ALEXANDRE CABRAL MOTA

"Model Checking CSPz: Techniques to Overcome State Explosion"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADORES: AUGUSTO CESAR ALVES SAMPAIO
PAULO HENRIQUE MONTEIRO BORBA

RECIFE, DEZEMBRO/2001
Mota, Alexandre Cabral
XIII, 141 P. : IL., FIG., TAB.


Inclui bibliografia e apêndices.


005.14 CDD (22.ED.) MEI2007-043
Model Checking CSP\textsubscript{$Z$}: Techniques to Overcome State Explosion

Alexandre Mota

A thesis submitted for the degree of Doctor of Philosophy at the Universidade Federal de Pernambuco.

Supervisors: Augusto Sampaio  Paulo Borba

Recife: December, 2001
This thesis is dedicated to,

Adriana,
my parents, and
my whole family and God.
Acknowledgements

This thesis would not have been possible without much help, technical and otherwise, from my supervisors Augusto Sampaio and Paulo Borba. In particular, together with Ana Cavalcanti, they provided me with a valuable research environment.

For many valuable informal discussions about formal methods and other subjects I would like to thank Kelvin, Alexandre Macedo, Adnan Sherif, Luís Meneses, Márcio, Carlos Alexandre, Gledson and Nelson. And to Adalberto who has implemented our Java translator from CSP\(_Z\) to CSP\(_M\).

I would like to thank Bryan Scattergood, Ranko Lazić and Heike Wehrheim for sending me earlier drafts of their current researches on CSP\(_M\), data independence and data abstraction, respectively.

I am grateful to David Deharbe, David Naumann, Jim Woodcock and He Jifeng for fruitful suggestions and criticisms about our guided data abstraction approach.

To my father, Deca, who taughted me the art of self-learning, very important for facing a PhD program. And to my mother, Eliane, for showing me how resignation is powerful.

My deepest gratitude goes to my wife Adriana for her love and dedication. In special, she has taughted me the pleasure of life beyond formal methods.

The work described in this thesis has been financially supported by the Brazilian government through one of its agencies, CNPq, grant 143720/98-8.
Resumo

Hoje em dia, notamos um crescente interesse na técnica de verificação de modelos por acadêmicos e por profissionais da indústria. A razão disso talvez esteja na habilidade dos verificadores de modelos—implementações de algoritmos de verificação de modelos—serem totalmente automáticos: não há intervenção do usuário durante a análise. Com certeza, trata-se de uma vantagem ausente em provadores de teoremas, por razões teóricas. Obviamente, porém, uma limitação importante da verificação de modelos—o problema da explosão exponencial de estados—é só poder ser aplicada a sistemas finitos. Assim, verificação de modelos e prova de teoremas podem ser considerados técnicas ortogonais e complementares.

Uma outra tendência é o reuso formal de recursos disponíveis. Normalmente, consegue-se reuso pela integração formal de teorias e ferramentas bem difundidas. A razão é quase óbvia: obtém-se uma teoria e/ou ferramenta mais expressiva com menor esforço, quando comparado a desenvolver tal teoria e/ou ferramenta do nada. O ponto-chave nesta área de pesquisa é produtividade: conceito bem conhecido para profissionais da indústria. Por exemplo, integrar verificadores de modelos e provadores de teoremas está sendo visto como um esforço crucial para a análise de sistemas.

Nesta tese, nosso objetivo maior é mostrar como aplicar verificação de modelos para CSP\(_Z\). CSP\(_Z\) é uma linguagem formal que integra duas linguagens difundidas: a álgebra processos CSP e o formalismo baseado em modelos Z. Porém, em vez de criar uma técnica de verificação de modelos para CSP\(_Z\) do nada, nós seguimos a mesma abordagem usada em CSP\(_Z\): uma estratégia de verificação de modelos para CSP\(_Z\) onde reusa-se a técnica de verificação de modelos para CSP já existente. Basta mostrar como um processo CSP\(_Z\) pode ser visto como um em CSP.

Tendo a habilidade natural de lidar com tipos de dados infinitos, é normal CSP\(_Z\) criar sistemas infinitos. Isso evita a aplicação direta de verificação de modelos. Pesquisando soluções para esse problema, obtivemos duas contribuições distintas mas complementares. Caracterizamos como os processos infinitos de CSP\(_Z\) podem ser tratados aplicando duas abordagens diferentes. Uma é baseada no trabalho de Lazić que lida com a parte em CSP, fundamenta-se no conceito de independência de dados. A outra é baseada em interpretação abstrata e lida com a parte em Z. Além disso, nós encontramos uma maneira de mecanizar a abordagem baseada em interpretação abstrata, visto que os trabalhos na literatura requerem que o usuário proponha a abstração de dados.

Apesar da abordagem acima ser eficaz, o sistema abstraido ainda pode ser muito complexo (apesar de finito) para ser analisado por verificação de modelos. Então, fizemos um esforço adicional para adaptar uma abordagem manual para análise modular de deadlock—originalmente concebida para CSP. Mecanizamos parte dessa análise e a adaptação foi provada formalmente.

Finalmente, consideramos um sistema real como estudo de caso. Nós mostramos como este sistema pode ser analisado aplicando as abordagens investigadas e refinamento de dados para Z através de uma metodologia geral proposta. Enquanto as abordagens de refinamento de dados e abstração de dados mostraram-se promissoras no sentido de obter um sistema finito, a abordagem modular revelou um ganho impressionante comparado a uma análise global do mesmo sistema.

Palavras-chave. Verificação de modelos, especificação formal, prova de teoremas, análise.
Abstract

Nowadays, we can appreciate an increasing interest on the technique of model checking by academics as well as by professionals from the industry. Perhaps, the reason for this interest resides in the ability of model checkers—model checking algorithm implementations—being fully automatic: there is no user intervention during analysis. Indeed, this is a well-known theoretical limitation of theorem proving. Obviously, however, an important disadvantage of model checking—\textit{the exponential state explosion problem}—is that it can only be applied to finite systems. Thus, model checking and theorem proving can be considered orthogonal and complementary techniques.

Currently, another attractive trend is formally reusing available resources. This is normally achieved through the formal integration of well-accepted theories and tools. The reason is almost obvious: one can obtain a more expressive theory and tool with a minor effort when compared with developing such a theory and tool from scratch. The key point in this research area is productivity; a well-known and attractive concept for people in the industry. For instance, the integration of model checkers and theorem provers is being considered as a crucial trade-off for system analysis.

In this thesis, our main goal is to present how to perform model checking for CSP\_Z. CSP\_Z is a recent formal language which integrates two wide-spread languages: the process algebra CSP and the model-based language Z. However, instead of developing a model checking technique for CSP\_Z from scratch, we follow the same motivation underlying the creation of CSP\_Z: we propose a model checking strategy for CSP\_Z based on the existing model checking technique designed for CSP. This is accomplished by showing how a CSP\_Z process can be seen as a pure CSP process.

Due to the natural ability to handle infinite data types, CSP\_Z normally yields infinite systems which refrain the users from applying model checking directly. By investigating solutions for this problem, we have obtained two distinct but complementary contributions. We characterise how infinite CSP\_Z processes can be handled by applying two different approaches. One is based on Lazić’s work to handle the CSP part, based on the concept of data independence. The other is based on abstract interpretation and is used to handle the Z part. Further, we have found a way of mechanising the approach used to handle the Z part whereas work in the literature requires the user to propose the data abstraction.

Although the preceding approach seems to be very effective, the abstracted system can still be too complex (even being finite) to be handled by model checking. As an additional effort towards analysing such systems, we have adapted a manual modular approach for deadlock analysis—originally conceived for pure CSP—to deal mechanically with CSP\_Z processes. This adaptation has been formally proven.

Finally, we have considered a complex real system as case study. We show how this system can be analysed by applying the investigated approaches and data refinement for Z throughout a proposed overall methodology. While the data refinement and data abstraction approaches has shown promising in the direction of obtaining a finite system, the modular approach has revealed an impressive gain compared to a global analysis of the abstracted system.
Keywords. Model checking, formal specification, theorem proving, analysis.
# Contents

1 Introduction
   1.1 Specifying Behaviour and Data Structures ............................................. 5
   1.2 Overview .......................................................... 6

2 CSP\(_Z\)
   2.1 A Simple Example .................................................. 9
   2.2 The Syntax of CSP\(_Z\) .................................................. 12
      2.2.1 Channels ......................................................... 12
      2.2.2 Basic processes .................................................. 13
      2.2.3 Binary and unary operators ....................................... 14
      2.2.4 Multiprefix ...................................................... 15
      2.2.5 Recursion ......................................................... 16
      2.2.6 Modular CSP\(_Z\) .................................................. 16
   2.3 Semantics .......................................................... 17
      2.3.1 Denotational Semantics ........................................... 17
      2.3.2 The Semantics of the Z Part ...................................... 23
      2.3.3 Combining CSP and Z .............................................. 24
   2.4 Data Refinement ..................................................... 25

3 Model Checking CSP\(_Z\) ........................................... 27
   3.1 Requirements for Model Checking CSP\(_Z\) using FDR .............................. 27
   3.2 Introducing a State Space .............................................. 29
      3.2.1 Z Data Types Representation ..................................... 29
3.3 CSP Behaviour as a Function of the State Space ........................................... 30
3.4 Modelling the Z part as a CSP Process ...................................................... 31
   3.4.1 Optimisation ................................................................. 33
3.5 Combining the CSP and the Z Parts ............................................................ 34
   3.5.1 Dealing with Termination ....................................................... 36
3.6 The Transformation Preserves the CSP\textsubscript{Z} Semantics ......................... 36
3.7 Converting the WDT Process ................................................................. 42
3.8 Conclusions ......................................................................................... 44

4 CSP\textsubscript{Z} Data Abstraction .................................................................... 45
   4.1 Data Independence .............................................................................. 47
   4.2 Abstract Interpretation ...................................................................... 48
   4.3 CSP\textsubscript{Z} Data Abstraction ...................................................... 52
      4.3.1 Generating Optimal Data Abstraction ..................................... 61
      4.3.2 Algorithm .............................................................................. 67
   4.4 Conclusions ....................................................................................... 83

5 Local Analysis ............................................................................................. 85
   5.1 Local Deadlock Analysis ..................................................................... 85
   5.2 Extending the Modular Approach towards CSP\textsubscript{Z} ......................... 92
   5.3 Conclusions ....................................................................................... 95

6 The Overall Strategy and a Case Study ......................................................... 97
   6.1 The Overall Strategy .......................................................................... 97
      6.1.1 Refine the Z part .................................................................... 97
      6.1.2 Transform CSP\textsubscript{Z} into CSP\textsubscript{M} .................................... 98
      6.1.3 Apply the Guided Data Abstraction ........................................ 98
      6.1.4 Apply the modular strategy for deadlock analysis ...................... 98
      6.1.5 Workflow of the strategy .......................................................... 99
   6.2 Case Study: The SACI-1 Microsatellite ................................................. 99
CONTENTS

6.2.1 Data Types ......................................................... 100
6.2.2 Fault-Tolerant Router ............................................ 102
6.2.3 Telecommand ....................................................... 102
6.2.4 Telemetry ........................................................... 103
6.2.5 Asynchronous Recover ............................................. 104
6.2.6 OBC Clock .......................................................... 105
6.2.7 SACI-1 .............................................................. 106
6.3 Investigating the SACI-1 Network .................................. 106
   6.3.1 Investigating the Telemetry Process ............................ 106
   6.3.2 The Modular Deadlock Analysis ............................... 113
6.4 Conclusions ......................................................... 116

7 Conclusion ............................................................... 119
   7.1 Related Work ....................................................... 122
   7.2 Future Research .................................................... 123

A Elements of CSP\textsubscript{M} ........................................... 125
   A.1 Sets ................................................................. 125
   A.2 Sequences ......................................................... 126
   A.3 Booleans ........................................................... 126
   A.4 Extra ............................................................... 126
   A.5 Process Expressions ............................................... 127

B SACI-1 in CSP\textsubscript{M} .............................................. 129
   B.0.1 SCLOCK in CSP\textsubscript{M} .................................... 129
   B.0.2 Telecommand in CSP\textsubscript{M} ............................... 130
   B.0.3 Telemetry in CSP\textsubscript{M} .................................. 132
   B.0.4 A_R in CSP\textsubscript{M} ........................................ 133
   B.0.5 FTR in CSP\textsubscript{M} ........................................ 134
   B.0.6 SACI-1 in CSP\textsubscript{M} ..................................... 135
List of Figures

1.1 Relationship among Chapters .................................................. 6

2.1 Parallelism for Traces ............................................................ 21

4.1 Graphical Representation of a Simple Network ............................... 50
4.2 LTS of the Z part of $P_{Clock}$ ................................................. 63
4.3 LTS of the Z part of $P$ ............................................................ 66
4.4 LTS of an Unstable process ...................................................... 67
4.5 Lattice of the preconditions .................................................... 73

5.1 Graphical representation of a network of three processes ................. 87
5.2 Disconnecting edges of a network ............................................. 91
5.3 Conflict-freedom of linked processes via disconnecting edge .............. 91
5.4 Essential components of a network .......................................... 92

6.1 SACI-1 Architecture Description .............................................. 101
6.2 Disconnecting edges of the SACI-1 communicating graph .................. 115
6.3 Pairs of processes which must be conflict-free ............................. 115
6.4 Essential components which must be deadlock-free ........................ 116
List of Tables

2.1 Semantic functions from the Z standard ........................................... 18
6.1 Overall Strategy Results ................................................................. 113
6.2 Summary of results: Modular × Global Analysis ............................... 117
Chapter 1

Introduction

An important ingredient for producing high quality software is formal methods. They are particularly indicated for complex and critical systems development, where the high costs of constructing such systems or the necessity of ensuring safety of human life justify their use. Among the tasks involved in the application of formal methods, the specification of the required system and the analysis of the properties of this specification deserve special attention.

Specifying a system means—in general—elaborating an unambiguous, precise, and concise description of the system. It is worth pointing out that, to accomplish such a task, it is essential to choose the most appropriate specification language. Appropriateness is related to the ability the language has to capture the aspects—data structures, concurrency, real-time, etc—the system exhibits.

Specification languages are normally and originally designed to describe some specific aspects. Therefore, the formal integration of such languages—research area known as linking theories—is a much more desirable and valuable goal. Linking theories is a recent trend in the area of formal methods. The general purpose of this effort is to combine concepts and notations in a uniform framework which allows one to capture more than one aspect of a system. For example, concurrent specification languages, such as CSP [14, 10] or CCS [59], can characterise precisely the behavioural aspects of a system, whereas they are not suitable for modelling concisely (and abstractly) the system data structures. This is because the data structures in these languages correspond to those of a programming languages. Another viewpoint is that languages such as Z [47, 35], VDM [15] and OBJ [29] have great expressive power to describe abstract data structures, but lack the notion of operation evaluation order. Currently, there is a lot of language integration proposals. Some examples are LOTOS [65], Temporal Logic and CSP [69], LOTOS and Z [21], CSPZ and CSP$_O$Z [16, 17], and Circus [30].

This thesis is based on the specification language CSP$_Z$. This is a formal integrated language which combines the process algebra CSP [14, 10] with the model-based language Z [47, 35]. Consequently, CSP$_Z$ is able to describe concurrency and data structure aspects of a system in a
same environment. Furthermore, CSP \( Z \) was designed in such a way that it is possible to describe concurrency and data structures orthogonally. That is, one is allowed to view each aspect of a system separately. Beyond its orthogonal description capabilities, the approach to CSP \( Z \) refinement is compositional in the sense that refining the CSP or the Z part (with some constraints) leads to the refinement of the whole specification [16, 17].

Concerning the analysis of properties, the major techniques are normally classified as theorem proving [31] and model checking [22]. The sort of logic used to specify a property determines what technique might be used. For instance, theorem proving can deal with non-decidable logics (that is, first-order or high-order logic) whereas model checking can only work with decidable logics like some variants of the propositional logic [48]. For this reason, theorem proving allows checking infinite state systems while model checking is only able to consider finite ones. Therefore, model checking is useless when the system being investigated exhibits an enormous (or worst, an infinite) number of states. Such limitation is known as the exponential state explosion problem. For instance, applying a model checker to a system with this problem will consume all memory resources of the computer without giving any useful response.

Despite this limitation, there is an increasing interest on model checking by academics as well as by people from the industry. Perhaps, the reason of this interest resides in the ability of model checkers to analyse systems fully automatically, that is, without user intervention. Contrarily, in general, this is theoretically impossible for theorem provers [31].

Traditionally, model checking means verifying automatically the satisfaction relation \( M \models p \). That is, does the model \( M \) (described as a transition system) satisfy the property \( p \) (described as a logical formula)? [22]. However, this satisfaction can be captured in another way. Roscoe [9, 10] has observed that the refinement \( S_p \preceq S_M \)—where \( S_p \) and \( S_M \) are CSP specifications and \( S_p \) is known to exhibit the desired property \( p \)—provides an equivalent response to \( M \models p \); this model checking variant is more precisely called of refinement checking. For instance, the name of the model checker (or better, the refinement checker) for CSP is Failures-Divergences Refinement (or FDR [42], for short). It is worth noting that a CSP process is transformed—via its operational semantics—into a labelled transition system (or LTS, for short) with the purpose of refinement checking [10].

Mota and Sampaio [4] have proposed a strategy for model checking CSP \( Z \) specifications emphasising deadlock analysis. Some ideas were discussed on how to implement this strategy using the FDR model checker through its notation \( CSP_M \) [13]. An independent effort to refine the ideas introduced in Mota and Sampaio’s work [4] is reported in the work of Fischer and Wehrheim [18] whose main contribution is a more structured presentation of the conversion to \( CSP_M \). The work reported in Fischer and Wehrheim’s work [18] uses CSP \( OZ \) (the Object-Oriented version of CSP \( Z \)) instead of CSP \( Z \), although inheritance has not been explored. Mota and Sampaio [6] further developed the strategy proposed in [4], considering the representation of all CSP \( Z \) operators into the FDR notation. As in Fischer and Wehrheim’s work [18], we show how the Z part of a CSP \( Z \) specification can be characterised as a CSP process, parameterised by the state space, using some
conversion patterns. The behaviour of a CSP\(_Z\) specification is then captured by the parallel composition of the CSP part and a CSP process which characterises the Z part. In this way we can combine CSP\(_Z\) specifications as ordinary CSP processes.

A subtle and important aspect of this translation strategy is to consider the termination state of the CSP part of the CSP\(_Z\) specification. This is missing in the work of Fischer and Wehrheim [18]. Our treatment of input and output channels and operations are also slightly different from the one given in the work of Fischer and Wehrheim, which considers only single event channels and methods (operations), whereas here we further develop this aspect giving full treatment to complex event channels and operations. Finally, in this thesis we prove that this translation preserves the semantics of CSP\(_Z\).

However, since CSP\(_Z\) naturally yields systems with the exponential state explosion problem, due to the use of infinite data types through Z, a more fundamental question arises: can model checking be extended for infinite state systems in some way?

In this direction, various techniques have been proposed—and are currently being carefully studied—to handle more general systems: local analysis [43, 37], data independence [54, 58], symmetry elimination [22], partial order reduction [22], automatic testing [55, 62], abstract interpretation [51, 52], integration of model checkers with theorem provers [68, 34], etc. Although these research areas seem promising, full automatic analysis of infinite state systems is still almost unexplored [22, 34]. Actually, the most powerful abstraction techniques still need user expertise for complete and adequate usage [49, 22, 34, 40]. User intervention is needed for elaborating a sufficiently abstract model of the system so that model checking can be applied successfully. More importantly, in the current literature—to the best of our knowledge—one cannot yet find a guide to generate such abstractions.

Wehrheim [26] has proposed a way of model checking infinite CSP\(_{\Omega Z}\) processes through data abstraction using a technique based on abstract interpretation. For CSP alone, Lazić [58] has considered model checking infinite data independent CSP processes. Investigating these works, we have found a flaw in some results of Wehrheim, based on the work of Lazić. The point is that Wehrheim performs the data abstraction analysis on the Object-Z part alone, assuming that the CSP part will behave accordingly independent on the data abstraction performed. Following the work of Lazić, we saw that it is not valid in general. Thus, we have highlighted the problem and its solution by presenting and proving extensions of some theorems presented in the work of Wehrheim.

Besides solving this problem, we have further developed the subject by showing that it is possible to mechanise the work of Wehrheim [26]. Mechanisation of abstract interpretation based approaches is very desirable because the most difficult task, which is finding abstract domains, is proposed by the user [22, 52]. The reason to use Wehrheim’s work [26] instead of, for example the works of Cleaveland and Riely [55] or Loiseaux et al [19], is that it already uses a CSP algebraic style which is very convenient for FDR. Indeed, the work of Wehrheim [26] can be seen as a CSP view of the works of Cleaveland and Riely [55] and Loiseaux et al [19].
CHAPTER 1. INTRODUCTION

We argue that a data dependent infinite CSP\textsubscript{Z} process can be transformed into a finite CSP\textsubscript{Z} process by replacing its original infinite types with finite subtypes and rewriting the expressions based on these subtypes [7]. This data abstraction satisfies the criteria presented by Wehrheim [26] and has the advantage of being fully mechanised. Our approach is a step forward, since we calculate the data abstraction, based on the process description only. Notably, the most promising works on this research area assume some kind of data abstraction determined by the user [55, 19, 49, 63, 40]. Instead of considering a completely unrelated abstract type [55, 19], we assume a related abstract type [63, 40].

Our abstract types are the subtypes of the original ones whereas the works of Shankar et al [63] and Stahl et al [40] consider booleans as abstractions. The use of booleans, as abstractions, is very attractive since booleans are finite data types. However, the choice of what infinite predicates the booleans replace is difficult. Therefore, these works need an initial and convenient boolean abstraction—given by the user—in order to be successful.

On the other hand, by choosing subtypes we have the advantage of building them without user intervention. Our approach starts by considering the subtypes as the original types themselves. By an iterative process, our approach tries to partition the original types into equivalence classes. Hence, the abstraction consists in selecting some elements of each partition to represent the whole equivalence class with respect to the process behaviour. If the number of partitions found is finite, then we obtain a useful data abstraction for model checking. Otherwise, the abstraction belongs to the same class of infinite state systems and then it is not feasible for model checking.

As an additional alternative to tackle the state explosion problem, we show how the network decomposition strategy, developed for CSP by Brookes and Roscoe [61] for manual deadlock analysis, can be adapted for CSP\textsubscript{Z}, allowing the analysis of CSP\textsubscript{Z} specifications in a modular fashion. Furthermore, we have found a way of using FDR to support the analysis. This new result is presented and proven. We present the advantages of applying such a modular analysis in a quantitative basis, making a comparison between a global and local analysis for our case study (see below). The result is presented on a table and the outcome is impressive for this system.

Another direction for analysing deadlock, outside the scope of this thesis, is to assume that the CSP\textsubscript{Z} specification was built using deadlock-free configurations patterns or it has special configurations (that is, rings, client-server, etc), as shown by Cunha and Justo [53] and by Martin [32].

Our case study is a CSP\textsubscript{Z} specification of a subset of the On-Board Computer (OBC) of a Brazilian artificial microsatellite (SACI-1) [28] developed by the Brazilian Space Research Institute (INPE). Using this case study we present an overall strategy for model checking by considering data refinement aspects, the mechanised data abstraction, and the modular analysis of deadlock.
1.1 Specifying Behaviour and Data Structures

In this section we briefly present alternative languages for specifying behavioural and data structure aspects of a system. In what follows, we intend to show why CSP\(_Z\) is an interesting language for specifying such systems.

Among the languages concerned only with data structure aspects, we can find \(Z\) \([47, 35]\), VDM \([15]\), OBJ \([29]\), and Abstract State Machines \([67]\) (ASM) among others. Although all these languages are directly related to model data structures, \(Z\), VDM, and ASM use the so-called model-based approach where mathematical objects (for example: sets, relations, etc) form the basis for specifying a system. On the other hand, OBJ applies the algebraic style aiming at modelling data structures by describing existing properties among them through axioms and equations. None of these languages, however, have specific constructs to describe behavioural aspects, such as: choice, parallelism, sequence, etc. Thus, these languages specify behavioural aspects in a rather difficult and implicit manner. As illustrated by Hayes \([27]\), \(Z\) has already been used to describe concurrent systems; however, it is worth noting that such a description is not so clear as it would be if concurrent constructs existed.

The other category of languages is primarily related to model behavioural aspects of a system. In this category we can find Petri Nets \([66]\), CSP \([14, 10]\), and CCS \([59]\). The main characteristic of all these languages is that the data structures supported are those of programming languages. Thus, while they have behaviour constructs which ease the description of concurrent and distributed systems, they cannot describe abstract data structures, easily.

A crucial difference between these languages is in the reasoning about process equivalence. Although the processes modelled by these languages can be interpreted as a directed graph, or an \(LTS\), CSP deals with process equivalence in a different way. In CSP, the main way of deciding process equivalence is via refinement relations based on traces, failures, and divergences, which are not primarily based on transition systems. On the other hand, CCS and Petri Nets take the approach of defining the meaning of a process to be an \(LTS\) and the equivalence to be given in terms of equivalence between \(LTSs\), via a bisimulation relation. It is worth noting that the bisimulation relation only deals with the events a process can do, while the refinement approach also considers non-admissible events and divergent ones.

The last class of specification languages we consider in this section is that of combined theories. That is, the languages in this class deal with more than one aspect of a system at the same time. The well-known language LOTOS \([65]\) is the basic example of such a language. LOTOS is a combination of the specification languages CCS and ACT ONE \([25]\). Thus, LOTOS can model behavioural aspects with CCS, and data structures algebraically with ACT ONE. Another examples in this category are CSP\(_Z\) \([16, 17]\), CSP\(_OZ\) \([17]\), and Circus \([30]\). All these languages deal with behaviour and data structures orthogonally, differing only in the way equivalence between processes is handled.
CHAPTER 1. INTRODUCTION

The main reason to choose CSP\textsubscript{Z}, instead of LOTOS resides on the theory of refinement underlying CSP and Z. With CSP\textsubscript{Z}, refining the behavioural part of a specification leads to the refinement of the whole specification, and refining the data structure part also leads to the refinement of the whole, provided some constraints are satisfied. Besides, the theory of refinement for CSP allows us to consider what a process can do, what it cannot do, and divergence aspects; while in LOTOS, the bisimulation relation of CCS is used. Since we are not planning to use object orientation, CSP\textsubscript{Z} seems more appropriate than CSP\textsubscript{OZ}. Finally, like CSP\textsubscript{Z} and CSP\textsubscript{OZ}, Circus is an attractive language based on a combination of CSP and Z, as well as on the theory of refinement for CSP and Z, but it is too recent to be considered in the present work.

1.2 Overview

![Figure 1.1: Relationship among Chapters](image_url)

This thesis has a modular organisation. Each chapter comprehends a self-contained module. Figure 1.1 illustrates the relationship among the main modules. Chapter 2 introduces the CSP\textsubscript{Z} language using an example, and subsequently providing its syntax and semantics as defined in [16, 17]. This chapter forms the basis for the subsequent chapters.

In Chapter 3 we describe a strategy for model checking CSP\textsubscript{Z} using the model checker FDR via a translation of CSP\textsubscript{Z} into CSP\textsubscript{M}. This translation is proven valid by Theorem 3.6.1. The format of the resulting CSP\textsubscript{M} process has revealed an interesting way of avoiding state explosion by separating data independent behaviour from data dependent one. This chapter is the main focus.
of this thesis as we are primarily interested in model checking CSP\textsubscript{Z}. However, as CSP\textsubscript{Z} deals with infinite data types in its channels and state space, we have to consider strategies to minimise and avoid the state explosion problem. That is the reason why we have included the techniques to avoid state explosion into the scope of this module (see Figure 1.1). Normally, one has to abstract a CSP\textsubscript{Z} specification in some way before performing model checking.

After characterising how model checking can be performed in the CSP\textsubscript{Z} setting, we consider in Chapter 4 our first two contributions regarding avoidance of state explosion by data abstraction. Firstly, we explain why some results of Wehrheim [26] are not valid in general and prove corrected versions—Lemmas 4.3.2 and 4.3.3—of these results. Further, we show that we can mechanise the data abstraction approach. We present an algorithm and prove its correctness in Theorem 4.3.3.

A modular approach to deadlock analysis of CSP specifications and its extension to handle CSP\textsubscript{Z} specifications are presented in Chapter 5. Our goal in this chapter is to show how the building blocks of the modular deadlock analysis can be determined with the aid of FDR—Lemmas 5.2.2 and 5.2.3.

Chapter 6 presents an overall strategy which integrates the results described in the previous chapters. Such a strategy has emerged as a result of our needs to perform some kind of abstraction in our case study: the SACI-1 OBC. We have concluded that our experience should be stated to provide material for subsequent investigation in model checking CSP\textsubscript{Z} processes. Another contribution of this chapter is characterising the biggest steps and their dependencies for the purpose of tool development and integration.

Finally, in Chapter 7, we consider what are the benefits of using an integrated language and the practical advantages and limitations of using FDR in this setting. We also present what still needs to be done as future research.
Chapter 2

CSP$_Z$

CSP$_Z$ is a formal integrated language which combines the process algebra CSP [14, 10] with the model-based language Z [47, 35]. Consequently, CSP$_Z$ is able to describe concurrency and data structures aspects of a system in a same environment. Further, CSP$_Z$ was designed in such a way that it is possible to describe concurrency and data structures orthogonally. That is, one is allowed to view each aspect of a system separately. Beyond its orthogonal description capabilities, CSP$_Z$ can also be refined independently, that is, the approach to refinement is compositional in the sense that refining the CSP or the Z part (with some constraints) leads to the refinement of the whole specification [16, 17].

This chapter is organised as follows. Firstly, we present an overview of CSP$_Z$ through an example which is part of the specification of our case study, fully described in Section 6.2. In Section 2.2 we present the syntax of CSP$_Z$. Its semantics is finally introduced in Section 2.3. Finally, some considerations about data refinement are discussed in Section 2.4.

2.1 A Simple Example

In this section we present a CSP$_Z$ specification of a simple process of the On-Board Computer (OBC) of a Brazilian artificial microsatellite (SACI-1) [28] developed by the Brazilian Space Research Institute (INPE) (see Section 6.2 for further details). The Watch-Dog Timer, or simply WDT, is responsible for waiting periodic reset signals that come from another OBC process, the Fault-Tolerant Router (FTR). If such a reset signal does not come, the WDT sends a recovery signal to the FTR asynchronously, to normalise the situation. This procedure is repeated three times; if, after that, the FTR still does not respond, then the WDT considers the FTR faulty, finishing its operation successfully.

A CSP$_Z$ specification (or modular process) is encapsulated into a spec and end_spec scope, where the name of the specification follows these keywords. The interface is the first part of a CSP$_Z$
specification and is used to declare the external channels (keyword chan) and the local (or hidden) ones (keyword lchan). Each list of communicating channels has an associated type. The general form of such type is \([v_1 : T_1; \ldots ; v_n : T_n]\) where \(v_1, \ldots, v_n\) are lists of variables and \(T_1, \ldots, T_n\) their respective types. This is actually a simplification of schema types in Z where a predicate \(P\) can be used to constrain the values that can be assigned to the variables. Channels without an associated type are simply events as conventioned by the CSP tradition. The types \(T_1, \ldots, T_n\) could be built-in or user-defined types; in the latter case, they can be declared outside the spec and end.spec scope, as illustrated by the given-set CLK introduced below and used to build the type of the channel clockWDT.

\[
\text{[CLK]}
\]

The WDT interface includes a communicating channel clockWDT (it can send or receive data of type CLK, which represents moments in time, through the variable clk), two external or visible events (reset and recover), and four internal or hidden events (timeOut, noTimeOut, failFTR, and offWDT).

\begin{verbatim}
spec WDT
  chan clockWDT: [clk : CLK]
  chan reset, recover
  lchan timeOut, noTimeOut, failFTR, offWDT
\end{verbatim}

The concurrent behaviour of a CSP\(_Z\) specification is introduced by the keyword main, where other equations can be added to obtain a more structured description: a hierarchy of processes. The equation main describes the WDT behaviour in terms of a parallel composition of two other processes, Signal and Verify, which synchronise in the event offWDT. The process Signal waits for consecutive reset signals (coming from the FTR process, described in Section 6.2) or synchronises with Verify (through the event offWDT) when the FTR goes down. The process Verify waits for a clock period, then checks whether a reset signal arrived at the right period or not via the choice operator (\(\sqcup\)). If a timeOut occurs then the WDT tries to send, at most for three times, a recovery signal to the FTR through the process A_R whose only purpose is to implement an asynchronous communication between the WDT and the FTR (see Section 6.2). If the FTR is not ready to synchronise in this event, after the third attempt, then Verify assumes that the FTR is faulty (enabling failFTR) and then synchronises with Signal (at offWDT), in which case both terminate (behaving like SKIP). From the viewpoint of the SACI-1 project, the WDT is turned off because it cannot restart (recover) the FTR anymore.

\begin{verbatim}
main=Signal \| Verify
  \{offWDT\}
Signal=(reset→Signal \sqcup offWDT→SKIP)
Verify=(clockWDT?clk→(noTimeOut→Verify
\end{verbatim}
The Z part complements the main equation by means of a state-space and operations which define the state change produced by each CSP event. The system state \( (\text{State}) \) has simply a declarative part recording the number of cycles that the WDT tries to recover the FTR, and the value of the last clock received. The initialisation schema \( (\text{Init}) \) asserts that the number of cycles begins at zero; prime (‘) variables are used to describe the resulting state. To enumerate the cycles allowed, the constant set \( \text{LENGTH} \) is added to be used in the declarative part of the state-space.

\[
\text{LENGTH} \equiv 0 \ldots 3 \\
\text{State} \equiv [\text{cycles} : \text{LENGTH}; \text{time} : \text{CLK}] \\
\text{Init} \equiv [\text{State}' | \text{cycles}' = 0]
\]

To fix a time out period we introduce the constant \( \text{WDTtOut} \) of type \( \text{CLK} \). To find out whether the current time of the clock is a time out, we introduce the constant \( \text{WDTP} \) which is a relation used to express when one element of \( \text{CLK} \) is a multiple of another.

\[
\text{WDTtOut} : \text{CLK} \\
\text{WDTP} : \text{CLK} \leftrightarrow \text{CLK}
\]

The following operations are defined as standard Z schemas (with a declaration part and a predicate which constrains the values of the declared variables) whose names originate from the channel names, prefixing the keyword \( \text{com}_c \). Informally, the meaning of a CSP\_Z specification is that, when a CSP event \( c \) occurs the respective Z operation \( \text{com}_c \) is executed, possibly changing the data structures. Further, when a given channel has no associated schema, this means that no change of state occurs. For events with an associated non-empty schema type, the Z schema must have input and/or output variables with corresponding names in order to exchange communicated values between the CSP and the Z parts. Hence, the input variable \( \text{clk}？ \) (in the schema \( \text{com}_c \text{clockWDT} \) below) receives values communicated through the \( \text{clockWDT} \) channel (with type \( [\text{clk} : \text{CLK}] \)). For schemas where prime variables are omitted, we assume that no modification occurs in the corresponding component; for instance, in the schema \( \text{com}_c \text{reset} \) below it is implicit that the time component is not modified \( (\text{time}' = \text{time}) \).

\[
\text{com}_c \text{reset} \equiv [\Delta \text{State} | \text{cycles}' = 0] \\
\text{com}_c \text{clockWDT} \equiv [\Delta \text{State}; \text{clk}？ : \text{CLK} | \text{time}' = \text{clk}?] \\
\]

Note that the precondition of the schema \( \text{com}_c \text{noTimeOut} \) below is given by the predicate \( \neg \text{WDTP}(\text{time}, \text{WDTtOut}) \), meaning that the current time is not a multiple of the time out constant, and therefore the time out has not yet occurred; complementarily, the precondition of \( \text{com}_c \text{timeOut} \) specifies the occurrence of a time out.
CHAPTER 2. CSP

\[
\text{com\_noTimeOut} \triangleq [\Xi \text{State} \mid \neg \text{WDTP}(\text{time}, \text{WDTtOut})]
\]

\[
\text{com\_timeOut} \triangleq [\Xi \text{State} \mid \text{WDTP}(\text{time}, \text{WDTtOut})]
\]

As already explained, the recovery procedure is attempted for 3 times, after which the WDT assumes that the FTR is faulty. This forces the occurrence of failFTR and then turns off the WDT process.

\[
\text{com\_recover} \triangleq [\Delta \text{State} \mid \text{cycles} < 3 \land \text{cycles}' = \text{cycles} + 1]
\]

\[
\text{com\_failFTR} \triangleq [\Xi \text{State} \mid \text{cycles} = 3]
\]

end_spec WDT

2.2 The Syntax of CSP\(_Z\)

In this section we present the syntax of CSP\(_Z\) according to [16, 17]. Since the syntax of CSP\(_Z\) is quite long we have modularised it into various related subsections in a top-down view.

The syntax of CSP\(_Z\) is an extension of the standard Z syntax with three new paragraphs: channel declarations and processes.

\[
\text{Paragraph}_C := \text{Paragraph}_Z \\
| \text{Channel} \\
| \text{Process} \\
| \text{ModularProc}
\]

A specification is then a combination of such paragraphs: \(\text{Paragraph}_C^*\). The term \(\text{Paragraph}_Z\) represents Z standard paragraphs which are described in [47, 35]. Here we concentrate on the new constructs \(\text{Channel}\) and \(\text{Process}\). Furthermore, the considerations about the term \(\text{ModularProc}\) will be left for Section 2.2.6 because it is a combination of \(\text{Paragraph}_Z\), \(\text{Channel}\) and \(\text{Process}\).

2.2.1 Channels

A channel declaration is introduced by the keyword \texttt{chan} followed by a name and an optional expression (denoting its type). Its grammar rule is given by

\[
\text{Channel} ::= \text{chan} \ \text{DeclNameList}[; \ \text{Expr}]
\]

where \(\text{DeclNameList}\) is just a comma separated list of names and \(\text{Expr}\) is a standard Z expression.

Channel declarations are like a distributed free type definition. This type is assumed to be \(\text{EVENTS}\) and is available (as the integer type \(\mathbb{Z}\)) in every CSP\(_Z\) specification.

A process is defined as
2.2. THE SYNTAX OF CSP_Z

\[
\text{Process ::= DefLHS = Expr}
\]

The term \textit{DefLHS} represents the process name together with optional parameters.

\[
\text{DefLHS ::= ProcName[(SchemaText)]}
\]

The \textit{SchemaText} is a standard Z declaration and a predicate restricting the possible values.

