Federal University of Pernambuco
Informatics Centre

Master in Computer Science

Efficient and Mechanised Analysis of Infinite CSP_Z Specifications: strategy and tool support

Adalberto Cajueiro de Farias

Master Dissertation

Recife, 3 April 2003

This dissertation is dedicated to,

my parents, my brothers,
my nephews and niece
and God.

Federal University of Pernambuco
Informatics Centre

Efficient and Mechanised Analysis of Infinite CSP_Z Specifications: strategy and tool support

Adalberto Cajueiro de Farias

Dissertation submitted for the degree of Master

at the Federal University of Pernambuco, Brazil

Surpevisors: Augusto Sampaio and Alexandre Mota

Recife, 3 April 2003

Acknowledgments

The contributions received during these last two years have been essential to the conclusion of this work. I am sincerely thankful for the effort, advice and help from my supervisors, Augusto Sampaio and Alexandre Mota (the *ninja*). They always have given me attention, even when I asked for help in a not scheduled time.

I would like to thank Fernanda Moreira and Ana Cavalcanti for the implementation of the Z parser in Java. The original Z grammar was sent them by Ian Toyn, a member of Z Standards Panel. In addition, I would like to thank Phil Armstrong, a member of Formal Systems, for having sent me the original CSP parser used by FDR. That help were very important to our implementation.

I also thank Márcio and Adnan Sherif (my partner for coffee and tea) for many informal, but fruitful discussions about formal methods and problems found during our researches.

It is my duty to thank my father, Aldemar, and my mother, Fátima, for everything, specially when I decided to abandon a military career to study computer science. Their pride felt because my good military position is now bigger because they can see seven years of investment producing a professional more prepared to face less dangerous problems than fireman activities. For many times when I was working until late, my father looked at the screen and said: "What is that? I really do not know how you understand that!". I lost the notion about how many times my mother got angry with me. The funniest situation occurred when I forgot to have lunch and she said: "You cannot allow the machine dominate you! Stop! Take a rest! Have lunch!".

Finally, I would like to tank the Brazilian government agency CNPq for having financed this work.

Resumo

Na modelagem de sistemas concorrentes, o uso de diferentes linguagens formais tem sido uma alternativa muito utilizada nos últimos anos. Álgebras de processos (como CSP e CCS) são adequadas para modelar comportamento, enquanto que linguagens baseadas em modelos matemáticos (como Z e VDM) são mais adequadas para descrever aspectos de dados.

As linguagens integradas surgiram com o intuito de prover suporte para lidar com diferentes aspectos ao mesmo tempo. CSP_Z, por exemplo, é uma notação integrada que faz uso de CSP e Z para especificar comportamento e dados de forma ortogonal. Sua semântica foi definida em termos de CSP, o que tornou possível reusar FDR, o verificador de modelos padrão para CSP, na verificação de CSP_Z.

Embora essa estratégia elimine a necessidade de implementar um verificador de modelos específico para CSP_Z, ela não é suficiente para resolver um problema conhecido como *explosão de estados*. Tal problema ocorre porque a técnica de verificação de modelos analisa todos os possíveis estados do sistema. Caso essa quantidade seja infinita ou muito grande, a aplicação da técnica torna-se impossível ou impraticável.

Dentre as diversas técnicas usadas para lidar com essa limitação, abstração de dados concentra-se em tranformar um sistema infinito em um finito e equivalente, tal que o mesmo possa ser verificado por ferramentas. Para CSP_Z, a abstração de dados faz uso da teoria de *independencia de dados* de Lazić e *interpretação abstrata* dos Cousots. A primeira teoria é aplicada na parte de CSP para garantir que seu comportamento é o mesmo, independente do tipo de dado manipulado pela especificação. A segunda teoria fornece suporte para reescrever programas utilizando objetos de um domínio abstrato (mais simples) ao invés de utilizar objetos de um domínio concreto. A idéia geral da abstração de dados para CSP_Z é fazer um refinamento de dados na parte de Z sem afetar propriedades originais da especificação.

O trabalho de Mota, relacionado à abstração de dados para CSP_Z, concentra-se apenas na preservação de propriedades da parte de dados (Z) da especificação. Isso significa que algumas situações manifestadas pela parte de CSP não são capturadas pela abordagem e a busca pela abstração explora mais possibilidades que o necessário.

Procurando contribuir com a pesquisa sobre abstração de dados para CSP_Z, nosso trabalho apresenta uma estratégia mais geral que abstrai uma especificação considerando as partes de CSP e Z. A primeira contribuição diz respeito à classe de problemas sobre a qual a técnica pode ser aplicada, considerando que situações específicas da parte de CSP são capturadas. Uma outra contribuição é o substancial aumento de eficiência do algoritmo.

Em termos de ferramentas de suporte, nós também apresentamos uma implementação em Java para nossa abordagem. Embora com algumas limitações, nossa ferramenta é um trabalho pioneiro na área, visto que as outras abordagens estudadas durante esta pesquisa requerem a intervenção do usuário na construção da abstração, ou utilizam protótipos que não encorajam o uso prático.

Abstract

In concurrent systems modelling, the use of different formal languages has been an alternative very used in the last years. Process algebras (like CSP and CCS) are adequate to model behaviour, while languages based on mathematical models (like Z and VDM) are more suitable to describe data aspects.

Integrated languages have appeared in order to provide support to deal with different aspects at the same time. Many of them were proposed to specify concurrent systems. CSP_Z, for example, is an integrated notation which makes use of CSP and Z to specify behaviour and data orthogonally. Its semantics was defined in terms of CSP, what made possible to reuse FDR, the standard CSP model checker, in CSP_Z verification.

Although this strategy eliminates the necessity of implementing a model checker specific for CSP_Z, it is not sufficient to solve a problem known as *state explosion*. Such a problem occurs because the technique of model checking analyses all possible states of the system. If the state space is infinite or too large, the application of model checking becomes impossible or impractical.

Among several techniques used to deal with such a limitation, data abstraction concentrates on transforming an infinite system into an equivalent finite one, such that it can be verified by tools. For CSP_Z, data abstraction uses the *data independence* theory from Lazić and the *abstract interpretation* theory from the Cousots. The first theory is applied to the CSP part in order to guarantee that its behaviour is the same, independently of the data type manipulated by the specification. The second theory gives support to rewrite programs by using objects from an abstract domain (simpler) instead of using objects from a concrete domain. The general idea of CSP_Z data abstraction is achieving a data refinement in the Z part without affecting the original properties of the specification.

The work of Mota, related to data abstraction for CSP_Z, concentrates on preserving properties of the data (Z) part of the specification. This means that some situations manifested by the CSP part are not captured by the approach and the search for the abstraction explores more possibilities than necessary.

Focused on producing contributions for CSP_Z data abstraction, our work presents a more general strategy which abstracts a specification by considering the CSP and the Z parts. The first contribution concerns the class of problems to which it can be applied, considering that specific situations of the CSP part are captured. Another contribution is a substantial increase of efficiency of the algorithm.

In terms of support tools, we also present an implementation in Java for our approach. Although with some limitations, our tool is a pioneer work on the area, since the other approaches studied during this research require user assistance when building the abstraction, or use prototypes that do not encourage their practical use.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since software development has been progressively changing from art to science, several techniques are emerging to enrich software processes as much as possible. The research area of Software Engineering has naturally appeared to study methodologies for the best practices of software. Several software processes (and, more specifically, life cycle models) have been proposed to guide the development. However, the increasing and dynamic complexity of systems raises more and more challenges to Software Engineering professionals. Methods and techniques have been constantly improved in order to supply ways of solving new problems.

