freud. *Civilisation and Its Discontents*. Complete Psychological Essays. The Hogarth Press Ltd., 2^{nd} edition, 1955.

[Fre02] Leonardo Freitas. JACK—A Process Algebra Implementation in Java. Master's thesis, Universidade Federal de Pernambuco, April 2002.

[Fre03] Leonardo Freitas. Exploring the Internal Object Model of FDR. not published, `Tcl/Tk` script, 2003.

[Fre04a] Leonardo Freitas. Formal Model Checking Architecture for *Circus*. Appendix A.5 in [Fre05e] (CD-ROM), 2004.

[Fre04b] Leonardo Freitas. Predicate Transition System—Automata Theory. Appendix A.3 in [Fre05e] (CD-ROM), 2004.

[Fre04c] Leonardo Freitas. Proving Theorems with Z/Eves. Appendix A.1.1.1 in [Fre05e] (CD-ROM), 2004.

[Fre04d] Leonardo Freitas. Sequential Refinement Search Algorithm Derivation for Model Checking *Circus*. Appendix A.6 in [Fre05e] (CD-ROM), 2004.

[Fre05a] Angela Freitas. From *Circus* to Java: Implementation and Verification of a Translation Strategy. Master's thesis, University of York, 2005.

[Fre05b] Leonardo Freitas. Animating a Simplified Operational Semantics for Model Checking *Circus* in ***Z/Eves***. Appendix A.4.2 in [Fre05e] (CD-ROM), 2005.

[Fre05c] Leonardo Freitas. CZT integration with *Circus*. Appendix A.2 in [Fre05e] (CD-ROM), 2005.

[Fre05d] Leonardo Freitas. Formalisation of the Operational Semantics for Model Checking *Circus* in ***Z/Eves***. Appendix A.4.1 in [Fre05e] (CD-ROM), 2005.

[Fre05e] Leonardo Freitas. *Model Checking Circus*. PhD thesis, Univeristy of York, October 2005.

[FW06] Leo Freitas and Jim Woodcock. An Archiecture for Integrating Refinement Model Checking with Theorem Proving. *Innovations in Systems and Software Engineering, a NASA Journal*, April 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Jonhson, and John Vlissides. *Design Pattern Elements of Reusable Object-Oriented Software*. Adison Wesley, 1995.

[GN99] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software Practice and Experience*, 0(S1):1—5, June 1999.

[Gol00] Michael Goldsmith. *FDR2 User's Manual version 2.67*. Formal Systems (Europe) Ltd., May 2000.

[Gol01] Michael Goldsmith. *Overview of FDR in [RSR$^+$01]*, chapter 4, pages 125—140. Addison Wesley, 2001.

[Gro92] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practioner Series. Prentice-Hall, 1992. ISBN 0-13-752833-7.

[Gro95] The RAISE Method Group. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, 1995. ISBN 0-13-752700-4.

[Gro05] The ICS Group. *ICS Manual (Version 2.0)*. SRI International, Computer Science Laboratory, SRI International 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, 2005.

[Hea97] Goverment Communication Headquarters. UNCLASSIFIED: E6, Use of Formality Discussion. G3A videotape n^o 68, 1997.

[HFR99] Neil Harrison, Brian Foote, and Hans Rohnert, editors. *Pattern Languages of Program Design*, volume 4 of *Software Pattern Series*. Addison Wesley, December 1999.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. International Series in Computer Science. Prentice-Hall, 1998. ISBN: 0134587618.

[HMU01] John Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2^{nd} edition, 2001. ISBN: 0201441241.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Process.* International Series in Computer Science. Prentice-Hall, 1985.

[Hol97] Gerard J. Holzmann. The Model-Checker SPIN. In *IEEE Transactions on Software Engineering*, volume 23:5, pages 1—17, 1997.

[Hui01] Marieke Huisman. *Reasoning about Java Programs in Higher-Order Logic using PVS and Isabelle.* PhD thesis, Universiteit Nijmegen, 2001.