\textbf{Example 2.2.1 (Use of Parameters)}

Let Buffer be a process that reads integers on a channel \textit{in} and writes them through a channel \textit{out}. Further, Buffer maintain an infinite sequence of integers which can only be removed if the sequence is not empty.

\[
\text{chan in, out : [n : Z]}
\]
\[
\text{Buffer(⟨⟩) = in?x → Buffer(⟨x⟩)}
\]
\[
\text{Buffer(⟨y⟩ □ S) = in?x → Buffer(⟨y⟩ □ S □ ⟨x⟩) □ out!y → Buffer(S)}
\]

After giving a name to a process using \textit{DefLHS} we need to build a process body. This is accomplished with the term \textit{Expr}. It is an extension of the standard Z grammar for expressions with a new nonterminal for process expressions. Its type must be \textit{PROCESS}, which is assumed to be in the scope of every CSP_Z specification like \textit{EVENTS}.

\[
\text{Expr ::= if Predicate then Expr else Expr}
\]
\[
\quad | \quad \text{ProcExpr}
\]

Since the definition of \textit{ProcExpr} is quite lengthy, it will be explained step by step in the following.

\textbf{2.2.2 Basic processes}

There are four basic processes:

\[
\text{ProcExpr ::= STOP | SKIP | Chaos(Expr) | div}
\]

The process \textit{STOP} represents a broken process (it cannot do any step). \textit{SKIP} means successful termination (this is achieved via performing a special √ action and then behaving like \textit{STOP}). \textit{Chaos(E)} can engage in arbitrary traces on the set of events \textit{E} or deadlock at any time (non-deterministically). Finally, the process \textit{div} diverges immediately (it engages in an infinite loop of invisible actions).
2.2.3 Binary and unary operators

The definition of binary and unary operators is also standard.

\[ ProcExpr ::= ChanExpr \rightarrow ProcExpr \quad \text{– prefixing} \]
\[ \quad | \quad ProcExpr \Box ProcExpr \quad \text{– external choice} \]
\[ \quad | \quad ProcExpr \cap ProcExpr \quad \text{– internal choice} \]
\[ \quad | \quad ProcExpr \triangleleft ProcExpr \quad \text{– sequential composition} \]
\[ \quad | \quad ProcExpr \mid| ProcExpr \quad \text{– generalised parallel} \]
\[ \quad | \quad ProcExpr \mid|\mid Expr \quad \text{– alphabetised parallel} \]
\[ \quad | \quad ProcExpr \mid| Expr SetExpr \quad \text{– interleaving} \]
\[ \quad | \quad ProcExpr[Expr]ProcExpr \quad \text{– linked parallel} \]
\[ \quad | \quad ProcExpr \setminus Expr \quad \text{– interrupt} \]
\[ \quad | \quad ProcExpr[Renaming] \quad \text{– renaming} \]

For prefixing \( e \rightarrow P \), when \( e \) occurs the process behaves like \( P[e] \), where \( P[e] \) means that all bindings created in \( e \) are used in \( P \). External and internal choice as well as interleaving and sequential composition are the same as is known from Hoare’s CSP [14]. In the generalised parallel composition \( P \mid| Q \), the processes \( P \) and \( Q \) run in parallel and synchronise on the set of events \( E \).

A renaming \( P[\text{new}_1/\text{old}_1, \ldots, \text{new}_n/\text{old}_n] \) substitutes events \( \text{old}_i \) to \( \text{new}_i \) and has the same syntax as in \( Z \); further, a relation can also be used.

\( \quad \text{Renaming ::= Word}/\text{Word}, \ldots, \text{Word}/\text{Word} \)
\[ \quad | \quad \text{RelExpr} \]

The linked parallel operator was introduced for FDR. It is a combination of parallel composition, renaming and hiding, generalising Hoare’s piping operator \( \gg \) [14]. For instance, suppose that \( P = \text{in}?x \rightarrow \text{out}!x \rightarrow P \) then \( P \gg P \) is given by

\[ (P[\text{mid}/\text{out}]) \mid| P[\text{in}/\text{mid}]) \setminus \{ | \text{mid} | \} \]
\[ \quad \{ | \text{in}, \text{mid} | \} \quad \{ | \text{mid}, \text{out} | \} \]

which simplifies, by renaming, to

\[ (P_{\text{in}}) \mid| P_{\text{out}}) \setminus \{ | \text{mid} | \} \]
\[ \quad \{ | \text{in}, \text{mid} | \} \quad \{ | \text{mid}, \text{out} | \} \]
2.2. THE SYNTAX OF CSP

where \( P_{in} = in \? x \rightarrow mid!x \rightarrow P_{in} \) and \( P_{out} = mid?x \rightarrow out!x \rightarrow P_{out} \). Expanding the parallelism we get

\[
P_{mid} \setminus \{ mid \}
\]

where \( P_{mid} = in \? x \rightarrow mid!x \rightarrow out!x \rightarrow P_{mid} \). Note that \( P_{mid} \) is keeping a copy of the value of \( x \) in the intermediate channel \( mid \). After hiding \( mid \) we have

\( P \)

That is, the piping mechanism is useful to keep intermediate values safe from external interference via the hiding abstraction operator (\). We model interruptions, \( P \triangle Q \) is used. It behaves like \( P \) until \( Q \) engages in some event. The hiding operator \( P \setminus E \) hides any event from the set of events \( E \).

Indexed operators

The following definitions are simply the indexed extensions of the previous operators.

\[
\text{ProcExpr ::= } \sqcap \text{SchemaText} \cdot \text{ProcExpr} \quad \text{– internal choice} \\
| \sqcup \text{SchemaText} \cdot \text{ProcExpr} \quad \text{– external choice} \\
| || [Expr]\text{SchemaText} \cdot \text{ProcExpr} \quad \text{– generalised parallel} \\
| || \text{SchemaText} \cdot [Expr]\text{ProcExpr} \quad \text{– alphabetised parallel} \\
| ||| \text{SchemaText} \cdot \text{ProcExpr} \quad \text{– interleaving} \\
| |||\text{SchemaText} \cdot \text{ProcExpr} \quad \text{– sequential composition} \\
| [Expr]\text{SchemaText} \cdot \text{ProcExpr} \quad \text{– linked parallel}
\]

2.2.4 Multiprefix

Multiprefixing is a construct taken from CSP\(_M\). The process \( c?x!(a + 1)?y!f(b)!d \rightarrow P \) receives values for \( x \) and \( y \) on the channel \( c \) and sends the values of the expressions \( (a + 1) \), \( f(b) \) and \( d \) through the same channel, and then behaves like \( P \). The actions are performed as a single event.

In CSP\(_M\), the input variable can be restricted in terms of the values they might assume. For example, \( P = in\?x : 1..10 \rightarrow STOP \) can only accept values within 1..10, blocking (refusing) those outside, like \( in\?12 \).

The precise syntax is as follows. Note that the schema text in a field expression must only introduce a single variable.

\[
\text{ProcExpr ::= } \text{Expr FieldExprList} \rightarrow \text{ProcExpr} \quad \text{– multiprefix} \\
\text{FieldExprList ::= } \text{FieldExpr}[\text{FieldExprList}] \\
\text{FieldExpr ::= } ?\text{Word} \\
| ?\text{SchemaText} \\
| !\text{Expr} \\
| !\text{SchemaText}
\]
2.2.5 Recursion

Recursion in CSP\(_{Z}\) is only introduced by a call to another process name instead of the \(\mu\) operator [14, 10]. Therefore, instead of writing

\[\mu P.a \rightarrow b \rightarrow P\]

we write

\[P = a \rightarrow b \rightarrow P\]

Its definition is given by

\[ProcExpr ::= ProcName([ZExpr])\]  – reference

The parameter \(ZExpr\) denotes the concrete parameters (see the right hand side of the process \(Buffer\) in Example 2.2.1). It is worth noting that processes can be referenced before use.

2.2.6 Modular CSP\(_{Z}\)

Besides introducing CSP\(_{Z}\) specifications by simple process expressions, we can work with modular structures. This is achieved by viewing a CSP\(_{Z}\) process as a combination of a CSP and a Z part. This kind of module was firstly used in [16] and also defined for CSP\(_{OZ}\) [17]. Its meaning is given in terms of the simple process expressions. Then, a modular CSP\(_{Z}\) process (CSP\(_{Z}\) process for short) is defined as

\[ModularProc ::= spec\ DefLHS\]

\[\begin{aligned}
Interface & = ProcExpr \\
main & = ProcExpr \\
Process^* & = SchemaExpr \\
State & = SchemaExpr \\
Init & = SchemaExpr \\
(com\_DeclName & = SchemaExpr^*)^* \\
end\_spec
\end{aligned}\]

Note that smaller pieces of CSP\(_{Z}\) syntax are used together to represent a whole CSP\(_{Z}\) module. A CSP\(_{Z}\) module is identified by the keywords \(spec/end\_spec\) which are used to delimit a scope. From this definition we simply need to further define \(Interface\) and explain the whole structure.

An interface is just a (possibly empty) list of channel declarations. Therefore,

\[Interface ::= Channel^*\]  – Global channels

\[LChannel^*\]  – Local channels

It is worth noting that we already saw the definition of \(Channel\) in Section 2.2.1. However, there is a new construct \((LChannel)\) which is now defined
2.3. SEMANTICS

$LChannel ::= lchan \textbf{DeclNameList}[: \textbf{Expr}]$

This is a new possibility of working with channels allowed by the modular structure. Its meaning is very simple: while it is visible inside the \texttt{spec/end\_spec} scope, it is invisible (hidden) from outside.

The keyword \texttt{main} denotes a special process which represents the initial process of the CSP part. The term \texttt{Process} allows the definition of various processes inside a \texttt{spec/end\_spec}. This is generally used to give a more hierarchical view of the CSP part. The keyword \texttt{State} denotes the state space of the Z part whereas \texttt{Init} is its initialisation. Finally, we have the (possibly empty) list of schema operations $(\texttt{com\_DeclName} \equiv \texttt{SchemaExpr})^*$.

Once we have described all elements of a CSP \texttt{Z} module we now describe its meaning in terms of the simpler CSP \texttt{Z} constructs which will be semantically defined in the following section.

This completes our considerations about the CSP \texttt{Z} syntax. For further details, please refer to Fischer [16, 17].

2.3 Semantics

We present here only the denotational semantics of CSP \texttt{Z} (see Fischer [16, 17] for the complete definition). Table 2.1 summarises the main symbols from the Z semantics used to give meaning of the CSP \texttt{Z} expression part.

2.3.1 Denotational Semantics

Normally, the denotational semantics of a CSP-based language is given in terms of the following sets: traces, failures, and divergences. In this section we present the denotational semantics of CSP \texttt{Z} using these sets.

**Definition 2.3.1 (Trace, Refusal, and Failure)**

\[ \Sigma \text{ be an alphabet. Then} \]

- \[ \text{Trace} == \text{seq}\Sigma \cup \{ t : \text{seq}\Sigma \cdot t \prec \{\sqrt{\}} \} \]
- \[ \text{Refusal} == \mathbb{P}(\Sigma \cup \{\sqrt{\}}) \]
- \[ \text{Failure} == \text{Trace} \times \text{Refusal} \]

And the semantics of each process is given by the following schema

\[
\begin{array}{l}
\textbf{ProcType} \\
\textbf{F} : \mathbb{P} \text{ Failure} \\
\textbf{D} : \mathbb{P} \text{ Trace}
\end{array}
\]
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{U}$</td>
<td></td>
<td>The Z semantics is based on the semantic universe $\mathbb{U}$ that provides semantic values for all Z values.</td>
</tr>
<tr>
<td>$\mathcal{W}$</td>
<td></td>
<td>The set $\mathcal{W}$ is a subset of $\mathbb{U}$ and it contains all semantic values for Z expressions.</td>
</tr>
<tr>
<td>$Model$</td>
<td>$Word \rightarrow \mathbb{U}$</td>
<td>A model is a finite association between names and values.</td>
</tr>
<tr>
<td>$Binding$</td>
<td>$Word \rightarrow \mathcal{W}$</td>
<td>A binding is a finite association between names and values of Z expressions.</td>
</tr>
<tr>
<td>$[\cdot]^P$</td>
<td>$Predicate \rightarrow \mathcal{P}Model$</td>
<td>It defines the meaning of Z predicates. The value of $[p]^P$ is the set of all models in which predicate $p$ is true.</td>
</tr>
<tr>
<td>$[\cdot]^e$</td>
<td>$Expr \rightarrow (Model \rightarrow \mathcal{W})$</td>
<td>The function $[\cdot]^e$ defines the meaning of Z expressions. The value of $[e]^e$ is a function computing the semantic value of $e$ in a given model.</td>
</tr>
<tr>
<td>$[\cdot]^T$</td>
<td>$Expr \rightarrow (Model \rightarrow \mathcal{P} \mathbb{U})$</td>
<td>The function $[\cdot]^T$ defines the semantics of a type. A type $t$ is mapped to the carrier set in a given model.</td>
</tr>
<tr>
<td>$\oplus$</td>
<td></td>
<td>For a given binding $b$ and a model $M$, the term $M \oplus b$ is the overriding of $M$ by $b$. Thus $$(M \oplus b)(n) = \begin{cases} M(n), \text{ for } n \notin \text{dom}(b) \ b(n), \text{ otherwise} \end{cases}$$</td>
</tr>
</tbody>
</table>

Table 2.1: Semantic functions from the Z standard
where $F$ captures the set of failures and $D$ the set of divergences.

**Definition 2.3.2 (Denotational CSP$_Z$ semantics)**

The following functions yield the different aspects of the semantics of process expressions.

- $F : \text{ProcExpr} \rightarrow (\text{Model} \rightarrow \mathcal{P} \text{Failure})$ (Failures)
- $D : \text{ProcExpr} \rightarrow (\text{Model} \rightarrow \mathcal{P} \text{Trace})$ (Divergences)

The function $[\cdot]^D : \text{ProcExpr} \rightarrow (\text{Model} \rightarrow \text{ProcType})$ gives the denotational semantics for process expressions. For a process expression $P$ and a model $M$, we have $M \in \text{dom}[P]^D$ iff $x \in \text{dom}M$ for all free variables and process identifiers $x$ used in $P$. We assume for all process expressions $P$ and models $M$, that

$[P]^D(M) = \text{Sem}_{\text{CSP}_Z}$

holds.

The function $T : \text{ProcExpr} \rightarrow (\text{Model} \rightarrow \mathcal{P} \text{Trace})$ computes the traces of a process expression. For a process expression $p$ and a model $M$, we have

$T[p](M) = \{ t : \text{Trace}; X : \text{Refusal} \mid (t, X) \in F[p](M) \bullet t \}$

Rather than defining $[\cdot]^D$ directly, we define the usual functions $F$ and $D$. For a detailed explanation about the following definitions please refer to Roscoe [10].

Let $M$ be a model. We use the abbreviation $\Sigma = M(\text{EVENTS})$.

The process $\text{Stop}$ can refuse every event initially and it cannot diverge

$F[\text{Stop}](M) = \{ X : \text{Refusal} \bullet (\langle \rangle, X) \}$
$D[\text{Stop}](M) = \emptyset$

whereas $\text{Skip}$ can initially refuse everything except $\sqrt{\cdot}$. After termination it behaves like $\text{Stop}$.

$F[\text{Skip}](M) = \{ X : \mathbb{P} \Sigma \bullet (\langle \rangle, X) \} \cup \{ X : \text{Refusal} \bullet (\langle \sqrt{\cdot} \rangle, X) \}$
$D[\text{Skip}](M) = \emptyset$

The divergent process can engage in any trace and, further, refuse everything. Recall the definition

of $\text{ProcType}$, then
For sequential composition, termination of the first component hands over control to the second.

\[
\mathcal{F}[\text{div}](M) = F \\
\mathcal{D}[\text{div}](M) = D
\]

The chaotic process can refuse any set of events, but it does neither diverge nor terminate.

\[
\mathcal{F}[\text{Chaos}(e)](M) = \{ t : \text{seq}[e]^c(M); X : \text{Refusal} \} \\
\mathcal{D}[\text{Chaos}(e)](M) = \emptyset
\]

The prefix \( e \rightarrow P \) can initially refuse everything except the event \( e \). After engaging in \( e \) it behaves like \( P \). The definition of multiprefix will be given after presenting the definition for distributed external and internal choices.

\[
\mathcal{F}[e \rightarrow P](M) = \{ X : \text{Refusal} | \ [e]^c(M) \notin X \cdot (\langle \rangle, X) \} \cup \\
\{ s : \text{Trace}; X : \text{Refusal} | (s, X) \in \mathcal{F}[P](M) \cdot (([e]^c(M)) \cap s, X) \}
\]

\[
\mathcal{D}[e \rightarrow P](M) = \{ t : \mathcal{D}[P](M) \cdot ([e]^c(M)) \cap t \}
\]

For the failures of external choice we have to consider four cases: Empty and non-empty traces, initial divergence and initial termination.

\[
\mathcal{F}[P \sqcap Q](M) = \{ X : \text{Refusal} | (\langle \rangle, X) \in \mathcal{F}[P](M) \cap \mathcal{F}[Q](M) \cdot (\langle \rangle, X) \} \cup \\
\{ f : \mathcal{F}[Q](M) \cup \mathcal{F}[P](M) | \text{first } f \neq \langle \rangle \} \cup \\
\{ X : \text{Refusal} | (\langle \rangle) \in \mathcal{D}[P](M) \cup \mathcal{D}[Q](M) \cdot (\langle \rangle, X) \} \cup \\
\{ X : \Sigma | (\langle \rangle) \in \mathcal{T}[P](M) \cup \mathcal{T}[Q](M) \cdot (\langle \rangle, X) \}
\]

\[
\mathcal{D}[P \sqcap Q](M) = \mathcal{D}[P](M) \cup \mathcal{D}[Q](M)
\]

Internal choice is much easier to define. It is simply based on set union.

\[
\mathcal{F}[P \sqcup Q](M) = \mathcal{F}[P](M) \cup \mathcal{F}[Q](M) \\
\mathcal{D}[P \sqcup Q](M) = \mathcal{D}[P](M) \cup \mathcal{D}[Q](M)
\]

For sequential composition, termination of the first component hands over control to the second.

\[
\mathcal{D}[P \triangledown Q](M) = \mathcal{D}[P](M) \cup \\
\{ s : T[P](M); t : \mathcal{D}[Q](M) | s \cap (\langle \rangle) \in T[P](M) \cdot s \cap t \}
\]

\[
\mathcal{F}[P \triangledown Q](M) = \{ s : \text{seq} \Sigma; X : \text{Refusal} | (s, X \cup \{ \langle \rangle \}) \in \mathcal{F}[P](M) \} \cup \\
\{ s : T[P](M); t : T[Q](M); X : \text{Refusal} | \\
(s, X) \in \mathcal{F}[Q](M) \cap (s \cap (\langle \rangle)) \in T[P](M) \cdot (s \cap t, X) \} \cup \\
\{ s : \mathcal{D}[P \triangledown Q](M); X : \text{Refusal} \cdot (s, X) \}
\]

To define the parallel operator we need the parallel operator for traces first. For a set of events \( E \) and traces \( s \) and \( t \), the set \( s \parallel t \) consists of all traces that are interleaving of \( s \) and \( t \) synchronized on \( E \) (see Figure 2.1). The process \( P \parallel Q \) diverges, if \( P \) or \( Q \) can engage in a divergent trace
synchronizing on the set of events $e$. To calculate the refusals of $P \parallel Q$ we have to consider two things: On the synchronization alphabet $e$, every event is refused if $P$ or $Q$ can refuse it. But for events not in $e$, events are refused if $P$ and $Q$ refuse it. This reflects the concept of generalised parallel composition: It is a combination of classical parallel composition as defined in Hoare’s book [14] and interleaving.
The events from renaming to ‘preserve’ these events:

Furthermore, all events in which every step

As \( P \) can engage in events from ran \( R \) that are not touched by \( P[R]Q \), we need an extra renaming to ‘preserve’ these events: \( R' = \{ x : \text{ran} R \cdot x \rightarrow x \} \). The relation \( R' \) is used to rename the events from \( P \) that are in ran \( R \).

\[
P[R]Q = (((P[R \cup R']) ||_{\text{ran} R} Q) \setminus \text{ran} R)[R'^\sim]
\]

The renaming \( [R'^\sim] \) restores the renaming \( [R'] \) (\( R'^\sim \) is the inverse of \( R' \)). Interruption is similar to sequential composition, where not only \( \sqrt{\text{\neg}} \), but every visible event hands over control to the second component.

\[
\mathcal{D}[P \triangle Q](M) = \mathcal{D}[P](M) \cup \{ t : \mathcal{T}[P](M) \cap \text{seq} \Sigma; s : \mathcal{D}[Q](M) \bullet t \sim s \}
\]

\[
\mathcal{F}[P \triangle Q](M) = \{ s : \mathcal{T}[P](M); X; Y : \text{Refusal} | (s, X) \in \mathcal{F}[P](M) \land \text{last} s \neq \sqrt{\text{\neg}} \land (\emptyset, Y) \in \mathcal{F}[Q](M) \bullet (s, X \land Y) \} \cup \{ f : \mathcal{F}[P](M) | \text{last(first} f) = \sqrt{\text{\neg}} \} \cup \{ s : \mathcal{T}[P](M); t : \mathcal{T}[Q](M); Y : \text{Refusal} | \text{last} s \neq \sqrt{\text{\neg}} \land (t, Y) \in \mathcal{F}[Q](M) \bullet (s \sim t, Y) \} \cup \{ s : \mathcal{D}[P \triangle Q](M); X : \text{Refusal} \}
\]

Hiding (\( P \setminus e \)) is based on removing all events of the set \( e \) from the traces of \( P \). Because we consider only stable states, only refusals of \( P \) that include \( e \) result in a refusal of \( P \setminus e \).
simplify the definitions, we assume that the process \( P \) corresponds a schema, given by Definition 2.3.3. Observe that the \( Z \) part is very simple and it is given by Definition 2.3.2. The semantics of the \( Z \) part is very simple and it is given by Definition 2.3.3. Observe that the \( Z \) part is very simple and it is given by Definition 2.3.2.

The distributed internal choice is based on distributed union.

\[
\mathcal{F}[P \setminus e](M) = \{ s : T[P](M) ; X : Refusal \mid (s, X \cup [e]^{c}(M)) \in \mathcal{F}[P](M) \}
\]

\[
\mathcal{D}[P \setminus e](M) = \{ s : D[P](M) ; t : Trace \mid s \mid (\Sigma \setminus [e]^{c}(M)) \cap t \} \cup \{ u : I[P](M) ; t : Trace \mid \text{dom}(u \uparrow (\Sigma \setminus [e]^{c}(M))) \in \mathcal{F}N \}
\]

Note that we use the straightforward application of filter (\( \lceil \cdot \rceil \)) on infinite traces. If \( s \) is a trace of \( P \) than every renaming of \( s \) is a trace of \( P[R] \). It is worth observing that \( P[R] \) is normally written as \( P[R] \), but for avoiding confusion with the semantic functions its formalisation is presented this way. The process \( P[R] \) can only refuse a set \( X \), if \( P \) can refuse all events related to \( X \). To simplify the definitions, we assume that \( R \) is total. A renaming \( R \) can always be totalised by \( R' = R \cup \{ a : EVENTS \mid a \notin \text{dom}R \bullet a \rightarrow a \} \).

The distributed external choice is more complicated because we have to consider more cases. The definition of \( F[\square e \bullet P] \) is split into four parts: initial refusals, refusals after engaging in some event, initial divergences and initial termination.

\[
\mathcal{F}[\square e \bullet P](M) = \{ X : Refusal \mid (\langle X \rangle, X) \in \bigcap \{ t : [e]^{c}(M) \bullet \mathcal{F}[P](M \oplus t) \} \} \cup \{ f : \{ t : [e]^{c}(M) \bullet \mathcal{F}[P](M \oplus t) \} \mid \text{first f} \neq \langle X \rangle \} \cup \{ X : Refusal \mid \langle X \rangle \in \mathcal{D}[\square e \bullet P](M) \bullet (\langle X \rangle, X) \} \cup \{ X : P \Sigma \mid \langle \sqrt{X} \rangle \in \bigcup \{ t : [e]^{c}(M) \bullet T[P](M \oplus t) \} \bullet (\langle X \rangle, X) \}
\]

where again \( [e]^{c}(M) \neq \emptyset \). For \( [e]^{c}(M) = \emptyset \), we have \( [\square e \bullet P](M) = [Stop](M) \).

### 2.3.2 The Semantics of the \( Z \) Part

The semantics of the \( Z \) part is very simple and it is given by Definition 2.3.3. Observe that the \( Z \) part does not exhibit divergence (\( D[P_{2}](M) = \emptyset \)). Its failures are built by sequential composition and precondition of schemas. The sequential composition means that to a trace, like \( \langle a, b, c \rangle \), corresponds a schema, given by \( \text{com}_{a} \parallel \text{com}_{b} \parallel \text{com}_{c} \). While the refusals originate from those schemas with the precondition disabled (not valid), in respect to a given state. For example, after a trace like \( \langle a, b, c \rangle \), the refusals are given by the events whose schema preconditions are disabled for
the state given by \( \text{com}_a \triangleright \text{com}_b \triangleright \text{com}_c \). Since after a schema (composition) has been performed the current state is given by \( \text{State}' \), the preconditions must be checked in this state. For this reason, the primed precondition—\((\text{pre com}_c)'\), for a channel \( c \)—is used. An implicitly information is captured by the predicate-based definition of this failures set: all channels with preconditions valid are ready to engage (that is, an external choice), whereas the state—after a channel has been engaged—is in general non-deterministic (that is, an internal choice). This implicit information is very important to understand how our translation—presented in the next chapter—has emerged.

**Definition 2.3.3 (Semantics of the Z Part)**

Let \( P_Z \) be the CSP\(_Z\) process

\[
\text{spec } P_Z; \ I; \ \text{State}; \ \text{Init}; \ Z; \ \text{end spec } P_Z
\]

where \( I \) is an interface, \( Z \) is a set of Z operations and there is no CSP part. Then, the semantics of \( P_Z \) is defined as follows

\[
\begin{align*}
\mathcal{F}[P_Z](M) &= \{(s, X) : \text{Failure} \mid c \in X \land \exists \text{State}' \cdot \text{comp}(s) \land \neg (\text{pre com}_c)' \}\^P \\
\mathcal{D}[P_Z](M) &= \emptyset
\end{align*}
\]

where \( \text{comp}(s) \) is given by

\[
\begin{align*}
\text{comp}(\langle \rangle) &= \text{Init} \\
\text{comp}(\langle c \rangle \triangleright s) &= \text{com}_c \triangleright \text{comp}(s)
\end{align*}
\]

\[\diamondsuit\]

### 2.3.3 Combining CSP and Z

The integration of the CSP part and the Z part is the most easy step on defining the semantics of CSP\(_Z\). The reason is that CSP\(_Z\) was designed as two orthogonal, but complementing, efforts, in the direction of linking theories and tools. Therefore, the simplest way of thinking about what means a CSP\(_Z\) module is just putting the two parts together. This is exactly what Definition 2.3.4 does, but formally; that is, a CSP\(_Z\) module is just an abbreviation for a parallel composition between the CSP part and the Z part.

**Definition 2.3.4 (Semantics of CSP\(_Z\))** Let \( P \) be the CSP\(_Z\) specification

\[
\text{spec } P; \ I; \ \text{main}; \ \text{State}; \ \text{Init}; \ Z; \ \text{end spec } P
\]

such that its CSP part is captured by the \( P_{CSP} \) process

\[
\text{spec } P_{CSP}; \ I; \ \text{main}; \ \text{end spec } P_{CSP}
\]
2.4. DATA REFINEMENT

and its Z part by the \( P_Z \) process

\[
\text{spec } P_Z; \ I; \ State; \ Init; \ Z; \ \text{end_spec } P_Z
\]

Then, the semantics of \( P \) is defined by

\[
[P]_D^D = [P_{CSP} \parallel P_Z]_D^D
\]

\( \Diamond \)

This completes the definition of the semantics for CSP\(_Z\).

2.4 Data Refinement

The most interesting issue is the relation between Z data refinement [47, 35] and CSP refinement [14, 10]. The semantics was defined in such a way that Z data refinement is consistent with CSP refinement [16, 17].

Data refinement was developed to transform specifications into implementations in a stepwise manner. The idea is to replace abstract data types of a specification with more concrete data types.

To achieve data refinement, the CSP part (processes) must not refer to any variables from the state space of Z schemas. We say a process \( P \) has no internal access to a schema \( S \) if no variable from \( S \) is used directly within \( P \).

**Definition 2.4.1 (Internal Access)**

Let \( A = (S, I, O) \) be an abstract data type. A process expression \( P \) has no internal access onto \( A \), if within \( P \) and within all processes \( P \) refers to

- no variable from \( S \) is used directly,
- the schemas from \( O \) and the schema \( I \) are only used for schema prefixing,
- the schema \( S \) is only used as a parameter of a process definition or as \( \theta S \) in recursive calls and
- no other expression refers directly or indirectly to some schema \( O \cup \{ S, I \} \).

\( \Diamond \)

The following lemma is taken from [17]. It relates Z data refinement with process refinement. Its proof can be found in [17].
Lemma 2.4.1 (Data Refinement Implies CSP refinement)

Let $A = (S_A, I_A, O_A)$ and $C = (S_C, I_C, O_C)$ be boundedly nondeterministic\textsuperscript{1} abstract data types such that $C$ is a data refinement of $A$. Let $\text{Rep}$ be the retrieve relation of $A$ and $C$. Let $P_A$ and $P_C$ be process expressions such that $P_A$ and $P_C$ have no internal access to $S_A$ and $S_C$. Furthermore, let $P_A$ and $P_C$ be syntactically equal, except that schemas of $A$ used in $P_A$ are replaced with the corresponding schemas of $C$ in $P_C$.

Then $P_C$ is a refinement of $P_A$ ($P_A \sqsubseteq P_C$) in all models $M$ that respect $\text{Rep}$

$$\forall M : [\text{Rep}]^P \bullet P_A \sqsubseteq_M P_C$$

\textsuperscript{1}An abstract data type is boundedly nondeterministic if the initialisation schema and each operation schema is boundedly nondeterministic.
Chapter 3

Model Checking CSP\(_Z\)

In general, a model checking strategy is oriented towards a specific language. Therefore, proposing a model checking strategy for a new language consists, normally, in building a completely new strategy. However, instead of developing a technique and a tool for model checking CSP\(_Z\) specifications from scratch, our effort is in the direction of reusing and linking theories and tools.

Traditionally, model checking means verifying automatically the satisfaction relation \(M \models p\), which states that the model \(M\) (described as a transition system) satisfies the property \(p\) (described as a logical formula) [22]. Roscoe [9] observed that this verification could be made via a refinement checking for CSP: \( (M \models p) \iff (S_p \subseteq S_M) \), where \(S_p\) and \(S_M\) are CSP specifications and \(S_p\) is built (or predefined in cases such as deadlock-freedom) as abstract as possible exhibiting the desired property \(p\). Since CSP\(_Z\) is a language whose semantics is based on the standard model of CSP (see Chapter 2), and refinement in the CSP part leads to refinement of the CSP\(_Z\) specification (see Section 2.4), it seems wise to extend the existing model checking technology for CSP in the FDR system [42] to CSP\(_Z\).

This chapter is organised this way. In Section 3.1 we present the requirements to be satisfied in order to apply the model checking strategy for CSP to CSP\(_Z\). The introduction of a state space should be considered and this is done in Section 3.2. The way the CSP part of a CSP\(_Z\) process is affected by the Z part is presented in Section 3.3. What the Z part means in terms of the CSP semantics is discussed in Section 3.4. Section 3.5 shows how the CSP and Z parts should be combined as well as discusses about the composition of CSP\(_Z\) processes.

3.1 Requirements for Model Checking CSP\(_Z\) using FDR

In order to be able to use FDR as a model checking tool for CSP\(_Z\) specifications, we need to answer the following questions which, together, record some necessary conditions for embedding a subset of Z into CSP. It is worth observing that this section is presented towards the CSP\(_M\) language.
The patterns have self explanations but a more detailed presentation of the CSP language can be obtained in Appendix A.

1. How to describe a state space in CSP?
2. How to constrain the CSP behaviour based on the state values?
3. How to completely characterise the Z part as a CSP process?
4. How to combine and synchronise the CSP part and the Z part of a CSP specification?

Each of these questions is answered in a separate subsection. For this purpose we use the WDT process of Section 2.1, reproduced here, to ease the illustration of our strategy.

```
[CLK]

spec WDT
  chan clockWDT: [clk : CLK]
  chan reset, recover
  lchan timeOut, noTimeOut, failFTR, offWDT

  main=Signal || Verify
    {offWDT}
  Signal=(reset→Signal □ offWDT→skip)
  Verify=(clockWDT?clk→(noTimeOut→Verify
       □ timeOut→(recover→Verify
       □ failFTR→offWDT→skip)))
  WDTtOut : CLK
  WDTP : CLK ↔ CLK

  LENGTH == 0 .. 3

State ≜ [cycles : LENGTH; time : CLK]
Init ≜ [State' | cycles' = 0]
com_clockWDT ≜ [∆State; clk? : CLK | time' = clk?]
com_recover ≜ [∆State | cycles < 3 ∧ cycles' = cycles + 1]
com_reset ≜ [∆State | cycles' = 0]
com_noTimeOut ≜ [ΞState | ¬ WDTP(time, WDTtOut)]
com_timeOut ≜ [ΞState | WDTP(time, WDTtOut)]
com_failFTR ≜ [ΞState | cycles = 3]

end_spec WDT
```
3.2 Introducing a State Space

The standard way of associating a state with a CSP process is through parameterisation. Indeed, parameterisation is a technique used to represent succinctly a family of processes with similar structure [10]. Part of the above CSP \( Z \) process is captured by the equation below:

\[
\text{WDT}(\text{State}) = \text{clockWDT}\text{?clk} \rightarrow \text{WDT}(\text{State}') \ldots
\]

where \( \text{State}' \) is a new state built in some way from \( \text{State} \) and \( \text{clk} \). Hence, each process carries some expression which represents its state space; this is essential to model CSP \( Z \) processes.

Generalising the above specific pattern, a \( Z \) state space can be represented as a set of tuples where the components are identified by their positions in the tuple. Then, we have the following pattern conversion (where \( \text{Inv}(v_1, \ldots, v_n) \) is the invariant of the system, that is, a predicate over \( v_1, \ldots, v_n \) which must be true during the whole life-cycle of the process):

\[
\text{State} = \{ (v_1, \ldots, v_n) | v_1 \leftarrow T_1, \ldots, v_n \leftarrow T_n, \text{Inv}(v_1, \ldots, v_n) \}
\]

where \( \leftarrow \) stands for \( \in \) and the commas mean logical and’s (\( \wedge \)). Instantiating this pattern for our example, we get:

\[
\text{State} = \{ (\text{cycles}, \text{time}) | \text{cycles} \leftarrow \{0,1,2,3,4\}, \text{time} \leftarrow \text{CLK} \}
\]

where \( \text{CLK} \) is a set of possible clocks.

The finiteness of the above set of pairs can be achieved due to our work (see Chapter 4.3) and the work of Lazić [58] which deals with data independent CSP processes.

The initialisation schema is written as a specific case of the \( \text{State} \) schema where the predicate \( \text{Inv} \) is replaced by a predicate \( \text{init} \) (of initialisation). For our example we have:

\[
\text{Init} = \{ (\text{cycles}, \text{time}) | (\text{cycles}, \text{time}) \leftarrow \text{State}, \text{cycles}==0, \text{time}==\text{noneClock} \}
\]

3.2.1 \( Z \) Data Types Representation

Although, in the WDT example, the translation of the state and of the initialisation schemas is immediate, in general there is no one-to-one correspondence between the elements \( T_i \) and \( T_i \), and \( \text{Inv}(\ldots) \) and \( \text{Inv}(\ldots) \). This is why we have used different fonts. Our objective is to emphasise
higher abstract patterns rather than discuss implementation issues of converting abstract types into more concrete types available in CSP\(_M\). In practice, for complex CSP\(_Z\) specifications one has to do a series of data refinement steps until a sufficiently concrete specification is achieved (see Section 6.1 for a possible strategy).

The representation of channel declarations, constants and free-types (when based on primitive types) is a straightforward syntactic conversion, as presented in [3]. However, for some elaborate data structures such as maps, relations, bags, etc, the conversion might be a complex data refinement task. The concrete representations must be built around the definition of sets and sequences, which are available in FDR.

Currently, we have two possible ways to deal with abstract data types: either apply data refinement until manageable data types have been found (this is the powerful and always possible way) or use the implementations of maps, relations, bags, etc, available in our Java transformational tool [1, 2] (this is not feasible for all practical situations).

A more detailed strategy for this abstract data type translation is out of the scope of this chapter. An approach to implementing model-based specifications using a functional notation can be found, for example, in [50].

### 3.3 CSP Behaviour as a Function of the State Space

Another concern is how to restrict the CSP behaviour based on the state space of the process. In Chapter 2, we saw that the Z part has a very curious feature: if a precondition of a Z schema is true, then the corresponding event is enabled, otherwise the event is refused.

In our particular example, the WDT process can accept the event `recover`, but `recover` is enabled only if the number of cycles is less than 3. To handle this kind of feature we use the CSP\(_M\) notation \(b \land P\) which stands for a conditional: if \(b\) then \(P\) else STOP:

\[
\text{cycles < 3 \& recover } \rightarrow \text{WDT(State')}
\]

where \(\text{cycles < 3} \ (\text{cycles < 3})\) is the precondition of \(\text{com\_recover}\), and \(\text{State'}\) is a new state taking into account the effect of the operation \(\text{com\_recover}\).

Conditionals based on the current value of components of the state space (as well as on the values of input variables) serve as a standard pattern to check the preconditions of Z operations described by schemas.
3.4 Modelling the Z part as a CSP Process

The Z part of a CSP\(_Z\) specification can be seen as a parameterised passive process (with the state as a parameter; that is, the translation of the state space \(\text{State}\) as parameter). The passive denotation means that it always offers all events with the precondition satisfied. In order to model the Z part completely, we must express: Z operations, state space initialisation and the composition of Z operations controlled by the CSP part (traces), through CSP\(_M\) constructs. Then, for Z operations we have the following conversion pattern (Recalling that \(\text{State} \equiv [v_1 : T_1; \ldots; v_n : T_n | P]\)):

\[
\downarrow
\]

\[\text{com}((v_1, \ldots, v_n), \text{Ch.In.Out}) = \{ (v_1', \ldots, v_n') \mid (v_1', \ldots, v_n') \leftarrow \text{State}, \right.
\]

\[\left. \text{pre}((v_1, \ldots, v_n), \text{In}), \text{pos}((v_1, \ldots, v_n), (v_1', \ldots, v_n'), \text{In}, \text{Out}) \}\]

which, specialised to our example, gives us:

\[\text{com}((\text{cycles}, \text{time}), \text{reset}) = \{ (\text{cycles'}, \text{time'}) \mid (\text{cycles'}, \text{time'}) \leftarrow \text{State}, \right.
\]

\[\left. \text{cycles'}==0, \text{time'}==\text{time} \}\]

\[\text{com}((\text{cycles}, \text{time}), \text{clockWDT.clk}) = \{ (\text{cycles'}, \text{time'}) \mid (\text{cycles'}, \text{time'}) \leftarrow \text{State}, \right.
\]

\[\left. \text{cycles'}==\text{cycles}, \text{time'}==\text{clk} \}\]

Note that \(\text{State}\) is now a fixed set defined previously, as illustrated by the rule in Section 3.2.