To deal with the production of large-scale software, there is a consensus among the many development processes: several phases are proposed in order to enhance the quality of the final product [79]. Depending on the kind of software being developed, more or less phases can be employed. Critical systems (or safety-related systems), for example, require a formal guarantee of consistency when evolving from one phase to the next, that is, the artefact in the following phase must satisfy the properties of the current one. The use of formalisms in those kinds of systems is justified by the serious consequences caused to human life when such systems do not work properly (or correctly).

Some development models—known as Formal Development Processes—define a phase called *specification.* In such a phase, the software is described by a *formal specification*—an unambiguous and abstract description, where some of its properties can be analysed, studied and guaranteed even before its implementation [17]. In this scenario, Formal Methods represent an important research area to supply methodologies and techniques to deal with such an approach. Although the use of formalisms still finds many obstacles, support tools have been very promising to diffuse Formal Methods, in the sense that their application becomes easier.

In a formal development process, a formal notation is used to build an abstract description of the system. Afterwards, techniques and tools are used in order to derive more concrete (or implementable) versions or prove desirable properties. Notations used to describe systems must provide a concise way of writing programs, such that their aspects are precisely and abstractly characterised.

Considering the Rational Unified Process [68], a widely adopted current practice of

software development, formalisms have been employed in many ways. For example, to guarantee refinement between UML [39] diagrams, to generate test cases based on the pre- and postconditions of the operations, to annotate UML objects with a formal language, etc.

The increasing complexity of systems has raised new questions about what aspects have to be dealt formally. Regarding concurrent systems, there are many aspects to be taken into account—behaviour, data, time, mobility, probability, etc. Current notations do not present enough expressiveness to deal with distinct aspects simultaneously. Instead, integrations of different formalisms, known as the research area of linking theories and tools, have been used as an effective alternative to deal with concurrent systems aspects modelling.

The idea behind linking theories consists in building a new theory as a formal combination of other existing and widely accepted theories. The result of such an integration is a more expressive theory with capabilities to model a larger class of problems. Furthermore, the reuse of the syntaxes and semantics of the constituent languages is a natural advantage and the reuse of existing tools represents an important motivation.

1.1 Computer-aided Verification

Software analysis is becoming more complex than developing the software itself. Its goal consists of comparing two documents—a specification (*Spec*) and an implementation (*Impl*) or a property to be satisfied—in order to guarantee that *Impl* must really work as specified in *Spec* [76]. Apart from many techniques, model checking and theorem proving are the two main approaches used in computer-aided verification.

1.1.1 Model Checking

Model checking is an automatic technique used to determine properties of a system by exploring their state spaces [64]. Formally speaking, it consists of checking the satisfaction relation $M \models p$, which states that M is a model for p, a logical formula.

Although this technique is completely automatic, it works only for well-delimited finite state systems, that is, their state spaces can only grow to an amount supportable by the current hardware technology. In the last few years, model checking has demonstrated to be very useful in practice, and applicable to real world problems from the Industry [11]. Several model checkers (like FDR [36], SPIN [38] and SMV [55]) are available to aid software verification.

1.1.2 Theorem Proving

Theorem proving is another technique employed to determine properties of a specification. Unlike model checking, this technique can be applied to specifications with infinite state spaces.

The price paid for using theorem proving is that, depending on the logic the system makes use and the deduction rules used during the proof, user assistance can be required. That is, in general, theorem proving is an interactive and laborious approach.

Like model checking, theorem proving has also demonstrated to be very useful in practice. Several examples employing support tools (like Z-Eves [60], PVS [63] or ACL2 [59]) are reported in the literature [12].

1.2 Concurrent Systems

Concurrency is a very common characteristic in systems today. Apart from many aspects presented by concurrent systems, behaviour and data are the most essential among them and can be viewed orthogonally. The existing notations to model behaviour have expressiveness to describe control flow (order of operations and interactions between processes). However, the expressiveness presented by those formalisms are not adequate to model data aspects. They work with primitive and concrete structures and, therefore, do not have support to model data structures abstractly. Examples of formal languages used to model behaviour are CSP [25, 13] and CCS [77]. The theory of Petri Nets [86] is also useful to specify behavioural aspects of a system.

On the other hand, formalisms like Z [62] and VDM [44] are suitable to describe data aspects because they are based on models. Mathematical objects (like sets, relations, functions, etc) are used to specify data structures. However, concurrency and behavioural aspects are not described by using these formalisms because they do not present enough expressiveness.

The integration of formalisms has been recently employed to deal with behaviour and data aspects separately. While process algebras are used to describe control flow, model-based languages are used to specify data aspects (data structures, state space and operations). There are many options of formal integrations in this category:

- CSP_Z [20, 22] – a combination of CSP and Z.
- CSP_{OZ} [21, 23] – a combination of CSP and Object-Z [74].
- ZCCS [5] – an integration of Z and CCS.
- RAISE [83] – a combination of VDM, ACT ONE [41] and OBJ [46] with the process algebras CSP and CCS.
- MOSCA [85] – an integration of CCS and VDM
- LOTOS [45] – a combination of CCS and ACT ONE.
- Circus [51] – a language based on Z, CSP and The Unifying Theory [26].

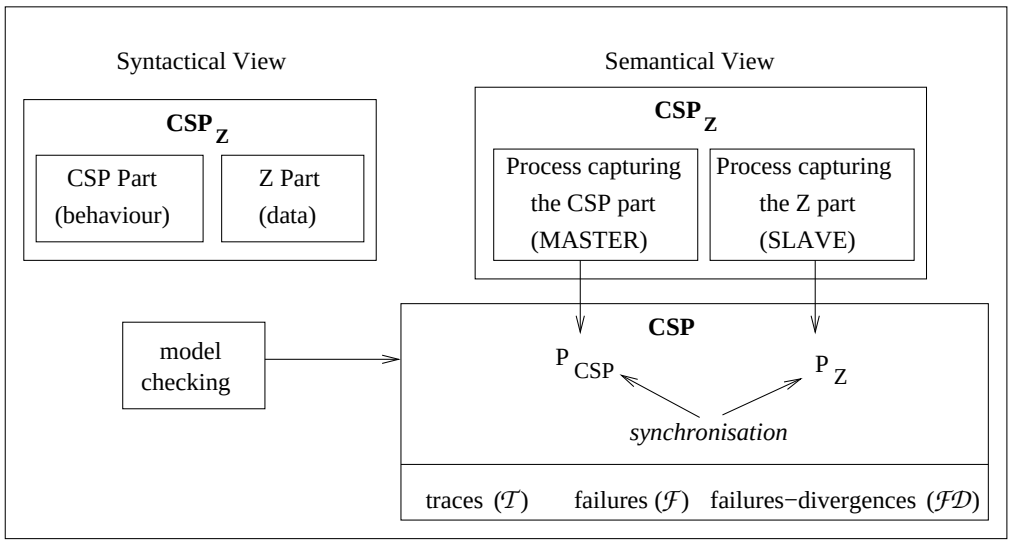


Figure 1.1: View of CSP_Z

Integrated languages are an elegant way of dealing with complex concurrent systems. The natural understanding of behaviour and data as orthogonal aspects permits to break the problem into two smaller and complementary ones.