[Jac00] Daniel Jackson. Automating First-Order Relational Logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, volume 25:6 of *ACM SIGSOFT Software Engineering Notes*, pages 130—139, November 2000.

[Jac02] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256—290, April 2002.

[JAX] Java Architecture for XML Binding (JAXB). java.sun.com/xml/jaxb/.

[JCP98] Java Community Process. jcp.org, 1998.

[JEd98] JEdit Java Editor. www.jedit.org, 1998.

[JG98] Geraint Jones and Michael Goldsmith. *Programming in occam 2.* International Series in Computer Science. Prentice-Hall, 2^{nd} edition, 1998.

[JLL02] He Jifeng, Zhiming Liu, and Xiaoshan Li. Towards a Refinement Calculus for Object Systems. In *Proceedings of the Conference ICCI2002*, pages 69—77. IEEE Computer Society Press, 2002.

[Jon83] C. B. Jones. Specification and Design of (Parallel) Programs. In *Proceedings of IFIP'83 North Holland*, pages 321—332, 1983.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 2^{nd} edition, April 1990.

[JP01] B. Jacobs and E. Poll. A Logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lecture Notes in Computer Science, pages 284—299. Springer, 2001.

[JUn05] JUnit: Java Unit Testing Framework. www.junit.org, 2005.

[Kan65] Immaunel Kant. *Logic.* University of Konigsberg, 1765.

[Kok98] Ilkka Kokkarinen. *A Veridication-Oriented Theory of Data in Labelled Transition Systems.* PhD thesis, Tampere University, Finland, 1998.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[Law04] Jonathan Lawrence. Practical Application of CSP and FDR to Software Design. In Ali E. Abdallah, Cliff Jones, and Jeff Sanders, editors, *25 Years of CSP*. FACJ, 2004.

[Laz99] Ranco S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University, Programming Research Group, 1999.

[LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, number 2805 in Lecture Notes in Computer Science, pages 855—874. Springer-Verlag, 2003.

[LBP05] M. A. Leuschel, M. Butler, and S. Lo Presti. *ProB User Manual version 1.1.4*. Declarative Systems and Software Engineering University of Southampton, UK and Softwaretechnik und Programmiersprachen University of Düusseldorf, Germany, 2005.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A behavioural interface specification for Java. Technical Report TR #98-06y, Department of Computer Science Iowa State University, 1998.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design, in Behavioural Specification for Businesses and Systems*, chapter 12, pages 175—188. Kluwer, 1999.

[Lea00] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison Wesley, 2 edition, February 2000.

[Lem03] Lemma-One. *ProofPower Tutorial*, 2003.

[LMC01] M. Leuschel, T. Massart, and A. Currie. How to Make FDR Spin: LTL Model Checking of CSP by Refinement. In J. N. Oliveira and P. Zave, editors, *Formal Methods Europe2001*, volume 2021, pages 99—118. Springer-Verlag Berlin, 2001.

[LOT] LOTOS at Ottawa web-site. www.csi.uottawa.ca/~lotos/.

[Low96] Gave Lowe. A Hierarchy of Authentication Specifications. Technical report, University of Leicester, 1996.

[Low97] Gave Lowe. *CASPER User Manual*. Oxford University, 1997.

[LPC+04] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry. *JML Reference Manual*. Iowa State University, August 2004. Revision 1.93.

[Mar97] A. C. Martin. Acyclic Visitor. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[McM93a] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM93b] K. L. McMillian. *The SMV System in [McM93a]*, chapter 4, pages 61–86. Kluwer Academic Publishers, 1993.

[MD00] B. Mahony and J. S. Dong. Timed Communicating Object-Z. *IEEE Transactions on Software Engineering*, 26(2):150—177, February 2000.

[MdC01] Yun Mai and Michael de Champlain. A Pattern Language To Visitors. In *The 8th Annual Conference of Pattern Languages of Programs (PLoP 2001)*, September 2001.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall, 2 edition, 1997.