It is worthwhile observing that the separation of pre- and post-conditions in the Z schema of the above rule is just a convenience of the conversion strategy, since schemas have a single predicate without such a separation. As already pointed out, the translation of predicates into executable boolean functions is not so simple, in general, as it might appear, but this is not the focus of this thesis; see, for example, [50].

Let \text{Interface} be the interface (set of external and internal channels) of a CSP\(_Z\) process \(P\). Then, its Z part without a state initialisation is given by:
CHAPTER 3. MODEL CHECKING CSP

\[
Z(\text{State}) = \\
\text{let} \\
\text{Conf} = \{(\text{com}(\text{State}, c), c) \mid c \leftarrow \text{Interface}\} \\
\text{within} \\
\left[\right] \text{Conf} \neq \{\} \& (\text{States,Comm}) : \text{Conf} @ \\
\text{States} \neq \{\} \& \\
\neg \left| \text{State}' : \text{States} @ \text{Comm} \rightarrow Z(\text{State}') \right.
\]

The idea behind representing the Z part as a CSP process is to collect all pairs of states and communication (due to the relational nature of the state space we can have more than one next state for a single communication). This is accomplished with this piece of CSP code:

\[
\{(\text{com}(\text{State}, c), c) \mid c \leftarrow \text{Interface}\}
\]

which might seem rather strange at a first glance. To explain how this set is generated it is fundamental to remind that, in CSP\(_M\), we have no distinction between input and output channels after expansion of their definitions; they are all normalised to a dotted format. For example: \(a?x\) is the same as \(\{a.e \mid e \in \text{Type}(a)\}\) whereas \(a!5\) is equivalent to \(a.5\). Now, observe that \(c\) comes from the \text{Interface} (all possible communication channels of the process) which is already in dotted form due to the use of \(\{\ldots\}\). Hence, after \(c\) has been assigned a dotted channel we can evaluate \(\text{com}(\text{State}, c)\) by pattern matching and, finally, the tuple \((\text{com}(\text{State}, c), c)\) is used to guarantee that a next state is associated to a specific communication \(c\) which is very important to the rest of the transformation pattern.

The construction \(\left[\right] \text{Conf} \neq \{\} \& (\text{States,Comm}) : \{\ldots\} @ \ldots\) is an indexed external choice where we introduce the guard \(\text{States} \neq \{\}\) to ensure that there is at least one valid next state. According the semantics of CSP\(_Z\) (see Section 2.3), the output and the next state might be non-deterministic only depending on the \text{com,Comm Z} operation. Then we use an indexed internal choice of possible next states (captured with \text{State'} \(: \text{States}) associated with a communication. Finally, we build the prefix \text{Comm} \rightarrow Z(\text{State}') to exhibit this behaviour.

To conclude the conversion of the Z part, we consider the valid initialisation states for the Z part by extending the above process to its final representation (where \text{Init} is an initial valid state):

\[
Z_{\text{CSP}} = \\
\text{let} \\
Z(\text{State}) = \\
\text{let} \\
\text{Conf} = \{(\text{com}(\text{State}, c), c) \mid c \leftarrow \text{Interface}\} \\
\text{within}
\]
3.4. MODELLING THE Z PART AS A CSP PROCESS

\[
\begin{align*}
\text{Conf} & \neq \{\} \& (\text{States}, \text{Comm}) : \text{Conf} @ \text{States} \neq \{\} \& \\
& |\sim| \text{State}' : \text{States} @ \text{Comm} \rightarrow Z(\text{State}') \\
\text{Init} & \neq \{\} \& \\
& |\sim| \text{iState} : \text{Init} @ Z(\text{iState})
\end{align*}
\]

An initialisation operation is like an output one, in the sense of generating the next states from which one must be chosen nondeterministically. Then we have an indexed internal choice of all possible initialisations.

The \text{let ... within ...} construct is used to write a recursive process in a more structured way. In particular, we will see in Section 3.5 that the \text{let ... within ...} construct is very useful to represent the notion of modularisation corresponding to the spec/end_spec markups.

3.4.1 Optimisation

The conversion pattern for schemas, introduced previously, might be inefficient for practical purposes. The reason is that the generic conversion pattern uses set comprehension to represent an arbitrary relation defined by the corresponding schema, and the implementation of a set comprehension in FDR involves expanding all possibilities (given by the types used in the definition) and then filtering the pairs for which the relation holds.

The opportunity for optimisation happens for specialised (schemas) operations, that is, instead of generic predicates we can find constant based predicates or \(\Xi\)-based predicates. In such cases, the schema predicate can be rewritten to a more specific predicate, provided the state invariant and precondition are satisfied. We illustrate this point using the schema \text{com\_FTR} of the WDT process. Its general translation is given by

\[
\text{com}((\text{cycles}, \text{time}), \text{failFTR}) = \{ (\text{cycles}', \text{time}') | (\text{cycles}', \text{time}') \leftarrow \text{State}, \\
\text{cycles}==3, \text{cycles}'==\text{cycles}, \text{time}'==\text{time} \}
\]

Now, observe that the natural number 3 is restricting, considerably, the set of possible pairs \((\text{cycles}', \text{time}')\). Furthermore, its \(\Xi\)-based nature \((\text{cycles}'==\text{cycles}, \text{time}'==\text{time})\) is producing an additional limitation. In particular, this predicate has a unique solution. Therefore, instead of the more general set comprehension we can convert the schema \text{com\_failFTR} into

\[
\text{com}((\text{cycles}, \text{time}), \text{failFTR}) = \{ (\text{cycles}, \text{time}) | \text{cycles}==3 \}
\]

Obviously, such transformations must be performed according to formal rules. For instance, the above transformation was achieved via the following steps
CHAPTER 3. MODEL CHECKING CSP

\[
\text{com}((\text{cycles}, \text{time}), \text{failFTR}) = \{ (\text{cycles'}, \text{time'}) | (\text{cycles'}, \text{time'}) \leftarrow \text{State}, \\
\text{cycles}=3, \text{cycles'}=\text{cycles}, \text{time'}=\text{time} \}
\]

[By the one-point rule]

\[
\text{com}((\text{cycles}, \text{time}), \text{failFTR}) = \{ (\text{cycles}, \text{time}) | \text{member}((\text{cycles}, \text{time}), \text{State}), \\
\text{cycles}=3 \}
\]

[Since \(\text{member}((\text{cycles}, \text{time}), \text{State}) = \text{true}\)]

\[
\text{com}((\text{cycles}, \text{time}), \text{failFTR}) = \{ (\text{cycles}, \text{time}) | \text{cycles}=3 \}
\]

Such optimisations might be quite significant in practice. For instance, such effort is essential in Chapter 6 where the global analysis phase of our case study (see Section 6.2) can only be performed after such an optimisation. A similar contribution was presented for the model checking community—in its early days—by McMillan [39] (firstly presented in his PhD thesis [38]). McMillan has shown that boolean structures—commonly used by the traditional model checking community—could be captured by ordered binary decision diagrams [56] (or OBDDs) which are a compressed version of the well-known truth-tables. From his proposal, the term symbolic model checking has emerged as a revolution (historical mark) for the model checking technique. Since then, more real complex systems could be analysed using model checking [22].

3.5 Combining the CSP and the Z Parts

The meaning of a CSP\(_Z\) specification unit is defined in [16] (see Chapter 2, Section 2.3) as the parallel composition of the CSP and Z parts synchronising at the channels declared in the interface. Let

\[
\text{spec } P
\]

\[
\text{Interface} \quad \text{(Represents the interface of } P\text{)}
\]

\[
\text{main} \quad \text{(Represents the CSP part)}
\]

\[
S_Z \quad \text{(Represents the Z part)}
\]

\[
\text{end_spec } P
\]

be a generic CSP\(_Z\) specification, where \(P\) is the name of the specification, \text{Interface} is its interface (i.e., it is the union of the set of visible channels introduced by the keyword \text{chan} and the set of local—outside invisible—channels introduced by the keyword \text{lchan}, see Chapter 2 for further details), \text{main} is its CSP part, and \(S_Z\) is its Z part.

According to our translation strategy into CSP\(_M\), the CSP part suffers almost no modifications in its structure, except for the output channels. Recall from our simple buffer specification
that the term `out!y`, in the `main` equation of the CSP$_Z$ process `Buffer`, has no meaning considered in isolation; it needs the participation of the Z part to behave accordingly. In order to communicate the output values generated by the Z part to the CSP part, one must enable the CSP part to generate all possible values, which depend on the type of the respective channel, and the Z part to simply select some of them.

So, when a term like `c!y` is found in the `main` equation it is transformed into `c?y`, which is justified by the equation `?x : A \rightarrow P(x) = \Box_{x \in A} x \rightarrow P(x)`, where `A \subseteq \Sigma` [10].

Summarising our conversion strategy, an arbitrary CSP$_Z$ specification, say `P`, will be translated to a CSP$_M$ process with the following skeleton.

```
P = let
   -- The Interface
   Channels = {|...|}
   lChannels = {|...|}
   Interface = union(Channels, lChannels)
   -- The CSP Part
   main = ...
   -- The Z Part
   State = {...}
   Init  = {...}
   com(sTuple, Comm) = {...}
   ...
   Z_CSP = ...
within (main [|Interface|] Z_CSP)\lChannels
```

In words, the behaviour of `P` is the parallel composition of the communication part (`main`) and the process which represents the Z part (`Z_CSP`); these processes synchronise in the interface `Interface`. The hiding in the above equation is necessary because CSP does not provide a means for modularisation in the sense of scope found in CSP$_Z$ specifications. Therefore, local channels in CSP$_Z$ need to be explicitly hidden when considered in CSP.

Finally, when considering the combination of CSP$_Z$ specifications one needs only to combine the top-level processes. The reason of this is because a CSP$_Z$ specification is translated into a pure CSP$_M$ specification, and therefore the CSP operators can be used to combine CSP$_Z$ specifications as ordinary CSP processes.

Surprisingly, perhaps, it is not necessary any additional conversion rule to deal with combining CSP$_Z$ specifications.
3.5.1 Dealing with Termination

It is worth noting that the process $Z_{CSP}$ which represents the Z part of a CSP\(_Z\) specification never terminates successfully. This might be a problem when the CSP part does terminate, since the parallel composition of the CSP and Z parts (as defined in the previous section) would lead to deadlock.

A simple solution to this problem is to add a new event to the alphabet of the process which will serve as a synchronisation point between the CSP and the Z parts. We denote this event by \texttt{terminate} and use it in the following way:

- Add \texttt{terminate} to \texttt{lChannels};
- Add the external choice $[] \texttt{terminate} \rightarrow \texttt{SKIP}$ to the $Z_{CSP}$ process;
- Remove the \texttt{terminate} event from the interface when computing the set
  $\{(\text{com}(\text{State}, c), c) \mid c \leftarrow \text{Interface}\}$
  on process $Z(\text{State})$;
- Concerning the CSP part, prefix every \texttt{skip} process with the special event \texttt{terminate}. This forces the synchronisation between the CSP and the Z parts upon \texttt{terminate}.

When the CSP part does not terminate successfully we can skip these steps concerning termination.

3.6 The Transformation Preserves the CSP\(_Z\) Semantics

In this section we show that the CSP\(_Z\) semantics is preserved by the transformations presented in the previous sections. It is worth noting that we are talking about implementation details. We are concerned with demonstrating that the patterns we use generate the same meaning as the semantics presented in Chapter 2.

In the following we prove that our transformation preserves the semantics of CSP\(_Z\). We concentrate on the Z part, since the equivalence between a CSP and a CSP\(_M\) representation, for a given process, is presented in Scattergood’s work [13].
Lemma 3.6.1 (The Z Part)

Let \( P \) be the CSP\(_Z\) specification

\[
\text{spec } P; \ I; \ main; \ State; \ Init; \ Z; \ \text{end_spec } P
\]

such that its Z part is captured by the \( P_Z \) process

\[
\text{spec } P_Z; \ I; \ State; \ Init; \ Z; \ \text{end_spec } P_Z
\]

Furthermore, assume that \( \text{Init}^\equiv(M) = \text{Init}^\equiv(M) \), \( \text{State}^\equiv(M) = \text{comp}(s)^\equiv(M) \), and the used operators and types of Z have a direct corresponding after translation. If \( P_{NF} \) is the CSP\(_M\) process resulting from the translation approach, regarding only the Z part of \( P \), then \( \text{[P} Z\text{]}^D = \text{[P} NF\text{]}^D \).

**Proof.** The proof consists in establishing the following equalities

\[
\mathcal{F}[P_{NF}](M) = \mathcal{F}[P_Z](M) \\
\mathcal{D}[P_{NF}](M) = \mathcal{D}[P_Z](M)
\]

But, recall from Section 2.3.2 that \( \mathcal{D}[P_Z](M) = \emptyset \). Furthermore, as long as \( P_{NF} \) does not have any hiding in its body, then \( \mathcal{D}[P_{NF}](M) = \emptyset \). Thus, we have only to prove that

\[
\mathcal{F}[P_{NF}](M) = \mathcal{F}[P_Z](M)
\]

First of all, from Section 3.4, we know that \( P_{NF} \) is given by the following standard pattern

\[
\text{let}
\quad Z\text{(State)} =
\quad \text{let}
\quad \quad \text{Conf} = \{(\text{com}(\text{State}, \text{c}), \text{c}) \mid \text{c} \leftarrow I\}
\quad \text{within}
\quad \quad \quad \quad \quad [] \text{Conf} \neq \{} \& (\text{States, Comm}) : \text{Conf} @
\quad \quad \quad \quad \quad \quad \text{States} \neq \{} \&
\quad \quad \quad \quad \quad \quad \quad \quad \text{\text{^\mbox{}^{-} State'}} : \text{States} @ \text{Comm} \to Z(\text{State}')
\quad \text{within Init} \neq \{} \&
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\text{\^\mbox{}^{-} iState} : Init} @ Z(\text{iState})
\]

From this pattern, we have two situations: \( \text{Init} = \{} \) and \( \text{Init} \neq \{} \). The former is trivial because

\[
\mathcal{F}[P_{NF}](M)
\]

\[
= \mathcal{F}[\{} \neq \{} \& \text{\^\mbox{}^{-} iState} : \{} @ Z(\text{iState})](M)
\]

\[
= \mathcal{F}[\text{Stop}](M)
\]

\[
= \emptyset
\]

\[
= \bigcup \{ t : \emptyset \cdot \mathcal{F}[Z(\text{iState})](M \oplus t) \}
\]

\[
= \mathcal{F}[P_Z](M)
\]
Indeed, in what follows, this base case is discarded because it always yields the analysis of the process
Stop. The latter implies that
\[ \mathcal{F}[P_{NF}](M) = \bigcup \{ t : [iState : Init]^c(M) \cdot \mathcal{F}[Z(iState)](M + t) \} \]
and
\[ \mathcal{F}[P_Z](M) = \{(s, X) : Failure \mid c \in X \land [\exists State' \cdot comp(s) \land \neg (pre\, com_c)]^P \} \]
be equal. By extensionality we have
\[ \forall c \cdot c \in \bigcup \{ t : [iState : Init]^c(M) \cdot \mathcal{F}[Z(iState)](M + t) \} \]
\[ \Leftrightarrow \]
\[ c \in \{(s, X) : Failure \mid c \in X \land [\exists State' \cdot comp(s) \land \neg (pre\, com_c)]^P \} \]

As each element of these sets is determined by a trace, the proof now follows by induction on the
size of the traces corresponding to each element of the set.

The base case for traces is \((\langle \rangle, X)\), such that \(X \neq \Sigma\). In this situation, we are analysing
the initialisation. As the case \((\langle \rangle, \{\} \})\) is trivial (this tuple is always present in the failures of any
process), we assume that \((\langle \rangle, Y)\) is present in both sets and check if \((\langle \rangle, Y \cup \{e\})\) can also be present
in both sets, for a refused communication \(e\). Thus,

\[ (\langle \rangle, Y \cup \{e\}) \in \bigcup \{ t : [iState : Init]^c(M) \cdot \mathcal{F}[Z(iState)](M + t) \} \]
\[ \Leftrightarrow \]
\[ (\langle \rangle, Y \cup \{e\}) \in \{(\langle \rangle, X) : Failure \mid c \in X \land [\exists State' \cdot Init \land \neg (pre\, com_c)]^P \} \]
\[ \equiv \text{By a law of \(\epsilon\)} \]
\[ (\langle \rangle, Y) \in \bigcup \{ t : [iState : Init]^c(M) \cdot \mathcal{F}[Z(iState)](M + t) \} \]
\[ \land (\langle \rangle, \{e\}) \in \bigcup \{ t : [iState : Init]^c(M) \cdot \mathcal{F}[Z(iState)](M + t) \} \]
\[ \Leftrightarrow \]
\[ (\langle \rangle, Y) \in \{(\langle \rangle, X) : Failure \mid c \in X \land [\exists State' \cdot Init \land \neg (pre\, com_c)]^P \} \]
\[ \land (\langle \rangle, \{e\}) \in \{(\langle \rangle, X) : Failure \mid c \in X \land [\exists State' \cdot Init \land \neg (pre\, com_c)]^P \} \]

By induction hypothesis, we have only to prove that
\[ (\langle \rangle, \{e\}) \in \bigcup \{ t : [iState : Init]^c(M) \cdot \mathcal{F}[Z(iState)](M + t) \} \]
\[ \Leftrightarrow \]
\[ (\langle \rangle, \{e\}) \in \{(\langle \rangle, X) : Failure \mid c \in X \land [\exists State' \cdot Init \land \neg (pre\, com_c)]^P \} \]

At this point, we have only to show that the communication \(e\) is refused by the process \(P_{NF}\). Thus,
we have to check whether \((States, Comm) = (com(State, e), e)\) provides \(States = \{\}\). That is,
\(com(State, e)\) returns the empty set. This happens when the translated precondition or postcondition
are not satisfied. As we are assuming a one-to-one correspondence in the data part, the
postcondition will never be invalid when the precondition is valid. Thus, \(com(State, e)\) returns
the empty set iff the translated precondition is invalid as expected.
3.6. THE TRANSFORMATION PRESERVES THE CSP\textsubscript{Z} SEMANTICS

Now, let us check the presence of \((t \triangleright (e), \{\})\) in both sets, assuming that \((t, \{\})\) is present.

\[(t \triangleright (e), \{\}) \in \mathcal{F}[\text{let} \]
\[
\quad \text{Conf} = \{(\text{com}(\text{State}, c), c) \mid c \in I\}
\]
\[
\quad \text{within}\]
\[
\quad \quad \text{if Conf} \neq \emptyset \text{ then}
\quad \quad \quad \Box (\text{States,Comm}) : \text{Conf} \bullet
\quad \quad \quad \quad \text{if State} \neq \emptyset \text{ then}
\quad \quad \quad \quad \quad \Box \text{State}' : \text{States} \bullet \text{Comm} \rightarrow Z(\text{State}')
\quad \quad \quad \text{else Stop}
\]
\[
\quad \text{else Stop}\] \(\mathcal{M}\)
\[
\Leftrightarrow
\]
\[(t \triangleright (e), \{\}) \in \{(s, X) : \text{Failure} \mid c \in X \land [\exists \text{State}' \bullet \text{com}(s) \land \neg (\text{pre com}_c')]'\}^P\]

Recall the analysis of \(\text{Init}=\{\}\), where the process becomes \text{Stop}. As this analysis is standard, we assume that it does not occur (for simplification). So, we reduce the above predicate to

\[(t \triangleright (e), \{\}) \in \mathcal{F}[\Box (\text{States,Comm}) : \{(\text{com}(\text{State}, c), c) \mid c \in I\} \bullet
\]
\[
\quad \Box \text{State}' : \text{States} \bullet \text{Comm} \rightarrow Z(\text{State}') \] \(\mathcal{M}\)
\[
\Leftrightarrow
\]
\[(t \triangleright (e), \{\}) \in \{(s, X) : \text{Failure} \mid c \in X \land [\exists \text{State}' \bullet \text{com}(s) \land \neg (\text{pre com}_c')]'\}^P\]

which is equivalent to prove the following

\[
\text{com}(\text{State}', e) \neq \emptyset \Leftrightarrow [\exists \text{State}' \bullet \text{com}(s) \land (\text{pre com}_e)]
\]

where \text{State}' is given by the successive application of the \text{com} functions from the \text{Init} state. This proceeds by hypothesis, where we have \([\text{State}]^c(M) = [\text{com}(s)]^c(M)\) and the preconditions have a direct corresponding.

The final situation is \((s, X \cup \{e\}),\) assuming \((s,X)\). But this is straightforward following the previous argumentation over preconditions. \(\diamond\)

As Parallelism acts like conjunction, the parallel composition between two processes diverges iff at least one of them diverges and the other does not forbid this behaviour. In what follows, we present Lemma 3.6.2 which establishes this fact formally.
Lemma 3.6.2 (Parallelism and Divergence)

Let \( P \) and \( Q \) be two processes. Assume that \( \alpha P = \alpha Q = I \) and \( D(Q) = \emptyset \). Then,
\[
D( P || Q ) = D(P) \cap T(Q)
\]

Proof. By definition, we have
\[
D[ P || Q ](M) = \{ u, v : \text{Trace} \mid (\exists s : T[P](M); t : T[Q](M) \bullet \ u \in \left( \left( s \ || \ I \right)^\epsilon(M) \cap \text{seq}\Sigma \right) \wedge \ (s \in D[P](M) \lor t \in D[Q](M)) \bullet u \preceq v \}
\]

By hypothesis, we have that \( D(Q) = \emptyset \)
\[
\{ u, v : \text{Trace} \mid (\exists s : T[P](M); t : T[Q](M) \bullet \ u \in \left( \left( s \ || \ I \right)^\epsilon(M) \cap \text{seq}\Sigma \right) \wedge \ (s \in D[P](M) \lor t \in \emptyset) \bullet u \preceq v \}
\]

As \( t \in \emptyset \equiv \text{false} \) and \( b \lor \text{false} = b \)
\[
\{ u, v : \text{Trace} \mid (\exists s : T[P](M); t : T[Q](M) \bullet \ u \in \left( \left( s \ || \ I \right)^\epsilon(M) \cap \text{seq}\Sigma \right) \wedge \ s \in D[P](M) \bullet u \preceq v \}
\]

By a law of conjunction
\[
\{ u, v : \text{Trace} \mid (\exists s : D[P](M); t : T[Q](M) \bullet \ u \in \left( \left( s \ || \ I \right)^\epsilon(M) \cap \text{seq}\Sigma \right) \bullet u \preceq v \}
\]

By hypothesis, the alphabet of \( P \) and \( Q \), and the synchronisation set \( I \) are equals. Recall of the rule for traces, \( (x) \preceq s \ || \ (x) \preceq t = \{ u : s \ || \ t \bullet (x) \preceq u \} \) (see Figure 2.1). This is the unique applicable rule due to the uniform alphabet hypothesis. Then,
\[
D[P](M) \cap T[Q](M) = (D[P] \cap T[Q])(M)
\]

\( \diamond \)

Theorem 3.6.1 (CSP\(_Z\) Translation)

Let \( P \) be the CSP\(_Z\) specification
\[
\text{spec} \ P; \ I; \text{main}; \ State; \ Init; \ Z; \text{end-spec} \ P
\]
provided that \( P_{CSP} \) captures the CSP part and \( P_Z \) the Z part. If \( P' \) is the CSP_M process resulting from the translation approach, such that \( P_{NF} \) captures the translation for the Z part, then \( [P]^D = [P']^D \).

**Proof.** By the definition of \([.]^D\), we have to prove

\[
\mathcal{F}[P_{CSP} \parallel P_Z](M) = \mathcal{F}[\text{main} \parallel P_{NF}](M)
\]

\[
\mathcal{D}[P_{CSP} \parallel P_Z](M) = \mathcal{D}[\text{main} \parallel P_{NF}](M)
\]

Recall from Section 2.3.2 that \( \mathcal{D}[P_Z](M) = \mathcal{D}[P_{NF}](M) = \emptyset \). And by Lemma 3.6.2, we have

\[
\mathcal{F}[P_{CSP} \parallel P_Z](M) = \mathcal{F}[\text{main} \parallel P_{NF}](M)
\]

\[
\mathcal{D}[P_{CSP}](M) \cap T[P_Z](M) = \mathcal{D}[\text{main}](M) \cap T[P_{NF}](M)
\]

The work of Scattergood [13] guarantees \( \mathcal{D}[P_{CSP}](M) = \mathcal{D}[\text{main}](M) \). By Lemma 3.6.1, we know that \( T[P_Z](M) = T[P_{NF}](M) \). Hence, we have only to prove

\[
\mathcal{F}[P_{CSP} \parallel P_Z](M) = \mathcal{F}[\text{main} \parallel P_{NF}](M)
\]

Applying the definition of parallel composition for both sides of the equality, we get

\[
\{s; t; u : \text{Trace}; Y; Z : \text{Refusal} \mid (s, Y) \in \mathcal{F}[P_{CSP}](M) \land (t, Z) \in \mathcal{F}[P_Z](M) \land Y \setminus ([e]^e(M) \cup \{\sqrt{v}\}) \land u \in s \parallel t \bullet (u, Y \cup Z)\}
\]

\[
\cup \{f : \text{Failure} \mid \text{first } f \in \mathcal{D}[P_{CSP} \parallel P_Z](M)\}
\]

\[
= \{s; t; u : \text{Trace}; Y; Z : \text{Refusal} \mid (s, Y) \in \mathcal{F}[\text{main}](M) \land (t, Z) \in \mathcal{F}[P_{NF}](M) \land Y \setminus ([e]^e(M) \cup \{\sqrt{v}\}) \land u \in s \parallel t \bullet (u, Y \cup Z)\}
\]

\[
\cup \{f : \text{Failure} \mid \text{first } f \in \mathcal{D}[\text{main} \parallel P_{NF}](M)\}
\]

As we already have proven, \( \mathcal{D}[P_{CSP} \parallel P_Z](M) = \mathcal{D}[\text{main} \parallel P_{NF}](M) \). Hence, we can simplify

\[
\{s; t; u : \text{Trace}; Y; Z : \text{Refusal} \mid (s, Y) \in \mathcal{F}[P_{CSP}](M) \land (t, Z) \in \mathcal{F}[P_Z](M) \land Y \setminus ([e]^e(M) \cup \{\sqrt{v}\}) \land u \in s \parallel t \bullet (u, Y \cup Z)\}
\]

the above equality to

\[
\{s; t; u : \text{Trace}; Y; Z : \text{Refusal} \mid (s, Y) \in \mathcal{F}[P_{CSP}](M) \land (t, Z) \in \mathcal{F}[P_Z](M) \land Y \setminus ([e]^e(M) \cup \{\sqrt{v}\}) \land u \in s \parallel t \bullet (u, Y \cup Z)\}
\]

\[
= \{s; t; u : \text{Trace}; Y; Z : \text{Refusal} \mid (s, Y) \in \mathcal{F}[\text{main}](M) \land (t, Z) \in \mathcal{F}[P_{NF}](M) \land Y \setminus ([e]^e(M) \cup \{\sqrt{v}\}) \land u \in s \parallel t \bullet (u, Y \cup Z)\}
\]

\[
= \{s; t; u : \text{Trace}; Y; Z : \text{Refusal} \mid (s, Y) \in \mathcal{F}[\text{main}](M) \land (t, Z) \in \mathcal{F}[P_{NF}](M) \land Y \setminus ([e]^e(M) \cup \{\sqrt{v}\}) \land u \in s \parallel t \bullet (u, Y \cup Z)\}
\]
CHAPTER 3. MODEL CHECKING CSP

By extensionality, we have only to prove
\[
\mathcal{F}[P_{CSP}](M) = \mathcal{F}[\text{main}](M) \\
\mathcal{F}[P_Z](M) = \mathcal{F}[P_{NF}](M)
\]
which is true by Lemma 3.6.1 (\(\mathcal{F}[P_Z](M) = \mathcal{F}[P_{NF}](M)\)) and by Scattergood [13] (\(\mathcal{F}[P_{CSP}](M) = \mathcal{F}[\text{main}](M)\)).

3.7 Converting the WDT Process

As a full example, in this section we present the conversion of the CSP\(_Z\) specification of the WDT process (see Section 2) into the FDR notation. In particular, this example illustrates the termination issue which must be taken into account in the translation, as explained above. It also shows that the CSP part can be structured using named processes apart from \text{main}.

In this translation the \text{CLK} given-set is implemented as a (finite) free-type (\text{datatype}) so that FDR can analyse the process. This is possible only due to satisfaction of the proof obligations about data abstraction as will be presented in the next chapter.

datatype CLK = noneClock | clk1 | clk2 | clk3 | clk4 | clk5 | clk6

channel reset, failFTR
channel recover
channel timeOut, noTimeOut
channel terminate
channel clockWDT : CLK

WDT =
let
    -- Interface of WDT
    Channels = \{|clockWDT, reset, recover|\}
    lChannels = \{|timeOut, noTimeOut, failFTR, terminate|\}
    Interface = union(Channels, lChannels)
    -- CSP part initialised by the main equation
    main = (Signal \{|terminate |\} Verify)
    Signal = reset \rightarrow Signal \[
        terminate \rightarrow \text{SKIP}
    \]
    Verify = clockWDT\?c \rightarrow (noTimeOut \rightarrow Verify \[
        timeOut \rightarrow (recover \rightarrow Verify) \\
        failFTR \rightarrow terminate \\
        \rightarrow \text{SKIP}
    \)]
    State = \{(cycles, time) | cycles <- \{0,1,2,3,4\}, time <- CLK\}
3.7. CONVerting the Wdt process

Init = { (0, noneClock) }

-- Auxiliary elements: Constant and Function
WDToUt = {clk3, clk6}
WDTP(time, timeout) = member(time, timeout)
com((cycles, time), reset) = { (0, time) }
com((cycles, time), clockWDT.clk) = { (cycles, clk) }
com((cycles, time), recover) = union(if cycles<3 then
  { (cycles+1, time) }
else {},
  if cycles==3 then
  { (0, time) }
else {})
com((cycles, time), noTimeOut) =
  if not WDTP(time, WDTtoUt) then
  { (cycles, time) }
else {}
com((cycles, time), timeout) = if WDTP(time, WDTtoUt) then
  { (cycles, time) }
else {}
com((cycles, time), failFTR) = if cycles==3 then
  { (cycles, time) }
else {}

-- Z Part given by the Z_CSP equation
Z_CSP =
let
  Z(State) =
  let
    Conf = {(com(State, c), c) | c <- diff(Interface, {terminate})}
  within
    [] Conf != {} & (States,Comm): Conf @
      States != {} &
      |_| State': States @ Comm -> Z(State')
    [] terminate -> SKIP
  within Init != {} &
    |_| iState: Init @ Z(iState)
within (main []|Interface|) Z_CSP)\Channels

Note the terminate event in the third line above from the bottom.
3.8 Conclusions

As CSP\(_Z\) is a very recent formal language, few tools have been defined for it. Our effort in this chapter is in the direction of reusing theories and tools in order to propose a Model Checker for CSP\(_Z\). We have achieved this goal by rewriting CSP\(_Z\) in terms of CSP, and then taking the CSP Model Checker FDR for free.

This contribution has been firstly announced in [3] as future work. In [4, 5] we have presented our earlier concrete ideas about Model Checking CSP\(_Z\). Although we have considered a rather simplified syntax it was rich enough to handle the processes of interest. Fischer and Wehrheim [18] extended Mota and Sampaio [4, 5] in the context of CSP\(_OZ\). However, it was in [6] that we have presented the complete strategy for model checking the full CSP\(_Z\) language, further extending [18] by allowing multiprefix communication and by highlighting the potential for optimisation. Indeed, the work of Mota and Sampaio [6] has proposed a way of expressing CSP\(_Z\) in terms of CSP, which is very close to the one independently proposed by [17], and summarised here in Chapter 2.

A subtle and important aspect of this translation strategy is to consider the termination state of the CSP part of the CSP-Z specification. This is missing in [18]. Our treatment of input and output channels and operations are also slightly different from the one given in [18], which considers only single event channels and methods (operations), whereas here we further develop this aspect, giving full treatment to complex event channels and operations.

The translation approach used in this thesis is a current trend to extend model checking of some languages to others, where model checking is originally unavailable. A very interesting research using the same direction is on model checking Java programs [36] via a translation of a subset of Java into the specification language PROMELA of the model checker SPIN [24].

As an additional contribution in this direction, the strategy proposed here has been implemented by Farias [1, 2]. The details of this work is out of the scope of the present thesis; it will be detailed in a master’s dissertation by Farias.
Chapter 4

CSP\textsubscript{Z} Data Abstraction

Impressive efforts have been carried out to compact various classes of transition systems while still preserving most of their properties; currently, even a simple model checker can easily analyse millions of states. However, many systems cannot be automatically analysed due to their size, which is normally induced by the use of infinite data types on communications and process parameters. Indeed, various techniques have been proposed and are still being carefully studied in order to handle a larger class of systems [22, 10]: local analysis, data independence, symmetry elimination, partial order reduction, automatic testing, abstract interpretation, integration of model checkers with theorem provers, etc. Although these research approaches seem promising, full automatic analysis of infinite state systems is still almost unexplored [34]. Actually, the most powerful abstraction techniques still need user expertise for complete and adequate usage. Furthermore, user intervention is needed for elaborating a sufficiently abstract model of the system so that model checking can be applied successfully; and, more importantly, in the current literature there is no strategy to generate such abstractions. The goal of the this chapter is to introduce such a strategy to handle infinite CSP\textsubscript{Z} processes.

In the previous chapter we presented an strategy which translates CSP\textsubscript{Z} to CSP\textsubscript{M}, for the purpose of model checking using FDR. However, the translation usually results in infinite state CSP processes due to the abstract data-types allowed by Z. This is the main motivation for defining the present strategy, since the infinite processes generated by the translation need to be transformed into finite ones, for analysis purposes.

Although, in principle, we could employ our strategy to other specification languages, the choice of CSP\textsubscript{Z} was based on the way a process is described and interpreted. And, further, to show its practical applicability, we use FDR [42]. Recall from Chapters 2 and 3 that every CSP\textsubscript{Z} process is seen as two independent and complementary parts: a behavioural one described in CSP and a data structure based modelled in Z. The behavioural part is assumed to be data independent while data dependent aspects are confined to the data structure part. Finally, recall from Section 3.4 that, although the data structure part is sufficiently expressive, it also assumes a simple (normal)
form which enables the mechanisation of the strategy to be relatively straightforward.

Our approach for defining an strategy combine some ideas about data independence and abstract interpretation for model checking in a slightly different manner from approaches available in the literature. Our approach is more specifically based on Lazić’s approach [58] to model checking data independent CSP processes and Wehrheim’s approach to data abstracting CSP\textsubscript{OZ} [26], although we concentrate on CSP\textsubscript{Z}. The reason to use Wehrheim’s approach instead of, for example the works of Cleaveland and Riely [55] or Sifakis et al [19], is that her approach already uses a CSP algebraic style which is very convenient for using FDR. Wehrheim’s work can be seen as a CSP view of the works of Cleaveland and Riely [55] or Sifakis et al [19]. We use Lazić’s work on the CSP part of a CSP\textsubscript{Z} process because we need only to apply our strategy on the Z part.

We claim that a data dependent infinite state CSP\textsubscript{Z} process can be transformed into a finite CSP\textsubscript{Z} process by using subtypes on its channel declarations, state, input and output variables, and also by rewriting some predicates (the postconditions of the schemas). We present an algorithm for our strategy in which decidable aspects are transferred to the user by means of integrating theorem provers to answer the algorithm’s questions [68]. Currently, the strategy supports model checking properties expressed via data independent processes. For example, it supports deadlock and livelock analysis (See Roscoe [10] and Lazić [58] for more details).

The major advantage of our approach is that, unlike related work in the literature, we calculate the data abstraction necessary for obtaining the finite process. Notably, the most promising works on this research area assume some kind of data abstraction determined by the user [55, 19, 63, 40]. Our approach can be successful when the predicates considered (predicates of com\_schemas) are expressed in a decidable logic, and the behaviour of the Z part is stable, that is, schemas are monotonic with respect to the state space changes. The former characteristic is related to the difficulty in discharging the proof obligations used to determine stability. The latter is a guarantee that the algorithm implementing our approach always terminates. For instance, another interesting point about our algorithm is that it assumes that the Z part of a process has a CSP (normal) form as introduced in Section 3.4. This turns the calculation of the data abstractions relatively straightforward. The question regarding stability is also used by other researchers—for example, by Schmidt [20] and by Levi [23]—in quite similar ways.

This chapter is organised as follows. The following section presents some background information about data independence. Abstract interpretation is introduced in Section 4.2. Section 4.3 introduces the basics of abstracting CSP\textsubscript{Z} processes. The main contribution of this chapter is presented in Section 4.3 where some examples are used for illustrative purposes and an algorithm which implements our strategy is presented. Finally, we present our conclusions and discuss topics for further research.
4.1 Data Independence

Informally, a data independent system $P$ [54, 58] (with respect to a data type $X$) is a system that does not perform any operations involving values of $X$; it only inputs such values, stores them, and compares them for equality. In that case, the behaviour of $P$ is preserved if any concrete data type (which admits equality) is substituted for $X$; in other words, $X$ is a data type parameter of $P$. This is defined more precisely in [58] as shown by the following definition:

**Definition 4.1.1** $P$ is data independent in a type $X$ iff

1. Constants do not appear in $P$, only variables appear, and
2. Only polymorphic operations are used, and
3. Only equality tests are used, and
4. More complex functions and predicates must be defined in terms of 2 and 3, and
5. No replicated constructs (such as indexed parallel compositions) over $X$ may appear in $P$, other than replicated nondeterministic choices over $X$.

Although data independence is a strong property, many concurrent systems fall in this category. For example, consider the following kinds of systems:

- Communication protocols;
- Memories and database systems are typically data independent with respect to the types of addresses and values;
- Agents and the spy in security protocols are almost all data independent with respect to the types of agent identities and keys;
- Nodes in a system consisting of many nodes are sometimes data independent with respect to the type of their identities, although the whole system is typically not.