We have chosen CSP_Z due to our experience in CSP and Z, and because the normal form proposed by Mota [8] for specifications written in that language has proved very promising to deal with verification, analysis and data refinement.

Figure 1.1 illustrates two views of CSP_Z. In the syntactical view, a specification is composed by two distinct parts: CSP (behavioural) and Z (data). The operational semantics of CSP_Z (semantical view) is described by the CSP semantics itself. Two processes—P_{CSP} and P_Z—capture the CSP and the Z parts respectively. They are completely synchronised, so that the CSP part acts as a master (controller) process and the Z one acts as a slave process, achieving data manipulation. Through one of the three CSP models (*traces*, *failures* or *failures-divergences*), the original specification can be analysed by model checking.

1.2.1 Verifying Properties in CSP_Z

Basing the semantics in that of CSP, the model checking of CSP_Z is achieved by describing a specification as a synchronisation of two smaller CSP processes, such that existing tools for CSP can also analyse CSP_Z. This approach was deeply investigated by Mota and Sampaio [8] who defined a strategy to convert a CSP_Z specification into an equivalent CSP process. An interesting case study can be found in [6] where a subset of an on-board computer of an artificial Brazilian micro-satellite was specified in CSP_Z and then verified using FDR [36].

The idea of model checking CSP_Z specifications using only FDR is not enough to deal with all classes of problems, especially those concerning infinite systems (due to the *state*

explosion problem). To overcome this limitation, several techniques have been proposed to be used before applying model checking: abstraction [7, 42, 28, 30], elimination of symmetry [61], binary decision diagrams [54], partial order methods [69], data independence [76], local analysis [48], integration of model checking with theorem proving [7, 87, 82], etc. In this work, we are interested in investigating data abstraction, a kind of abstraction which allows one to transform an infinite system into a finite one, while still preserving most of its properties.

In this challenging search for a technique to treat infinite CSP specifications, Lazić [76] proposed a theory called *data independence*, which deals with refinement checking between infinite, but data independent, processes. Broadly, a data independent system is one that works similarly, independently of the data type manipulated by it. This imposes that a data independent system must use operations which work with any data type (polymorphic operations). Lazić showed that refinement checking between data independent processes can be made by considering a sufficient subset of the original data type manipulated by them. This strategy has been successfully applied to some practical examples; however, it is not able to deal with processes which are data dependent.

The work of Wehrheim [42] has investigated the same problem for CSP_{OZ} specifications; however, her work consists on applying (manually) data abstraction on the Z part and it does not mention any constraint upon the CSP part (in terms of data independence). This means that her technique can be applied to CSP_{OZ} processes, although the behavioural part presents data dependence aspects. In fact, it is not valid because if the behavioural part has some data dependence, then it must be considered when applying the data abstraction technique [7].

Mota [7, 9] used the theory of Lazić and the results obtained by Wehrheim to investigate an automatic way of abstracting CSP_Z specifications. Before model checking an infinite CSP_Z process, his strategy applies the data independence theory to guarantee that the CSP part will not be affected when abstracting the data type manipulated by the whole process, and the abstract interpretation theory to the Z part to build a new one (by abstracting values and operations), such that the combination of the CSP part with the new Z part preserves the properties of the original process (see Figure 1.2). This abstraction technique consists of replacing an infinite data type with a finite one (an inverse refinement), so that model checking can be successfully applied to the resulting CSP_Z process. The strategy is based on the discovery of infinite and stable behaviour of the data part, that is, the possibility of executing infinitely the same sequence of operations. It is worth commenting that, if the process does not present an infinite and stable behaviour, the algorithm is nonterminating. Furthermore, the algorithm does not captures the effects of the CSP part over the Z one.

Although the automatic strategy proposed by Mota [9] represents an important step towards tool support development for data abstraction, it abstracts an original process by considering only the data (Z) part. Besides exploring more possibilities than necessary, his approach does not capture some situations manifested by the behavioural (CSP) part.

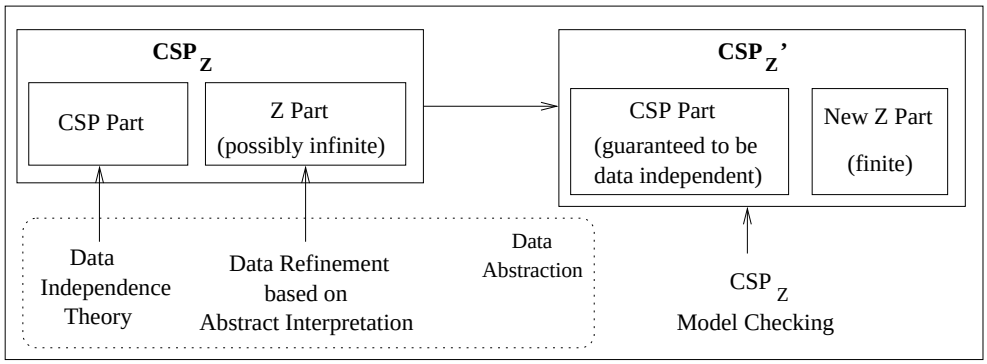


Figure 1.2: Data Abstraction for CSP_Z

1.3 Scope of this Work

This work investigates an extension to the data abstraction technique proposed in [7, 9] by also considering the behavioural part when abstracting a specification. Now, the idea of considering a CSP_Z specification as a compound process (see Figure 1.1) is strongly observed. As the CSP part plays a major role, establishing the control flow, one does not have to explore all possible states permitted by the Z part.

Our approach has a series of advantages: not all possibilities need to be explored, but only those ones allowed by the behavioural part; our algorithm to find a CSP_Z data abstraction is at least as efficient as (and often more efficient than) that one proposed by Mota [7, 9] and often converges more quickly; our data abstraction approach is obtained more easily because we use concrete data provided by the expansion of the CSP_Z process; we also capture special aspects of the behavioural part (successful termination, deadlock and livelock). It is worth pointing out that our algorithm behaves the same as that of Mota only when the CSP part is a recursive external choice of all the events (operations) of the system; in such case, the CSP part does not improve any restriction on the Z one, but this is very unlikely to happen.

In the model checking viewpoint, our approach represents the necessary states of a system instead of considering the whole state space. Moreover, we also present a robust and flexible implementation in Java which applies our formal approach through a friendly graphical user interface. The tool interacts with the Z-Eves [60] theorem prover in order to determine properties which cannot be dealt by model checking. In this context, we also make use of model checking and theorem proving in an integrated approach.

1.4 Organisation of this Work

This work is organised as follows. Chapter 2 introduces the CSP_Z language by explaining both CSP and Z separately. The syntax of CSP_Z used in this work is a subset of the original one [20, 22]. The normal form of a CSP_Z process is also given in order to reinforce the adequacy of a process to be verified by model checking, through the application of

a consolidated conversion strategy [8]. Furthermore, such a normal form facilitates the application of a guided data abstraction technique.