[MFMU05] Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT Support for Z Extensions. In *Fifth International Conference on Integrated Formal Methods*, number 3771 in Lecture Notes in Computer Science, pages 227—245. Springer-Verlag, 2005.

[MH00] Jeremy M. R. Martin and Yvonne Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. *Communicating Process Architectures*, 2000.

[Mil80] R. Milner. *A Calculus for Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Mil90] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1990.

[MK99] Jeff Magee and Jeff Krammer. *Concurrency: State Models & Java Programs*. Worldwide Series in Computer Science. Addison Wesley, April 1999.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[Mos04a] Peter Mosses. Modular Structural Operational Semantics (SOS). *Journal of Logic and Algebraic Programming*, 60-61:196—228, 2004.

[Mos04b] Peter Mosses. SOS and Variable Scope. e-mail discussion on the subject for using local scope for *Circus* in SOS., 2004.

[Mot97] Alexandre Mota. Formalization and Analysis of the SACI-1 micro satelite in CSP-Z. Master's thesis, Universidade Federal de Pernambuco, 1997. in Portuguese.

[Mot01] Alexandre Mota. *Model Checking CSP-Z: Techniques to Overcome State Explosion*. PhD thesis, Universidade Federal de Pernambuco, December 2001.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification*, volume 1. Springer-Verlag, 1992.

[MP95] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Safety*, volume 2. Springer-Verlag, 1995.

[MRB98] Robert Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design*, volume 3 of *Software Pattern Series*. Addison Wesley, June 1998.

[MS97] Irwin Meisels and Mark Saaltink. *Z/Eves 1.5 Reference Manual*. ORA Canada, September 1997. TR-97-5493-03d.

[MS01] Alexandre Mota and Augusto Sampaio. Model Checking CSP-Z. *Science of Computer Programming, Elsevier*, 40:59—96, 2001.

[MU05] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK*, pages 13—15. Springer-Verlag, April 2005.

[Nor97] Martin E. Nordberg III. Default and Extrinsic Visitor. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[NPW03] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[Nuk05] Gift Nuka. *Mechanisation for the Relational Calculus*. PhD thesis, University of Kent at Canterbury, 2005. To appear.

[OCW05] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying Theories in ProofPowerZ. draft, Univeristy of York, 2005.

[OGM04] Ourgrid UFCG, ourgrid.lsd.ufcg.edu.br/files/manual_2.1.pdf. *MyGrid User Manual (v2.1)*, 2004.

[OGP02] OurGrid Project Website. www.ourgrid.org, 2002.

[Oli03] Marcel Oliveira. *Circus*web-site. www.cs.york.ac.uk/circus, 2003.

[Oli06] Marcel Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, University of York, 2006. To appear in the beggining of 2006.

[OV96] Rimon Orni and Uzi Vishkin. Two Computer Systems Paradoxes: Serialize-to-Parallelize, and Queuing Concurrent-Writes. Technical report, University of Maryland, Department of Eletrical Engineering, 1996.

[Pan00] Z Standard Panel. Formal Specification, Z Notation, Syntax, Type and Semantics—Consensus Working Draft 2.6. Technical Report JTC1.22.45, BSI Panel IST/5/-/19/2 (Z Notation) and ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z), August 2000. www.cs.york.ac.uk/~ian/zstan/.

[PCB03] Daniel Paranhos, Walfredo Cirne, and Francisco Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing*, August 2003.

[Pel94] Doron Peled. Combining Partial Order Reductions with On-the-fly Model Checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377—390, London, UK, 1994. Springer-Verlag.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981. ISBN 0-13-661983-5.

[PNW] Petri Net Worlds web-site. www.daimi.au.dk/PetriNets/.

[PY96a] Atanas N. Parashkevov and Jay Yantchev. ARC—A Tool for Efficient Refinement and Equivalence Checking for CSP. In *IEEE* 2^{nd} *International Conference on Algorithms and Architectures for Parallel Processing ICA3PP*, 1996.