The work of Lazić [58] deals with the refinement relation between two data independent processes by means of the cardinality of their data independent types. Suppose $P \sqsubseteq_M Q$ has to be checked, for some model $M$, such that $P$ and $Q$ are infinite state and data independent. Further, consider that $P$ and $Q$ are data independent in respect to a unique type $X$. Thus, Lazić analyses the processes $P$ and $Q$ and, depending on which obligations according to Definition 4.1.1 $P$ and $Q$ satisfy, a constraint like $\#X \geq N$, such that $N \in \mathbb{N}_1$, is found. Then, Lazić assures—via theorems

---

$\mathbb{N}_1$ stands for the set of natural numbers without zero ($\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$).
based on $N$—that $P \sqsubseteq M$ $Q$ holds iff $\#X \geq N$. Finally, the refinement is checked in FDR [42] by replacing the symbolic type $X$ for a concrete type $C$, satisfying the constraint on $X$.

The simplest situation—used in this thesis—occurs when $P$ and $Q$ satisfy only the item 1. of Definition 4.1.1. In such cases, the type $X$ might have at least one element (For example, $X = \{0\}$). When equality tests and/or polymorphic operations appear, then the cardinality can be greater than 2. Suppose the use of the equality test $x_1 = x_2$, where $x_1, x_2 \in X$. In this case, the refinement holds iff $\#X \geq 2$. For example, if $X$ is replaced for $\{0\}$ then $x_1 = x_2$ is never falsified. Since equality tests are used on conditionals, the property concerning the alternative $x_1 \neq x_2$ is lost.

Based on this criteria, Lazić classifies CSP processes according to these cardinalities. Such cardinalities are called thresholds and their classification threshold collections.

The above results form the basis to analyse the CSP part of a CSP$_Z$ specification, since the CSP part always satisfy the required condition for data independence. Definition 4.1.2 states the kind of data independence we are focusing.

**Definition 4.1.2 (Trivially Data Independent)** A trivially data independent CSP process $P$ is a data independent process which has no equality tests, nor polymorphic operations. Then, $\#X \geq 1$ is a sufficient threshold for all data type $X$ independent in $P$, in order that all properties of $P$ be preserved.

Definition 4.1.3 is used to guarantee that the CSP$_Z$ specification we are analysing has the simplest data independent process description for its CSP part.

**Definition 4.1.3 (Partially Data Independent)** A CSP$_Z$ specification is partially data independent if its CSP part is trivially data independent.

After stating exactly what the CSP part can be, we concentrate on the Z part using a more general approach in what follows.

## 4.2 Abstract Interpretation

In this section we present some background material necessary to understand how abstract interpretation works and why it is an attractive theory.

Abstract interpretation is a program analysis methodology based on the notion of galois connections (or closure operators), and often used in compiler design and many other contexts [51, 52]. The role of abstract interpretation is to interpret a program in an abstract domain using abstract operations. Hence, the principal advantage is to obtain useful information about a concrete system by executing its abstract version instead.
4.2. ABSTRACT INTERPRETATION

For model checking, the idea is to construct an abstract model from a given specification in such a way that abstract properties can be proven and then useful information of the concrete specification is obtained [22]. This approach can avoid the state space explosion problem by replacing infinite data types by finite ones. The unique drawback of this approach—which is related to the precision of the considered abstraction—is that deterministic operations might become nondeterministic and specific communications might not be observed anymore. Hence, the more precise the abstraction, the more properties about the original system are preserved.

In the following we present some basic definitions necessary for understanding galois connections (see [11] for more information).

**Definition 4.2.1 (Poset)**

A poset (partially ordered set) \( \langle L, \sqsubseteq \rangle \) is a set \( L \) equipped with a binary relation \( \sqsubseteq \) on \( L \), such that for all \( x, y \in L \) the following holds

- \( x \sqsubseteq x \) (Reflexive);
- \( x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y \) (Antisymmetric);
- \( x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z \) (Transitive).

\[ \diamond \]

**Example 4.2.1** The structure \( \langle \mathbb{N}, \leq \rangle \) is a poset where \( \mathbb{N} \) is the set of natural numbers and \( \leq \) is the usual less than order on natural numbers.

\[ \diamond \]

**Definition 4.2.2 (Upper/Lower Bound)**

Let \( \langle L, \sqsubseteq \rangle \) be a poset and let \( S \subseteq L \). An element \( x \in L \) is an upper bound of \( S \) if \( s \sqsubseteq x \) for all \( s \in S \). A lower bound is defined dually. The set of all upper bounds of \( S \) is denoted by \( S^u \) whereas \( S^l \) is used for lower bounds:

\[ S^u = \{ x \in L \mid \forall s \in S \bullet s \sqsubseteq x \} \quad S^l = \{ x \in L \mid \forall s \in S \bullet x \sqsubseteq s \} \]

\[ \diamond \]

If \( S^u \) has a least element, say \( x \), then \( x \) is called the lub (least upper bound) of \( S \). Dually, if \( S^l \) has a greatest element, say \( y \), then \( y \) is called the glb (greatest lower bound) of \( S \). It is common to use \( \text{lub}(x,y) \)—the least element of \( \{x,y\} \)—as \( x \sqcup y \) (read as “\( x \) join \( y \)”) and \( \text{glb}(x,y) \)—the greatest element of \( \{x,y\} \)—as \( x \sqcap y \) (read as “\( x \) meet \( y \)”). Extending these operations to sets we have \( \bigcup S \) and \( \bigcap S \), respectively.
Definition 4.2.3 (Lattice)
Let \( (L, \sqsubseteq) \) be a poset with \( L \) a non-empty set. Then,

- If \( x \sqcup y \) and \( x \sqcap y \) exist for all \( x, y \in L \) then \( (L, \sqsubseteq) \) is called a lattice;
- If \( \sqcup S \) and \( \sqcap S \) exist for all \( S \subseteq L \) then \( (L, \sqsubseteq) \) is called a complete lattice.

Example 4.2.2 The structure \( (P\{0,1\}, \subseteq) \) is a lattice. In particular, it is a complete lattice and its graphical representation can be seen in Figure 4.1.

\[ \begin{array}{c}
\{0,1\} \\
\{0\} \\
\{1\} \\
\{} \\
\end{array} \]

Figure 4.1: Graphical Representation of a Simple Network

Definition 4.2.4 (Monotonic Map)
Let \( (L, \sqsubseteq_L) \) and \( (M, \sqsubseteq_M) \) be posets. Let \( f : L \rightarrow M \) be a function then \( f \) is said to be monotonic iff

\[ \forall x, y \in L \cdot x \sqsubseteq_L y \Rightarrow f(x) \sqsubseteq_M f(y) \]

Example 4.2.3 Successor, \( \text{succ}(x) = x + 1 \), is a commonly used monotonic function.

Definition 4.2.5 (Galois Connection)
Let \( (A, \sqsubseteq_A) \) and \( (C, \sqsubseteq_C) \) be two lattices. If there exist monotonic maps \( \alpha : C \rightarrow A \) (abstraction function) and \( \gamma : A \rightarrow C \) (concretisation function) such that

- \( \alpha \circ \gamma(a) \sqsubseteq_A a \);
- \( c \sqsubseteq_C \gamma \circ \alpha(c) \).
then the representation \( \langle C, \sqsubseteq_C \rangle \xleftarrow{\alpha} \langle A, \sqsubseteq_A \rangle \) is said to be a galois connection. The maps \( \alpha \) and \( \gamma \) are also called adjunctions. Further, from the above conditions we have

\[
\begin{align*}
\alpha &= \lambda X : C \bullet \downarrow \{ Y \in A \bullet X \sqsubseteq \gamma(Y) \} \\
\gamma &= \lambda Y : A \bullet \downarrow \{ X \in C \bullet \alpha(X) \sqsubseteq Y \}
\end{align*}
\]

\( \diamond \)

Note that in the abstract interpretation terminology the order \( \sqsubseteq \) is defined such that \( x \sqsubseteq y \) means \( x \) is more precise than \( y \). Hence, \( \alpha \circ \gamma(a) \sqsubseteq_A a \) means \( \alpha \circ \gamma(a) \) is the best approximation for \( a \) and \( c \sqsubseteq_C \gamma \circ \alpha(c) \) means the application of \( \gamma \circ \alpha \) adds no information to \( c \). The lattice \( \langle A, \sqsubseteq_A \rangle \) represents the lattice of properties of the system, having \( \langle C, \sqsubseteq_C \rangle \) as the usual semantic domain.

In the tradition of abstract interpretation one has to establish adjunctions such that they form a galois connection and, for all concrete objects and operations, propose abstract versions for them. Moreover, this proposal must be done in such a way that the operators (concrete and abstract) are compatible in some sense; this compatibility originates the notions of soundness (safety) and completeness (optimality) \([55, 57]\). For example, let \( f : C \rightarrow D \) be a concrete operation defined over the concrete domains \( C \) and \( D \). Let an abstract interpretation be specified by the following galois connections \( \langle C, \sqsubseteq_C \rangle \xleftarrow{\alpha_{C,A}} \langle A, \sqsubseteq_A \rangle \) and \( \langle D, \sqsubseteq_D \rangle \xleftarrow{\alpha_{D,B}} \langle B, \sqsubseteq_B \rangle \). In addition, let \( f^* : A \rightarrow B \) be the corresponding abstract semantic operation for \( f \). Then, \( f^* \) is sound for \( f \) if \( \alpha_{D,B} \circ f \sqsubseteq f^* \circ \alpha_{C,A} \). In abstract interpretation, completeness is meant as the natural strengthening of the notion of soundness, requiring its reverse relation to hold as well. Hence, \( f^* \) is complete for \( f \) iff \( \alpha_{D,B} \circ f = f^* \circ \alpha_{C,A} \).

A very interesting result about abstract interpretation is that one can find out the best correct approximation of an operation \( f \), if it exists. It is a straightforward application of the theory developed by the Cousots \([51]\), where an abstract operation \( f^* \) is a correct approximation of \( f \) iff \( \alpha_{D,B} \circ f \circ \gamma_{A,C} \sqsubseteq f^* \). Thus, defining a function \( f^{A,B} \) as

\[
\begin{align*}
f^{A,B} = \alpha_{D,B} \circ f \circ \gamma_{A,C}
\end{align*}
\]

leads to an automatically sound abstract version of \( f \). That is, in order to define an abstract interpretation, what is really crucial to determine are the abstract domains and adjunctions because the abstract versions of the objects and operations can be defined constructively. As pointed out in a recent work by Giacobazzi and Ranzato \([57]\), the level of precision of an abstract interpretation is only dependent on the chosen abstract domains. An important result of Giacobazzi and Ranzato \([57]\) shows that \( f^{A,B} \) is always sound but not always complete. Indeed, it is shown that if one can find a complete abstract operation \( f^* \) then it must coincide with the best correct approximation \( f^{A,B} \) as follows

\[
\begin{align*}
\gamma_{A,C} = \lambda x : x & \quad \text{(since } \alpha_{C,A} \circ \gamma_{A,C} = \lambda x : x) \\
\alpha_{D,B} \circ f \circ \gamma_{A,C} & = \quad \text{(by completeness of } f^*) \\
f^{A,B} & = \quad \text{as follows}
\end{align*}
\]
4.3 CSP\textsubscript{Z} Data Abstraction

In this section we present what is the meaning of performing a data abstraction to a CSP\textsubscript{Z} process and what are the conditions (proof obligations) under which a CSP\textsubscript{Z} data abstraction can be considered optimal (complete). The material of this section is based on the work of Wehrheim [26] with additional improvements (see [26] for more details).

Let \( P \) be a partially data independent modular CSP\textsubscript{Z} process, and \textbf{Interface} the set of channel names (interface) of \( P \). Recall from Section 2.2.6 that if \( P \) is modular then \( P = P_{\text{CSP}} \parallel P_{\text{Z}} \). To abstract a CSP\textsubscript{Z} process \( P \) is basically to find an abstract interpretation for the data domains of the channels and state of \( P \), that is, define new domains and new operations for \( P \). Thus, let \( D \) be the data domains of state variables and \( M_c \) the data domain (type of the communicating values) of channel \( c \) (\( c \in \text{Interface} \)). By convention, we assume that communications are split into input (\( M_c^{\text{in}} \)) and output (\( M_c^{\text{out}} \)) communications. Then we assume that the \texttt{com}\textsubscript{c} operations in CSP\textsubscript{Z} have the following signature.

\[
\langle \cdot | \text{com}_c | \rangle : D \times M_c^{\text{in}} \rightarrow \mathcal{P}(D \times M_c^{\text{out}})
\]

Following the abstract interpretation approach, we build abstract \texttt{com}\textsubscript{c} schemas in terms of abstract data domains and abstract interpretations of primitive operations. Thus, let \( D^A \) and \( M^A_c \) be abstract data domains of variables and channels, and let \( h \) and \( r_c \) be abstraction functions. These functions are our left adjunctions \( \alpha \) while our concretisations functions are simply the corresponding identity maps.

\[
\begin{align*}
  h : D &\rightarrow D^A \\
  r_c : M_c &\rightarrow M^A_c
\end{align*}
\]

It is worth noting that communication abstractions (\( r_c \)) are specifically defined for communicating channels. Events do not have communication abstractions; the identity map is used instead.

An abstract interpretation \( \{ | \cdot | \} \) of \texttt{com}\textsubscript{c} operations operates on abstract data domains.

\[
\{ | \text{com}_c | \} : D^A \times M_c^{\text{in},A} \rightarrow \mathcal{P}(D^A \times M_c^{\text{out},A})
\]

By definition [51, 52], an abstract interpretation is compositional, then \( \{ | \text{com}_c | \} \) is built by the abstraction of inner operations. For example: let \( s, s_1, s_2 \) be state variables of type sequence then a predicate \( s' = s_1 \bowtie s_2 \) (in a \texttt{com}\textsubscript{c} schema) is abstracted to \( s'^A = s_1^A \bowtie s_2^A \).

Before introducing the definitions of safe and optimal data abstractions, we have to present the definition of a useful operator to handle with powersets of pairs of values.
4.3. CSP \( Z \) DATA ABSTRACTION

Definition 4.3.1 (Operator \( \times \))

\[
\begin{array}{c}
\times \colon (X \to Z) \times (Y \to W) \to P(X \times Y) \to P(Z \times W) \\
\forall h : X \to Z; r_c : Y \to W; O : P(X \times Y) \bullet \\
(h \times r_c) = \{d^A : Z; m^A : W \mid \exists d : Z; m : Y \bullet (d, m) \in O \land \\
h(d) = d^A \land r_c(m) = m^A \bullet d^A \mapsto m^A\}
\end{array}
\]

Recall from Section 4.2 that abstract domains and operations might be found such that the new interpretation is a safe or optimal abstraction of the original system. These designations come from the work of Cleaveland and Riely [55] meaning that the abstract interpretation is sound or complete, respectively [51, 57]. In the following we present what that means for CSP \( Z \).

Definition 4.3.2 (Safe abstraction) An abstract interpretation \( \{ \cdot \} \) is safe according to abstractions \( h \) and \( r_c \) iff

\[
\forall d^A : D^A; m^A : M^A \bullet \{\text{com}_c \}(d^A, m^A) = \bigcup_{(d,m)}(h \times r_c)(\{\text{com}_c \}(d, m))
\]

where \( (d,m) \in D \times M \) such that \( d^A = h(d), m^A = r_c(m) \).

The previous definition requires that the abstract interpretation for the \( \text{com}_c \) operations capture more information than its original version actually does. It is reasonably obvious why this is required, since the abstraction of a process must have more behaviours (although non-deterministic) than its refinement, in the sense of the theory of process refinement for CSP. Indeed, the non-determinism is provided by the distributed union applied to the abstraction \( (h \times r_c) \) of the original interpretation.

A stronger requirement is stated by the following definition. Instead of introducing non-determinism, we have a precise relationship between the abstract version of an operation and the abstraction of its original result.

Definition 4.3.3 (Optimal Abstraction) An abstract interpretation \( \{ \cdot \} \) of \( \text{com}_c \) is optimal with respect to abstraction functions \( h \) and \( r_c \) iff

\[
\forall d : D; m : M \bullet \{\text{com}_c \}(h(d), r_c(m)) = (h \times r_c)(\{\text{com}_c \}(d, m))
\]

By convention, the abstract version of \( P \) is denoted by \( P^A \), assuming abstract interpretation \( \{ \cdot \} \). The process \( P^A \) is obtained by substituting the types (data domains) of the channels used in the interface by the abstract ones (images of \( r_c \)) and by replacing all \( \text{com}_c \) and \( \text{pre}_c \) operations (which implies replacing inner operations such as \(+, \cap, \leq\), etc.) by their abstract versions.
CHAPTER 4. CSP\(_Z\) DATA ABSTRACTION

Definition 4.3.3 is indeed an equivalence in terms of Z data refinement (\(P_2^Z \equiv P_Z \iff P_2^Z \subseteq P_Z\) and \(P_Z \subseteq P_2^A\)). However, since we are not dealing with \(P_Z\) isolated we also have to consider \(P_{\text{CSP}}\).

Recall from Section 2.4 that a Z data refinement must not interfere in the interface of the process, otherwise we cannot use Lemma 2.4.1. Therefore, to obtain optimal data abstraction for \(P\) in terms of optimal data abstraction for \(P_Z\) we have to guarantee at least that the possible modifications in the interface—caused by the data abstraction of \(P_Z\)—are extended to \(P\), since process refinement is not defined for differing alphabets. To accomplish this, a renaming \(R\) based on the abstraction communication functions is defined.

**Definition 4.3.4 (Interface Abstraction)**

Let \(r_c\) be (safe/optimal) communication abstractions for \(c \in \text{Interface}\) (\(r_c : M_c \rightarrow M_c^A\)). Then the interface abstraction is given by the renaming \(R\)

\[
R = \bigcup_{c \in \text{Interface}} \{m : M_c; \ m^A : M_c^A | r_c(m) = m^A \bullet c.m \mapsto c.m^A\}
\]

The renaming \(R\) is the identity map for events. That is, \(\forall e : \text{EVENTS} \bullet e R e\).

The following law is an important result about applying a renaming which is indeed an injective function. In this case, the renaming is completely conservative and its distribution over the parallel operator is allowed.

**Law 4.3.1 (\(f[\cdot] - || - \text{dist})\)**

\[
f[P || Q] = f[P] || f[Q] \text{ if } f \text{ is injective.}
\]

Definition 4.3.5 is introduced to give support to the lemmas presented subsequently in this section about data abstractions.

**Definition 4.3.5 (Renaming Independence)**

Let \(P\) be a CSP\(_Z\) process and \(Q\) be a data independent CSP process, such that \(\Sigma\) is the alphabet of \(P\) and \(Q\), and the minimum threshold for \(Q\) is \(N\) (\(N \in \mathbb{N}\)), for all channels. Furthermore, let \(R\) be a renaming such that \(\forall c : \Sigma \bullet \text{ran}(\{c\} < R) \geq N\), and \(\Sigma'\) be a new alphabet with \(\Sigma' = \text{ran} R\) (where \(P\) is simply a short form for \(P_\Sigma\)). If

\[
Q_\Sigma \subseteq P_\Sigma \leftrightarrow Q_{\Sigma'} \subseteq P_\Sigma[R]
\]

and the unique difference between \(Q_\Sigma\) and \(Q_{\Sigma'}\) is the specific events, then \(Q\) is said to be renaming independent.

In order to deal with the CSP and the Z part individually through a renaming, we need to show that the renaming we are using and the data independence characteristic of the CSP part enables such a distributivity between renaming and parallelism.
Lemma 4.3.1 Let \( P \) be a partially data independent \( \text{CSP}_Z \) specification with interface \( I \), such that \( P_{\text{CSP}} \) represents its CSP part and \( P_Z \) its \( Z \) part. Furthermore, let \( R \) be an interface abstraction over \( P \). Then,

\[
P[R] = P_{\text{CSP}}[R] \parallel P_Z[R]
\]

\[
\text{ran } R
\]

Proof. By the definition of \( P_Z \), we have

\[
\left( \begin{array}{c}
\begin{array}{c}
\pre \text{com}_{a_1} \& a_1 \rightarrow P_{\text{com}_{a_1}}(\text{State}) \\
\square \begin{array}{c}
\pre \text{com}_{a_2} \& a_2 \rightarrow P_{\text{com}_{a_2}}(\text{State}) \\
\vdots \\
\square \begin{array}{c}
\pre \text{com}_{a_n} \& a_n \rightarrow P_{\text{com}_{a_n}}(\text{State})
\end{array}
\end{array}
\end{array}
\end{array}
\right)
\]

\[
[ R ] = P_{\text{CSP}}[ R ] \parallel P_Z[ R ]
\]

\[
\text{ran } R
\]

Let us consider the following situations.

1. Let \( b \) and \( c \) be two distinct events, such that \( P_{\text{CSP}} \) and \( P_Z \) can synchronise on them. Recall from Definition 4.3.5 that, for \( b \) and \( c \), the renaming \( R \) corresponds to the identity map, which is naturally an injective function. Thus, applying Law 4.3.1 we get

\[
P[R] = (P_{\text{CSP}} \parallel P_Z)[R] = P_{\text{CSP}}[R] \parallel P_Z[R].
\]

2. Let \( d \) be a channel such that \( d.m \) and \( d.m' \) are two distinct events ready for communication of \( P_{\text{CSP}} \) and \( P_Z \). This analysis takes two further possibilities

(a) The abstraction of \( d.m \) and \( d.m' \) can be distinct, that is

\[
r_d(d.m) = d.m_1^A \land r_d(d.m') = d.m_2^A \land d.m_1^A \neq d.m_2^A
\]

Thus, \( R \) becomes injective and follows the same analysis of case 1.

(b) The abstractions can result the same abstract value, that is

\[
r_d(d.m) = r_d(d.m') = d.m^A
\]

This is the most difficult case because we do not have an injective function as a renaming. The unique transitions we can have for \( \text{CSP}_Z \) processes, in this situation, are

\[
\begin{array}{c}
P_{\text{CSP}} \xrightarrow{d.m} P'_{\text{CSP}}, P_Z \xrightarrow{d.m} P'_Z \\
P_{\text{CSP}} \parallel P_Z \xrightarrow{d.m} P'_{\text{CSP}} \parallel P'_Z
\end{array}
\]

\[
\begin{array}{c}
P_{\text{CSP}} \xrightarrow{d.m'} P''_{\text{CSP}}, P_Z \xrightarrow{d.m'} P''_Z \\
P_{\text{CSP}} \parallel P_Z \xrightarrow{d.m'} P''_{\text{CSP}} \parallel P''_Z
\end{array}
\]
From these transitions, we have that

\[
(P_{\text{CSP}} \| P_{\text{Z}})[R] \xrightarrow{d.m^A} \begin{cases} (P'_{\text{CSP}} \| P'_{\text{Z}})[R] \\ I \\ \text{or,} \\ (P''_{\text{CSP}} \| P''_{\text{Z}})[R] \\ I \end{cases}
\]

On the other hand, by applying the renaming to the CSP and Z parts we have

\[
P_{\text{CSP}}[R] \xrightarrow{d.m^A} \begin{cases} P'_{\text{CSP}}[R] \\ \text{or,} \\ P''_{\text{CSP}}[R] \end{cases} \quad \text{and} \quad P_{\text{Z}}[R] \xrightarrow{d.m^A} \begin{cases} P'_{\text{Z}}[R] \\ \text{or,} \\ P''_{\text{Z}}[R] \end{cases}
\]

By applying the parallel operator, now synchronising on \((\text{ran } R)\), these transitions would yield four combinations. However, due to the abstraction \(r_d(d.m) = r_d(d.m') = d.m^A\), we have that the Z part is equivalent in the sense of the preconditions enabled before and after these transitions have occurred. Thus, by the renaming we have \(P'_{\text{Z}}[R] \equiv P''_{\text{Z}}[R]\). For the CSP part, we know that \(P_{\text{CSP}}\) is trivially data independent and \(R\) (Definition 4.3.4) does not eliminate any channel from the interface (\(R\) satisfies the trivially data independent requirement). Thus, it follows that \(P'_{\text{CSP}}[R] \equiv P''_{\text{CSP}}[R]\).

Then

\[
(P_{\text{CSP}}[R] \| P_{\text{Z}}[R]) \xrightarrow{\text{ran } R} (P'_{\text{CSP}}[R] \| P'_{\text{Z}}[R])
\]

And, from the above observations, we also have that \((P'_{\text{CSP}} \| P'_{\text{Z}})[R] \equiv (P''_{\text{CSP}} \| P''_{\text{Z}})[R]\) as long as the parallel operator, for CSP\(_{\text{Z}}\) processes, acts exactly like a conjunction. ♦

The Laws 4.3.2 and 4.3.3 are introduced by Roscoe in [10] and used in the subsequent lemmas to justify the results concerning the Z part of a CSP\(_{\text{Z}}\) specification.

**Law 4.3.2** \([\rightarrow \text{ dist}]\)

\[
(P \sqcap Q)[R] = P[R] \sqcap Q[R]
\]

**Law 4.3.3** \([\rightarrow \Box \rightarrow \text{ dist}]\)

\[
(P \Box Q)[R] = P[R] \Box Q[R]
\]

The following lemmas relate the original and abstract versions of a CSP\(_{\text{Z}}\) process. These lemmas are extensions—concerning only the CSP part—of theorems proposed by Wehrheim in [26]. The motivation for proposing these extensions is that Wehrheim’s theorems can be invalid if we allow equality tests and/or polymorphic operations in the CSP part.
Lemma 4.3.2 Let $P$ be a partially data independent $CSP_Z$ specification and $P^A$ its abstract version, defined by a safe abstract interpretation $\{ \cdot \}$ with interface abstraction given by the renaming $R$. Then $P^A \sqsubseteq_T P[R]$.

Proof. From Lemma 4.3.1, we have

$$P[R] = P_{CSP}[R] \parallel P_Z[R]$$

By the definition of $P_Z$, we get

$$P_{CSP}[R] \parallel \begin{array}{c}
\text{pre com}_{a_1} & \& a_1 \to P_{com_{a_1}}^Z(State) \\
\vdots & \vdots \\
\text{pre com}_{a_n} & \& a_n \to P_{com_{a_n}}^Z(State)
\end{array}$$

From Laws 4.3.2 and 4.3.3, we rewrite the above process as

$$P_{CSP}[R] \parallel \begin{array}{c}
\text{pre com}_{a_1} & \& a_1 \to P_{com_{a_1}}^Z(State) \\
\vdots & \vdots \\
\text{pre com}_{a_n} & \& a_n \to P_{com_{a_n}}^Z(State)
\end{array}$$

For the abstract version, we already have

$$P_{CSP}^A \parallel \begin{array}{c}
\text{pre com}_{a_1}^A & \& a_1^A \to (P_{com_{a_1}}^Z(State))^A \\
\vdots & \vdots \\
\text{pre com}_{a_n}^A & \& a_n^A \to (P_{com_{a_n}}^Z(State))^A
\end{array}$$

From the definition of data independence, as long as $\forall a : \Sigma \bullet (\{a\} \triangleleft R) \neq \emptyset$ (which is valid by the Definition 4.3.4) then $P_{CSP}[R] = P_{CSP}^A$. However, to analyse the $Z$ part, we have to employ the Definition 4.3.2. Let $a_i$ be a communication, then

$$\forall d^A : D^A, \; m^A : M^A \bullet \{ \text{com}_{a_i} \} (d^A, m^A) = \bigcup_{(d, m)} (h \times r_c(\{ \text{com}_{a_i} \})(d, m))$$

where $(d, m) \in D \times M$ such that $d^A = h(d), m^A = r_c(m)$. According to the previous definition for $a_i$, if $(P_{com_{a_i}}^Z(State))^A[R]$ can change the current state $S$ to $S'$ or $S''$, but not both at the same time, the abstract version $(P_{com_{a_i}}^Z(State))^A$ can, due to the distributed union. That is, in
the sense of CSP, the process $\left(P_{Z \text{com\_a}_i(\text{State})}^A\right)$ can exhibit more behaviours (nondeterministically) than $\left(P_{Z \text{com\_a}_i(\text{State})}^A\right)[R]$. As a result of this more general nondeterministic behaviour, we get

$$T \left(\left(P_{Z \text{com\_a}_i(\text{State})}^A\right)[R]\right) \subseteq T \left(\left(P_{Z \text{com\_a}_i(\text{State})}^A\right)\right)$$

which—by definition—means that $P_{Z \text{com\_a}_i(\text{State})}^A \subseteq T P_{Z \text{com\_a}_i(\text{State})}[R]$. And, by correspondence between the CSP parts, we have $P_{Z \text{com\_a}_i(\text{State})}^A \subseteq T P_{Z \text{com\_a}_i(\text{State})}[R]$. ♦

Lemma 4.3.2 shows clearly that $P_{Z \text{com\_a}_i(\text{State})}^A$ captures more information than $P_{Z \text{com\_a}_i(\text{State})}^A$ actually does. That is the reason why $P_{Z \text{com\_a}_i(\text{State})}^A$ is put on the left side of the process refinement order relation $\subseteq_T$: it is more abstract than $P_{Z \text{com\_a}_i(\text{State})}^A$. Moreover, not only we can analyse only trace-based and renaming independent properties with this result, but the useful information we can get still depends on the refinement check. For example, let $Q \subseteq_T P$ be a desired refinement check, where $Q$ is (infinite) data and renaming independent and $P$ is (infinite) partially data independent. The first step is to check $Q \subseteq_T P_{Z \text{com\_a}_i(\text{State})}^A$ (since we have $P_{Z \text{com\_a}_i(\text{State})}^A \subseteq_T P_{Z \text{com\_a}_i(\text{State})}[R]$, guaranteed by Lemma 4.3.2, and the refinement relation $\subseteq$ is transitive). If it is valid then $Q \nsubseteq_T P$ is, as well, by the renaming independent hypothesis. Otherwise, if $Q \nsubseteq_T P_{Z \text{com\_a}_i(\text{State})}^A$ then it is not necessarily the case of $Q \nsubseteq_T P$, because the transitivity only helps when the first refinement check is valid. This fact will be precisely characterised on Theorem 4.3.1.

A stronger result is obtained by the following lemma. Since the following result is based on an equality relation then the refinement check is useful even if the first refinement check, illustrated in the preceding paragraph, is not valid. This fact is presented formally on Theorem 4.3.2.

**Lemma 4.3.3** Let $P$ be a partially data independent CSP $Z$ specification and $P_{Z \text{com\_a}_i(\text{State})}^A$ its abstract version defined by optimal abstract interpretation $\{\cdot\}$ with interface abstraction given by the renaming $R$. Then $P_{Z \text{com\_a}_i(\text{State})}^A = F D P_{Z \text{com\_a}_i(\text{State})}^A$.

**Proof.** Similar to Lemma 4.3.2, except for the Definition 4.3.3—in respect to a given communication $a_i$—which gives us

$$\forall d : D; m : M \bullet \{\cdot\} \text{com\_a}_i \{h(d), r_c(m)\} = (h \times r_c)(\{\cdot\} \text{com\_a}_i \{d, m\})$$

This equality implies

$$\text{precom\_a}_n^A = \text{precom\_a}_n \land \left(P_{Z \text{com\_a}_i(\text{State})}^A\right)^A = \left(P_{Z \text{com\_a}_i(\text{State})}^A\right)[R]$$

From the above result, we know that the preconditions contribute to the refusals (failures). We also know that the $Z$ part has no divergence. Thus, by a pairwise analysis of the CSP $Z$ processes before and after abstraction, we obtain

$$F(P_{Z \text{com\_a}_i(\text{State})}^A) = F(P_{Z \text{com\_a}_i(\text{State})}^A)$$
$$D(P_{Z \text{com\_a}_i(\text{State})}^A) = D(P_{Z \text{com\_a}_i(\text{State})}^A) = \emptyset$$
$$F(P_{Z \text{com\_a}_i(\text{State})}^A) = F(P_{Z \text{com\_a}_i(\text{State})}^A)$$
$$D(P_{Z \text{com\_a}_i(\text{State})}^A) = D(P_{Z \text{com\_a}_i(\text{State})}^A)$$

Therefore, the result holds.
4.3. CSP\textsubscript{Z} DATA ABSTRACTION

which is the definition for \( P[R] =_{FD} P^A \).

As previously highlighted, in general Lemmas 4.3.2 and 4.3.3 concern only renamed versions of the CSP\textsubscript{Z} original processes. Thus, only those properties preserved via renaming might be checked. Wehrheim [26] still tries to avoid this limitation via algebraic manipulation but the problem of infinity occurs again. Furthermore, observe that Lemmas 4.3.2 and 4.3.3 provide partial information, since the theory of process refinement requires that the alphabet of the processes—involved in a refinement—must be equal. Therefore, before the refinement relation \( Q \sqsubseteq P \) can be performed, a basic obligation is to have \( \alpha Q = \alpha P \). However, the abstraction of an infinite process consists exactly in changing the original alphabet to a finite one. Hence, for a refinement check like \( Q \sqsubseteq P \), if \( P \) is abstracted to \( P^A \) we cannot check even \( Q \sqsubseteq P^A \) directly, \( \alpha Q \neq (\alpha P^A = \alpha P[R]) \).

Consequently, the following theorems are new results with respect to the work developed by Wehrheim [26], taking the work of Lazić [58] as support. They state exactly what it is needed to use Lemmas 4.3.2 and 4.3.3 in practice. That is, when we can change the alphabets of the two processes involved in a refinement check without changing the desired properties.

**Theorem 4.3.1 (Safe Data and Renaming Independence)**

Let \( P \) be an infinite partially data independent CSP\textsubscript{Z} process and \( Q \) be an infinite CSP data independent process with minimum threshold of \( N \) (\( N \in \mathbb{N}_1 \)). Furthermore, assume that \( Q \) is renaming independent and the alphabet (interface) of \( P \) and \( Q \) is \( \Sigma \). If \( P^A \) is a safe abstraction for \( P \) according to an abstract interpretation \( \{ \cdot \} \), such that the abstraction renaming \( R \) provides the minimum threshold of \( N \) for all channels (\( \forall c : \Sigma \bullet \text{ran}(\{c\} \triangleleft R) \geq N \)), then

\[
Q_{\Sigma} \sqsubseteq_T P^A \Rightarrow Q \sqsubseteq_T P
\]

**Proof.** The proof follows by the preceding results

1. \( Q_{\Sigma} \sqsubseteq_T P^A \) [By hypothesis]
2. \( P^A \sqsubseteq_T P[R] \) [By Lemma 4.3.2]
3. \( \Rightarrow Q_{\Sigma} \sqsubseteq_T P[R] \) [By steps 1., 2. and the transitivity of \( \sqsubseteq \)]
4. \( \Rightarrow Q \sqsubseteq_T P \) [By Definition 4.3.5 and \( \forall c : \Sigma \bullet \text{ran}(\{c\} \triangleleft R) \geq N \)]

The main difference between the Theorem 4.3.1 and 4.3.2 is that the former is successful only when the abstract refinement is satisfied. If the abstract refinement is not satisfied we do not know nothing about the refinement of the concrete processes. The latter, however, is a stronger result and applies when the refinement is (not) satisfied.
Theorem 4.3.2 (Optimal Data and Renaming Independence)

Let $P$ be an infinite partially data independent $\text{CSP}_Z$ process and $Q$ be an infinite $\text{CSP}$ data independent process with minimum threshold of $N$ ($N \in \mathbb{N}_1$). Furthermore, assume that $Q$ is renaming independent and the alphabet (interface) of $P$ and $Q$ is $\Sigma$. If $P^A$ is optimal abstraction for $P$ according to an abstract interpretation $\| \cdot \|$, such that the abstraction renaming $R$ provides the minimum threshold of $N$ for all channels ($\forall c : \Sigma \bullet \text{ran}(\{c\} \triangleleft R) \geq N$), then

$$Q_{\Sigma'} \sqsubseteq_{FD} P^A \Leftrightarrow Q \sqsubseteq_{FD} P$$

Proof. The proof follows by the preceding results

1. $Q_{\Sigma'} \sqsubseteq_{FD} P^A$ [By hypothesis]
2. $P^A =_{FD} P[R]$ [By Lemma 4.3.3]
3. $\Leftrightarrow Q_{\Sigma'} \sqsubseteq_{FD} P[R]$ [By steps 1. and 2.]
4. $\Leftrightarrow Q \sqsubseteq_{FD} P$ [By Definition 4.3.5 and $\forall c : \Sigma \bullet \text{ran}(\{c\} \triangleleft R) \geq N$]

Simple examples for infinite data and renaming independent $\text{CSP}$ processes are the deadlock-free process

$$DF = \sqcap x : \Sigma \bullet DF$$

and the livelock-free process

$$LF = \text{Chaos}(\Sigma) \sqcap \text{STOP}$$

In particular, the FDR tool uses exactly these processes when one has to check deadlock or livelock-freedom (See Roscoe [10] for more details). That is, let $P$ be a $\text{CSP}$ process. Thus, $P$ is deadlock-free iff $DF \sqsubseteq_{FD} P$. Therefore, without giving a full-list of infinite data and renaming independent $\text{CSP}$ processes, our results already apply directly for these—very usual—properties.

The following example illustrates what means a data abstraction with a simple and concrete $\text{CSP}_Z$ process.

Example 4.3.1 (A Data Abstraction)

Let $P$ be the $\text{CSP}_Z$ process

spec $P$

```
chan a, b : N
main = a?x -> b?y -> main
```
4.3. CSP DATA ABSTRACTION

\[\text{State} \triangleq [c : \mathbb{N}]\]

\[\text{com}_a \triangleq [\Delta \text{State}; x? : \mathbb{N} \mid c' = x']\]

\[\text{com}_b \triangleq [\Delta \text{State}; y? : \mathbb{N} \mid c \ast y? > 0 \land c' = y']\]

end_spec

Furthermore, let \( A = \{\text{pos}, \text{nonPos}\} \) be our abstract domain with abstraction functions

\[r_a = r_b = h = \{n : \mathbb{N} \mid n > 0 \cdot n \mapsto \text{pos}\} \cup \{n : \mathbb{N} \mid n \leq 0 \cdot n \mapsto \text{nonPos}\}\]

Because we have interface abstraction we build the renaming \( R \) as

\[R = \{n : \mathbb{N} \mid n > 0 \cdot a.n \mapsto a.\text{pos}\} \cup \{n : \mathbb{N} \mid n \leq 0 \cdot a.n \mapsto a.\text{nonPos}\} \cup \{n : \mathbb{N} \mid n > 0 \cdot b.n \mapsto b.\text{pos}\} \cup \{n : \mathbb{N} \mid n \leq 0 \cdot b.n \mapsto b.\text{nonPos}\}\]

and the abstract version of the operators are

\[s_1 \triangleq s_2 = \begin{cases} \text{pos}, & s_1 = s_2 = \text{pos} \\ \text{nonPos}, & \text{otherwise} \end{cases}\]

\[s_1 \triangleright s_2 = \begin{cases} \text{true}, & s_1 = \text{pos} \land s_2 = \text{nonPos} \\ \text{false}, & \text{otherwise} \end{cases}\]

Then, applying \( r \) to the channels, \( h \) to state variable \( c \) and constants (such as 0) and using the abstract operators, we get

\[\text{spec } P^A\]

\[\text{chan } a,b : A\]

\[\text{main } = a?x \rightarrow b?y \rightarrow \text{main}\]

\[\text{State}^A \triangleq [c : A]\]

\[\text{com}_a^A \triangleq [\Delta \text{State}^A; x? : A \mid c' = x']\]

\[\text{com}_b^A \triangleq [\Delta \text{State}^A; y? : A \mid c \ast y? > \text{nonPos} \land c' = y']\]

end_spec

and \( P[R] \) is simply \( P \) using the renaming \( R \) determined above.