In Chapter 3 we present our main contribution: an extension to Mota's mechanised strategy for CSP_Z data abstraction. First, we give a view of data independence and abstract interpretation for CSP_Z. Corollaries 3.1 and 3.2 introduce the notions of safe and optimal abstractions for CSP_Z processes. Then, we present Theorem 3.1, which formalises the advantage of our approach over Mota's one [7, 9]. Afterwards, we present an algorithm which mechanises our idea, several examples of its application, and the limitations of our approach.

The tool implementing our strategy is presented in Chapter 4. In the same chapter, there is a section dedicated to Design Patterns [33], which improves the structure of implementations in the object-oriented paradigm. After that, we present the screen and dialogs of the tool as well as details about configuration and interaction with other tools (model checkers and theorem provers).

Finally, in Chapter 5 we present the benefits obtained with this work and some topics for future research, concerning the extension of the technique itself and improvements to the tool.

We also provide an appendix with useful definitions, proofs and extra information used by this work.

Chapter 2

CSP_Z Notation

The application of Formal Methods in software development includes the use of formal notations for modelling systems in an abstract manner. This abstract model is known as *formal specification*, from which important properties can be extracted through the application of techniques and tools. For concurrent systems, two aspects need to be considered: behaviour and data. Several notations have been developed in order to supply expressiveness to model control flow and data aspects. Process algebras such as CSP [25, 13] and CCS [77] are useful to model behavioural aspects, while languages like Z [62] and VDM [44] are adequate to model data structures. Most integrated notations combine a process algebra and a model-based language. Some examples of integrated languages are ZCCS [5] (an integration of Z and CCS), RAISE [83] (combination of VDM, ACT ONE [41] and OBJ [46] with the process algebras CSP and CCS), MOSCA [85] (integration of CCS and VDM), LOTOS [45] (a combination of CCS and ACT ONE), CSP_{OZ} [21] (a combination of CSP and Object-Z [74]), Circus [51] (a language based on Z, CSP and The Unifying Theory of Programming [26]) and CSP_Z [20, 23] (a combination of Z and CSP).

The CSP_Z language makes use of Z to model data types, state and operations in a very abstract and natural manner, because Z is based on set theory and first order logic. The basic structures of Z are *schemas*—abstract descriptions containing declarations, pre- and postconditions. Although Z presents expressiveness to model operations as mathematical entities (schemas), it is not suitable to model order of execution or interaction among them. Unlike Z, CSP allows us to describe interactions between operations in an easy manner. The notion of *process* and *communication* makes the language suitable to describe behavioural aspects and synchronisation. Processes can interact among themselves in many different ways in order to produce new processes.

Combining these complementary features of CSP and Z, CSP_Z was defined as a conservative[1] integration of them, such that CSP is used to model process interaction and Z is used to manipulate data structures, state and operations.

Syntactically, a CSP_Z specification treats such aspects—process and data—into two separate and complementary parts which give a modular structure. The semantics of the

[1] In the sense that most of their structures are preserved.

language [20, 21] has been defined in terms of CSP. The data part is viewed as a process which synchronises with the behavioural one. Based on that, the behaviour of a CSP_Z specification was defined in terms of a *labelled transition system* (LTS), a set of nodes and transitions between them. This representation permits to apply the technique of model checking after deriving a CSP_M (the machine-readable version of CSP accepted by the tool FDR [36]) process from a CSP_Z specification [8]. Therefore, CSP_Z can be also analysed using the model checker FDR.

This chapter is organised as follows. Section 2.1 gives an overview of CSP, including its most important operators and the notion of models and refinement between processes. Section 2.2 presents the Z language and introduces the notion of refinement in Z. Section 2.3 presents a reduced version of the CSP_Z abstract syntax and an example of a specification. Section 2.4 shows the strategy proposed by Mota and Sampaio [8] for converting a CSP_Z specification into CSP_M. Moreover, we also reproduce a theorem from [7] stating that the translation preserves the semantics and then present the normal form of a CSP_Z process. Such a reduced form is important to explain the behaviour of a CSP_Z process (Section 2.5) in terms of a *labelled transition system.* Finally, Section 2.6 presents our conclusions about this chapter.

2.1 CSP

According to Roscoe [13], CSP was designed to be a notation and a theory for describing and analysing systems whose primary interest arises from ways in which different components interact at the level of communication. Each component is treated as a process and a communication is an event which processes must agree on. A communication can be viewed as a transaction or synchronisation between two or more processes rather than as necessarily being the transmission of data one way.

Communication events happen in a *channel,* which represents a *wire* between processes. During a synchronisation, values can be communicated from one process to another through channels.

Based on communication and event concepts, several constructs are defined. Here, we only show the main structures of CSP (see [13, 36] for a complete description).

2.1.1 Channels

Channels are declared in order to define events. Example:

$$channel\ a, b$$
$$channel\ c : \{1, 2, 3\}$$

Channels a and b do not support data types; they represent events. On the other hand, channel c accepts a data type whose values come from the set $\{1, 2, 3\}$. Therefore, three distinct events can be captured from c: $c.1$, $c.2$ and $c.3$. Input and output communications are characterised by the symbols ? and !, respectively, as for example in $c?x$ and in $c!y$.

2.1.2 Processes

A process is a behavioural description of a computation (in terms of events). For any process P, αP denotes its *alphabet*, the set of events which can be performed by P. The set of all events in CSP is called Σ (Sigma).

Basic Processes

Some processes are considered elementary (not built from others). *STOP* denotes a process which cannot perform any event (it represents a *deadlock*). On the other hand, *SKIP* denotes a process which terminates successfully.

Prefixing

The prefix operator is the simplest way of creating a process. For example,

$$P = a \rightarrow b \rightarrow STOP$$

defines the process P which accepts the event a, performs it and behaves like $b \rightarrow STOP$. Before reaching *STOP*, P can perform the sequences of events (traces): $\langle\rangle, \langle a\rangle$ or $\langle a, b\rangle$. When data types are considered, input and output events can be performed as in

$$Q = a?x : \{1, 2\} \rightarrow b!x \rightarrow STOP,$$

where Q is a process which accepts an input value put into a, outputs it into b and then deadlocks. The alphabet of Q is the set $\{a.1, a.2, b.1, b.2\}$. Along its execution, Q can perform the following traces: $\langle\rangle$, $\langle a.1\rangle$, $\langle a.2\rangle$, $\langle a.1, b.1\rangle$ and $\langle a.2, b.2\rangle$. It is worth noting that values communicated on channels can augment substantially the events performed by a process. For example, $a?x : \{1\}$ represents one event, whereas $a?x : Int$ is an infinite set of events because *Int* represents the integers in CSP.

Input and output events can be combined into a multiprefix construct as in

$$P = a?x!f(x)?y!d \rightarrow Q$$

where P is a process which receives the input values x and y on the channel a and sends outputs $f(x)$ and d through the same channel. Afterwards, it behaves like Q.

External Choice

The external choice operator represents a choice between two processes which is made based on the environment synchronisation. For example, the process

$$a \rightarrow STOP \,\Box\, b \rightarrow c \rightarrow STOP$$

offers a and b as initial events. Depending on the interaction with other processes (the environment), it performs a or b and then behaves accordingly. Thus, it can exhibit the following traces: $\langle\rangle, \langle a\rangle, \langle b\rangle$ or $\langle b, c\rangle$.