[PY96b] Atanas N. Parashkevov and Jay Yantchev. ARC—A Verification Tool for Concurrent Systems. In *Proceedings of the Third Australasian Parallel and Real-Time Conference.* Brisbane, Australia, 1996.

[QDC03] S. C. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation of TCOZ in Unifying Theory of Programming. In *Proceedings of Formalt Methods Europe, Pisa*, number 3582 in Lecture Notes in Computer Science. FME, Springer-Verlag, 2003.

[Qin04] QinetiQ Malvern Ltd. www.qinetiq.co.uk, 2004.

[RFA] RAISE FAQ web-ste. spd-web.terma.com/Projects/RAISE/faq.html.

[RGG+95] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. *First TACAS in LNCS Springer-Verlag*, 1019(1), 1995.

[Rie00] Dirk Riehle. *Framework Design: A Role Modeling Approach.* PhD thesis, Swiss Federal Institute of Technology Zurich, Universität Hamburg, 2000.

[Ris00] Linda Rising. *The Pattern Almanac 2000.* Software Design Pattern Series. Addison Wesley, May 2000.

[Ros94a] A. W. Roscoe, editor. *A Classical Mind: Essays in Honour of C. A. R. Hoare.* International Series in Computer Science. Prentice-Hall, 1994.

[Ros94b] A. W. Roscoe. *Model Checking CSP in [Ros94a]*, chapter 21, pages 353—378. Prentice-Hall, 1994.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* International Series in Computer Science. Prentice-Hall, 1997.

[RSR+01] Peter Ryan, Steve Schneider, Bill Roscoe, Michael Goldsmith, and Gave Lowe. *Modelling and Analysis of Security Protocols.* Addison Wesley, 2001.

[RTE] RAISE web-site at Terma. spd-web.terma.com/Projects/RAISE/.

[RUN] RAISE web-site at UNU/IIST. www.iist.unu.edu/raise/.

[Rus57] Bertrand Russell. *In Praise of Idleness.* Routledge Classics. Routledge & Karan Paul, 2^{nd} edition, 1957.

[Rus00] John Rushby. From Refutation to Verification. *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE XIII/PSTV XX)—Pisa Italy*, pages 369—374, October 2000.

[Rus04] Bertrand Russell. *Sceptical Essays*. Routledge Classics, 2004.

[Saa99a] Mark Saaltink. *Z/Eves 2.0 Mathematical Toolkit*. ORA Canada, October 1999. TR-99-5493-05b.

[Saa99b] Mark Saaltink. *Z/Eves 2.0 User's Guide*. ORA Canada, 1999. TR-99-5493-06a.

[Sca92] J. B. Scattergood. A Parser for CSP. Technical report, Oxford University, December 1992.

[Sca98] J. B. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, Oxford University, The Queen's College, 1998.

[Sch98] Steve Schneider. Security Properties and CSP. Technical report, Royal Holloway, University of London, 1998.

[Sch00] Steve Schneider. *Concurrent and Real-Time Systems, The CSP approach*. World Wide Series in Computer Science. John Wiley & Sons, 2000.

[Sch02] Steve Schneider. *The B-Method—An Introduction*. Palgrave, 2002. ISBN 0-33-79284-X.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An Eletronic Purse—Specification, Refinement, and Proof. Technical Report PRG-126, Oxford University, 2000.

[Sha] N. Shankar. Machine Assisted Specification and Verification. Lecture notes available at www.csl.sri.com/~shankar.

[Sha96] N. Shankar. Model Checking and Theorem Proving in PVS, Lecture 7 Slides, 1996. Lecture 7 slides at Marktoberdorf Summer School.

[Sha02] N. Shankar. Mechanised Verification Methodologies, August 2002. In Summer School in Specification, Verification, and Refinement—Turku Finland.

[Sil99] António Manuel Ferreira Rito Silva. *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, March 1999.