Considering Example 4.3.1, we can prove that \( P^A =_{\mathcal{FD}} P[R] \) is valid. A more complex predicate, such as \( x > y \), would require a larger domain, such as \( OA = \{\text{pos}, \text{zero}, \text{neg}\} \) with obvious interpretations.

4.3.1 Generating Optimal Data Abstraction

In this section we present a first attempt to develop a strategy to mechanically derive optimal data abstraction. This strategy is informally introduced using simple examples and then the formal part is presented.
Before presenting the examples, let us introduce some simple, but clarifying, concepts. We call by a property a conjunction involving the preconditions for all schemas declared in the Z part. Hence, if the Z part of a CSP\textsubscript{Z} specification has the schema operations \texttt{com\textsubscript{a}}, \texttt{com\textsubscript{b}}, and \texttt{com\textsubscript{c}} then

\begin{align*}
\text{pre com\textsubscript{a}} \land \neg \text{pre com\textsubscript{b}} \land \text{pre com\textsubscript{c}} \\
\text{or} \\
\neg \text{pre com\textsubscript{a}} \land \neg \text{pre com\textsubscript{b}} \land \text{pre com\textsubscript{c}}
\end{align*}

are valid properties, while

\begin{align*}
\text{pre com\textsubscript{a}} \\
\text{or} \\
\neg \text{pre com\textsubscript{b}} \land \text{pre com\textsubscript{c}}
\end{align*}

are not. A property requires that all preconditions are present because the set of all properties forms the \textit{lattice of the preconditions} for the process. Later in this section, we present this idea more formally. For now, it is simply a new concept that must be introduced.

The terms \textit{stable}, \textit{periodic}, and \textit{cyclic} are used interchangeably, through this section, to mean that some traces—and, consequently, properties—repeat forever.

Finally, although not explicitly stated we are assuming that the Z part has the (normal) form introduced in Section 3.4. As previously pointed out, this makes our mechanical data abstraction relatively straightforward.

It is worth pointing out that, by only using the Z part of a CSP\textsubscript{Z} process to determine an abstraction, we are requiring a more stronger property than it actually needs to be. That is, it can be the case that we cannot find an abstraction for a process because we have only considered the Z part, but if we consider both parts—acting as a whole—perhaps we could derive an abstraction; indeed, more easily. In general, the CSP part acts as a filter to the Z part and vice-versa. In the present work, we have not considered this situation for simplicity of presentation.

Initially, we present an example taken from Wehrheim [26], where the data abstraction was proposed by the user. We demonstrate that following our informal strategy we are able to calculate such a data abstraction.

\textbf{Example 4.3.2 (Infinite Clock)}

\begin{quote}
\textit{Let $P_{\text{Clock}}$ be an infinite CSP\textsubscript{Z} process given by}
\end{quote}

\begin{verbatim}
\textbf{spec} $P_{\text{Clock}}$
\textbf{chan} tick, tack
\textbf{main} = $\Box$ e : \{tick, tack\} $\bullet$ e $\rightarrow$ main
\end{verbatim}
4.3. CSP\textsubscript{Z} DATA ABSTRACTION

\[
\begin{align*}
\text{State} & \equiv [n : N] \\
\text{Init} & \equiv [\text{State} \mid n' = 0] \\
\text{com\_tick} & \equiv [\Delta \text{State} \mid n \mod 2 = 0 \land n' = n + 1] \\
\text{com\_tack} & \equiv [\Delta \text{State} \mid n \mod 2 = 1 \land n' = n + 1]
\end{align*}
\]

end_spec

Set the abstraction data domain to be equal to the concrete one. Set abstraction function \( h \) to be the identity map. Recall from Section 4.3 that the abstractions \( r_{\text{tick}} \) and \( r_{\text{tack}} \) are not defined since there is no communication. Therefore we already know that we do not need a renaming (interface abstraction). Our first step is very simple: apply the corresponding transition rules (see Section 2.3) to the Z part until the set of enabled preconditions in the current state has already occurred in an earlier state. This step yields the LTS of Figure 4.2. Note that the precondition \( \text{pre\_com\_tick}, n \mod 2 = 0 \) (\( n \) is even) is valid in \( n = 0 \) and \( n = 2 \). At this point we perform our second step: try to prove that this repetition is permanent. Let \( \text{conj} \) be a conjunction of preconditions (property) and \( \text{comp} \) be a sequential composition as follows

\[
\begin{align*}
\text{conj} & \equiv \text{pre\_com\_tick} \land \neg \text{pre\_com\_tack} \\
\text{comp} & \equiv \text{com\_tick} \circ \text{com\_tack}
\end{align*}
\]

Then the general predicate to be proven is

\[
\forall \text{State}; \text{State}' \mid \text{conj} \land \text{comp} \Rightarrow \text{conj}'
\]

This predicate (we call it by stability predicate) can be proven by theorem provers like Z-Eves [46] or ACL2 [44], for example.

Our third step checks the proof status of the stability predicate. If it is valid then the abstraction function \( h \) is modified. Furthermore, this validity assures an equivalence relation—under the conjunction of preconditions (the property)—between the states before and after the sequential composition of \( \text{com\_} \) operations, including the operations inside the schema composition. That is,
as long as \( \text{com\_tick} \land \text{com\_tack} \) is stable then the next possible schema operation must be \( \text{com\_tick} \), and after this the next must be \( \text{com\_tack} \), and so on. Thus, from the above predicate we build the equivalence relation

\[
E_{\text{tick}} = \{ n : \mathbb{N} \mid n \mod 2 = 0 \cdot n \mapsto n + 2 \}^*
\]

\[
E_{\text{tack}} = \{ n : \mathbb{N} \mid n \mod 2 = 1 \cdot n \mapsto n + 2 \}^*
\]

which, for each partition, we take the minimum elements to construct the abstraction function. It is worth noting that \( n \mod 2 = 0 \) is a reduced form for the property \( \text{pre\_com\_tick} \land \neg \text{pre\_com\_tack} \), the same way \( n \mod 2 = 1 \) is a simplification for \( \neg \text{pre\_com\_tick} \land \text{pre\_com\_tack} \).

\[
h(n) = \begin{cases} 
0, & 0 \ E_{\text{tick}} n \\
1, & 1 \ E_{\text{tack}} n 
\end{cases}
\]

That is, the abstraction function is induced by the equivalence relation built. After that, we discard this execution path and try to explore another one, repeating the previous steps. Since our example does not have any other paths to explore, we start the final step which builds the abstract domains and abstract operators. For our example, the abstract domain is simply \( A = \{0, 1\} \) (the image of \( h \)), and the abstract version of the successor operator is simply the application of the abstraction (\( \alpha = h \)) and concretisation (\( \gamma = i_A \)) functions as follows

\[
\{ | \lambda x : \mathbb{N} \cdot x + 1 | \} = \alpha \circ (\lambda x : \mathbb{N} \cdot x + 1) \circ \gamma = \lambda x^A : A \cdot h(x^A + 1)
\]

That is, the abstraction is obtained by replacing the concrete domains, applying the abstraction function \( h \) to the constants, and the concrete operators are abstracted by an application of the abstraction function to the result. It is worth observing that our strategy is done in such a way that we have not to abstract the preconditions: the abstraction of the precondition is the precondition itself (identity). The reason for this is that our abstract domains are always the subsets of the original types determined from the lattice of the preconditions (repetition of the set of preconditions enabled).

Note that this abstraction is optimal by construction. The absence of communication abstractions (renaming) yields an equivalence under Z data refinement and process refinement, that is, \( P_{\text{Clock}} \equiv_{\mathcal{FD}} P^A_{\text{Clock}} \) (see Lemma 4.3.3).

It is worth noting that the stability predicate originates from the lattice of the preconditions of the Z part: all preconditions disabled lead to deadlock whereas all preconditions enabled lead to full nondeterminism. This lattice is known as the lattice of properties in the terminology of abstract interpretation [52]. If, during the symbolic execution of the Z part, we achieve a point (trace) such that after it the set of preconditions (a given property in the lattice of preconditions) is always the same, the domain used until that point can be seen as a representative for the future values because they all have the same property.

The following example is slightly more difficult than the previous one with respect to preserving useful properties, such as, deadlock-freedom.
Example 4.3.3 (A Precise Loop)

Let $P$ be a CSP$_Z$ process given as

$$
\text{spec } P
\begin{align*}
\text{chan } a, b \\
\text{main } = a \rightarrow \text{main} \Downarrow b \rightarrow \text{main}
\end{align*}
$$

$$
\begin{align*}
\text{State } &\equiv \{c : \mathbb{N}\} \\
\text{Init } &\equiv \{\text{State}') | c' = 0\} \\
\text{com}_a &\equiv \{\Delta \text{State} | c \leq 5 \land c' = c + 1\} \\
\text{com}_b &\equiv \{\Delta \text{State} | c \geq 5 \land c' = c + 1\}
\end{align*}
$$

end_spec

Again we start by setting the abstract domain as $\mathbb{N}$ and $h$ to be the identity. After that we begin the exploration of the LTS (of the Z part) in a lazy fashion, observing the repetition of the set of valid preconditions. To ease the explanation, observe Figure 4.3. This figure shows that we need 6 expansions, and respectively 5 stability predicates with status false, in order to get a stable path. Let $\text{conj}$ be the property being repeated and $\text{comp}$ the sequential composition where this is happening

$$
\begin{align*}
\text{conj } &\equiv \neg \text{precom}_a \\
\text{comp } &\equiv \text{com}_b
\end{align*}
$$

and the general predicate to be proven is

$$
\forall \text{State}; \text{State}' | \text{conj } \bullet \text{comp } \Rightarrow \text{conj}'
$$

From this predicate we achieve the following unique equivalence relation, since the sequential composition is built by only one schema operation.

$$
E = \{c : \mathbb{N} | c > 5 \bullet c \mapsto c + 1\}^*
$$

where $c > 5$ is the reduced form for $\neg \text{precom}_a \land \text{precom}_b$. The abstraction function is built in terms of the least elements of each partition. Then

$$
\begin{align*}
h(c) = \begin{cases} 
6, & c \in E \\
6, & c + 1 \\
c, & \text{otherwise}
\end{cases}
\end{align*}
$$

which determines the abstract data domain $A = 0 \ldots 6$ and $\{n + 1\} = \lambda n^A : A \bullet h(n^A + 1)$.

The following example is presented to demonstrate that there are situations where our strategy fails. These situations are exactly when at least one part of the process is unstable, that is, it does not admit an infinite periodic behaviour.
Example 4.3.4 (Safe abstraction)

spec ND

\( \text{chan } a, b, c : \mathbb{N} \)
\( \text{main} = a!v_1 \rightarrow \text{main} \) \( \boxplus b!v_2 \rightarrow \text{main} \) \( \square c!v_3 \rightarrow \text{main} \)

\( \text{State} \doteq [x : \mathbb{N}] \)
\( \text{Init} \doteq [\text{State}' \mid x' = 1] \)
\( \text{com}_a \doteq [\Delta \text{State}; v_1! : \mathbb{N} \mid x \leq 5 \land v_1! = x \land x' = x + 1] \)
\( \text{com}_b \doteq [\Delta \text{State}; v_2! : \mathbb{N} \mid x \geq 5 \land v_2! = x \land x' = x - 3] \)
\( \text{com}_c \doteq [\Delta \text{State}; v_3! : \mathbb{N} \mid (x < 4 \lor x \geq 6) \land v! = x \land x' = 2 \times x] \)

end_spec

We set \( \mathbb{N}^A \) to \( \mathbb{N} \) and \( h, r_a, r_b, r_c \) to \( v_0 \). Then, we expand the Z part (see Figure 4.4). Various stability predicates are unsuccessfully checked until \( x = 6 \) is achieved. We find optimal abstraction from \( x = 6 \) to infinite via \text{com}_c, however from \( x = 6 \) via \text{com}_b the property does not stabilise and the stability predicates are checked forever; that is, in this example the algorithm is non-terminating.

However, we can find—in a manual way—a data abstraction for this process although using some principles of the algorithm. We can build a safe abstraction by introducing non-determinism into the schema operation \text{com}_b. For instance, this operation must work in the following way: at \( x = 6 \) its application can yield \( x = 3 \) (its traditional interpretation) as well as \( x = 6 \) again. This is required in order to capture—abstractly—the behaviour of this process. That is, suppose that \( x = 50 \), hence \text{com}_c can be applied normally (we already have determined that it is stable, that is, \( \neg \text{pre com}_a \land \text{pre com}_b \land \text{pre com}_c \) is always valid through \text{com}_c). But the application of the \text{com}_b schema operation will decrease the value of \( x \)—in finitely many steps—until the property \( \neg \text{pre com}_a \land \text{pre com}_b \land \text{pre com}_c \) is no longer valid. This capacity of coming from a future stable place to a finite one introduces non-stability and, therefore, cannot be captured via our mechanical data abstraction approach, for the present moment.

By considering the abstract domain \( \mathbb{N}^A = 0 \ldots 6 \), we can simply find a safe abstraction where
4.3. CSP\textsubscript{Z} DATA ABSTRACTION

\[
h = r_a = r_b = r_c = \lambda x : \mathbb{N} \cdot \begin{cases} 
6, & \text{if } x \geq 6 \\
x, & \text{otherwise}
\end{cases}
\]

with abstract versions for the operations given by

\[
\{ \lambda x : \mathbb{N} \cdot x - 3 \} = (\lambda x : 0.5 \cdot x - 3) \cup (\{6\} \times \{3, 6\})
\]

which is now relational. And the remainder operations are abstracted as usually

\[
\{ \lambda x : \mathbb{N} \cdot x + 1 \} = \lambda x : \mathbb{N}^4 \cdot h(x + 1)
\]

\[
\{ \lambda x : \mathbb{N} \cdot 2 \times x \} = \lambda x : \mathbb{N}^4 \cdot h(2 \times x)
\]

Recall from Lemma 4.3.2 that, as long as this process has communication abstraction and is a safe abstraction then only trace-based data and renaming independent properties can be checked.

![Figure 4.4: LTS of an Unstable process](image)

From now on, the formal part of our data abstraction proposal will be presented.

### 4.3.2 Algorithm

In this section we present the algorithm for CSP\textsubscript{Z} data abstraction. It is described in a functional style and some functions are based on pattern matching. The main part of the algorithm concentrates on the function \texttt{findAbstraction}. The other functions are independent modules called at specific points by \texttt{findAbstraction}.

From Examples 4.3.2 and 4.3.3, we can observe the necessity to know about the current state, trace, and property of a given process. Recall from Section 3.4 that, due to the characteristics of
the Z part of a CSP\(_{Z}\) specification, we can represent all this information using channel names. It is worth noting that although traces are semantically built by events (channel names and values), we are not using a semantic model in this section. We are simply characterising symbolically the elements we need to describe the functionality of the algorithm. Therefore, via channel names we can build a (symbolic) trace (a sequence of channel names), the current state (a sequential schema composition where the schemas are built by prefixing the keyword \texttt{com} in front of the channel names) and a property (as a set of channels). For example, suppose a CSP\(_{Z}\) process with interface \{a, b, c\} (without considering values for simplification). Therefore, an example of a trace for this process could be \langle a, b, b, c \rangle, the current state corresponding to this trace would be 

\[
\text{Init} \#[\text{com}_a \# \text{com}_b \# \text{com}_b \# \text{com}_c]
\]

and, finally, a property could be characterised by the set \{a, b\}, which means

\[
\text{pre com}_a \land \text{pre com}_b \land \neg \text{pre com}_c
\]

that is, if a channel does not belong to the property set, then we take the negation of the precondition of its corresponding \texttt{com} schema.

We introduce a short name for power of channel names, since it appears in more than one place.

\[
PCh == \mathcal{P} \text{ChanName}
\]

We use it to define the following elements: an acceptance set, a label, and a property.

\[
\begin{align*}
\text{AcceptanceSet} & == PCh \\
\text{Label} & == \text{ChanName} \\
\text{Property} & == PCh
\end{align*}
\]

It is worth observing, again, that we are not using a semantic model. Our acceptance set is simply a set which contains channel names and the algorithm will manipulate this set in some way. Thus, the unique moment this acceptance set corresponds to the acceptance set of the semantic models (see Roscoe [10]) is when it is first created, as it will be explained in what follows.

To capture traces, we have to employ sequences. Since, at each element of a trace, the next alternatives (acceptance set) can change as well as the current property, we define a compound structure. This new structure can capture all this information at once and is defined as follows.

\[
\text{Path} == \text{seq} (\text{AcceptanceSet} \times \text{Label} \times \text{Property})
\]

Its components are:

- An acceptance set (\text{AcceptanceSet}), which is used to inform the algorithm what alternatives it can follow, at each state. In particular, when a new element of the sequence \text{Path} is created
by `findAbstraction`, this set is filled by all channels which have their associated preconditions valid. Every time a new trace must be considered, one element of this set is removed and used in the next element (Label) of the Path structure. This removal process is used to notify—precisely, when the set is empty—the algorithm to discard the current tuple and try a previous one; that is, all alternatives from this state on were already checked;

- A label (Label) to characterise the current transition. It is also used to build schema compositions according to the current trace up to this point. As explained previously, it changes according to the removed channel from the acceptance set;

- And, finally, a property (Property) which is static and is used to keep the property (as a set of channel names), at each state. When a new tuple is created by `findAbstraction`, the set `Property` is initialised with the same content as the set `AcceptanceSet`. However, as it was explained, once created at each element of the sequence, this set remains unchanged whereas the acceptance set varies in the course of the algorithm.

First, we present the function `findAbstraction`; the kernel of our guided data abstraction. Its first call is given as

\[ \text{findAbstraction}((\text{accS}_\text{init}, \tau, \text{accS}_\text{init})) \]

where `accS_init = validOpers ∅ Interface` and `accS_init ≠ ∅`. If `accS_init` is empty, then the previous call becomes simply `findAbstraction ∅`. That is, if `accS_init = ∅` then the Z part has a deadlock for the empty trace. This is equivalent to the proof of initialisation employed in the tradition of the Z language [47].

The trivial part of the function `findAbstraction` concerns its base case (∅). When it achieves this situation, no further progress (expansion) is possible and then it returns an identity map, according to the data domains of the Z part (we assume that D is the type related to the schema `State`). This identity will be overridden recursively by the abstractions previously found, yielding the desired outcome. It is worth noting that in the Examples 4.3.2, 4.3.3 and 4.3.4 the identity map was set first.

When the Path structure is not empty then its role is to expand the Z part, according to the alternatives found at `accS_curr`. The first condition checked is whether the current acceptance set is empty (`accS_curr = ∅`). If this set is empty then the current tuple of the Path structure must be discarded and a previous one must be taken, recursively (`findAbstraction t`). This condition is what guarantees the termination of this function in case no infinite trace with unstable predicates is possible.

If the current acceptance set is not empty, the function `findAbstraction` calculates the next transition, based on the event chosen to be engaged (`\( t_{next} \in accS_{curr} \)`). A next transition can assume two differing forms:

1. `t_{next}`: It is used when an abstraction can be performed or the next acceptance set is empty;
2. \texttt{t\_further}: It is used when no abstraction can be performed.

After checking that the next acceptance set, via the selected event, is not empty (\texttt{accS\_next} \neq \emptyset), we attempt to obtain a repeated property (\texttt{accS\_next} \in \text{ran}(\text{ran} \ t\_next)). If this is the case, then it calls the function \texttt{checkStability}, in order to abstract this trace. If an abstraction is possible, we calculate a new \textit{Path} structure—\texttt{t\_new}—by calling the function \texttt{newExploration}. Otherwise, the further expansion is chosen to be applied recursively to \texttt{findAbstraction}, as in the case the property has not been repeated.

\begin{verbatim}
findAbstraction :: Path \rightarrow (D \rightarrow D^A)
findAbstraction () = id
findAbstraction ((accS\_curr, l\_curr, prop\_curr)) \_ t=
    if accS\_curr = \emptyset then findAbstraction t
    else
        let
            l\_next \in accS\_curr
            t\_next = ((accS\_curr \setminus \{l\_next\}, l\_next, prop\_curr)) \_ t
            accS\_next = validOpers t\_next Interface
            t\_further = ((accS\_next, \tau, accS\_next)) \_ t\_next
        in
            if accS\_next \neq \emptyset then
                if accS\_next \in \text{ran}(\text{ran} \ t\_next) then
                    let
                        user = checkStability t\_next accS\_next
                        t\_new = newExploration t\_next accS\_next
                    in
                        case user of
                            optimal : findAbstraction t\_new \oplus optimalAbs t\_next accS\_next
                            none : findAbstraction t\_further
                            else findAbstraction t\_further
                else
                    findAbstraction t\_further
            else
                findAbstraction t\_next
\end{verbatim}

Recall from Section 2.3.2 that the acceptable events of the Z part are constrained by the preconditions of the respective \texttt{com\_} schemas. That is, if, for a given channel \texttt{c}, the precondition of \texttt{com\_c} is valid, then \texttt{c} is ready to engage into a possible communication with the CSP part. Otherwise, \texttt{c} is refused in the Z part and consequently in the CSP part too, because they cannot synchronise at this moment.

In order to capture this information, the function \texttt{validOpers} is defined. It takes a path and an acceptance set as input. Every call to \texttt{validOpers} uses the interface of the CSP\_Z specification as the acceptance set by default (see its use in the function \texttt{findAbstraction}). The result of this
function is the set of channels (subset of the interface) which has the precondition valid for the
current state (built using \texttt{buildComp}). The mechanics of this function is straightforward: it chooses
a channel from the acceptance set used as input and checks whether its \texttt{com}_e schema has a valid
precondition. If it has, then the corresponding channel is kept in the output. Otherwise, it is
simply discarded and the recursion proceeds. As shown in Example 4.3.5, the current state is built
by the call \texttt{buildComp t \emptyset}.

\[
\text{validOpers} :: \quad \text{Path} \to \text{AcceptanceSet} \to \text{AcceptanceSet}
\]
\[
\text{validOpers} t \emptyset = \emptyset
\]
\[
\text{validOpers} t \text{accS}_{\text{curr}} =
\]
\[
\text{let } e \in \text{accS}_{\text{curr}}
\]
\[
\quad \text{if } \left[\left[ \text{buildComp} t \emptyset \Rightarrow (\text{pre com}_e)\right]\right] = \left[\left[ \text{false}\right]\right] \quad \text{then }
\]
\[
\emptyset
\]
\[
\quad \text{else } \quad \{e\} \cup (\text{validOpers} t \text{accS}_{\text{curr}} \setminus \{e\})
\]

Recall from Section 2.3 (particularly at Table 2.1) that \([p]^P\) means the semantic interpretation of
the predicate \(p\). Therefore, in general, the clause \([\left[\text{buildComp} t \emptyset \Rightarrow (\text{pre com}_e)\right]]^P \neq [\left[ \text{false}\right]]^P\) needs some
theorem proving support. However, for some processes, where all variables have an associated
value, it is possible to obtain the same result via direct application of the current state to the given
preconditions.

Recall from Section 2.3.2 that, for a given trace, we have a corresponding Z schema composition
of the schemas involved in the trace. For example, suppose that the trace \(\langle a, b, c \rangle\) has occurred
then the state of the system is given by \(\text{Init} \circ \text{com}_a \circ \text{com}_b \circ \text{com}_c\). The function \texttt{buildComp},
presented in what follows, has this purpose. Given a trace embedded into a path structure and a
set of channels corresponding to a given property of the system, the function \texttt{buildComp} returns a
schema composition according to this trace. It is worth noting that it builds a schema composition
up to where the given property occurs. This is used since, in our data abstraction approach, we
have to check whether a given trace repeats periodically. In order to characterise what trace we
are dealing with, we need to inform the property which is repeating.

\[
\text{buildComp} :: \quad \text{Path} \to \text{Property} \to \text{SchemaExpr}
\]
\[
\text{buildComp} \emptyset \text{prop} = \text{Init}
\]
\[
\text{buildComp} \langle \text{accS}_{\text{curr}}, e, \text{prop}_{\text{curr}} \rangle \triangleright t \text{prop} =
\]
\[
\quad \text{if } \text{prop} = \text{prop}_{\text{curr}} \quad \text{then }
\]
\[
\text{com}_e
\]
\[
\quad \text{else } \quad (\text{buildComp} t \text{prop}) \circ \text{com}_e
\]
CHAPTER 4. CSP$_{Z}$ DATA ABSTRACTION

Example 4.3.5 (Using buildComp)

Consider the CSP$_{Z}$ specification of Example 4.3.2. When the CSP part performs the tick event, the state at this point is given by

\[
\text{buildComp} \langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \emptyset
\]

The structure \(\langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle\) means that:

- \(\emptyset\): there are no more alternatives to explore;
- \(\text{tick}\): the event currently performed;
- \(\{\text{tack}\}\): set characterising the property \((\neg \text{pre\_com\_tick} \land \text{pre\_com\_tack})\).

This call results in the schema composition \(\text{Init}_9\text{com\_tick}\), which is the current state after tick. Therefore, the call “\(\text{buildComp}\ t\ \emptyset\)” gives the current state for the trace kept in the Path structure \(t\); this empty set is used because we need the whole trace and not a (suffix) trace, corresponding to a given property. By another viewpoint, the empty set is used because it characterises a deadlock induced by the state, that is, the conjunction of the negation of the preconditions for all channels (or the bottom element—\(\bot\)—of the lattice of Figure 4.5; see later in this section). Since a deadlock does not repeat (at the first occurrence the process stops), this set is very convenient to be used as a special case.

The Examples 4.3.2, 4.3.3 and 4.3.4 were analysed this way.

Example 4.3.6 (Using validOpers)

Let us calculate the valid operations for the same situation as presented in Example 4.3.5. Since the interface of the process \(P_{\text{Clock}}\) is given by the set \(\{\text{tick}, \text{tack}\}\), we have

\[
\text{validOpers} \langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \{\text{tick}, \text{tack}\}
\]

which expands to (assuming that the first channel chosen was \(\text{tick}\))

\[
\text{if } [(\text{buildComp} \langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \emptyset) \Rightarrow (\text{pre\_com\_tick})^P = [false]^P \text{ then } \emptyset \text{ else } \{\text{tick}\}) \cup (\text{validOpers} \langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \{\text{tick}, \text{tack}\} \setminus \{\text{tick}\})
\equiv
\text{if } [(\text{Init}_9\text{com\_tick}) \Rightarrow (\text{pre\_com\_tick})^P = [false]^P \text{ then } \emptyset \text{ else } \{\text{tick}\})
\cup (\text{validOpers} \langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \{\text{tack}\})
\equiv
\emptyset \cup (\text{validOpers} \langle (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \{\text{tack}\})
\equiv (e \in \{\text{tack}\} = \text{tack})
\text{if } [(\text{buildComp} \langle ((\emptyset, \text{tick}), \{\text{tack}\}) \emptyset) \Rightarrow (\text{pre\_com\_tack})^P = [false]^P \text{ then } \emptyset}
\]

\]
4.3. CSP\textsubscript{Z} DATA ABSTRACTION

else \{tack\}) ∪ \{\text{validOpers } (\{∅, tick, \{tack\}\}) \} ∪ (tack) \setminus \{tack\})
\equiv (if \ [(\text{Init}_{\text{com\_tick}}) ⇒ (\text{pre\_com\_tack})]^P = [false]^P \text{ then } ∅ \text{ else } \{tack\})
\cup (\text{validOpers } (\{∅, tick, \{tack\}\}) \ Φ)
\equiv \{tack\} ∪ ∅

resulting in \{tack\}.

Recall from Section 4.3.1 that we define a property to be a conjunction of preconditions. Figure 4.5
presents a lattice (the standard ordering being implication) with all the properties that the \text{Z} part
of CSP\textsubscript{Z} specification with interface \{a_1, a_2, \ldots, a_n\} can exhibit. Furthermore, to simplify the
figure we have suppressed the negations. Thus, for all elements (conjunction) of the lattice, if
some precondition does not appear, then we mean that its negation takes place. In particular, the
bottom element (⊥) represents
\neg \text{pre\_com\_a}_1 ∧ \neg \text{pre\_com\_a}_2 ∧ \ldots ∧ \neg \text{pre\_com\_a}_n

![Figure 4.5: Lattice of the preconditions](image)

**Example 4.3.7 (The lattice of preconditions)** As a concrete example, let the interface of
a CSP\textsubscript{Z} specification be given by the set of channels \{a, b, c\}. If a schema \text{com\_a} has a valid
precondition at a given state but the other schemas, \( \text{com}_b \) and \( \text{com}_c \), do not have then the property which characterises precisely this state is given by

\[
\text{pre com}_a \land \neg \text{pre com}_b \land \neg \text{pre com}_c
\]

\( \Diamond \)

This kind of information is generated by the functions \texttt{validGuards} and \texttt{invalidGuards}.

Recall from Examples 4.3.2, 4.3.3 and 4.3.4 that we have to explore the Z part as much as possible. This is done by calculating the valid schemas (function \texttt{validOpers}) at each state and choosing one of the available channel names (events) to build a new path. However, we can achieve a point where either no further progress can be performed (a deadlock situation or a fully explored state), or an abstraction is achieved (we can stop from expanding). At this point, we have to look back and check whether there exists another path to follow.

The function \texttt{newExploration} is defined to perform the task explained above. It takes a path structure as input as well as a property. Associated to the \texttt{Path} structure only, we have two possibilities: Either it is empty and we return an empty sequence (there is no further paths to check), or it is not empty and the resulting \texttt{Path} structure depends on the given property. The first two situations deal with the current property of the \texttt{Path} structure being equal to the property given as input. That is, we have found the element of the \texttt{Path} structure which is keeping the information concerning the previous repeated property. In this case, there are two choices based on the current alternatives (acceptance set): either the current tuple must be discarded (\( \text{alts}_{\text{curr}} = {} \)), or this tuple still has a possible alternative to be considered (\( \text{alts}_{\text{curr}} \neq {} \)). In both situations, the function \texttt{findAbstraction} deals with the resulting \texttt{Path} structure. The last case is exactly when we have not found the repeated property then we discard the current tuple and deal with the remainder sequence recursively. Obviously, this function is total and always terminates. The termination guarantee is related to the \texttt{Path} structure becoming closer to ({}), at each recursive step.

\[
\begin{align*}
\text{newExploration} & : : \text{Path} \rightarrow \text{Property} \rightarrow \text{Path} \\
\text{newExploration} \langle \rangle \ prop & = \langle \rangle \\
\text{newExploration} \langle (\text{accS}_{\text{curr}}, e, \text{prop}_{\text{curr}}) \rangle \sim t \ prop & = \\
\quad \text{if } \ prop = \text{prop}_{\text{curr}} \land \text{accS}_{\text{curr}} = \emptyset \text{ then} \\
\quad \quad t \\
\quad \text{else} \\
\quad \quad \text{if } \ prop = \text{prop}_{\text{curr}} \land \text{accS}_{\text{curr}} \neq \emptyset \text{ then} \\
\quad \quad \quad \langle (\text{accS}_{\text{curr}}, e, \text{prop}_{\text{curr}}) \rangle \sim t \\
\quad \quad \text{else} \\
\quad \quad \quad \text{newExploration} \ t \ prop
\end{align*}
\]

Example 4.3.8 (Using \texttt{newExploration})
Suppose that we have achieved the following point
\[
\langle (\emptyset, \text{tack}, \{\text{tick}\}), (\emptyset, \text{tick}, \{\text{tack}\}) \rangle
\]
Thus, applying \texttt{newExploration} to this path with the property \{\text{tack}\} we have
\[
\texttt{newExploration} \langle (\emptyset, \text{tack}, \{\text{tick}\}), (\emptyset, \text{tick}, \{\text{tack}\}) \rangle \{\text{tack}\}
\]
which reduces to \(\langle \rangle\). An important observation—clearly stated in this example—is that the structure Path grows from right to left, contrary normally to what one expects about traces. The reason is that it is more conventional to take the head of a sequence than its last element (this question is specifically dealt at function \texttt{findAbstraction}).

The function \texttt{checkStability} is the most important among the auxiliar functions. Its purpose is to transfer the undecidability problem, related to the check for stability, to the user, via application of theorem proving. In this sense we are integrating model checking with theorem proving, and hence working in the direction of theory and tool integration as stated by Pnueli [68]. This function has a rather different output, corresponding to the kind of user response. It is worth observing that, when we mention a user, we are referring to some external interaction: a human being, a theorem prover, etc. For example, we can have a predicate which can be proven fully automatic by a theorem prover without a human being intervention. Therefore, our strategy can be fully automatic as long as the predicates considered belong to a class of a decidable logic [54, 60, 44, 31, 62, 12].

Therefore, before presenting the function \texttt{checkStability}, we introduce the possible user responses, by means of a free-type definition. It can be optimal and the abstraction is a total surjective function, or we must expand this path since it is not stable.

\[
\text{USER ::= } \text{optimal} \quad \text{– The abstraction is optimal}
| \quad \text{none} \quad \text{– We cannot abstract this trace}
\]

The function \texttt{checkStability} is based on the validity of the predicate \(\forall \text{State}; \text{State}' | \text{conj} \bowtie \text{comp} \Rightarrow \text{conj}'\), where \text{conj} captures the stable property (as a conjunction of valid and invalid preconditions) and \text{comp} stands for the stable sequential composition of \text{com} operations.

```plaintext
checkStability :: Path \rightarrow Property \rightarrow USER
checkStability \text{ t prop } =
let
    \text{conj} = \text{validGuards prop} \land \text{invalidGuards (Interface \setminus prop)}
    \text{comp} = \text{buildComp t prop}
    \text{stable} = \forall \text{State}; \text{State}' | \text{conj} \bowtie \text{comp} \Rightarrow \text{conj}'

    \text{if } [\text{stable}]^P = [\text{true}]^P \text{ then}
        \text{optimal}
    \text{else}
        \text{none}
```

Diamond symbol: \(\Diamond\)
The function \texttt{validGuards} generates only the conjunction of the valid preconditions. Its definition is very simple: either it considers a single channel and returns a single precondition expression built around this channel, or it takes a set with more than one channel element and returns a recursively built conjunction of preconditions.

\begin{verbatim}
validGuards :: AcceptanceSet \rightarrow ZPred
validGuards \emptyset = true
validGuards accS\textsubscript{curr} =
  let
    e \in accS\textsubscript{curr}
  \in
    pre com\textsubscript{e} \land (validGuards accS\textsubscript{curr} \setminus \{ e \})
\end{verbatim}

Complementarily, the function \texttt{invalidGuards} generates the conjunction of the invalid preconditions. Its definition is identical to the previous function except for adding a negation ($\neg$) in front of every generated precondition.

\begin{verbatim}
invalidGuards :: AcceptanceSet \rightarrow ZPred
invalidGuards \emptyset = true
invalidGuards accS\textsubscript{curr} =
  let
    e \in accS\textsubscript{curr}
  \in
    \neg pre com\textsubscript{e} \land (invalidGuards accS\textsubscript{curr} \setminus \{ e \})
\end{verbatim}

\textbf{Example 4.3.9} (Using \texttt{validGuards} and \texttt{invalidGuards})

Consider the same situation as presented in Examples 4.3.5 and 4.3.6. Since \textit{tack} is the event ready to occur, the property at this point is given by

\[ \texttt{validGuards} \{ \textit{tack} \} \land \texttt{invalidGuards} \{ \textit{tick}, \textit{tack} \} \setminus \{ \textit{tack} \} \]

which results in \((\texttt{pre com\_tack} \land \neg \texttt{pre com\_tick})\).

\textbf{Example 4.3.10} Using \texttt{checkStability}

Let us take the Example 4.3.3, when the property (characterised by the set) \{ \textit{a} \} repeats for the first time. Thus, we make the following call

\[ \texttt{checkStability} \langle (\emptyset, \textit{a}, \{ \textit{a} \}) \rangle \{ \textit{a} \} \]
Note that the tuple with the repeated property is not inserted into the Path structure yet. Now, we expand the previous call to

\[
\begin{align*}
\text{comp} & = \text{com}_a \\
\text{conj} & = \text{precom}_a \land \neg \text{precom}_b \\
\text{stable} & = \forall \text{State; State}' | \text{conj} \bullet \text{comp} \Rightarrow \text{conj}'
\end{align*}
\]

Checking \([\text{stable}]^P\), we get false. Therefore, we cannot abstract this domain and then we must expand it further.

When the function \text{checkStability} determines that the current trace can be abstracted, then we have to know how to produce the expected data abstraction. We have to find a map between a small (finite) set and an infinite one. For optimal abstractions, this map is a total surjective function. For instance, consider the Example 4.3.2. In this example, it was possible to abstract the trace \(\langle \text{tick}^0, \text{tack}^1, \text{tick}^2, \text{tack}^3, \ldots \rangle\) by \(\langle \text{tick}^0, \text{tack}^1 \rangle^n\) because we could find the total and surjective map

\[
h(n) = \begin{cases} 
0, & 0 \text{ E tick } n \\
1, & 1 \text{ E tack } n
\end{cases}
\]

where

\[
\begin{align*}
\text{E tick} & = \{ n : \mathbb{N} | n \text{ mod } 2 = 0 \bullet n \mapsto n + 2 \}^* \\
\text{E tack} & = \{ n : \mathbb{N} | n \text{ mod } 2 = 1 \bullet n \mapsto n + 2 \}^*
\end{align*}
\]

Before presenting the function which generate optimal data abstraction, we have to introduce three new auxiliary functions. The first is the function \text{buildTrace} which is identical to the function \text{buildComp}, except for its value of return.