Internal Choice

This operator represents a nondeterministic choice. For example, the process

$$P = a \rightarrow STOP \sqcap b \rightarrow c \rightarrow STOP$$

decides to engage on a or b without considering any interaction. Therefore, an agreement on a between P and another process can raise a deadlock because P can reject it. The choice is made as an internal action, called τ-transition, in which one can make an analogy that the process plays a coin and decides the following behaviour by itself.

Sequential Composition

This process combinator allows one to express an idea of sequential execution in the sense that one process is executed until it terminates and only then another one is executed. The process $P;Q$ runs P until it terminates and then runs Q. If P never terminates or deadlocks then Q will never be executed. For example, $a \rightarrow STOP;\ b \rightarrow SKIP$ never performs b, whereas $SKIP;\ b \rightarrow SKIP$ always does it.

Conditional Choice

Conditional choice is analogous to the conditional statements existing in most of programming languages; it is represented by $P \lhd b \rhd Q$. First of all, the boolean condition b is evaluated. If it is true then the process behaves like P else it behaves like Q.

Conditional choice can also be represented by a boolean guard. For example, $b \ \& \ P$ is the same as $P \lhd b \rhd STOP$.

Alphabetised Parallelism

When processes are placed in parallel, they can interact by performing events in common. The alphabetised parallelism is a binary operator which allows to establish these interactions by giving the synchronisation alphabets of both processes. For example, in

$$P = a \rightarrow b \rightarrow SKIP$$
$$Q = b \rightarrow c \rightarrow SKIP$$
$$R = P \ {}_{\{a,b\}}||_{\{b,c\}} \ Q$$

$\{a, b\}$ and $\{b, c\}$ are the synchronisation alphabets of P and Q, respectively. The synchronisation occurs on events from the intersection $\{a, b\} \cap \{b, c\}$, that is $\{b\}$. The events outside this intersection can be performed independently by the processes whose synchronisation alphabet contains them. For example, the event a is performed by P and the event c is performed by Q, independently.

Interleaving

With this operator, processes run completely independent of each other, that is, $P \mid\mid\mid Q$ does not have any agreement. For example, $a \rightarrow SKIP \mid\mid\mid a \rightarrow b \rightarrow SKIP$ has two possibilities of behaviour:

- $a \rightarrow (SKIP \mid\mid\mid a \rightarrow b \rightarrow SKIP)$ - the left process performs the event a.
- $a \rightarrow (a \rightarrow SKIP \mid\mid\mid b \rightarrow SKIP)$ - the right process performs the event a.

The interleave operator is very adequate for specifying processes that will be executed by distinct resources.

Generalised Parallelism

The generalised parallel operator is a simple and general manner of placing processes into parallel. Unlike the alphabetised parallelism operator, we only give the *synchronisation interface*—a set containing all events which the component processes must synchronise on. Events outside the interface can be performed independently, depending on the alphabets of the component processes. For example, the process

$$a \rightarrow b \rightarrow SKIP \underset{\{b\}}{\mid\mid} c \rightarrow b \rightarrow SKIP$$

produces an agreement only on b. The events a and c are performed, independently, by the left- and the right-hand processes, respectively.

It is worth mentioning that the interleave and the alphabetised parallel operators can be expressed by using generalised parallelism:

- $P \mid\mid\mid Q = P \underset{\{\}}{\mid\mid} Q$
- $P \; {}_X\!\mid\mid\!{}_Y \; Q = P \underset{X \cap Y}{\mid\mid} Q$

We also observe that, in the original version of CSP presented by Hoare [25], parallelism is represented by the operator $\mid\mid$. In $P \mid\mid Q$, the synchronisation depends on the alphabets of P and Q. In the CSP version of Roscoe [13], the same operator means full synchronisation and it can be expressed as $\underset{\Sigma}{\mid\mid}$ or as ${}_\Sigma\!\mid\mid\!{}_\Sigma$.

Hiding

Sometimes a process performs internal events and does not require any agreement on it. If such events are left visible to the environment, other processes can synchronise on them, causing interference, and thus producing undesirable behaviour or side effects (for example, deadlock). The CSP hiding operator allows to hide events from the environment such that no process can synchronise on them. For example, the process

$$(a \rightarrow b \rightarrow c \rightarrow SKIP) \backslash \{b, c\}$$

can perform the events b and c independently of the environment. Therefore, for the environment, it is similar to $a \rightarrow SKIP$.

Renaming

Sometimes, CSP processes can have a similar syntactical structure, differing only by the event names. A simple renaming of events would be enough to build a new process. For example, $b \rightarrow STOP$ can be built from $a \rightarrow STOP$ by simply renaming a to b,

$$b \rightarrow STOP = (a \rightarrow STOP)[\![b/a]\!]$$

where $[\![b/a]\!]$ denotes the substitution of the event a for the new event b.

Recursion

In CSP, recursion can be expressed either by the special operator μ or by making a process to reference itself or another recursive process. Let Q, Q' and P be CSP processes. By the following definitions, Q and Q' are equivalent, although their syntactical definitions are different.

$$Q = e \rightarrow P$$
$$P = a \rightarrow b \rightarrow Q$$
$$Q' = \mu X.(e \rightarrow a \rightarrow b \rightarrow X)$$

The semantics of μ consists of unfolding an expression when the binded variable is reached along the same expression, that is, the execution of $\mu X.F(X)$ causes $F(X)$ to execute and replace all occurrences of X with $\mu X.F(X)$.

Some important processes

There are some non-basic processes which represent standard and useful properties. They are:

$$RUN(A) = ?x : A \rightarrow RUN(A)$$
$$Chaos(A) = STOP \sqcap (?x : A \rightarrow Chaos(A))$$
$$\mathbf{div} = (\mu X.(?e : \Sigma \rightarrow X))\backslash\Sigma$$

The RUN process represents the most non-deterministic deadlock-free CSP process. The $Chaos$ process is similar to RUN except for breaking at any moment. The **div** process requires special attention: it performs any event from Σ and then hides this action from the environment. Therefore, although the process is computing something, no one can see what is happening within it. This situation is known as *livelock* and it is an undesirable behaviour of a process.

2.1.3 Initials and Afters

At each step of its execution, a process can offer a set of events. These events are called *initials* and they are defined for each CSP construct. Appendix A.1 contains the definition of *initials* of a CSP process and its application to the main constructs. The following examples give a brief idea about the *initials* of a process:

- $initials(STOP) = \{\}$;
- $initials(a \rightarrow P) = \{a\}$;
- $initials(a \rightarrow P \,\Box\, b \rightarrow Q) = initials(a \rightarrow P \sqcap b \rightarrow Q) = \{a, b\}$

In the third equation, although there is an internal decision, both a and b can be initially performed by the processes.

To consider the behaviour of a process after a given trace, we use the *afters* operator ($/$), which gives the behaviour of a process after performing a trace. For example,

$(c \rightarrow STOP \,\Box\, a \rightarrow b \rightarrow STOP)/\langle a \rangle$

produces $b \rightarrow STOP$.

2.1.4 Models

According to Roscoe [13], CSP can be viewed as a notation to describe concurrent systems as well as a collection of mathematical models and reasoning methods useful to study processes which interact with each other by communication. This section gives a brief view of the three CSP models: *traces* ($\mathcal{T}$), *failures* ($\mathcal{F}$) and *failures-divergences* ($\mathcal{FD}$). They define the *essence* of specifications in CSP.