[SJ02] Adnan Sherif and He Jifeng. Toward a Time Model for *Circus*. In C. George and H. Miao, editors, *ICFEM 2002*, number 2495 in Lecture Notes in Computer Science, pages 613—624. Springer-Verlag, 2002.

[SM05] Mark Saaltink and Irwin Meisels. The Core Z/Eves API (DRAFT). Technical Report TR-99-5540-xxa, ORA Canada, September 2005.

[Smi00] Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[Spe05] Spec# web-site. research.microsoft.com/specsharp, 2005.

[Spi98] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1998.

[SSRB00] Douglas Schimidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Network Object*, volume 2 of *Software Design Patterns Series*. John Willey & Sons, 2000.

[ST02] Steve Schneider and Helen Treharne. CSP Theorems for Communicating B Machines. Technical Report CSD-TR-02-12, Royal Holloway University of London, www.cs.rhul.ac.uk/~helen, December 2002.

[ST04] Steve Schneider and Helen Treharne. Verifying Controlled Components. In Eerke Boiten, John Derick, and Graeme Smith, editors, *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 87—107. Springer, 2004.

[SWC02] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451—470. Springer-Verlag, 2002.

[Tan05] Xinbei Tang. *Mobile Processes in Circus*. PhD thesis, University of York, 2005. To appear.

[UM03] Mark Utting and Petra Malik. Community Z Tools. czt.sourceforge.net, 2003.

[UTS+03] Mark Utting, Ian Toyn, Jing Sun, Andrew Martin, Jin Song Dong, Nicholas Daley, and David Currie. ZML: XML Support for Standard Z. In *ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003. Proceedings*, pages 437—456. Springer-Verlag, 2003.

[Utt05] Mark Utting. *Jaza User Manual and Tutorial*. University of Waikato, The University of Waikato Hamilton, New Zealand, 2005.

[VCK96] John M. Vlissides, James O. Coplein, and Norman L. Kerth, editors. *Pattern Languages of Program Design*, volume 2 of *Software Pattern Series*. Addison Wesley, August 1996.

[VDM] VDM Archive web-site. www.csr.ncl.ac.uk/vdm/.

[vEVD89] P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[WB00] David Watt and Deryck F. Brown. *Programming Language Processors in Java: Compilers and Interpreters*. Prentice Hall, 2000.

[WC01a] J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[WC01b] J. C. P. Woodcock and A. L. C. Cavalcanti. The Steam Boiler in a Unified Theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC01)*, pages 291—298. IEEE Computer Society Press, December 2001.

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—203. Springer-Verlag, 2002.

[WC04] J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In E. A. Boiten, J. Derrick, and G. Smith, editors, *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40—66. Springer-Verlag, 2004. Invited tutorial.

[WCF05a] Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas. Animating a Simplified Operational Semantics for Model Checking *Circus* in Jaza. Appendix A.4.3 in [Fre05e] (CD-ROM), 2005.

[WCF05b] Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas. Operational Semantics for Model Checking *Circus*. In *Proceedings of Formal Methods Europe*, number 3582 in Lecture Notes in Computer Science, pages 237—252, 2005.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, 1996.

[Win87] Jannette Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Language and Systems*, 9(1):1—24, January 1987.

[Wit21] Ludwig Wittgenstein. *Tractatus Logico-Philosophicus*. Routledge Classics. Routledge & Karan Paul, 2^{nd} edition, 1921.

[Woo02] J. C. P. Woodcock. Unifying Theories of Parallel Programming. In *Logic and Algebra for Engineering Software*. IOS Press, 2002. Also Keynote speech in ICFEM 2002: 4th International Conference on Formal Engineering Methods, Shanghai, IEEE Computer Society Press.

[Woo03] Jim Woodcock. Dependable Systems Evolution. www.nesc.ac.uk, 2003.

[Woo05] Jim Woodock. Unifying theories of programming course module. University of York Undergraduate Course, 2005.

[Xav06] Manuela Xavier. *Circus* Type-checker. Master's thesis, Universidade Federal de Pernambuco, 2006. In preparation.