The second is the function \text{cShiftT}. It makes a cyclic shift in a trace. By simplicity, \text{cShiftT} always shifts the elements of the trace to the left, that is, let \(s = \langle a, b, b, c \rangle\) be a trace and 2 is the number of shifts. Then, \text{cShiftT} \(s\ 2\) returns the trace \(\langle b, c, a, b \rangle\).
Finally, we have the third auxiliary function `buildSeqC`. It returns a sequential schema composition from the trace of input. That is, `buildSeqC ⟨a, b, c⟩` returns `com_a o com_b o com_c`.

\[
\text{buildSeqC} :: \text{seq ChanName} \rightarrow \text{SchemaExpr}
\]
\[
\text{buildSeqC} (e) = \text{com}_e
\]
\[
\text{buildSeqC} (e) \circ s = \text{com}_e \circ (\text{buildSeqC} s)
\]

The function `optimalAbs` generates an optimal data abstraction. It is simply a mapping between one fixed value, according to the equivalence relations of the periodic property, and their infinite equivalents. Observe that the least value refers to the first element of the stable trace. The remainder values are obtained via sequential composition, since the future sequential compositions repeat indefinitely as verified by `checkStability`. Note that, differently from the Examples 4.3.2 and 4.3.3, the function `optimalAbs` builds the abstraction function using the Z notation, that is, instead of values we are working more generally with bindings.

\[
\text{optimalAbs} :: \text{Path} \rightarrow \text{Property} \rightarrow (D \rightarrow D^A)
\]
\[
\text{optimalAbs} t \text{ prop } =
\]
\[
\text{let}
\]
\[
\text{stable } = \text{buildTrace} t \text{ prop }
\]
\[
1 \leq j \leq \#\text{stable}
\]
\[
\text{EqRel}(j) = \{[\text{buildSeqC}(\text{cShiftT stable} (j - 1))]^\varepsilon\}
\]
\[
\text{abs}_j \in \text{EqRel}(j)
\]
\[
\bullet
\]
\[
\bigcup_{i=1}^{\#\text{stable}} \{s : \text{EqRel}(i) \bullet s \mapsto \text{abs}_i\}
\]

It is worth noting that, unlike the Examples 4.3.2 and 4.3.3, in the definition of `optimalAbs` we are building the equivalence relations (`EqRel`) implicitly. The difference is that while in the examples we deal with values directly, in this definition we are working with bindings (association between names and values) provided by the Z language (see Spivey [47] or Woodcock and Davies [35] for further details). Another point is that, working this way, we do not have to worry about the precise names of the variables of the state space (`State`).

**Lemma 4.3.4 (Local Optimal Abstraction)**

Let \( P \) be a CSP\(_Z\) process, after translation, \( t \) be a Path structure, and \( \text{prop} \) be a property for the Z part of \( P \) which occurs in \( t \). Furthermore, assume that \( \forall \text{State}; \text{State}' | \text{conj} \bullet \text{comp} \Rightarrow \text{conj}' \) is valid, where

\[
\text{conj} = \text{validGuards} \text{ prop} \land \text{invalidGuards} (\text{Interface} \setminus \text{prop})
\]
\[
\text{comp} = \text{buildComp} t \text{ prop} = \text{com}_e \circ \cdots \circ \text{com}_a_k
\]
4.3. \textsc{ CSP}$_Z$ DATA ABSTRACTION

Then, the call $\text{optimalAbs} \ t \ \text{prop}$ yields optimal data abstraction in respect to the operation $\text{com}_c$ satisfying the precondition $\text{conj}$.

\textbf{Proof.} From the definition of the function $\text{optimalAbs}$, we have

\[
\text{optimalAbs} \ t \ \text{prop} = \bigcup_{i=1}^{\#\text{stable}} \{ s : \text{EqRel}(i) \cdot s \mapsto \text{abs}_i \}
\]

where

\[
\begin{align*}
\text{stable} &= \text{buildTrace} \ t \ \text{prop} \\
1 &\leq j \leq \#\text{stable} \\
\text{EqRel}(j) &= \{ [[\text{buildSeqC}(\text{cShiftT} \ \text{stable} \ (j-1))]^c \}
\end{align*}
\]

As long as $\text{prop}$ occurs in the structure $t$, $\text{buildTrace}$ can be applied successfully and always returns a non-empty sequence. Thus, $\#\text{stable} \geq 1$ and $\text{optimalAbs} \ t \ \text{prop} \neq \emptyset$. From

\[
\forall \text{State}; \ \text{State}' \mid \text{conj} \cdot \text{comp} \Rightarrow \text{conj}'
\]

we have that $\text{com}_c \circ \text{com}_a \circ \ldots \circ \text{com}_a \circ \text{com}_a \circ \ldots$, which repeats indefinitely.

In order to prove that $\text{optimalAbs} \ t \ \text{prop}$ yields optimal data abstraction we have to consider the Definition 4.3.3.

\[
\forall d : D; \ m : M \mid \text{conj} \cdot \{ \{ \text{com}_c \} \langle h(d), r_c(m) \rangle \rangle = (h \times r_c)(\{ \text{com}_c \} \langle d, m \rangle)
\]

The function $\text{optimalAbs}$ generates the building blocks for the construction of $h$ and $r_c$. Thus, if $s \mapsto \text{abs}_i \in \text{optimalAbs}$, then $h \ s = \text{abs}_i$ as well. So, by expanding both sides of the above equality, we get

\[
\begin{align*}
\{ \text{com}_c \} \langle h(d), r_c(m) \rangle &= (h \times r_c)(\{ \text{com}_c \} \langle d, m \rangle) \\
(h \times r_c)(\{ \text{com}_c \} \langle d d^A, mm^A \rangle) &= (h \times r_c)(\{ \text{com}_c \} \langle d, m \rangle)
\end{align*}
\]

\[
\{ d^A : D^A; \ m^A : M^A \mid (\exists d'd' : D; \ mm': M^A \bullet (dd', mm') \in (\{ \text{com}_c \} \langle d d^A, mm^A \rangle) \land \\
\quad d^A = h(d'd') \land m^A = r_c(mm') \bullet d^A \mapsto m^A \}
\]

\[
\{ d^A : D^A; \ m^A : M^A \mid (\exists d' : D; \ m' : M \bullet (d', m') \in (\{ \text{com}_c \} \langle d, m \rangle) \land \\
\quad d^A = h(d') \land m^A = r_c(m') \bullet d^A \mapsto m^A \}
\]

Since $dd^A \in \text{EqRel}(j) \land d \in \text{EqRel}(j)$, as well as $mm^A \in \text{EqRel}(j) \land m \in \text{EqRel}(j)$, for some $j$, we also have that $dd' \in \text{EqRel}(k) \land d' \in \text{EqRel}(k)$, as well as $mm' \in \text{EqRel}(k) \land m' \in \text{EqRel}(k)$, for some $k$, due to the definition of $\text{optimalAbs}$. Therefore, $h(d'd') = h(d') = d^* \ (d^* = d^A$ or $d^* = d^A)$, and $r_c(mm') = r_c(m') = m^*$ ($m^* = m^A$ or $m^* = m^A$).

\[\Box\]

Now, we present a proof that our mechanised data abstraction provides optimal data abstraction if the first call to the function $\text{findAbstraction}$ actually stops, or terminates successfully.

First, we present an intermediate result to ease the presentation of the Theorem 4.3.3.
Lemma 4.3.5 (Overhidden Preserved Abstraction)

Let $P$ be a CSP $\mathbb{Z}$ process, after translation, $t$ and $t'$ be Path structures, and $\text{prop}$ be a property for the $\mathbb{Z}$ part of $P$. If the overhidden

\[ \text{findAbstraction} \ t \oplus \text{optimalAbs} \ t' \ \text{prop} \]

can be applied and terminates successfully then it yields optimal data abstraction.

**Proof.** In order for this overhidden to be applicable, we must have that $\text{checkStability} \ t' \ \text{prop}$ is also applicable (that is, $\forall \text{State}; \text{State'} | \ \text{conj} \bullet \text{comp} \Rightarrow \text{conj}'$ is valid), resulting the value optimal. And from Lemma 4.3.4, the function $\text{optimalAbs}$ yields optimal data abstraction. To conclude the proof, it rests only to check whether the overhide operator is conservative in respect to optimal abstraction.

Thus, let us consider an induction over the number of overhidings performed by $\text{findAbstraction}$

- **Base case** ($i_D$): this case always occur because we are assuming that $\text{findAbstraction} \ t$ terminates successfully. The above overhiding becomes

  \[ \lambda \ d : D \bullet d \oplus \text{optimalAbs} \ t' \ \text{prop} \]

  which is optimal by Lemma 4.3.4 and because the identity is (the worst) optimal. The identity maps exactly the cases where $\forall \text{State}; \text{State'} | \ \text{conj} \bullet \text{comp} \Rightarrow \text{conj}'$ is not valid.

- **Induction case** ($i_D \oplus \text{optimalAbs} \ t^1 \ \text{prop} \oplus \ldots \oplus \text{optimalAbs} \ t^{(n-1)} \ \text{prop}$): in this situation, the overhiding becomes

  \[ \left( (i_D \oplus \text{optimalAbs} \ t^1 \ \text{prop}^{(1)} \oplus \ldots \oplus \text{optimalAbs} \ t^{(n-1)} \ \text{prop}^{(n-1)}) \oplus \text{optimalAbs} \ t^n \ \text{prop}^{n} \right) \oplus \text{optimalAbs} \ t' \ \text{prop} \]

  This overhiding needs the analysis of two distinct situations:

  - The properties $\text{prop}$ or $\text{prop}^n$ are equal to one of the previous properties $\text{prop}^1, \ldots, \text{prop}^{(n-1)}$: we need only to show that, for example, the property $\text{prop}$ has already been considered previously. The analysis concerning $\text{prop}^n$ can be classified as the current analysis of $\text{prop}$ or as the next case, where disjunctness is applied. Thus, without loss of generality, assume that $\text{prop} = \text{prop}^1$, and the others are disjoints

    \[ (i_D \oplus \text{optimalAbs} \ t^1 \ \text{prop}^{(1)} \oplus \ldots \oplus \text{optimalAbs} \ t^n \ \text{prop}^{n}) \oplus \text{optimalAbs} \ t' \ \text{prop} \]

    \[ \equiv (i_D \oplus (\text{optimalAbs} \ t^1 \ \text{prop}^{(1)} \cup \ldots \cup \text{optimalAbs} \ t^n \ \text{prop}^{n})) \oplus \text{optimalAbs} \ t' \ \text{prop} \]

    \[ \equiv (i_D \oplus (\text{optimalAbs} \ t^n \ \text{prop}^{n} \cup \ldots \cup \text{optimalAbs} \ t^1 \ \text{prop}^{1})) \oplus \text{optimalAbs} \ t' \ \text{prop} \]

    \[ \equiv (i_D \oplus (\text{optimalAbs} \ t^{n} \ \text{prop}^{n} \cup \ldots \cup \text{optimalAbs} \ t^1 \ \text{prop}^{1})) \cup \text{optimalAbs} \ t^1 \ \text{prop}^{1} \oplus \text{optimalAbs} \ t' \ \text{prop} \]

  Thus, we only need to analyse this pairwise overhiding. By definition, we have that the predicates

  $\forall \text{State}; \text{State'} | \ \text{conj} \bullet \text{comp} \Rightarrow \text{conj}'$

  $\forall \text{State}; \text{State'} | \ \text{conj} \bullet \text{comp}^{\text{new}} \Rightarrow \text{conj}'$
are valid. The difference is only related to the syntactic ordering of the sequential composition of schemas \( \text{comp} \) (EqRel(1)) and \( \text{comp}^{\text{new}} \) (EqRel^{\text{new}}(1)), which is semantically irrelevant. That is, for the above predicates to be valid for different sequential compositions then, for instance, \( \text{com}_{a_i} \circ \text{com}_{a_j} \Leftrightarrow \text{com}_{a_j} \circ \text{com}_{a_i} \), for arbitrary \( a_i \) and \( a_j \). Therefore, the overhiding
\[
\bigcup_{i=1}^{\#\text{stable}} \{ s : \text{EqRel}(i) \bullet s \mapsto \text{abs}_i \} \oplus \bigcup_{i=1}^{\#\text{stable}^{\text{new}}} \{ s : \text{EqRel}^{\text{new}}(i) \bullet s \mapsto \text{abs}_i^{\text{new}} \}
\]
becomes
\[
\bigcup_{i=1}^{\#\text{stable}} \{ s : \text{EqRel}(i) \bullet s \mapsto \text{abs}_i \} \oplus \bigcup_{i=1}^{\#\text{stable}^{\text{new}}} \{ s : \text{EqRel}^{\text{new}}(i) \bullet s \mapsto \text{abs}_i^{\text{new}} \}
\]
or (by the definition of overhiden)
\[
\bigcup_{i=1}^{\#\text{stable}} \{ s : \text{EqRel}(i) \bullet s \mapsto \text{abs}_i^{\text{new}} \}
\]
which is optimal by Lemma 4.3.4.

The properties \( \text{prop} \) or \( \text{prop}^{\text{n}} \) are not equal to any of the previous properties: this case is more easier because the abstractions are disjoint and then we have
\[
((t_1 \oplus \text{optimalAbs } t_1 \text{ prop}^{\text{1}} \oplus \ldots \oplus \text{optimalAbs } t_1 \text{ prop}^{(n-1)} \oplus \text{optimalAbs } t_1 \text{ prop}^{\text{n}}) \oplus \text{optimalAbs } t_1 \text{ prop}^{\text{new}}) \quad \text{[By termination]}
\]
\[
\equiv ((t_1 \oplus \text{optimalAbs } t_1 \text{ prop}^{\text{1}} \oplus \ldots \oplus \text{optimalAbs } t_1 \text{ prop}^{(n-1)} \cup \text{optimalAbs } t_1 \text{ prop}^{\text{n}}) \cup \text{optimalAbs } t_1 \text{ prop}^{\text{new}}) \quad \text{[By disjointness]}
\]
\[
\equiv ((t_1 \oplus \text{optimalAbs } t_1 \text{ prop}^{\text{1}} \oplus \ldots \oplus \text{optimalAbs } t_1 \text{ prop}^{(n-1)} \cup \text{optimalAbs } t_1 \text{ prop}) \cup \text{optimalAbs } t_1 \text{ prop}^{\text{n}}) \quad \text{[By } \cup\text{-commutativity]}
\]
which is optimal by induction, by Lemma 4.3.4, and by disjointness.

\[\Box\]

Now comes the main result of this section.

**Theorem 4.3.3 (Optimal Abstraction)**

Let \( P \) be a CSP\(_Z\) process. If the function \( \text{findAbstraction} \), applied to the Z part of \( P \), terminates then it yields optimal data abstraction.

**Proof.** The proof follows by induction on the size of the Path structure.

- Base case (\( \langle \rangle \)): trivial since \( \text{findAbstraction} \) returns the identity map, which is (the worst) optimal data abstraction in the sense of the abstract interpretation terminology.
CHAPTER 4. CSP\textsubscript{Z} DATA ABSTRACTION

• Induction case \((\langle \text{accS}_{\text{curr}}, \text{lt}_{\text{curr}}, \text{prop}_{\text{curr}} \rangle \triangleq t)\): the proof now follows from a case analysis based on the conditional branches of the function \text{findAbstraction}, considering the abbreviations of \text{accS}_{\text{curr}}, \text{lt}_{\text{next}}, \text{t}_{\text{next}}, \text{t}_{\text{further}}, \text{accS}_{\text{next}}, \text{t}_{\text{new}} given above in the algorithm

1. \text{accS}_{\text{curr}} = \emptyset: this is the most simple situation because it calls recursively \text{findAbstraction} \text{t} which returns optimal data abstraction via the induction hypothesis.

2. \text{accS}_{\text{curr}} \neq \emptyset \land \text{accS}_{\text{next}} \neq \emptyset: its truth depends only on the analysis of case 4 (below).

3. \text{accS}_{\text{curr}} \neq \emptyset \land \text{accS}_{\text{next}} = \emptyset: this branch implies that the call \text{findAbstraction} \text{t}_{\text{next}} occurs. From the Algorithm, we know that the previous call can be rewritten as

\[
\text{findAbstraction} \langle \langle \text{accS}_{\text{curr}} \setminus \{\text{lt}_{\text{next}}\}, \text{lt}_{\text{next}}, \text{prop}_{\text{curr}} \rangle \rangle \triangleq t
\]

which admits two situations

(a) The call \text{findAbstraction} \text{t}_{\text{next}} occurs repeatedly, because \text{accS}_{\text{next}} = \emptyset remains true at each call: after \(m\) calls, the constraint \text{accS}_{\text{curr}} = \emptyset becomes true, due to the successive subtraction \text{accS}_{\text{curr}} \setminus \{\text{lt}_{\text{next}}\}. When this happens, we are reported to case 1.

(b) The set \text{accS}_{\text{next}} can become non-empty and then the proof follows from the analysis of cases 4. and 5.

4. \text{accS}_{\text{curr}} \neq \emptyset \land \text{accS}_{\text{next}} \neq \emptyset \land \text{accS}_{\text{next}} \in \text{ran(} \text{ran t}_{\text{next}} \text{)}: in this situation we have the overhidden

\[
\text{findAbstraction} \text{t}_{\text{new}} \oplus \text{optimalAbs} \text{t}_{\text{next}} \text{accS}_{\text{next}}
\]

By the induction hypothesis we have that \text{findAbstraction} \text{t}_{\text{new}} is optimal abstraction. Therefore, by Lemma 4.3.5 the previous overhidden yields optimal data abstraction.

5. \text{accS}_{\text{curr}} \neq \emptyset \land \text{accS}_{\text{next}} \neq \emptyset \land \text{accS}_{\text{next}} \notin \text{ran(} \text{ran t}_{\text{next}} \text{)}: this branch yields the recursive call \text{findAbstraction} \text{t}_{\text{further}}. There are two situations to consider here:

(a) At each future call, the constraint \text{accS}_{\text{curr}} \neq \emptyset \land \text{accS}_{\text{next}} \neq \emptyset \land \text{accS}_{\text{next}} \notin \text{ran(} \text{ran t}_{\text{next}} \text{)} is true: this cannot happen because we are assuming that the algorithm terminates successfully.

(b) As long as case 5.(b) cannot happen indefinitely, the proof follows by the analysis of cases 2., 3., or 4.

\diamond

Currently, we have a Haskell prototype\footnote{It is located at http://www.cin.ufpe.br/~acm/stable.hs} for this algorithm. It was integrated to the theorem prover Z-Eves \cite{46} using a file based strategy. The function \text{checkStability} generates the
predicates—to be proven by Z-Eves—and the user controls every step guiding the approach. Indeed, Example 4.3.4 was built using the prototype. The prototype is very specific in the sense that the Z part—for each new process—must be introduced via a functional characterisation of the \texttt{com} operations \cite{4, 6}. A more robust implementation of the algorithm is a topic for further research.

## 4.4 Conclusions

Our original goal was to model check CSP\textsubscript{Z} processes \cite{4, 6}. This effort has presented another difficulty: how to model check infinite state systems since they emerge naturally in CSP\textsubscript{Z} specifications. The works of Lazić \cite{58} and Wehrheim \cite{26} have been adopted as a basis for this work due to their complementary contributions to our aim. However, both works had some kind of limitation: Lazić’s work \cite{58} allows only to check data independent refinements, whereas Wehrheim’s work \cite{26} left to the user the task of proposing abstract domains and operations (the most difficulty part of a data abstraction). In this sense, we believe that the results reported here contribute in the following way to the following works: to our earlier work \cite{4, 6}, by enabling model checking of infinite CSP\textsubscript{Z} processes; to Lazić’s work \cite{58}, by capturing data dependencies in the Z part of a CSP\textsubscript{Z} specification; and to Wehrheim’s work \cite{26}, by fixing a little problem in the formulation of some results, by characterising more precisely how the abstractions results can be used to check properties of infinite systems, and by mechanising her non-guided data abstraction strategy.

Another result was to find a flaw in some results of Wehrheim’s work \cite{26} based on Lazić’s work \cite{58}. It is related to the CSP part of a CSP\textsubscript{Z} process; Wehrheim’s work \cite{26} does not discriminate what CSP elements the CSP part can use. Thus, if equality tests are allowed, the CSP part can have stronger dependencies than those of the Z part. This was fixed on Lemmas 4.3.2 and 4.3.3 by considering the CSP part to be trivially data independent. This guarantees the Z part to be the most dependent one.

In the direction of mechanisation, our approach is similar to the works of Stahl \cite{40} and Shankar \cite{63}. The main difference is that, while they use boolean abstraction (replace predicates and expressions with boolean variables), we use subtype abstraction (replace types for subtypes and abstract operations for operations closed under the subtypes). This choice makes our work free from user intervention and can yield safe and optimal abstractions, but it offers some limitations if the state variables are strongly coupled. On the other hand, Stahl and Shankar can handle weakly coupled variables, due to the boolean abstraction strategy, however they need an initial user support and focus on safe abstractions. The normal form of a CSP\textsubscript{Z} specification \cite{4, 6}, which was obtained naturally from the definition of the language, has played an important role in this part of our work, by allowing the Z part to be more easily analysed. Both their approach and ours need theorem proving support and follow the research direction of tool and theory integration \cite{68}.

For future research we intend to investigate compositional results for optimal abstractions, analyse further properties beyond those allowed in Lazić’s work \cite{58}, classify processes according
to the predicates they use, and incorporate the abstraction algorithm into our mechanised model checking strategy in order to handle infinite CSP\textsubscript{Z} processes with minimum user assistance.
Chapter 5

Local Analysis

In the previous chapter we have discussed how to overcome state explosion by means of representing an infinite labelled transition system by a finite equivalent one, as long as the infinite LTS is sufficiently stable/regular. However, even after this data reduction, the resulting LTS can still be too large to be handled as a whole by FDR. In order to further reduce the problem size, we present a modular strategy, specifically designed for deadlock analysis, completely orthogonal to the data abstraction techniques presented in the previous chapter.

An interesting alternative to the material presented in this chapter is considering that the system being analysed was built using a strategy to avoid deadlock by construction. The works of Martin [32], and Cunha and Justo [53] have shown that it is possible to avoid deadlock by construction via composition of deadlock-free communication patterns. However, our work is in the direction of minimising user effort when specifying a CSP$_\mathsf{Z}$ system. Therefore, we are not considering a system built via a deadlock-free construction strategy. Otherwise, we don’t need to apply the material of this chapter.

We begin this chapter by briefly presenting the deadlock analysis technique developed by Brookes and Roscoe[61] in Section 5.1. In Section 5.2 we show how this technique can be extended for CSP$_\mathsf{Z}$ as well as how FDR can be used to do this kind of local deadlock analysis.

5.1 Local Deadlock Analysis

Brookes and Roscoe’s approach considers only CSP processes that do not diverge, are triple-disjoint (there is no channel shared by more than two processes) and have a static topology (recursion might not introduce new behaviour). These requirements allow a simpler mathematical treatment, while it is not too severe in practice. For instance, practical applications are expected to be divergence-free.

Assuming divergence-freedom, we can understand precisely CSP processes with only the Failures model, represented as $\mathcal{F}[P]$ where $P$ is a process. The Failures model represents a process
$P$ as a set of pairs $(s, X)$, where $s$ is a trace of $P$ and $X$ is a refusal set of $P$, such that after $P$ performs $s$ it may refuse to engage in any event of $X$.

A very simple still formal definition of deadlock for a process is given below by considering the refusal set as the whole alphabet (all possible events) of a process.

**Definition 5.1.1** The process $P$ can **deadlock** after the trace $s$ iff $(s, \alpha P) \in F[P]$. ♦

The next definition characterises deadlock-freedom by avoiding the worst possible case, i.e, a process is deadlock-free only when there is no trace which has as refusal set the whole alphabet of the process.

**Definition 5.1.2** The process $P$ is **deadlock-free** iff $\forall s \in (\alpha P)^* \cdot (s, \alpha P) \not\in F[P]$. ♦

We can see a complex concurrent system as a composition of more simple parallel processes. This specific organisation of processes receives a particular name in the next definition.

**Definition 5.1.3** A network is a parallel combination of processes. Let $V$ be a network composed of the processes $P_1, \ldots, P_n$ then $V = \langle P_1, \ldots, P_n \rangle$. ♦

**Example 5.1.1** Let Client and Server be two processes with the usual functionality as one expects. Since the client-server system is generally defined as a parallel composition between them, this network is denoted as $V = \langle \text{Client}, \text{Server} \rangle$. ♦

Basically, the kind of networks which we are interested in the rest of this chapter are those which satisfy the definition below. In the sequel, let $V$ be a network such that $V = \langle P_1, \ldots, P_n \rangle$, and $i, j, k$ range from 1 to $n$.

**Definition 5.1.4** A network is triple-disjoint iff $\alpha P_i \cap \alpha P_j \cap \alpha P_k = \emptyset$, $i \neq j \neq k$. ♦

**Example 5.1.2** Let $P = a \rightarrow b \rightarrow P$, $Q = b \rightarrow c \rightarrow Q$ and $R = c \rightarrow d \rightarrow a \rightarrow R$ be CSP processes. The network $V = \langle P, Q, R \rangle$ is triple-disjoint because $\alpha P \cap \alpha Q \cap \alpha R = \{a, b\} \cap \{b, c\} \cap \{c, d\} = \emptyset$. ♦

A network can be easily transformed into a graph, as established by the next definition. This is very useful as a hint to find out possible deadlock regions.

**Definition 5.1.5** A graphical view of a network is a communication graph where the processes are the nodes and an arc exists between two nodes iff $\alpha P_i \cap \alpha P_j \neq \emptyset$, $i \neq j$. ♦
Example 5.1.3 Let $P$, $Q$ and $R$ be CSP processes as in Example 5.1.2. Then, its graphical representation according to Definition 5.1.5 is given in Figure 5.1.

CSP uses the concept of alphabet for exhibiting the possible communications of its processes, we introduce now the concept of vocabulary which is simply a specific use of the alphabet of the processes in a pairwise manner.

![Figure 5.1: Graphical representation of a network of three processes](image)

**Definition 5.1.6** The vocabulary $\Lambda$ of a network $V$ is the set $\bigcup\{\alpha P_i \cap \alpha P_j \mid 1 \leq i < j \leq n\}$.  

**Example 5.1.4** Let $P$, $Q$ and $R$ be CSP processes as in Example 5.1.2. Then, the vocabulary of the network $V = \langle P, Q, R \rangle$ is 

$$\bigcup\{\alpha P \cap \alpha Q, \alpha P \cap \alpha R, \alpha Q \cap \alpha R\} = \bigcup\{\{b\}, \{a\}, \{c\}\} = \{a, b, c\}$$

Note that this set is different from the conventional set $\Sigma = \{a, b, c, d\}$.  

**Definition 5.1.7** A state $\sigma$ of a network $V$ is a trace $s$ of $V$ and an indexed tuple $(X_1, \ldots, X_n)$ of refusal sets $X_i$ such that for each $i$, $(s \mid \alpha P_i, X_i) \in \mathcal{F}[P_i]$.  

A process has many states depending on the traces performed. In the occurrence of an event, the process changes its state. The above definition express this for a network, i.e., for a set of processes. Note that the filter operator $(\mid)$ is used to give the state of the network in terms of each constituent process.

**Example 5.1.5** Let $V = \langle P, Q, R \rangle$ be a network defined by the processes of Example 5.1.2. Suppose $V$ performs the trace $s = \langle a, b \rangle$ then its current state is...
\( \sigma = (s, (X_P, X_Q, X_R)) \), where \( X_P = \{ b \}, X_Q = \{ b \} \) and \( X_R = \{ a, d \} \)

and

\[
\begin{align*}
(s \upharpoonright \alpha P, X_P) &= (\langle a, b \rangle, \{ b \}) \in \mathcal{F}[P] \\
(s \upharpoonright \alpha Q, X_Q) &= (\langle b \rangle, \{ b \}) \in \mathcal{F}[Q] \\
(s \upharpoonright \alpha R, X_R) &= (\langle \rangle, \{ a, d \}) \in \mathcal{F}[R]
\end{align*}
\]

\( \blacksquare \)

A simple extension of the Definition 5.1.1 is given below in terms of the preceding definition.

**Definition 5.1.8** A network \( V \) can deadlock after \( s \) iff there is a state \( (s, (X_1, \ldots, X_n)) \) of \( V \) for which \( \bigcup_{i=1}^{n} \alpha_P = \bigcup_{i=1}^{n} X_i \).

\( \blacksquare \)

**Example 5.1.6** Applying the previous definition to Example 5.1.5 we get

\[
\begin{align*}
\bigcup \{ \alpha P, \alpha Q, \alpha R \} &= \bigcup \{ X_P, X_Q, X_R \} \\
&\equiv \bigcup \{ \{a, b\}, \{b, c\}, \{c, d\} \} = \bigcup \{ \{b\}, \{a, d\} \} \\
&\equiv \{a, b, c, d\} \neq \{a, b, d\}
\end{align*}
\]

which clearly shows that this network can progress.

\( \blacksquare \)

Now we are ready to introduce the main building blocks of the modular deadlock analysis of [61]. They deal with the possibility of two processes be blocked by themselves.

**Definition 5.1.9** Let \( \sigma = (s, X) \) be a state and \( \Lambda \) be the vocabulary of the network \( V \). A pair of indices \( (i, j) \) (with \( i \neq j \)) is:

- A request if \( (\alpha_P_i \setminus X_i) \cap \alpha_P_j \neq \varnothing \) (\( P_i \xrightarrow{\sigma} P_j \) or \( P_i \xrightarrow{\sigma \Lambda} P_j \));

- A strong request if \( \varnothing \neq (\alpha_P_i \setminus X_i) \subseteq \alpha P_j \) (\( P_i \xrightarrow{\sigma} P_j \) or \( P_i \xrightarrow{\sigma \Lambda} P_j \));

- An ungranted request if it is a request and \( \alpha_P_i \cap \alpha P_j \subseteq X_i \cup X_j \) (\( P_i \xrightarrow{\sigma} P_j \) or \( P_i \xrightarrow{\sigma \Lambda} P_j \)).

- An ungranted strong request if it is a strong request and \( \alpha P_i \cap \alpha P_j \subseteq X_i \cup X_j \) (\( P_i \xrightarrow{\sigma} P_j \) or \( P_i \xrightarrow{\sigma \Lambda} P_j \)).

\( \blacksquare \)
Example 5.1.7 Let $V$ be a network and $\sigma$ a state of $V$ as given by Example 5.1.2. From that state we have the following requests:

- $P \not\rightarrow Q$ because $(\alpha P \setminus X_P) \cap \alpha Q \equiv \{a, b\} \setminus \{b\} \cap \{b, c\} \equiv \emptyset = \emptyset$
- $Q \not\rightarrow R$ because $(\alpha Q \setminus X_Q) \cap \alpha R = \{b, c\} \setminus \{b\} \cap \{a, c, d\} = \{c\} \neq \emptyset$
  - and $\emptyset \neq (\alpha Q \setminus X_Q) \subseteq \alpha R \equiv \emptyset \neq (\{b, c\} \setminus \{b\}) \subseteq \{a, c, d\} \equiv \emptyset \neq \{c\} \subseteq \{a, c, d\}$
  - and $\alpha Q \cap \alpha R \subseteq X_Q \cup X_R \equiv \{b, c\} \cap \{a, c, d\} \subseteq \{b\} \cup \{a, d\} \equiv \{c\} \not\subseteq \{a, b, d\}$
- $R \not\rightarrow P$ because $(\alpha R \setminus X_R) \cap \alpha P \equiv \{a, c, d\} \setminus \{a, d\} \cap \{a, b\} \equiv \emptyset = \emptyset$

An extension of Definition 5.1.2 is given a name in the following definition.

Definition 5.1.10 A network $V$ is busy if all of its node processes are deadlock-free.

Example 5.1.8 Obviously the network of Example 5.1.2 is busy.

The building blocks of modular deadlock analysis is given by conflicts and strong conflicts. The first represents the possibility of only one of two processes making progress. The latter is used to capture a situation in which neither process can communicate anymore; not even with their neighbors.

Definition 5.1.11 A state $\sigma$ of the pair $(P, Q)$ is $\Lambda$-conflict if $P \not\rightarrow Q$ and $Q \not\rightarrow P$, and strong $\Lambda$-conflict if $P \not\rightarrow Q$ or $Q \not\rightarrow P$ with respect to the set of requestable events $\Lambda$.

Example 5.1.9 In order to present a network with a conflict we take the network of Example 5.1.2, preserving $P$ and modifying $Q$ and $R$ as follows. Let $Q = b \to d \to c \to Q$ and $R = c \to d \to a \to R$ be processes forming the network $V = (P, Q, R)$. Suppose again that $V$ achieves state $\sigma$ from Example 5.1.5. From this state we can deduce the pairwise state $\sigma'$ between $Q$ and $R$. Calculating the conditions of Definition 5.1.9 for the subnetwork $V' = (Q, R)$:

- $Q \not\rightarrow R$ because $\alpha Q \setminus X_Q \cap \alpha R = \{b, c, d\} \setminus \{b, d\} \cap \{a, c, d\} \equiv \{c\} \neq \emptyset$
  - and $\emptyset \neq (\alpha Q \setminus X_Q) \subseteq \alpha R \equiv \emptyset \neq (\{b, c, d\} \setminus \{b, d\}) \subseteq \{a, c, d\} \equiv \emptyset \neq \{c\} \subseteq \{a, c, d\}$
  - and $\alpha Q \cap \alpha R \subseteq X_Q \cup X_R \equiv \{b, c, d\} \cap \{a, c, d\} \subseteq \{c\} \not\subseteq \{c\}$
- $R \not\rightarrow Q$ because $\alpha R \setminus X_R \cap \alpha Q = \{a, c, d\} \setminus \{a, d\} \cap \{b, c, d\} \equiv \{c\} \neq \emptyset$
  - and $\emptyset \neq (\alpha R \setminus X_R) \subseteq \alpha Q \equiv \emptyset \neq (\{a, c, d\} \setminus \{a, d\}) \subseteq \{b, c, d\} \equiv \emptyset \neq \{b, c, d\}$
  - and $\{c, d\} \subseteq \{c, d\}$ as above
we get a strong conflict between Q and R as expected. Further, we have a strong conflict between R and P and between P and Q in that state. This is a consequence of the local deadlock between Q and R is founded on a strong conflict, preventing Q or R to progress on alternative traces.

The following definition is an extension of the preceding one, and Definition 5.1.13 is a generalisation of these to network of processes.

**Definition 5.1.12** A pair \( \langle P, Q \rangle \) is free of \( \Gamma \)-conflict if none of its states is a \( \Gamma \)-conflict.

**Example 5.1.10** The pairs \( \langle P, Q \rangle \), \( \langle Q, R \rangle \) and \( \langle R, P \rangle \) from Example 5.1.2 are conflict-free as expected.

**Definition 5.1.13** A network \( V \) with vocabulary \( \Lambda \) is (strong) conflict-free iff for all \( i \neq j \) the pair \( \langle P_i, P_j \rangle \) is free of strong \( \Lambda \)-conflict.

**Example 5.1.11** From Example 5.1.10 we know that the network of Example 5.1.2 is conflict-free. However, the modified network of Example 5.1.9 is not since we have a strong conflict between Q and R.

Deadlock analysis is not easy even for small networks, hence it is convenient to introduce some way to divide the whole problem into small ones and conclude deadlock-freedom from these parts.

**Definition 5.1.14** The edges (nodes) \( V_1, \ldots, V_k \) are the disconnecting edges of the network \( V \) iff they are nodes of the communication graph of \( V \) whose removal would increase the number of connected components (partitions).

**Example 5.1.12** Let \( V = \langle V_1, \ldots, V_6 \rangle \) be a network. Its communication graph is presented in Figure 5.2, where either \( P_3 \) or \( P_5 \) is the possible disconnecting edge. Note that \( P_3 \) and \( P_5 \) can be removed simultaneously because only one of them will increase the number of remaining connected components.

**Definition 5.1.15** The essential components of \( V \) are the connected components of the graph that remains after all disconnecting edges were removed.

**Example 5.1.13** From Figure 5.2 we can identify the essential components which are depicted in Figure 5.4. Once again the essential components depend upon the disconnecting edge selected for removal. In the present situation, if we remove \( P_3 \) the essential components will be \( V_1 = \langle P_1, P_2 \rangle \) and \( V_2 = \langle P_4, P_5, P_6 \rangle \). Otherwise, if \( P_5 \) is removed then \( V_1 = \langle P_1, P_2, P_3 \rangle \) and \( V_2 = \langle P_4, P_6 \rangle \). Figure 5.3 shows the processes involved in a conflict-freedom proof obligation. In order to obtain deadlock information in a compositional fashion, it is necessary guarantee the conflict-freedom between the components linked via the disconnecting edges.
The essence of the local deadlock analysis is given by the following theorem. It states deadlock-freedom of a network in terms of the deadlock-freedom of its essential components.

**Theorem 5.1.1** Suppose $V$ is a busy network with essential components $V_1, \ldots, V_k$ where the pair of processes joined by each disconnecting edge are conflict-free with respect to the vocabulary $\Lambda$. Then if each $V_i$ is deadlock-free, so is $V$.

The above theorem establishes a connection between deadlock-freedom and pairwise conflict-freedom of the essential components of a network. The conflict-freedom constraint is necessary because if one essential component blocks then it can infect others if the edge linking two essential components has conflict. This is a very important result because for large networks one can attempt to arrange them such that they can be partitioned into simpler ones. Then Theorem 5.1.1 tells us that it suffices to check for deadlock-freedom of the essential components.

This deadlock analysis strategy is compositional in the sense that we verify smaller processes and use Theorem 5.1.1 to conclude the deadlock-freedom of their parallel compositions. In the next section we show how FDR can be used both to verify conflict-freedom of processes linked by disconnecting edges, and deadlock-freedom of essential components.
5.2 Extending the Modular Approach towards CSP\(_Z\)

According to the requirements of the formal strategy for deadlock analysis presented above, a network can only be investigated in a modular way if it is divergence-free, triple-disjoint, uses an associative parallel operator such as \(\parallel\) (defined in [14]) and has a static topology.

If one can prove that a network has all these properties then the results of the preceding section can be used. In what follows we show the conformity obligations for such results to generalise for CSP\(_Z\) specifications.

1. **Triple-disjointness for CSP\(_Z\)**: To guarantee triple-disjointness of a CSP\(_Z\) network we consider each CSP\(_Z\) process (component of this network) as a unit. This is necessary because when two CSP\(_Z\) specifications share a same event \(c\), we actually have four processes sharing this event, since the CSP and the Z parts of each process has the event \(c\) (see Section 2.2.6).

   The required property holds if and only if no more than two such units share a same event. Therefore, just like the check is performed in CSP, we need only to look at the interface of this network. The fact that its internal representation is the parallel composition of smaller processes is not relevant.

2. **Associativity of the CSP\(_Z\) parallel operator**: The parallel operator (specifically the generalised or interface parallel) is symmetric and distributive, however it is only weak associative in the sense that, in general, only the following associativity law is allowed (see [10] for more details).

   \[ P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \]

   \[
   \begin{array}{cccc}
   P & \parallel & Q & \parallel & R \\
   X & X & X & X & X \\
   \end{array}
   \]

   We argue that its general associativity is guaranteed by the previous requirement. The
following lemma shows that the associativity of the generalised parallelism can be extended for more than one synchronisation set since the three processes are triple-disjoint.

**Lemma 5.2.1 (Associativity of ||)**
Let $V$ be a triple-disjoint network. Then the generalised parallelism is associative. Thus, for all processes $P, Q$ and $R$ of $V$ we have

$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$$

provided $X \cup Y = X' \cup Y'$.