Traces

The most common way of understanding a process is looking at the sequences of events (traces) performed by it. For each process P, $traces(P)$ is the set containing all (possible) sequences of events performed by P. For example:

- $traces(STOP) = \{\langle\rangle\}$. The empty trace is the only way of representing a process which cannot perform any event;
- $traces(a \rightarrow b \rightarrow STOP) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$. This process may have communicated nothing yet, performed only a, or a and then b;
- $traces(a \rightarrow STOP \,\Box\, b \rightarrow STOP) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$. The process initially offers a or b;
- $traces(\mu X.(a \rightarrow X)) = \{\langle a\rangle^n \mid n \in \mathbb{N}\}$. This process can perform as many a's as its environment likes.

Some operations are defined over traces:

- Concatenation ($s \frown t$). The usual sequence concatenation. Example: $\langle a\rangle \frown \langle b, c\rangle$ produces $\langle a, b, c\rangle$;
- Exponentiation (s^n). Means n-fold concatenations: $s^0 = \langle\rangle$ and $s^{n+1} = s^n \frown s$;

- Prefix relation ($s \leq t$). s is prefix of t if there is a sequence (possibly empty) w such that $t = s \frown w$.

For any process P, $traces(P)$ will always have the following properties:

- $traces(P)$ is non-empty: it always contains the empty trace $\langle\rangle$;
- $traces(P)$ is prefix-closed: if $s \frown t$ is a trace performed by P then at some earlier time, the trace s was also performed by P.

The set of all possible representations of a process using traces is called the *traces* model ($\mathcal{T}$), the simplest CSP model which gives information about the visible events of a process.

Failures

Sometimes it is not sufficient analysing processes by looking at its traces because they do not give a complete description of its behaviour. Sometimes, processes can reject events and this subtle feature raises an important difference in terms of what a process *can* do and *must* do. For example, the processes $(a \rightarrow b \rightarrow STOP) \sqcap STOP$ and $a \rightarrow b \rightarrow STOP$ have the same traces, even though the second is allowed to do nothing at all, no matter what we offer to it. Indeed, the second one decides whether to offer a or to reject anything. Therefore, they are completely different in terms of refusals.

For all processes P, $refusals(P)$ is defined to be the set of events P cannot engage in. Under this viewpoint, the *failures* model ($\mathcal{F}$) permits to analyse processes according to its failures, a set of pairs (s,X) where $s \in traces(P)$ and $X \in refusals(P/s)$. The set $failures(P)$ can be calculated from the *failures diagram*—a directed graph where a node denotes the refusals (at that point) and an edge denotes a performed event. Figure 2.1 shows such diagrams for $a \rightarrow b \rightarrow STOP$ and $(a \rightarrow b \rightarrow STOP) \sqcap STOP$, respectively.

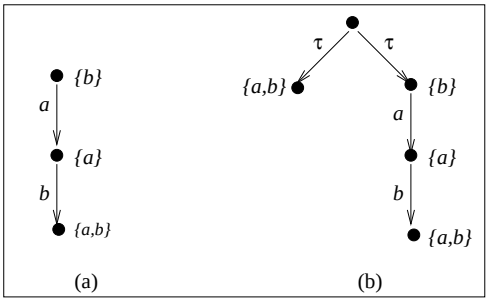


Figure 2.1: Failures diagram

Calculating their failures from the above figure we obtain:

$failures(a \rightarrow b \rightarrow STOP) = \{(\langle\rangle, \{b\}), (\langle a\rangle, \{a\}), (\langle a, b\rangle, \{a, b\})\}$

$failures((a \rightarrow b \rightarrow STOP) \sqcap STOP) = \{(\langle\rangle, \{a, b\}), (\langle\rangle, \{b\}), (\langle a\rangle, \{a\}), (\langle a, b\rangle, \{a, b\})\}$

Although they perform the same traces ($\{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$), the second process has more failures than the first one. This shows that the first process is as good as the second one in terms of traces, and better in terms of failures (it has less failures). Moreover, it is worth pointing out that, as presented by Roscoe [13], the failures of a process are calculated for each CSP construct, in an analogous way to the traces calculation. The failures diagram is presented only to give a graphical view of the failures.

Failures-Divergences

Although the *failures* model allows one to analyse processes in terms of failures, it is not sufficient to analyse invisible actions. Sometimes a process can execute infinitely many actions without showing any progress to the environment—*divergence*. A divergent process neither does anything useful nor refuses anything; it is similar to an infinite loop whose action is doing nothing.

The simplest example of a divergent process is $\mu X.(a \rightarrow X) \backslash \{a\}$. The only really satisfactory way of dealing with divergence is to record the set of traces on which a process can diverge. Once a process diverges, we assume it can perform any trace, refuse anything, and diverge on any latter trace. Thus, $divergences(P)$ is defined to be the set of P's divergences and contains not only the traces s on which P can diverge, but also all extensions $s \frown t$ of such traces. For example, consider the processes $P = a \rightarrow \mathbf{div} \Box b \rightarrow STOP$ and $Q = a \rightarrow \mathbf{div} \Box b \rightarrow \mathbf{div}$. Calculating their failures and divergences we obtain:

- $failures(P) = \{(\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$
- $failures(Q) = \{(\langle a\rangle, \{a, b\}), (\langle b\rangle, \{a, b\})\}$
- $divergences(P) = \{\langle a\rangle\}$
- $divergences(Q) = \{\langle a\rangle, \langle b\rangle\}$

Note that, although Q and P have the same failures, Q diverges more than P.

In the *failures-divergences* model ($\mathcal{FD}$)—the standard CSP model—a process is denoted by a tuple ($failures_\perp$, $divergences(P)$) where $failures_\perp$ is an extension of $failures(P)$ in the sense that it works with divergences:

$$failures_\perp(P) = failures(P) \cup \{(s, X) \mid s \in divergences(P)\}.$$

In fact, when a process is diverging, it can be refusing everything. Therefore, X can be any event from Σ.

2.1.5 Refinement

The notion of refinement between CSP processes depends on the model under which they are analysed. In the *traces* model, the relation $P \sqsubseteq_{\mathcal{T}} Q$ means "Q does not perform any trace different from P". In the *failures* model, $P \sqsubseteq_{\mathcal{F}} Q$ means "Q fails less than P".

Finally, in the *failures-divergences* model, $P \sqsubseteq_{\mathcal{FD}} Q$ means "Q fails and diverges less than P". These meanings are better expressed by the following definitions.

Definition 2.1 (Traces Refinement)

$$P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow traces(P) \supseteq traces(Q) \qquad \diamondsuit$$

Definition 2.2 (Failures Refinement)

$$P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow failures(P) \supseteq failures(Q) \qquad \diamondsuit$$

Definition 2.3 (Failures-Divergences Refinement)

$$P \sqsubseteq_{\mathcal{FD}} Q \Leftrightarrow failures_{\perp}(P) \supseteq failures_{\perp}(Q) \wedge divergences(P) \supseteq divergences(Q) \qquad \diamondsuit$$

The notion of *determinism* is based on the $\mathcal{FD}$ model: a process P is defined to be deterministic if and only if $divergences(P) = \{\}$ and $s \frown \langle a \rangle \in traces(P) \Rightarrow (s, \{a\}) \notin failures(P)$. In other words, it cannot diverge, and never has the choice of both accepting and refusing any action. Furthermore, the notion of equivalence between processes is derived from refinement as follows:

$$P \equiv_M Q \Leftrightarrow P \sqsubseteq_M Q \wedge Q \sqsubseteq_M P,$$

where M is one of the CSP models $\mathcal{T}, \mathcal{F}$ or $\mathcal{FD}$.

2.2 Z

The Z notation is based on set theory and first-order logic, allowing one to describe abstract data types and operations on these types. The language is suitable to model the following aspects:

1. Static aspects:
   - state-space;
   - invariant (a constraint preserved during all the system execution).
2. Dynamic aspects:
   - operations;
   - relationships between their inputs and outputs;
   - state changes.

In general, a Z specification is composed by paragraphs and has the following structure: data types and definitions, state, initialisation and operations. As the notation has many structures, we give a simplified presentation (see [62] for details).

2.2.1 Types

The most common way of defining a type in Z is using *Given Set*s, *Free Types*, and *Abbreviations*. A *Given Set* introduces a new type without worrying about its internal representation. For example, $[EVEN]$ defines a new type $EVEN$ which can be used along the specification. An *Abbreviation* introduces a new name for a type which has been previously defined. For example, $T == EVEN$ defines a type T as being the same as $EVEN$. *Free Types* are useful to describe union of values and recursive structures. For example:

$$ANSWER ::= \text{SUCCESS} \mid \text{ERROR}$$
$$TLIST ::= nil \mid cons\langle\langle T \times TLIST \rangle\rangle$$
$$TTREE ::= nil \mid tree\langle\langle T \times TTREE \times TTREE \rangle\rangle$$

The type $ANSWER$ has two possible values: SUCCESS or ERROR. $TLIST$ is a recursive definition of a list whose elements have the type T. $TTREE$ is a recursive definition for a tree of elements whose type is also T.

2.2.2 Definitions

Usually, definitions include generic functions and axiom descriptions used along the specification.

Axiomatic Definitions

An axiomatic definition usually includes an object and a constraint upon it. For example:

$$EVEN : \mathbb{P}\,\mathbb{Z}$$
$$\forall z : \mathbb{Z} \bullet z \in EVEN \Leftrightarrow z \bmod 2 = 0$$

Generic Definitions

A generic definition is a generic form of an axiomatic definition in which generic types can be parameters. It is useful to define generic operations, as in the definition of set inclusion below, which is polymorphic regarding the element type, represented by X.

$$[X]$$
$$\_ \subseteq \_ : \mathbb{P}\,X \leftrightarrow \mathbb{P}\,X$$
$$\forall s, t : \mathbb{P}\,X \bullet s \subseteq t \Leftrightarrow \forall x : X \bullet x \in s \Rightarrow x \in t$$

State and Initialisation

The system state and its initialisation are characterised by schemas, which are organisational structures of Z and include declarations and a predicate. For example:

State

$e : \mathbb{P}\ T$

$\#e \leq 10$

Init

$State'$

$e' = \varnothing$

The schema *State* defines the system state as a structure composed by components (variables). In the above example, only one component (e) was declared. There is also a constraint (*invariant*) to be guaranteed initially as well as when any state change occurs: the cardinality of e must be less than or equal to 10.

The schema *Init* defines initial values for the state components. In the example, it consists of initialising e with the empty set.

Schemas can also be described in a horizontal style:

$State \mathrel{\widehat=} [e : \mathbb{P}\ T \mid \#e \leq 10]$

$Init \mathrel{\widehat=} [State' \mid e' = \varnothing]$

2.2.3 Operations

Specifying operations in Z consists of describing them by using schemas, such that the declarations include information about state change (Δ or Ξ), input ($x?$) and output ($y!$) variables, and the predicate contains pre- and postconditions. For example:

insert

$\Delta State$

$x? : T$

$e' = e \cup \{x?\}$

The operation *insert* changes the state ($\Delta State$) and receives an input parameter ($x? : T$). Its (implicit) precondition is $\#e \leq 10$ (which could have been made explicit, although redundant: consequence of the state invariant) and its postcondition simply adds the input value to e. Variables decorated with $'$ denotes its value after the operation is executed. The Δ operator corresponds to the following expansion:

$$\begin{aligned} \Delta State &= [State;\ State'] \\ &= [e, e' : \mathbb{P}\ T \mid \#e \leq 10 \land \#e' \leq 10] \end{aligned}$$

On the other hand, operator Ξ corresponds to the an expansion which does not change the state:

$$\begin{aligned} \Xi State &= [State;\ State' \mid State' = State] \\ &= [e, e' : \mathbb{P}\ T \mid \#e \leq 10 \land \#e' \leq 10 \land e' = e] \end{aligned}$$

Now applying the notion of Δ to the *insert* schema, we obtain its expanded version:

insert

$e, e' : \mathbb{P}\, T$
$x? : T$

$\#e \leq 10 \land$
$\#e' \leq 10 \land$
$e' = e \cup \{x?\}$

The operation *remove* receives and element to be removed from e. Its precondition consists of checking whether the input value belongs to e or not. If so, it is removed from e, otherwise the value of e' is arbitrary.

remove

$\Delta State$
$x? : T$

$x? \in e$
$e' = e \setminus \{x?\}$

In addition, schemas can be combined in order to produce more powerful operations. For example, let OpA and OpB be two operation schemas. The composition $OpA \fatsemi OpB$ means that OpA is executed first and then OpB is executed. This sequential execution imposes that the state produced by the execution of OpA is used as initial state for OpB, and this intermediate state is hidden:

$$OpA \fatsemi OpB = \exists\, State_i \bullet OpA[State_i/State'] \land OpB[State_i/State].$$

2.2.4 Z Refinement

There are two types of refinement in Z: operation (algorithmic) and data. The former consists of making operations more applicable and deterministic, whereas the latter consists of replacing abstract data types with concrete ones. In this work we only give a brief explanation about such refinements. See [52, 62] for a complete explanation and real world examples.

Operation Refinement

Usually, operations defined by schemas are not complete in the sense that they are not enabled (applicable) for all input values and states. For example, recall that the operation *remove* presented in previous section can be executed only in cases where $x? \in e$. The algorithmic refinement consists of increasing the applicability of an operation, such that it can be executed in more cases (by weakening its precondition) or making it more deterministic, producing more predictable results (by strengthening its postcondition). For example, let

us define the operations *remove_error* and *remove_complete*, such that *remove_error* can be applied in cases where *remove* cannot, and *remove_complete* is defined in terms of *remove* and *remove_error*.

remove_error

$$\Xi State$$
$$x? : T$$
$$msg! : ANSWER$$

$$x? \notin e$$
$$msg! = ERROR$$

$$remove\_complete \mathrel{\widehat=} remove \lor remove\_error$$

The operation *remove_complete* can be executed in more cases than *remove* by acting as *remove_error*. Therefore, *remove_complete* was obtained by an operation refinement of *remove*.

Data Refinement

The idea behind data refinement is replacing abstract data structures by concrete ones which are closer to an implementation language.

When applying data refinement to a Z specification, state, initialisation and operations must be affected. The following schemas show an example: on the left, we have the simple specification we have concentrated in previous examples, where the schemas deal with an abstract data structure (set), while on the right the structure was replaced with a more concrete one (sequence).