**Proof.** Let $e \in X$. Then, when $e$ occurs we have one of the following transitions:

- $P \parallel X \parallel (Q \parallel Y \parallel R) \xrightarrow{(e)} P' \parallel X \parallel (Q' \parallel Y \parallel R)$ (synchronisation between $P$ and $Q$)
- $P \parallel X \parallel (Q \parallel Y \parallel R) \xrightarrow{(e)} P' \parallel X \parallel (Q \parallel Y \parallel R')$ (synchronisation between $P$ and $R$)

By the hypothesis of triple-disjointness we have that only one can happen (never both). Therefore, we can partition the transitions into two sets: one containing the participation of $P$ and $Q$, and the other of $P$ and $R$. Hence, let $e_q$ and $e_r$ be events of $X$ such that $Q \xrightarrow{(e_q)} Q'$ and $R \xrightarrow{(e_r)} R'$. Thus, the synchronisation sets $X'$ and $Y'$ are, respectively, $\{e_q | Q \xrightarrow{(e_q)} Q'\}$ and $Y \cup \{e_r | R \xrightarrow{(e_r)} R'\}$.  

3. **Divergence-freedom of the Z part:** Concerning divergence-freedom, recall that we have adopted the blocking view (see Section 2.3). Due to this fact the Z part (a distributed external choice of all the events in the interface) does not introduce divergence. Regarding the CSP part, we have to check that it does not diverge. This can be done using FDR, for example.

4. **Static topology of CSP$_Z$:** The static topology property, required in [61], is essential only for a non-mechanised analysis. The reason is that keeping track of the network’s structure during execution (by hand) is unmanageable. Nevertheless, a tool such as FDR automatically controls the execution flow and, therefore, the static topology becomes no longer a relevant requirement.

It is important to notice that the conditions 1 and 3 above are proof obligations for all applications; before using the modular approach to model-checking one must verify whether the above proof obligations hold. On the other hand, Conditions 2 and 4 are application independent. As explained above, Condition 2 is discharged by Lemma 5.2.1 and Condition 4 by the fact that the reasoning is conducted mechanically.

With FDR we can also prove whether two processes, say $P$ and $Q$, are conflict-free through refinement in the traces model. Let $\alpha P$ and $\alpha Q$ be the alphabets of $P$ and $Q$, and $S = \alpha P \cap \alpha Q$
be the synchronisation set (the interface). First we define how one can check whether there is an ungranted request from \( P \) to \( Q \), denoted \( P \xrightarrow{\sigma} Q \). Recall that this happens when \( P \) offers an event in \( S \) and \( Q \) cannot engage in it. This can be easily achieved in FDR by checking that \( P \) has a trace (of events in \( S \)) which is not a trace of \( Q \). Formally, we have:

**Lemma 5.2.2 (Ungranted Request in FDR)**

Let \( P \) and \( Q \) be two processes and \( S \) be the set of common events between \( P \) and \( Q \) (\( S = \alpha P \cap \alpha Q \)). Then \( P \xrightarrow{\sigma} Q \) is checked in the FDR tool by the following refinement check

\[
Q \setminus (\alpha Q \setminus S) \not\sqsubseteq T P \setminus (\alpha P \setminus S)
\]

which means that, in the traces model, \( P \) is not a refinement of \( Q \) when both are restricted to \( S \).

**Proof.**

\[
\exists \sigma = (s, (X_P, X_Q)) \bullet (\alpha P \setminus X_P) \cap \alpha Q \neq \emptyset \land \alpha P \cap \alpha Q \subseteq X_P \cup X_Q \\
[\equiv] \exists t : T(P/s) \bullet \text{head } t \in \alpha Q \land \langle \text{head } t \rangle \notin T(Q/s) \quad \text{[Since } P \text{ is infinite]} \\
[\equiv] \exists t : T(P \setminus (\alpha P \setminus S)/s) \bullet t \neq \langle \rangle \land \langle \text{head } t \rangle \notin T(Q/s) \quad \text{[Restricting } P \text{ to } S] \\
[\equiv] \exists s \downarrow t : T(P \setminus (\alpha P \setminus S)) \bullet t \neq \langle \rangle \land \langle \text{head } t \rangle \notin T(Q) \quad \text{[By triple-disjointness]} \\
[\equiv] T(P \setminus (\alpha P \setminus S)) \not\subseteq T(Q) \quad \text{[By 1.]} \\
[\equiv] T(P \setminus (\alpha P \setminus S)) \not\subseteq T(Q \setminus (\alpha Q \setminus S)) \quad \text{[Restricting } Q \text{ to } S] \\
[\equiv] Q \setminus (\alpha Q \setminus S) \not\sqsubseteq T P \setminus (\alpha P \setminus S) \quad \text{[By } \sqsubseteq \text{-def]} \\
\diamond
\]

Therefore, to determine if there is an ungranted request from \( Q \) to \( P \) we apply Lemma 5.2.2 as \( P \setminus (\alpha P \setminus S) \not\sqsubseteq T Q \setminus (\alpha Q \setminus S) \).

As we are concerned with conflict-freedom, we must check whether there are no simultaneous ungranted requests between \( P \) and \( Q \). The following lemma presents this result formally. It is a direct application of Definition 5.1.12.
5.3. CONCLUSIONS

Lemma 5.2.3 (Conflict-Freedom in FDR)

Let $P$ and $Q$ be two processes and $S$ be the set of common events between $P$ and $Q$ ($S = \alpha P \cap \alpha Q$). Then conflict-freedom between $P$ and $Q$ is checked in the FDR tool by the following predicate

$$Q \setminus (\alpha Q \setminus S) \sqsubseteq_T P \setminus (\alpha P \setminus S) \lor P \setminus (\alpha P \setminus S) \sqsubseteq_T Q \setminus (\alpha Q \setminus S)$$

Proof.

$$\neg (\exists \sigma \cdot P \xrightarrow{\sigma} Q \land Q \xrightarrow{\sigma} P) \quad [\text{Def. 5.1.12}]
\left[\equiv\right] \neg (Q \setminus (\alpha Q \setminus S) \sqsubseteq_T P \setminus (\alpha P \setminus S) \land P \setminus (\alpha P \setminus S) \sqsubseteq_T Q \setminus (\alpha Q \setminus S)) \quad [\text{Lemma 5.2.2}]
\left[\equiv\right] \neg (Q \setminus (\alpha Q \setminus S) \sqsubseteq_T P \setminus (\alpha P \setminus S) \lor P \setminus (\alpha P \setminus S) \sqsubseteq_T Q \setminus (\alpha Q \setminus S)) \quad [\neg - \exists]
\left[\equiv\right] Q \setminus (\alpha Q \setminus S) \sqsubseteq_T P \setminus (\alpha P \setminus S) \lor P \setminus (\alpha P \setminus S) \sqsubseteq_T Q \setminus (\alpha Q \setminus S)$$

$\square$

5.3 Conclusions

It is well-known that Model Checking is a technique limited to finite state problems. Henceforth, various strategies are being proposed in the literature [10, 22] in order to help Model Checkers to analyse systems as large as possible. The local analysis strategy can be seen as one of the most promising ones, because it has an impressive gain when it can be applied. We appreciate this gain in the next chapter, where we present in Table 6.3.2 a comparative study between a global and a local analysis of deadlock for our case-study. A work on classical Model Checking for local analysis is reported by Huhn [43].

Since the strategy of local analysis is normally oriented towards specific properties, the local analysis of deadlock is successful. Although it is not computable, in general it can turn to be computable by imposing some restrictions on the classes of processes to be analysed [10].

In this chapter, we have established that the local deadlock analysis of Section 5.1, previously defined for pure CSP, could be extended to the CSP$_Z$ setting. Further, as we always aim at applying the theoretical results in practice, we have shown at the end of Section 5.2 how the main definitions of Section 5.1 could be determined via FDR. These contributions were presented by Mota and Sampaio in [6].

After characterising how to mechanically determine ungranted requests and conflict-freedom (see Definitions 5.2.2 and 5.2.3), we are ready to start the full mechanisation of this process via integration with FDR as it is shown in Section 6.3. Meanwhile we have found the Deadlock Checker [33]. Although this tool already implements the deadlock analysis technique of Brookes and Roscoe [61] (briefly described in Section 5.1), it is very restricted in comparison to FDR. Therefore, as future work, we plan to integrate the Deadlock Checker with FDR.
Chapter 6

The Overall Strategy and a Case Study

In this chapter we present the overall strategy, and a case study which illustrates the application of the strategy. The overall strategy has emerged from the material presented in Chapters 2, 3, 4 and 5; its goal is to avoid the state explosion problem for CSP\( \mathcal{Z} \) specifications. The case study is built from five CSP\( \mathcal{Z} \) processes which, combined in parallel with the WDT process (see Chapter 2), result in a specification that represents a simplified behaviour of the On-Board Computer of the first Brazilian microsatellite for scientific applications (SACI-1 OBC\(^1\)).

This chapter is organised as follows. In the following section we show how the strategies presented in previous chapters can be combined into an overall strategy to analyse complex CSP\( \mathcal{Z} \) specifications. In Section 6.2 we present the main processes of our case study. In Section 6.3 we apply this overall strategy to our case study.

6.1 The Overall Strategy

In this subsection we present a complete strategy to apply the techniques shown in this thesis. We have ordered the steps suggesting the most natural way one can follow the proposed strategy.

6.1.1 Refine the Z part

As presented in Chapter 2, the data refinement of the Z part of a CSP\( \mathcal{Z} \) process (when preserving the CSP part) implies in the failures-divergences refinement of the whole CSP\( \mathcal{Z} \) process. This means that one can refine the data part consecutively, in a stepwise manner, until the data have a clear correspondence with those existing in CSP\(_M\) (without considering the finiteness nature of

\(^1\)A more detailed specification of this case study can be found in [3].
CHAPTER 6. THE OVERALL STRATEGY AND A CASE STUDY

the data types of CSP_M).

This step follows the standard strategy for data refinement in ordinary program development.

6.1.2 Transform CSP_Z into CSP_M

According to Chapters 2 and 3, there is a formal way of interpreting a CSP_Z process as a CSP (CSP_M) one. Then, the main goal of this step is to consider a CSP_Z specification as a CSP (CSP_M) one, without worrying about infiniteness of the original data types and expressions.

This is another delicate step, since it is not so direct to transform Z specifications into functional programming. However, recall that the goal of Step 6.1.1 is to refine the Z part until its data have a clear correspondence with those existing in CSP_M. Therefore, the current step can be applied straightforwardly and fully mechanised.

In particular, there is a trade-off between these first steps in order to mechanise the strategy as much as possible. It can be guided by observing that the CSP_Z to CSP_M translator [1, 2] already considers a possible implementation of the Z toolkit, hence almost all aspects of the Z toolkit are already available in our Java tool. Only those aspects which do not have a direct implementation need to be the focus of the first step.

6.1.3 Apply the Guided Data Abstraction

From the previous step, we have CSP_Z processes (possibly) with infinite data types. The infinite aspect is interesting, because we do not have to worry about capturing finiteness through data refinement at the first step; this is the focus of the present step.

To the infinite CSP_Z processes, we apply the data abstraction approach of Chapter 4. The goal here, as already stated in Chapter 4, is to find finite subsets for all types used in the CSP_Z processes. According to Section 4.3.1, if the CSP_Z processes have finite equivalence classes for their Z parts, then this subset search can be fully mechanised except for the proof of relatively simple predicates.

6.1.4 Apply the modular strategy for deadlock analysis

Even after finding finite representations for the CSP_M processes, the specification can still be too large to be handled by FDR as a unique process. Further, recall from Lemma 4.3.3 that the equivalence between the abstract version of a process and its original version might need a renaming, in the general situation. However, deadlock analysis is preserved under renaming [10]. Therefore, we can apply the approach of Chapter 5 in order to analyse the whole process as various smaller components, independently of renaming.
6.1.5 Workflow of the strategy

Although we have presented the overall strategy using the previous steps ordering, it is perfectly possible to apply these steps in another ways. However, some ordering dependencies might be observed; for example, Step 6.1.2 must always be applied before Step 6.1.4 since this last step relies on the use of FDR. Further, it is worth noting that none of the steps are obligatory; hence, if a specification uses data structures closer to those supported by CSP_M, then Step 6.1.1 could be avoided. A similar situation could arise if a specification can be globally analysed for deadlock-freedom; then Step 6.1.4 does not need to be performed, and so on.

An alternative ordering to the above sequence could be started with Step 6.1.2, followed by Step 6.1.3. After that, Step 6.1.1 could be applied if the data structures are too abstract with respect to those of CSP_M. Finally, Step 6.1.4 could be carried out.

6.2 Case Study: The SACI-1 Microsatellite

The SACI-1 On-Board Computer (OBC) is a fault-tolerant distributed processing system which combines software and hardware components [8, 28]. Its main parts are: its architecture, its fault-tolerant router and its application processes. Due to its fault-tolerant aspects, the SACI-1 was designed with redundant components (see Figure 6.1).

The architecture of this system (see Figure 6.1) is formed by three transputers and three replicated I/O interfaces: SRI (interface designed for Experiments Acquisition from space), UAC (interface designed to control satellite positioning—spin and attitude) and TC/TM (interface for communication to and from Earth, respectively). These interfaces are interconnected such that at a failure of some component the system is automatically reconfigured, possibly losing some of its functions.

The Fault-Tolerant Router (FTR)—designed with a reconfigurable routing algorithm—is responsible for recovering from failures. It is seen as the operating system of the satellite in the sense that most functions are provided through it, abstracting other processes from configuration details.

The Watch Dog Timer or simply WDT (Recall from Section 2.1) was designed to periodically verify the CPU state and in case of a failure it does some tasks in order to try to recover from the problem. If the problem is unrecoverable then WDT communicates this fact to the other CPUs and respective interfaces that take the respective actions.

Finally, we have the application processes that concurrently control the functions of the satellite achieving its mission. They are: Space-Ground Communication, Telecommand, Telemetry, Experiments Data Acquisition, Experiments Communication, Data Compression, Housekeeping, Attitude Control and Spin Control.
The OBC was designed to stay active even if just one CPU is alive. This is achieved through MUX (multiplexors) circuits which establishes a new connection whenever required (see Figure 6.1, dashed lines).

Each CPU has a copy of all application processes necessary to the SACI-1 mission, however just one copy of a given application process is activated at a given moment. Each FTR uses a Routing Table which determines if this FTR will treat messages from a given application process (in this case this process is considered local) or not (in which case it is remote). So, for a given process, only one FTR has an entry in its Routing Table determining that this process is local, and the other two FTRs treat this process as remote, disabling local communications with theirs copies of this process.

If some application process does not proceed, maybe due to software problems\(^2\), then the FTR detects this problem and recovers through a software reset. Before this action, however, the FTR communicates to the Housekeeping process that this error has occurred. After restart and on the right period, the housekeeping data is sent to Earth and a new process might be loaded. If a new process is to be loaded into main memory its bits are sent from Earth via the Telecommand process and then installed on the CPUs. This kind of activity is called dynamic loading.

However, in this chapter we take a more abstract view of this system (See Mota [3] for more details about the SACI-1 formalisation). We consider here a simplification of the real configuration. We consider two SACI-1 application processes, responsible for the communication between Earth and the satellite, and vice-versa, respectively: the Telecommand (TC) and the telemetry (TM) process. We also describe two auxiliary processes: one to implement an asynchronous communication between the WDT and the FTR (A_R) and the Clock Controller (SCLOCK).

### 6.2.1 Data Types

Before describing the processes themselves we introduce the following types. The free-type Message is defined in terms of the constructors TC and TM, for classifying the messages as telecommands or telemetries. The element nullMsg is defined for initialisation purposes. Both kinds of messages carry out a parameter Fields which is built as the cartesian product of Data and Integers, where Data is another free-type with three values (nullData denoting an empty message, sendTM standing for messages which should be sent to Earth by the TM process, and extra to represent other kinds of messages). The integer component is used for recording message ordering.

\[
\begin{align*}
\text{Message} & ::= \text{nullMsg} \mid \text{TC} \langle\langle \text{Fields} \rangle\rangle \mid \text{TM} \langle\langle \text{Fields} \rangle\rangle \\
\text{Fields} \quad &= \quad \text{Data} \times \mathbb{Z} \\
\text{Data} \quad &= \quad \text{nullData} \mid \text{sendTM} \mid \text{extra}
\end{align*}
\]

\(^2\)division by zero generates an interruption, communication impossibility is detected by the FTR through activity performance, etc.
6.2. CASE STUDY: THE SACI-1 MICROSATELLITE

Figure 6.1: SACI-1 Architecture Description
6.2.2 Fault-Tolerant Router

The FTR is responsible for some internal tasks, safely routing messages from process to process and for periodically sending a reset signal to the WDT. In order to model the FTR as close as possible to its original conception we consider that it can stop temporarily or permanently. In a temporary stop (where a failure occurs), the FTR can be reanimated through a recover signal. However, in a permanent one (where a fatal failure occurs) the FTR cannot be restarted.

spec FTR

chan clockFTR:[clk : CLK]
chan TC_FTR, FTR_TM:[msg : Message]
chan reset, recover_ftr, failure, fatal
lchan task
main=(Normal △ Problem)
Normal=(clockFTR?clk→(reset→Normal
□ task→Normal
□ TC_FTR?msg→FTR_TM!msg→Normal))
Problem=(failure→recover_ftr→main □ fatal→stop)

State ≜ [message : Message; time : CLK]
Init
State′
message′ = nullMsg
time′ = nonClock

com_task
Ξ State

WDTRstP : CLK

com_reset
Ξ State

WDTP(time, WDTRstP)

com_clockFTR

Ξ State

Δ State
clk? : CLK
time′ = clk?

end_spec FTR

Note that no Init schema was introduced in the specification of FTR. When this happens, any possible initialisation of the state variables is allowed.

6.2.3 Telecommand

Communications from Earth to the satellite are handled by the telecommand process. Although in the original description it performs a series of tasks (such as decoding, verification for errors, etc) to send the received message to other SACI-1 processes, here we simplify its functionality to
receive messages from Earth, and send to the destination process, which can be either the telemetry process or itself. These messages are sent through the FTR, when the destination is the telemetry process (a remote communication). When the message is to the Telecommand process itself, the treatment is totally local and is omitted here. One interesting aspect of this process is that it maintains a counter of how many messages were received. This value is sent to the Earth when the destination is the telemetry process.

```plaintext
spec TC
chan waitEarth, TC_FTR:[msg : Message]
chan remoteTask, localTask
main = waitEarth?msg → (localTask → main
    □ remoteTask → TC_FTR!msg → main)

State
message : Message
counter : Z

Init
State'
counter' = 0
message' = nullMsg

com_localTask
∀State
∃f : Fields • message = TC(f)

com_remoteTask
∀State
∃f : Fields • message = TM(f)

com_TC_FTR
△State
msg1 : Message
∃d : Data; n : Z •
message = TM(d, n)
msg1 = TM(d, counter)
counter' = 0

com_waitEarth
△State
msg? : Message
message' = msg?
counter' = counter + 1

end_spec TC
```

6.2.4 Telemetry

Data about the satellite are called telemetry, such as: temperature, voltage, some process status, etc. The telemetry process is responsible to maintain the most recent collection of stored telemetry and send it to the Earth in response to a request from the FTR. Since the telemetry storage is of
limited size and the information must be as recent as possible, if data are received and the storage is full, old data must be lost in order to allow the storage of more recent data; this is achieved in the \texttt{com\_storeTM} schema.

\begin{verbatim}
spec TM
    chan sendEarth, FTR_TM: [msg : Message]
    lchan emptyTM, moreTM, storeTM
    main=FTR_TM?msg→(SENDTM □ storeTM→main)
    SENDTM=emptyTM→main □ moreTM→sendEarth!msg→SENDTM

    | State ____________ | maxTM : \mathbb{Z} |
    | STM : \mathbb{N} → Message |
    | actMsg : Message |
    | #STM ≤ maxTM |

    | Init ____________ |
    | STM' = ∅ |
    | actMsg' = nullMsg |

    |Δ State ____________ |
    | msg? : Message |
    | actMsg' = msg? |

    | com\_emptyTM ⇔ |
    |∀ State | STM = ∅ |

    | com\_moreTM ⇔ |
    |∀ State | STM ≠ ∅ |

    |Δ State ____________ |
    | ((#STM ≥ maxTM ∧ |
    | STM' = (\{min(dom STM)\} ⊕ STM) ∪ |
    | \{i → actMsg\}) |
    | ∨ (#STM < maxTM ∧ |
    | STM' = STM ∪ |
    | \{(max(dom STM) + 1) → actMsg\}) |

end_spec TM
\end{verbatim}

\section*{6.2.5 Asynchronous Recover}

This process serves only to support an asynchronous communication between the WDT and the \textit{FTR}. The event \texttt{recover} coming from the \textit{WDT} is always enabled while the event \texttt{recover\_ftr} can only happen if at least one \texttt{recover} has occurred (\texttt{flag} = \texttt{true}). Hence, initially, no event \texttt{recover\_ftr} can occur.
6.2. CASE STUDY: THE SACI-1 MICROSATELLITE

spec $A_R$
\begin{align*}
\text{chan} & \text{ recover, recover}_{ftr} \\
\text{main} &= \text{IntRequest} ||| \text{IntExecution} \\
\text{IntRequest} &= \text{recover} \rightarrow \text{IntRequest} \\
\text{IntExecution} &= \text{recover}_{ftr} \rightarrow \text{IntExecution}
\end{align*}

\begin{align*}
\text{State} &\quad \text{Init} : B \\
\text{flag} &\quad \text{State'} \\
\text{flag'} &\quad = \text{false}
\end{align*}

end_spec $A_R$

6.2.6 OBC Clock

As CSP$_Z$ cannot capture precisely the temporal aspects of a system, we establish some conventions in order to characterise the SACI-1 as a system dependent of time. We model a process which exhibits events, carrying clock values, which control the behaviour of the WDT and of the FTR. By using a function $incC$ on clock we abstract from the way time is represented.

spec $SCLOCK$
\begin{align*}
\text{chan} & \text{clockWDT, clockFTR} : [clk : CLK] \\
\text{lchan} & \text{tic} \\
\text{main} &= \text{sendClock} \triangle \text{updateClock} \\
\text{sendClock} &= \text{clockWDT}!clk \rightarrow \text{sendClock} \quad \square \text{clockFTR}!clk \rightarrow \text{sendClock} \\
\text{updateClock} &= \text{tic} \rightarrow \text{main}
\end{align*}

\begin{align*}
\text{State} &\quad \text{Init} : \text{CLK} \\
\text{time} &\quad \text{State'} \\
\text{time'} &\quad = \text{noneClock} \\
\text{com}_{\text{clockWDT}} &\quad \text{com}_{\text{clockFTR}} \trianglerighteq \text{com}_{\text{clockWDT}} \\
\text{noneClock} &\quad \text{IncC} : \text{CLK} \rightarrow \text{CLK} \\
\text{IncC} &\quad \text{CLK} \rightarrow \text{CLK} \\
\text{com}_{\text{clockWDT}} &\quad \text{com}_{\text{clockFTR}} \trianglerighteq \text{com}_{\text{clockWDT}} \\
\text{com}_{\text{tic}} &\quad \trianglerighteq [\Delta \text{State} | \text{time'} = \text{IncC}(\text{time})]
\end{align*}

end_spec $SCLOCK$
Observe that the current value of the time variable is sent to the WDT or to the FTR processes at any moment. Note also that the time variable can be incremented by an interruption of the tic event.

6.2.7 SACI-1

The simplified behaviour of the SACI-1 microsatellite is given by an alphabetised parallel composition (||) of the previous five CSP\_Z components. In this specification, the elements inside the brackets of the parallel operator are the synchronisation points. The definition of the SACI-1 process illustrates that specification units can be combined just like ordinary CSP processes.

\[
\text{SACI-1} = (((WDT || A_R || FTR || SCLOCK) || TC) || TM}
\]

where \(X_1 = \{\text{recover}\}, X_2 = \{\text{reset, recover_ftr}\}, X_3 = \{\text{clockWDT, clockFTR}\}, X_4 = \{\text{TC_FTR}\}\) and \(X_5 = \{\text{FTR_TM}\}\).

In Appendix B we present the translation of each of the SACI-1 processes into the FDR notation, except for the WDT whose translation was already given in Section 3.7.

6.3 Investigating the SACI-1 Network

Section 6.2 has presented the SACI-1 OBC through its processes. They are all infinite state, either because the communications are infinite or the state has variables of infinite types. Therefore, in this section we investigate these processes according to the strategy of Section 6.1, so that we can apply model checking via FDR. The whole strategy will be presented for the process of Section 6.2.4: the Telemetry process. For the remainder processes, we give a description of the steps carried out (see Table 6.3.1).

6.3.1 Investigating the Telemetry Process

In what follows, we give a detailed presentation of the strategy of Section 6.1 applied to the Telemetry process. We begin with a data refinement, followed by a data abstraction and, finally, by a transformation from CSP\_Z to CSP\_M. The modular deadlock analysis is applied to all processes jointly, as a network.

Data Refinement

The goal of data refinement is to replace an abstract data type by a more concrete one. The choice for which concrete data types to use is—in general—an user responsability because it needs some expertise. We have chosen the Telemetry process since it is a simple example of how to
apply this step. Recall from Section 6.2.4 that it uses an abstract data type to store messages: a
partial function. However, from its informal and formal description we can see that it also uses a
mechanism to characterise an ordering: \( qtdSTM \). We can replace the partial function description
by a sequence-based one. Therefore, our task is to show that the Z part preserves the semantics
when using sequences instead of partial functions.

Following the standard data refinement approach for Z [47, 35], we have to propose a new state
space (\( \text{SeqState} \)) and a relation (\( \text{Retrieve} \)) between \( \text{SeqState} \) and the state presented in Section 6.2.4
(referred here by \( \text{TM}.\text{State} \)). After that, we have to propose a new operation—based on \( \text{SeqState} \)—
for each operation of the Telemetry process. Finally, we must prove that this new description is
compatible with the previous one via the proof obligations of the Z data refinement theory [47].

Hence,

\[
\begin{align*}
\text{SeqState} & \\
STM_{seq} & : \text{seq Message} \\
currMsg & : \text{Message} \\
\#STM_{seq} & \leq \text{TM}.\text{maxTM}
\end{align*}
\]

\[
\begin{align*}
\text{Retrieve} & \\
\text{TM}.\text{State} & \\
\text{SeqState} & \\
squashSTM & = STM_{seq} \\
actMsg & = \text{currMsg}
\end{align*}
\]

Now, we propose an initialisation schema and corresponding Z operations\(^3\) (based on the new
state space \( \text{SeqState} \)). Thus,

\[
\begin{align*}
\text{Init}_{seq}' & \\
\text{SeqState}' & \\
STM_{seq}' & = \langle \rangle \\
currMsg' & = \text{nullMsg}
\end{align*}
\]

\[\text{com\_emptyTM}_{seq} \equiv [\exists \text{SeqState} \mid STM_{seq} = \langle \rangle] \]

\[\text{com\_moreTM}_{seq} \equiv [\exists \text{SeqState} \mid STM_{seq} \neq \langle \rangle] \]

\[\text{com\_sendEarth}_{seq} \quad \Delta \text{SeqState} \]

\[
\begin{align*}
\text{msg}! & : \text{Message} \\
\text{msg}! & = \text{head} \ STM_{seq} \\
STM_{seq}' & = \text{tail} \ STM_{seq}
\end{align*}
\]

\[\text{com\_FTR\_TM}_{seq} \quad \Delta \text{SeqState} \]

\[
\begin{align*}
\text{msg}? & : \text{Message} \\
\text{currMsg}' & = \text{msg}?
\end{align*}
\]

\[\text{com\_storeTM}_{seq} \quad \Delta \text{SeqState} \]

\[
\begin{align*}
\left(\#STM_{seq} & \geq \text{maxTM} \land \\
\text{STM}_{seq}' & = \text{tail} \ STM_{seq} \sim (\text{currMsg})\right) \\
\lor \\
\left(\#STM_{seq} & < \text{maxTM} \land \\
\text{STM}_{seq}' & = STM_{seq} \sim (\text{currMsg})\right)
\end{align*}
\]

\(^3\)Recall that we are assuming equalities between unprimed and primed variables when omitted to save space.
The final effort in this step is to prove that the current Z description is a data refinement of the Z part of Section 6.2.4. It is worth noting that the proofs presented here are given using the theorem prover Z-Eves [46]. For each operation, we have to show that its current and previous versions are compatible before and after application. These obligations are captured by Lemmas 6.3.3 and 6.3.7, instantiated for operation sendEarth_{seq}. Lemma 6.3.2 concerns the proof obligation for the initialisation of the Z part.

Lemma 6.3.2 refers to the consistent initialisation for the abstract and concrete versions of the process $TM$. Our first attempt to prove this lemma fails because Z-Eves cannot infer that $\langle \rangle = \text{squash}[X] \Rightarrow T = \emptyset$. Thus, we introduce this result in the data refinement as Lemma 6.3.1 (precisely, as a forward rule [46]).

**Lemma 6.3.1** ($\text{nullSquash}[X]$)

$\langle \rangle = \text{squash}[X] \Rightarrow T = \emptyset$

The first thing to do is stating the theorem to be proven: try lemma initRef. Always that a theorem involves schemas, the first proof command is prove by reduce. This proof command starts by expanding all schema names for their definitions and then an iterative process takes place. This iterative process concerns the application of some specific Z-Eves proof commands followed by a simplification step. The process ends exactly when the current predicate is equal to the previous one. Since Lemma 6.3.1 was introduced in the proof context, Z-Eves yields a true after the first proof command.

**Lemma 6.3.2** (initRef)

$\text{Init}_{seq} \land \text{Retrieve}' \Rightarrow \text{TM.Init}$

**Proof.** The proof follows via Z-Eves commands.

1. try lemma initRef;
2. prove by reduce;

The following lemmas illustrate the standard data refinement for one operation of $TM$. For the remainder operations the lemmas and proofs are similar. Lemma 6.3.3 refers to the applicability criteria. That is, if the operation sendEarth can be applied (its precondition holds) then the operation sendEarth_{seq} can also be applied. In Lemma 6.3.3, the command split is used to inform Z-Eves to pay special attention on a specific predicate.

**Lemma 6.3.3** (appSend)

pre $TM.\text{com}_{\text{sendEarth}} \land \text{Retrieve} \Rightarrow \text{pre com}_{\text{sendEarth}_{seq}}$

**Proof.** The proof follows via Z-Eves commands.
Complementarily, Lemma 6.3.7 guarantees that the result of the application of operation $sendEarth$ is compatible—in the sense of the $Retrieve$ relation—to the result of the operation $sendEarth_{seq}$. To ease the proof of Lemma 6.3.7, the following auxiliary lemmas are introduced in the proof context.

**Lemma 6.3.4 (minHead)\[\text{\text{]}\]**

\[\text{head}(\text{squash}[X]S) = S(\text{min}(\text{dom}S)) ↔ \text{true}\]

**Lemma 6.3.5 (minTail)\[\text{\text{]}\]**

\[\text{\{}\text{min}(\text{dom}S)\}\prec S = \text{tail}(\text{squash}[X]S)\]

**Lemma 6.3.6 squashNeutral\[\text{\text{]}\]**

\[S \in \text{seq}X \Rightarrow \text{squash}[X]S = S\]

After introducing the above lemmas in the proof context, Lemma 6.3.7 is proven by the usual command `prove by reduce`.

**Lemma 6.3.7 (corrSend)**

\[\text{pre } TM.\text{com}_{sendEarth} \land \text{com}_{sendEarth_{seq}} \land Retrieve \land Retrieve' \Rightarrow TM.\text{com}_{sendEarth}\]

**Proof.** The proof follows via Z-Eves commands.

1. try lemma corrSend;
2. prove by reduce;
Mechanised Data Abstraction

The purpose of this section is to present our mechanised data abstraction approach applied to the Z part of the Telemetry process. We consider the Z part presented in the previous section, since we are applying the default workflow (in particular, we describe the abstraction step using Z notation to ease of presentation). We follow the algorithm proposed in Section 4.3.2, presenting the proof obligations we have to check to guarantee stability. Indeed, the interactions of the algorithm are being presented in the format of our Haskell prototype. As a result, we have a data abstraction for the currently considered process. The algorithm is executed by calling the function \texttt{findAbstraction} as follows

\[
\texttt{findAbstraction } \langle(\langle\texttt{accS}_\text{Init}, \tau\rangle, \texttt{accS}_\text{Init})\rangle
\]

where \(\texttt{accS}_\text{Init} = \texttt{validOpsers}()\) \(\textbf{Interface} = \{\texttt{emptyTM}, \texttt{FTR}_\text{TM}, \texttt{storeTM}\}\). Note that we have suppressed the subscript \(\text{seq}\) of the Z part components because it is irrelevant for the present explanation.

The first interaction given by the algorithm is

\[
\begin{align*}
\texttt{comp} & \triangleq \texttt{com}_\text{FTR}_\text{TM} \\
\texttt{conj} & \triangleq \texttt{pre com}_\text{FTR}_\text{TM} \land \texttt{pre com}_\text{emptyTM} \land \texttt{pre com}_\text{storeTM} \land \\
& \quad \neg \texttt{pre com}_\text{sendEarth} \land \neg \texttt{pre com}_\text{moreTM}
\end{align*}
\]

Followed by the question: \textbf{Is } \forall \texttt{State}; \texttt{State}' | \texttt{conj} \cdot \texttt{comp} \Rightarrow \texttt{conj}' \texttt{true}? After submitting this predicate to Z-Eves, we get \texttt{true} using the usual commands, as illustrated in the previous subsection. Thus, a new interaction takes place

\[
\begin{align*}
\texttt{comp} & \triangleq \texttt{com}_\text{emptyTM} \\
\texttt{conj} & \triangleq \texttt{pre com}_\text{FTR}_\text{TM} \land \texttt{pre com}_\text{emptyTM} \land \texttt{pre com}_\text{storeTM} \land \\
& \quad \neg \texttt{pre com}_\text{sendEarth} \land \neg \texttt{pre com}_\text{moreTM}
\end{align*}
\]

Followed by the question: \textbf{Is } \forall \texttt{State}; \texttt{State}' | \texttt{conj} \cdot \texttt{comp} \Rightarrow \texttt{conj}' \texttt{true}? Again, we get \texttt{true} by interacting with Z-Eves. After giving the answer \texttt{yes} to the algorithm, we get

\[
\begin{align*}
\texttt{comp} & \triangleq \texttt{com}_\text{storeTM} \uplus \texttt{com}_\text{sendEarth} \\
\texttt{conj} & \triangleq \texttt{pre com}_\text{FTR}_\text{TM} \land \texttt{pre com}_\text{emptyTM} \land \texttt{pre com}_\text{storeTM} \land \\
& \quad \neg \texttt{pre com}_\text{sendEarth} \land \neg \texttt{pre com}_\text{moreTM}
\end{align*}
\]

And the question \textbf{Is } \forall \texttt{State}; \texttt{State}' | \texttt{conj} \cdot \texttt{comp} \Rightarrow \texttt{conj}' \texttt{true}? receives an \texttt{yes}, again. The next interaction is

\[
\begin{align*}
\texttt{comp} & \triangleq \texttt{com}_\text{FTR}_\text{TM} \\
\texttt{conj} & \triangleq \texttt{com}_\text{sendEarth} \land \texttt{pre com}_\text{FTR}_\text{TM} \land \texttt{pre com}_\text{storeTM} \land \\
& \quad \texttt{pre com}_\text{storeTM} \land \neg \texttt{pre com}_\text{emptyTM}
\end{align*}
\]
Note that this time the property has changed but we obtain true. Another interaction with the algorithm gives

\[
\begin{align*}
comp & = \text{com\_moreTM} \\
conj & = \text{pre com\_sendEarth} \land \text{pre com\_FTR\_TM} \land \text{pre com\_moreTM} \land \\
& \quad \text{pre com\_storeTM} \land \neg \text{pre com\_emptyTM}
\end{align*}
\]

It worth observing that the current property is the same as the previous. Through the operation com\_moreTM this property does not change and Z-Eves returns true. By giving yes to the algorithm we get

\[
\begin{align*}
comp & = \text{com\_storeTM} \\
conj & = \text{pre com\_sendEarth} \land \text{pre com\_FTR\_TM} \land \text{pre com\_moreTM} \land \\
& \quad \text{pre com\_storeTM} \land \neg \text{pre com\_emptyTM}
\end{align*}
\]

which has proof status true and the algorithm terminates with the following data abstraction

\[Message^A = \{\text{nullMsg}\}\]

**Transformation from CSP\_Z to CSP\_M**

The transformation phase is straightforward, except for some predicates. Thus, after applying the macro patterns to the Telemetry process we have to check if all predicates have a direct representation in CSP\_M. This was performed for the Telemetry process yielding the following result.

```plaintext
datatype Dest  = TCp | TMp
datatype Data  = sendTM | extra
nametype Counter = {0..2}

nametype Message = Dest.Data.Counter

channel FTR\_TM: Message
cchannel sendEarth: Message
cchannel emptyTM, moreTM, storeTM, sendSTM

TM =
let
  -- The Interface
  Channels = \{FTR\_TM, sendEarth, emptyTM, moreTM, storeTM, sendSTM\}
  lChannels = {}
  Interface = union(Channels, lChannels)
```

```plaintext
```
-- The CSP Part

main = FTR_TM\d.m.c -> (sendSTM -> SENDTM [] storeTM -> main)
SENDTM = emptyTM -> main [] moreTM -> sendEarth?d.m.c -> SENDTM

-- The Z Part

maxTM = 1
SEQM = FSeq(Message, maxTM)
State = { (STM, actMsg) | STM <- SEQM, actMsg <- Message }
Init = { (<>, actMsg') | actMsg' <- Message }
com((STM, actMsg), FTR_TM.msg) = { (STM, msg) }
com((STM, actMsg), sendSTM) =
  if {n | n <- Counter,
      (actMsg==(TMp.sendTM.n)) or
      (actMsg==(TCp.sendTM.n))}!=
  then { (STM, actMsg) } else {}
com((STM, actMsg), storeTM) =
  if {d | d <- Data, n <- Counter, d!=sendTM,
       ((actMsg==(TMp.d.n)) or
       (actMsg==(TCp.d.n)))}!=
  then
    union(if #STM<maxTM then
       { (STM^<actMsg>, actMsg) }
    else {},
    if #STM=maxTM then
      { (tail(STM)^<actMsg>, actMsg) }
    else {})
  else {}
com((STM, actMsg), emptyTM) = if STM==<> then
  { (STM, actMsg) }
else {}
com((STM, actMsg), moreTM) = if STM!==<> then
  { (STM, actMsg) }
else {}
com((STM, actMsg), sendEarth.msg) = if STM!==<> and msg==head(STM)
  then
    { (tail(STM), actMsg) }
  else {}

Z_CSP =
let
  Z(State) = [] (States,Comm):
  { (com(State, c), c) | c <- Interface }
6.3. INVESTIGATING THE SACI-1 NETWORK

<table>
<thead>
<tr>
<th>Process</th>
<th>Description</th>
</tr>
</thead>
</table>
| **FTR** | The data refinement step is not necessary  
The data abstraction gives  
\(Message^A = \{nullMsg\}\)  
\(CLK^A = \{WDTRstP, noneCLK\}\), such that \(\neg WDTP(\text{noneClk, WDTRstP})\)  
The translation step is presented in Appendix B |
| **TC** | The data refinement step is not necessary  
The data abstraction gives  
\(Message^A = \{nullMsg, TC(nullData, 0), TM(nullData, 0)\}\)  
\(Z^A = \{0\}\)  
The translation step is presented in Appendix B |
| **A_R** | The data refinement step is not necessary  
The data abstraction step is not necessary  
The translation step is presented in Appendix B |
| **WDT** | The data refinement step is not necessary  
The data abstraction gives  
\(CLK^A = \{WDTRstP, noneCLK\}\), such that \(\neg WDTP(\text{noneClk, WDTRstP})\)  
The translation step is presented in Appendix B |
| **SCLOCK** | The data refinement step is not necessary  
The data abstraction gives  
\(CLK^A = \{WDTRstP, noneCLK\}\), such that \(\neg WDTP(\text{noneClk, WDTRstP})\)  
The translation step is presented in Appendix B |

Table 6.1: Overall Strategy Results

\[ @ \text{States}!={} & \]
\[ |\sim| \text{State'}: \text{States} @ \text{Comm} \rightarrow Z(\text{State'}) \]
\[ \text{within } |\sim| \text{iState}: \text{Init} \circ Z(\text{iState}) \]
\[ \text{within } (\text{main } [\text{Interface}] Z_{\text{CSP}})\backslash \text{Channels} \]

6.3.2 The Modular Deadlock Analysis

After applying data refinement wherever possible, followed by the mechanised data abstraction and the translation steps, we can apply the modular deadlock analysis. This step was not applied to the processes in isolation. The reason is that the modular deadlock analysis was (obviously) proposed for groups of processes (networks). Therefore, in this section we apply this step to the set of processes considered in our description of the SACI-1 OBC.

Recall from Chapter 5 that the modular deadlock analysis is built of smaller steps which
must be followed in order. They are

**Step 1.** Verify whether the network is busy. That is, check all processes individually against deadlock. If it is not busy, then at least one of its processes has a deadlock, and the strategy is not applicable.

For our network, we can load the *FTR* process into FDR and verify that this process has a deadlock state, specifically for the trace \( \langle \tau, \text{fatal} \rangle \) where \( \tau \) denotes an internal event. The *WDT* process also deadlocks after:

\[
\langle \text{clockWDT}.\text{clk6}, \tau, \text{recover}, \text{clockWDT}.\text{clk6}, \tau \rangle
\]

These deadlock states do not happen to be a problem because they are expected from the requirements of the OBC. The *FTR* process was modelled having this behaviour in mind; the cause of this mal-function comes from external problems (radiations, hardware failures, etc). With a similar argument, the *WDT* can only try to recover the *FTR* process for three times.

The purpose of our analysis is to show that the traces above are the only possible deadlock states. Therefore, we continue our analysis by removing the deadlock possibility of the *FTR* process (event fatal) and allowing the *WDT* to attempt recovering the *FTR* an arbitrary (possibly infinite) number of times. Thus, by loading the processes *WDT* and *FTR* (modified as explained above), *TC*, *TM*, *A_R* and *SCLOCK* in FDR we get a “√” (meaning success) for each assertion of the form

\[
\text{assert } P: [\text{deadlock free [F]}] \quad (\text{Is } P \text{ deadlock-free according to the Failures-model?})
\]

indicating that the network is busy.

**Step 2.** Draw the communication graph of the network and check for possible disconnecting edges. Figure 6.2 presents the communication graph for the SACI-1 OBC. The edges \( \text{de}_1 \) and \( \text{de}_2 \) are disconnecting edges because their removal increases the number of graph components, namely: \( \langle \text{TC} \rangle, \langle \text{TM} \rangle \) and \( \langle \text{FTR, WDT, A_R, CLK} \rangle \).

**Step 3.** Check the conflict-freedom between the processes linked by the disconnecting edges. Figure 6.3 shows the processes required to be conflict-free; these are the processes inside the regions \( \text{cf}_1 \) and \( \text{cf}_2 \). Recall from Section 5.2 that we can check these conflict-freedom obligations via FDR through the following assertions:

1. \text{assert } \text{FTR}\text{\textbackslash diff}(\text{Events},\{\text{TC}_\text{FTR}\}) \quad [T= \text{TC}\text{\textbackslash diff}(\text{Events},\{\text{TC}_\text{FTR}\})]
2. \text{assert } \text{TC}\text{\textbackslash diff}(\text{Events},\{\text{TC}_\text{FTR}\}) \quad [T= \text{FTR}\text{\textbackslash diff}(\text{Events},\{\text{TC}_\text{FTR}\})]
3. \text{assert } \text{FTR}\text{\textbackslash diff}(\text{Events},\{\text{FTR}_\text{TM}\}) \quad [T= \text{TM}\text{\textbackslash diff}(\text{Events},\{\text{FTR}_\text{TM}\})]
4. \text{assert } \text{TM}\text{\textbackslash diff}(\text{Events},\{\text{FTR}_\text{TM}\}) \quad [T= \text{FTR}\text{\textbackslash diff}(\text{Events},\{\text{FTR}_\text{TM}\})]

gives us three “√” for assertions 1, 3 and 4 and an “X” for assertion 2 stating that the pair of processes \( \langle \text{FTR, TC} \rangle \) and \( \langle \text{FTR, TM} \rangle \) are conflict-free with only one ungranted request from *FTR* to *TC*. 
Step 4. Check whether the essential components are deadlock-free. Figure 6.4 shows the essential components for our case study. We have to load the subnetwork

\[
\langle FTR, WDT, A_R, SCLOCK \rangle
\]

into FDR and check for deadlock. For this subnetwork, deadlock-freedom was obtained after eliminating the expected deadlock possibilities of the WDT and the FTR, as previously explained.

To give an idea of the effectiveness of this modular approach to deadlock analysis, Table 6.3.2 summarises the results obtained by a modular and by a global analysis of deadlock using the FDR model-checker. A global analysis of the SACI-1 specification takes 201.168 states (see Table 6.3.2). Concerning the incremental analysis, first we proved that the network is busy, checking FTR,
TC, TM, WDT, A_R and SCLOCK against deadlock. Then we checked that \((TC, FTR)\) and \((TM, FTR)\) are conflict-free, and then we proved that kernelSACI1 (see the group ec3 in Figure 6.4) is deadlock-free. So, by Theorem 5.1.1 SACI1 is also deadlock-free. The total number of states of the incremental proof is 3,426, which is only 1,7% of the number of states taken by the global analysis. Regarding the number of transitions, we found a percentage of 0,8%.

A last observation is that, for some specifications, only the modular approach can be used to check the desired results; in this paper we used a simplified case study to demonstrate the approach, but for real and complex systems, a global analysis might not be possible using FDR, because of the exponential increase of the number of states. Actually, as mentioned previously, without the optimisation introduced in Section 3.4.1, the global analysis caused a virtual memory overflow for a SPARC station with 711MB of virtual memory, while the modular analysis produced the expected result with 50,876 states and 280,005 transitions.

6.4 Conclusions

In this chapter, we have shown how the material of the previous chapters could be integrated in an overall strategy. In this strategy, we have proposed a standard step-based ordering with a simple variation, and a step dependency analysis. We have considered a case study—a subset of the On-Board Computer of the SACI-1 microsatellite [28, 8]—for illustrating the overall strategy.

We have taken the CSP\(_Z\) specification of the Telemetry process, from our case study, to describe the overall strategy in full detail. In particular, the last step of this strategy concerns modular deadlock analysis, which is suitable for analysing groups of processes (networks of processes). Therefore, in this last step, we have considered all processes described in this work together. For instance, this modular analysis has revealed an impressive gain when compared to a default
Concerning the Telemetry process, we have applied the standard data refinement and the mechanised abstraction approaches. Finally, we have shown how this process looks like in pure CSP (more precisely, in CSP\(_M\)).

The material presented in this chapter was achieved using tool support. We have used FDR for model checking, Z-Eves for theorem proving (regarding data refinement and the mechanised abstraction algorithm), and our Haskell prototype.
Chapter 7

Conclusion

In this thesis we have shown how to perform model checking for CSP\(_Z\) [4, 5, 6]. Initially, we have thought of proposing a completely new approach to model checking tailored by the CSP\(_Z\) needs; however, we realised that a more fruitful and productive contribution could be given in the same direction CSP\(_Z\) was designed [16, 17], that is, by reusing theories and tools. Therefore, our first step was to show how a CSP\(_Z\) process could be seen as a pure CSP process, via a translation approach. The interesting point about this translation approach is concerned with the Z part, which is transformed into a kind of CSP normal form process. Further, we have proven that our translation strategy is correct with respect to the semantics of CSP\(_Z\). Finally, since the specification language CSP already has a model checker—the FDR tool [13]—we can readily use FDR to model check CSP\(_Z\) processes, after translation. This translation strategy was implemented by Farias [2].

During the elaboration of the translation strategy, we have observed that we could optimise some conversion patterns in order to generate smaller structures. In particular, without the use of this optimisation phase, we could not analyse our case study in a global way (see Chapter 6), for the purpose of making a comparison to the modular approach; this is further discussed later in this text. Moreover, such optimisations (compressions) are very attractive and desirable by the model checking community. For instance, one of the biggest revolutions in this research area was presented by McMillan [39] (firstly investigated in his PhD Thesis [38]), by realising that, by using a symbolic representation, much larger systems could be analysed. This symbolic representation—known as ordered binary decision diagrams [56] (or simply OBDDs)—is an optimised (condensed) equivalent manner of representing boolean expressions.

It is well-known—from the literature—that the model checking technique can only be used for finite systems [10, 22]. However, the increasing interest of the academic and industrial communities on this technique, and the possibility of the system specifications to become infinite have led the researchers to propose alternatives to tackle this undesired limitation. More specifically, CSP\(_Z\) is a language which has a natural way of building infinite state systems, since its model-based part can deal with infinite data types [47, 35]. All proposals are underlay on a very basic principle:
abstraction [22]. However, it is worth observing that some classes of abstractions lose some properties of the system, whereas others do not. In this thesis, we have investigated three of such abstractions: a data independent based—which does not lose properties but are restricted to data independent systems [54, 58]—, a data abstraction based—which can lose some properties of the system but can deal with more complex processes [52, 55, 19, 26, 57, 23]—and a modular based—which does not lose properties, but works only for a delimited class of processes [61, 53, 32, 43, 37]. In general, the abstractions which can lose some properties of the system can deal with larger classes of processes, whilst those which do not lose properties are restricted to certain classes of processes [10, 22].

The data independent and abstraction approaches were analysed cojointly, because CSP Z can be seen as formed of a data independent (the CSP part) and a data dependent (the Z part) parts, as previously explained. Moreover, we have focused on research works which are more closely related to CSP [58, 26]. The importance of the work of Lazić [58] was to reveal a flaw in some results of Wehrheim [26]. The point is that Wehrheim has assumed that the CSP part cannot influence the Z part. Although this assumption is very reasonable, we have observed—based on the work of Lazić—that it is not valid in general. Our argument is simple: suppose that the CSP part of a CSP Z specification is written in such a way that it needs satisfaction of some minimal data requirements. If the data abstraction for the Z part does not satisfy these requirements, then the CSP part might lose some of its properties. Therefore, in the present work, we have avoided this problem by requiring that the CSP part be written in such a way that it conforms to any data abstraction chosen for the Z part. We have proposed extensions for the corresponding results of Wehrheim and proven them.

Besides showing how the work of Lazić complements that of Wehrheim, we have given another contribution in the direction of data abstraction. We have shown that it is possible to mechanise the search for a data abstraction [7], a task normally transferred to the user and based on heuristics [22]. The translation of a CSP Z specification into a CSP process, where the Z part assumes a kind of normal form, as previously explained, was crucial to allow the mechanisation to be relatively straightforward. The key point resides on three relevant aspects:

- The format of the Z part: an external choice over all the events (operations) turned out to be very convenient to be explored;
- Our data abstraction proposal assumes that we can replace the original infinite data types being used with corresponding finite subtypes. These subtypes originate from the partitions built from the preconditions of the Z schemas;
- A subtype can be built when we can guarantee that the behaviour of the Z part is periodic. That is, an infinite sequence like \(\langle a^0, a^1, a^2, \ldots \rangle\) can be replaced with the sequence \(\langle a^0, a^0, a^0, \ldots \rangle\) when the superscripts (states) are proven equivalent in some sense (for example, being \(\geq 0\)). Besides, the final benefit of this abstraction is that a finite labelled
transition system—corresponding to the infinite CSP process $\mu P.a^0 \rightarrow P$—captures this latter sequence.

In respect to guaranteeing that a given behaviour is periodic, we have followed the research direction strongly advocated by Pnueli [68], where he suggests the integration of model checking with theorem proving. Although we are basing our work on theorem proving, which is not fully automatic in general, when dealing with non-decidable logics [31], we have proposed a guided strategy instead of depending on user expertise. In particular, we can restrict the kind of predicates allowed in the Z part and work with decidable languages, like Presburger arithmetic [12], propositional logic with quantifiers [60], etc.

To experiment our mechanical data abstraction, we have developed a simple prototype in Haskell [64]. To use the prototype, we have to transform the Z part of a CSP$_Z$ specification into a functional style, like the transformation presented in Chapter 3 (more precisely in Section 3.4). After establishing the input for the first call, the prototype runs alone by investigating the Z part. When it achieves a point where some property repeats, it interacts with the user by presenting the stability predicate (with additional information)—in a \LaTeX [41] style, suitable to be loaded directly by Z-Eves—the user has to prove. Currently, we have to answer a yes or no to the prototype in order it can know if the submitted predicate is valid. We have used the prototype on Examples 4.3.2, 4.3.3, and 4.3.4, as well as in our case study (see Chapter 6).

Concerning modular abstraction, we have shown how the network decomposition strategy, developed for CSP by Brookes and Roscoe [61] for manual deadlock analysis, can be adapted for CSP$_Z$, allowing the analysis of CSP$_Z$ specifications in a modular fashion. Further, we have found a way of using FDR to support the analysis. This new result was presented and proven. We have discussed the advantages of applying such a modular analysis quantitatively, making a comparison between a global and local analysis of our case study. Although the result of this comparison was related to a single system, its outcome was impressive. Another direction for analysing deadlock, not considered in this thesis, is to assume that the CSP$_Z$ specification was built using deadlock-free configuration patterns or that it has special configurations (that is, rings, client-server, etc), as shown by Cunha and Justo [53] and by Martin [32]. This latter approach is more general than that used in this work, but assumes some user expertise for developing such special systems; an assumption discarded in our approach.

We have also considered a case study; it consists of a subset of the On-Board Computer (OBC) of a Brazilian artificial microsatellite (SACI-1) [28] developed by the Brazilian Space Research Institute (INPE). Based on this case study, we have presented an overall strategy for model checking by considering data refinement aspects, the mechanised data abstraction, and the modular analysis of deadlock with FDR support.

In summary, apart from defining a model checking strategy for an integrated language which combines a process algebra (CSP) with a model-based formalism (Z), we have hopefully contributed to the state-of-the-art of controlling the state explosion problem and, more specifically, analysing
CHAPTER 7. CONCLUSION

(possibly) infinite systems.

7.1 Related Work

Our initiative of translating CSP$_Z$ into CSP for model checking purposes [4, 5, 6] can also be appreciated in the works of Fischer and Wehrheim [18], and Havelund [36]. While we concentrate on CSP$_Z$, Fischer and Wehrheim deal with CSP$_OZ$, and Havelund gives a more audacious step towards presenting a model checking strategy for Java programs (although Havelund considers a restricted subset of Java, as expected).

Concentrating on CSP$_OZ$ [17], Wehrheim [26] has pointed out the possible results that could be achieved via a data abstraction approach to CSP$_Z$. The major difference to our work is that we have also considered the possible influence of the CSP part—taking the work of Lazić [58] as a basis—on the Z part, as well as the way the abstract domains and operations can be found. In respect to the work of Lazić [58], we can analyse a larger class of CSP processes: the partially dependent processes. This is a class which contains the data independent processes. This related advantage is the result of an integrated approach which combines the works of Wehrheim [26] and Lazić [58].

Regarding the mechanised aspect of our approach for data abstraction, there are several researchers tackling the same challenge. The most fruitful strategy—to the best of our knowledge—deals with applying boolean abstractions [49, 40]. By a suitable choice of a predicate (invariant), infinite predicates and expressions are replaced by boolean variables according to the invariant chosen. The generation of the abstracted system is mostly mechanical via fixpoint computations which, for some strategies, always terminate. However, once again the key point is on discovering an appropriate data abstraction (invariant), which is still a user responsibility [49]. Our approach tackles this problem automatically (or requires user intervention just during theorem proving) and does not demand that the user propose the abstraction.

Concerning modular analysis, our work—which is an adaptation of the technique developed by Brookes and Roscoe [61]—is comparable to the work of Martin and Jassim [33] and that of Cunha and Justo [53]. The difference between these works resides on the tool support. We reuse the FDR tool (which is a general purpose tool for CSP), Cunha and Justo [53] is a manual proposal, while Martin and Jassim propose a completely new and specific tool. Another difference is the class of processes on which we can perform deadlock analysis. Martin and Jassim as well as Cunha and Justo can analyse a larger class of processes by considering special network configurations (for example, rings, client-server, etc).
7.2 Future Research

Although the research undertaken in this thesis will hopefully be useful to the model checking community, much more work has to be done. In what follows, we present some of the tasks we wish to perform in the near future.

- **Implement the algorithm proposed in Chapter 4**: In a practical context, a first work to be accomplished is the implementation of the data abstraction algorithm, proposed in Chapter 4. Currently, we have a simple prototype which was primarily conceived to give some aid in the analysis of the case study of Section 6.2. Therefore, the goal is building an integrated environment where theorem provers, like Z-Eves [46], ACL2 [44], or PVS [63], could work together with model checkers, like FDR [42] or SPIN [24], as much transparently to the user as possible.

- **Implement the Strategy of Chapter 6**: After achieving the previous goal, we intend to implement the overall strategy presented in Chapter 6. The idea is to build a workbench supporting this strategy. We already have a translator from CSP\_Z to CSP [2]; the theorem provers previously cited can be used to guarantee data refinements; available model checkers can also be used.

- **Generalising the Renaming used on Lemmas 4.3.2 and 4.3.3**: Recall that the Lemmas 4.3.2 and 4.3.3 consider renamed versions of the original infinite processes. Therefore, in general, only those properties preserved via renaming can be analysed. However, there is no catalog of properties preserved via renaming. For example, we know that deadlock and livelock are renaming independent, but determinism is not, in general [10]. Therefore, another research direction is to explore what are the constructors a process can use in order to preserve properties via renaming. Perhaps, an interesting starting point can be the work of Leuschel et al [45]. This research presents relationships between traditional model checking and refinement checking. Further, it shows that it is possible to generate a CSP process from a logical formula, in some cases.

- **Compositional Results for Lemma 4.3.3**: According to Wehrheim’s work [26], one can only obtain compositional results regarding trace based properties. Since this is a very limited set of properties, we plan to investigate how to achieve similar results for failures and failures-divergent based properties. We expect to accomplish this based on the conclusions of the previous goal—concerning specifically Lemma 4.3.3.

- **Unstable Processes**: Recall from Chapter 4 that our data abstraction algorithm concerns only optimal abstractions. The reason is that we have related optimality with stability (or periodicity). That is, if all expansions of the Z part are stable, then we can make this process finite by capturing the infinite periodic traces by recursive equivalent ones. However, when
the process is unstable, that is, there exists at least one expansion of the Z part which is not periodic (non-deterministic), our algorithm becomes non-terminating. Therefore, we intend to investigate this problem more closely. To the best of our knowledge, there is not any related work which solves this problem. There are two alternatives we think could be promising to solve this problem: one is to isolate the source of non-determinacy and see if it belongs to a data independent class; the other is seeking if a non-deterministic solution—concerned to trace based properties—solves the problem.

- **Classify Processes According to their Degree of Decidability:** Recall from Chapter 4 that our algorithm relies on proof obligations over, generally, non-decidable predicates. Since this is contrary to the advantage of model checking—fully automatic—, an obvious progress would be to classify processes according to the decidability of the underlying language of predicates. For example, data independent processes can be analysed by the approach of Lazić [58], processes using Presburger arithmetic [12] could be analysed automatically through some Presburger arithmetic solver, quantified propositional based processes could be analysed by satisfiability problem solvers [60], etc.

- **Integrate the Deadlock Checker with FDR in a single environment:** Martin and Jassim [33] developed a tool called the Deadlock Checker. This is a very specific tool, designed for deadlock (and livelock) modular analysis. Thus, unlike FDR, the Deadlock Checker does not fit as an ideal tool for our purposes, when considered in isolation. However, having the Deadlock Checker as a particular function for FDR is very attractive. Therefore, we intend to integrate both tools in a single environment to get the best of each of them.
Appendix A

Elements of CSP\textsubscript{M}

In this appendix we present the main elements of the scripting language CSP\textsubscript{M}. For a deep view of this language, please refer to [13, 10].

A.1 Sets

\{1, 2, 3\} Set literal
\{m..n\} Closed ranges
\{m..\} Open ranges
\text{union}(a,b) Set union
\text{inter}(a,b) Set intersection
\text{diff}(a,b) Set difference
\text{Union}(A) Distributed union
\text{Inter}(A) Distributed intersection
\text{member}(x,a) x belongs to \(a\)
\text{card}(a) Cardinality
\text{empty}(a) Checks for emptyness
\text{set}(a) Convert a sequence to a set
\text{Set}(a) Powerset of a set
\text{Seq}(a) Set of sequences over a set (infinite if the set is not empty)
\{x_1,...,x_n \mid x \ll a, b\} Set comprehension
APPENDIX A. ELEMENTS OF CSP

A.2 Sequences

\(\langle\rangle\), \(\langle 1, 9\rangle\) 
Sequence literals

\(\langle m..n\rangle\) 
Closed ranges

\(\langle m..\rangle\) 
Open ranges

\(s^t\) 
Concatenation

\# s, or length(s) 
Tests whether \(s\) is empty

head(s) 
First element

tail(s) 
\(s = head(s) \cap tail(s)\)

concat(S) 
Distributive concatenation of sequences

elem(x,s) 
\(x \in \text{ran } s\)

\(\langle x_1, \ldots, x_n | x \leftarrow s, b\rangle\) 
Sequence comprehension

A.3 Booleans

true, false 
Boolean literals

b1 and b2 
Boolean and

b1 or b2 
Boolean or

not b 
Boolean negation

b1==b2, b1!=b2 
Equality operations

b1<b2, b1>b2, b1<=b2, b1>=b2 
Ordering operators

if b then e1 else e2 
Conditional operator

A.4 Extra

Extra

(4, \(\langle\rangle\), \{2\}) 
Tuples

let ... within ... 
Local definitions

\(\backslash x_1, \ldots, x_n \@ e\) 
Lambda definition

Int, Bool 
Simple types

nametype n=e 
Abbreviation

datatype n=e1 | ... | en 
Free-type
### A.5 Process Expressions

<table>
<thead>
<tr>
<th>CSP</th>
<th>CSP&lt;sub&gt;M&lt;/sub&gt;</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>STOP</td>
<td>Deadlock</td>
</tr>
<tr>
<td>SKIP</td>
<td>SKIP</td>
<td>Successfull termination</td>
</tr>
<tr>
<td>a → P</td>
<td>a → P</td>
<td>Simple prefix</td>
</tr>
<tr>
<td>a?x → P</td>
<td>a?x → P</td>
<td>Input prefix</td>
</tr>
<tr>
<td>a!v → P</td>
<td>a!v → P</td>
<td>Output prefix</td>
</tr>
<tr>
<td>P⧵Q</td>
<td>P ⊕ Q</td>
<td>External choice</td>
</tr>
<tr>
<td>P ⊓ Q</td>
<td>P</td>
<td>~</td>
</tr>
<tr>
<td>P\ A</td>
<td>P\A</td>
<td>Hiding</td>
</tr>
<tr>
<td>P ≦ b ▷ Q</td>
<td>if b then P else Q</td>
<td>Conditional choice</td>
</tr>
<tr>
<td>P ≦ b ▷ stop</td>
<td>b &amp; P</td>
<td>Boolean guard</td>
</tr>
<tr>
<td>P[</td>
<td>X</td>
<td>] Q</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P &gt;&gt; Q</td>
<td>P [c &lt;-&gt; c'] Q</td>
<td>Piping</td>
</tr>
<tr>
<td>P(s)</td>
<td>P(s)</td>
<td>Parametrisation</td>
</tr>
<tr>
<td>P(f(s))</td>
<td>let s'=f(s) within P(s')</td>
<td>Local declaration</td>
</tr>
<tr>
<td>P; Q</td>
<td>P;Q</td>
<td>Sequential composition</td>
</tr>
<tr>
<td>\i∈T P_i</td>
<td>[] i:T @ P(i)</td>
<td>Process indexing&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>P_i</td>
<td>P(i)</td>
<td>Parametrisation</td>
</tr>
</tbody>
</table>
Appendix B

SACI-1 in CSP_M

In this section we present the translation of the CSP_Z specification of the SACI-1 into FDR.

B.0.1 SCLOCK in CSP_M

The first process to be translated is the SCLOCK, whose specification was presented in the beginning of this section. In this translation the CLK given-set is declared as a free type (datatype) so that FDR can analyse the process.

datatype CLK = noneClock | clk1 | clk2 | clk3 | clk4 | clk5 | clk6

channel clockWDT,clockFTR: CLK
channel tic

SCLOCK =
let
    -- The Interface
    Channels = {|clockWDT,clockFTR|}
    lChannels = {|tic|}
    Interface = union(Channels, lChannels)
    -- The CSP Part
    main = sendClock [] updateClock
    sendClock = clockWDT?clk -> sendClock
    [] clockFTR?clk -> sendClock
    updateClock = tic -> main
    -- The Z Part
    -- IncC definition
    IncC(noneClock) = clk1
APPENDIX B. SACI-1 IN CSP

\[
\begin{align*}
\text{IncC(clk1)} & = \text{clk2} \\
\text{IncC(clk2)} & = \text{clk3} \\
\text{IncC(clk3)} & = \text{clk4} \\
\text{IncC(clk4)} & = \text{clk5} \\
\text{IncC(clk5)} & = \text{clk6} \\
\text{IncC(clk6)} & = \text{noneClock} \\
\text{State} & = \{ \text{time} | \text{time} \leftarrow \text{CLK} \} \\
\text{Init} & = \{ \text{noneClock} \} \\
\text{com(time, clockWDT.clk)} & = \text{if clk==time then } \{ \text{time} \} \text{ else } \{} \\
\text{com(time, clockFTR.clk)} & = \text{com(time, clockWDT.clk)} \\
\text{com(time, tic)} & = \{ \text{IncC(time)} \} \\
Z_{\text{CSP}} & = \\
\text{let} \\
& \quad \text{Z(State)} = \text{[] } (\text{States,Comm}): \\
& \quad \text{\{} \text{com(State, c), c } | \text{ c } \leftarrow \text{Interface} \} \\
& \quad \text{@ States}!={\} \& \\
& \quad \text{\{} \text{~| State'': States } @ \text{Comm } \rightarrow \text{Z(State')} \} \\
& \quad \text{within } \text{~| iState: Init } @ \text{Z(iState)} \\
& \quad \text{within (main []|\text{Interface}|] Z_{\text{CSP}}\backslash l\text{Channels}}
\end{align*}
\]

\textbf{B.0.2 Telecommand in CSP}_M

The next converted process is the Telecommand which uses preconditions to constrain the CSP behaviour.

\begin{align*}
\text{datatype Dest} & = \text{TCp | TMP} \\
\text{datatype Data} & = \text{sendTM | extra} \\
\text{nametype Counter} & = \{0..2\} \\
\text{nametype Message} & = \text{Dest.Data.Counter} \\
\text{channel TC_FTR: Message} \\
\text{channel waitEarth:Message} \\
\text{channel localTask, remoteTask} \\
\text{TC} & = \\
& \text{let} \\
& \quad \text{-- The Interface} \\
& \quad \text{Channels} = \{\text{TC_FTR, waitEarth}\} \\
& \quad \text{lChannels} = \{\text{localTask, remoteTask}\}
\end{align*}
Interface = union(Channels, lChannels)
-- The CSP Part
main = waitEarth?d.m.c -> (localTask -> main
  [] remoteTask -> TC_FTR?d.m.c -> main)

-- The Z Part
State = { (message, counter) | message <- Message,
           counter <- Counter }
Init = { (message', 0) | message' <- Message }
com((message, counter), TC_FTR.msg) =
  { (message', counter') | (message', counter') <- State,
    d <- Data, n <- Counter,
    message==(TMp.d.n),
    msg==(TMp.d.counter), counter'==0,
    message'=message }
com((message, counter), waitEarth.msg) =
  union(if counter<2 then
    { (message, counter+1) }
  else {},
  if counter==2 then
    { (message, 0) }
  else {})
com((message, counter), localTask) =
  if {d | d <- Data, n <- Counter,
    message==(TCp.d.n)}!={} then
    { (message, counter) }
  else {}
com((message, counter), remoteTask) =
  if {d | d <- Data, n <- Counter,
    message==(TMp.d.n)}!={} then
    { (message, counter) }
  else {}
Z_CSP =
  let
    Z(State) = [] (States,Comm) <- { (com(State, c), c) |
      c <- Interface } @ States != {} &
    |~| State': States @ Comm -> Z(State')
  within |~| iState: Init @ Z(iState)
  within (main []|Interface|) Z_CSP\lChannels
B.0.3 Telemetry in CSP\textsubscript{M}

The Telemetry process is the next, with a slightly more complex state-space, where is used a sequence.

```plaintext
datatype Dest = TCp | TMp
datatype Data = sendTM | extra
nametype Counter = {0..2}
nametype Message = Dest.Data.Counter

channel FTR_TM: Message
channel sendEarth: Message
channel emptyTM, moreTM, storeTM, sendSTM

TM =
  let
    -- The Interface
    Channels = {[FTR_TM, sendEarth, emptyTM, moreTM, storeTM, sendSTM]}
    lChannels = {}
    Interface = union(Channels, lChannels)
    -- The CSP Part
    main = FTR_TM?d.m.c -> (sendSTM -> SENDTM [] storeTM -> main)
    SENDTM = emptyTM -> main [] moreTM -> sendEarth?d.m.c -> SENDTM
    -- The Z Part
    maxTM = 1
    SEQM = FSeq(Message, maxTM)
    State = { (STM, actMsg) | STM <- SEQM, actMsg <- Message }
    Init = { (<>, actMsg’) | actMsg’ <- Message }
    com((STM, actMsg), FTR_TM.msg) = { (STM, msg) }
    com((STM, actMsg), sendSTM) =
      if {n | n <- Counter,
         (actMsg==(TMP.sendTM.n)) or
         (actMsg==(TCP.sendTM.n))}!={}
      then { (STM, actMsg) } else {}
    com((STM, actMsg), storeTM) =
      if {d | d <- Data, n <- Counter, d!=sendTM,
         ((actMsg==(TMP.d.n)) or
         (actMsg==(TCP.d.n)))}!={}
      then
```
union(if #STM<maxTM then
   { (STM^<actMsg>, actMsg) }
else {},
if #STM=maxTM then
   { (tail(STM)^<actMsg>, actMsg) }
else {})
com((STM, actMsg), emptyTM) = if STM==<> then
   { (STM, actMsg) }
else {}
com((STM, actMsg), moreTM) = if STM!=<> then
   { (STM, actMsg) }
else {}
com((STM, actMsg), sendEarth.msg) = if STM!=<> and msg==head(STM)
   then
      { (tail(STM), actMsg) }
   else {}

Z_CSP =
let
 Z(State) = [] (States,Comm):
   { (com(State, c), c) | c <- Interface }
 @ States!=={} &
   |~| State': States @ Comm -> Z(State')
within |~| iState: Init @ Z(iState)
within (main [] Interfaces] Z_CSP\1Channels

B.0.4 A_R in CSP_M

The simplest process of all is A_R. It is used to introduce asynchronism between the WDT and the FTR which can be implemented using the interleave operator.

channel recover, recover_ftr

A_R =
let
    -- The Interface
    Channels = { recover, recover_ftr }
 1Channels = {} 
 Interface = union(Channels, 1Channels)
    -- The CSP Part
**APPENDIX B. SACI-1 IN CSP**

\[
\begin{align*}
\text{main} & = \text{IntRequest} \ ||\ || \text{IntExecution} \\
\text{IntRequest} & = \text{recover} \rightarrow \text{IntRequest} \\
\text{IntExecution} & = \text{recover}_ftr \rightarrow \text{IntExecution} \\
\text{-- The Z Part} \\
\text{State} &= \{ \text{flag} | \text{flag} \leftarrow \text{Bool} \} \\
\text{Init} &= \{ \text{false} \} \\
\text{com}(\text{flag}, \text{recover}) &= \{ \text{true} \} \\
\text{com}(\text{flag}, \text{recover}_ftr) &= \text{if flag}==\text{true} \text{ then } \{ \text{false} \} \text{ else } \{} \\
\text{Z}_\text{CSP} &= \\
\text{let}
\quad \text{Z}(\text{State}) &= [] (\text{States,Comm}): \{ (\text{com}(\text{State}, c), c) | \\
\quad & c \leftarrow \text{Interface} \} \circ \text{States} \neq \{\} & \\
\quad \text{if} \text{State}' \neq \{\} & \text{&} \\
\quad \text{if} \text{State}' : \text{States} \circ \text{Comm} \rightarrow \text{Z}(\text{State}') \\
\quad \text{within} \text{if} \text{State}: \text{Init} \circ \text{Z}(\text{Init}) \\
\quad \text{within} \text{if} \text{main} [\{\text{Interface}\}] \text{Z}_\text{CSP}\text{lChannels}
\end{align*}
\]

**B.0.5 FTR in CSP**

The last process converted is the \textit{FTR}. Observe that the same implementation time out previously explained is adopted here for enabling \textit{failure} or \textit{fatal} events.

\[
\begin{align*}
\text{datatype} & \text{CLK} = \text{noneClock} | \text{clk1} | \text{clk2} | \text{clk3} | \text{clk4} | \text{clk5} | \text{clk6} \\
\text{datatype} & \text{Dest} = \text{nullDest} | \text{TCp} | \text{TMp} \\
\text{datatype} & \text{Data} = \text{nullData} | \text{sendTM} | \text{extra} \\
\text{nametype} & \text{Counter} = \{0..2\} \\
\text{nametype} & \text{Message} = \text{Dest}.\text{Data}.\text{Counter} \\
\text{channel} & \text{clockFTR}: \text{CLK} \\
\text{channel} & \text{TC}_\text{FTR}, \text{FTR}_\text{TM}: \text{Message} \\
\text{channel} & \text{failure}, \text{fatal}, \text{reset} \\
\text{channel} & \text{recover}_\text{ftr}, \text{task} \\
\text{FTR} &= \\
\text{let}
\quad \text{-- The Interface} \\
\quad \text{Channels} &= \{ \{ \text{clockFTR}, \text{reset}, \text{failure}, \text{recover}_\text{ftr}, \}
\quad & \text{TC}_\text{FTR}, \text{FTR}_\text{TM} \} \\
\quad \text{lChannels} &= \{ \{ \text{task}, \text{fatal} \} \} \\
\quad \text{Interface} &= \text{union}(\text{Channels}, \text{lChannels})
\end{align*}
\]
135

-- The CSP Part
main = Normal [> Problem
Normal = clockFTR?clk -> (reset -> Normal
[] task -> Normal
[] TC_FTR?d.m.c -> FTR_TM?d.m.c
-> Normal)

Problem = (failure -> recover_ftr -> main [] fatal -> STOP)
-- The Z Part
WDTRstP = {clk3, clk6}
WDTP(time, timeout) = member(time, timeout)
State = { (message, time) | message <- Message, time <- CLK }
Init = { (message’, noneClock) | message’ <- Message }
com((message, time), clockFTR.clk) = { (message, clk) }
com((message, time), reset) = if WDTP(time, WDTRstP) then
{ (message, time) } else {}
com((message, time), task) = if not WDTP(time, WDTRstP) then
{ (message, time) } else {}
com((message, time), TC_FTR.msg) = { (msg, time) }
com((message, time), FTR_TM.msg) = if msg==message then
{ (message, time) }
else {}
com((message, time), failure) = { (message, time) }
com((message, time), recover_ftr) = { (message, time) }
com((message, time), fatal) = { (message, time) }
Z_CSP =
let
Z(State) = [] (States,Comm): { (com(State, c), c) | c <- Interface } @ States != {} &
|~| State': States @ Comm -> Z(State')
within |~| iState: Init @ Z(iState)
within (main []Interface] Z_CSP)\Channels

B.0.6 SACI-1 in CSP

Now we have the top-level specification of the SACI-1’s kernel process, which is the parallel composition of four processes. Finally, we have our kernel process combined with the Telecommand and the Telemetry processes.

kernelSACI1 = ((WDT []{recover} A_R
[{{reset, recover_ftr}] FTR)
APPENDIX B. SACI-1 IN CSP

\[
\text{SACI}_1 = (\text{kernelSACI}_1 [\text{clockWDT, clockFTR}]) \text{SCLOCK} \\
\text{SACI}_1 = (\text{kernelSACI}_1 [\text{TC_FTR}]) \text{TC} [\text{FTR_TM}] \text{TM}
\]
Bibliography


Tese de Doutorado apresentada por Alexandre Cabral Mota a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “Verificação de Modelos para CSPz: Técnicas para Evitar a Explosão de Estados”, orientada pelo Prof. Augusto César Alves Sampaio e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Roberto Freire Cunha  
Departamento de Sistemas de Computação - CIn/UFPE

Prof. Ana Lúcia Caneca Cavalcanti  
Departamento de Ciência da Computação – CIn/UFPE

Prof. Carlos André Guimarães Ferraz  
Departamento de Informação e Sistemas – CIn/UFPE

Prof. Daltro José Nunes  
Departamento de Informática Aplicada / UFRGS

Prof. George Roger Ribeiro Justo  
University of Westminster

Visto e permitida a impressão.
Recife, 20 de dezembro de 2001.

Prof. AUGUSTO CESAR ALVES SAMPAIO  
Coordenadora da Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.