State

$$e : \mathbb{P}\,\mathbb{N}$$

$$\#e \leq 10$$

CState

$$s : \mathrm{seq}\,\mathbb{N}$$

$$\#s \leq 10$$

Init

$$State'$$

$$e' = \varnothing$$

CInit

$$CState'$$

$$s' = \langle\,\rangle$$

insert

$$\Delta State$$
$$x? : \mathbb{N}$$

$$e' = e \cup \{x?\}$$

c_insert

$$\Delta CState$$
$$x? : \mathbb{N}$$

$$x? \notin ran\ s$$
$$s' = s \frown \langle x? \rangle$$

Note that all definitions were rewritten to the new type (sequence) which is easily implemented by using arrays, an actual structure supported by most programming languages.

The schema *retrieve* is a relation between the two state schemas and establishes the correspondence between them. In this case, as sequences are modelled as a mapping from $\mathbb{N} \setminus \{0\}$ to any type, the set e must be equal to the range of the corresponding sequence (that is, $e = ran\, s$).

retrieve

$State$
$CState$

$e = ran\ s$

Using the Z notation, types, state, initialisation and operations can be easily modelled as mathematical objects. However, the constructs of Z are not adequate to model order (or interaction) between operations. Therefore, the idea of integrating languages is a potential solution to this limitation because different aspects are modelled by using distinct notations: a process algebra and a model-based language. The following sections show an integration between Z and CSP which has this purpose.

2.3 CSP_Z Grammar

The CSP_Z syntax is relatively complex because all structures of CSP and Z are considered. Therefore, instead of using the original version of the grammar given by Fischer [20, 23], we deal with a simplified version which is accepted by the tool presented in [1, 2]. Keywords are in **bold** font. Square brackets denote optionality and * means 0 or more occurrences of a term.

A CSP_Z specification is limited by the keywords **spec** and **end_spec**. It has an identifier (ProcId), a synchronisation interface (Interface) and two distinct parts (CSPart and ZPart):

Specification ::= **spec** ProcId Interface CspPart ZPart **end_spec** ProcId

The specification identifier (ProcId) denotes a process name; it can be just a name or a name with parameters:

ProcId ::= Identifier | Identifier **(** Parameters **)**

The interface (Interface) corresponds to a set of events and includes channels and local channels. The difference between them is the visibility: channels are visible by another process, whereas local channels are not.

Interface ::= Channel* | LocalChannel*
Channel ::= **chan** ListIdentifier [: ChannelType]
LocalChannel ::= **lchan** ListIdentifier [: ChannelType]

ListIdentifier is a list of identifiers, separated by comma, and ChannelType has the same structure as the declaration part of a Z schema:

ChannelType ::= [Identifier : Type (; Identifier : Type)*]

The type of an identifier (Type) can be an identifier or a type defined as in Z (e.g. $\mathbb{P}\,\mathbb{N}$, seq T).

On the other hand, the CSP part (CspPart) contains paragraphs which are process definitions; it must include a special process, named *main*, representing the CSP part as a whole.

CspPart ::= CspParagraph*
CspParagraph ::= ProcIdent = Process

Because there are many CSP operators, the language is rather flexible to build processes. Table 2.1 shows the main forms of building processes, including an example written in CSP_M, the machine-readable version of CSP accepted by the tool FDR [36]. Figure 2.2 shows the CSP_Z grammar as a whole.

CSP	CSP_M	**Name**
Process ||| Process	`P ||| Q`	Interleaving
Process □ Process	`P [] Q`	External Choice
Process ⊓ Process	`P |~| Q`	Internal Choice
Process || Process	`P || Q`	Synchronous Parallelism
Process $\|\|_X$ Process	`P [X||Y] Q`	Alphabetized Parallelism
Process $\|\|_X$ Process	`P [|X|] Q`	Generalized Parallelism
Process ; Process	`P ; Q`	Sequential Composition
Event → Process	`a -> Q`	Prefixing
STOP	`STOP`	Deadlock
SKIP	`SKIP`	Successful Termination
Chaos(Expr)	`CHAOS(A)`	Chaos Process
div	`div`	Divergent Process
Process ◁ Cond ▷ Process	`if b then P else Q`	Conditional Choice
Cond & Process	`b & P`	Boolean Guard
P(f(s))	`let s'=f(s) within P(s')`	Local Declaration

Table 2.1: CSP process definitions

The Z part includes a list of paragraphs (ZPart ::= ZParagraph*) defining data types, state, initialisation and operations. The Z structures considered here have already been introduced in Section 2.2: given sets, abbreviations, free types, etc. Refer to [62, 52] for a complete description of Z.

Looking at the grammar, one can see that a CSP_Z specification has the general form:

spec Name
Interface; CSP part ; Z part
end_spec Name

Specification	::=	**spec** ProcId Interface CspPart ZPart **end_spec** ProcId
ProcId	::=	Identifier | Identifier **(** Parameters **)**
Interface	::=	Channel* LocalChannel*
Channel	::=	**chan** ListIdentifier [: ChannelType]
LocalChannel	::=	**lchan** ListIdentifier [: ChannelType]
ChannelType	::=	**[** Identifier : Type (; Identifier : Type)* **]**
CspPart	::=	CspParagraph*
CspParagraph	::=	ProcIdent = Process
Process	::=	Process ∣∣∣ Process | Process □ Process
	|	Process ⊓ Process | Process ∥ Process
	|	Process $_X\|\|_Y$ Process | Process $\overset{\|\|}{_X}$ Process
	|	Process ; Process | Event → Process
	|	STOP | SKIP | Chaos(Expr)
	|	Process ⩤ Cond ⩥ Process | Cond & Process
ZPart	::=	ZParagraph*
ZParagraph	::=	GivenSet | FreeTypes | Abbreviation
	|	AxiomaticDefinition | GenericAxiomaticDescription
	|	SchemaDefinition | GenericSchemaDefinition

Figure 2.2: CSP_Z grammar

It is worth pointing out that, for each CSP event, there exists one schema whose name is composed by the *com_* prefix and the respective event name. This standard creates a strict association between a schema execution and its corresponding event performance: while the CSP part performs an event e, the Z one executes the schema *com_e*.

In the following, we present a specification of a simple clock which performs two events infinitely: *tick* and *tack*. An internal counter is incremented when any of these events is performed. This example has also been used by Wehrheim [42], except that here, all preconditions are true, that is, the Z part is ready to execute *com_tick* and *com_tack* all the time.

Example 2.1 (A simple clock)

spec Clock

chan tick, tack	(CSP part: channels)
$main = tick \rightarrow tack \rightarrow main$	(CSP part: main process)
$State \mathrel{\widehat=} [n : \mathbb{N}]$	(Z part: state declaration)
$Init \mathrel{\widehat=} [State' \mid n' = 0]$	(Z part: initialisation)
$com\_tick \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$	(Z part: operation)
$com\_tack \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$	(Z part: operation)

end_spec Clock

2.4 CSP_Z Model Checking

The principle of model checking is automatically verifying if a given formula p is satisfied on a specific finite domain M. In other words, the problem of model checking is determining the satisfaction relation $M \models p$, which states that M is a model for p. In terms of CSP, this checking can be achieved by refinement [14]: $(M \models p) \Leftrightarrow (S_P \sqsubseteq S_M)$, 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 the semantics of CSP_Z is based on the standard semantic model of CSP, model checking CSP_Z is an extension of model checking of CSP. The language had its model checking well studied by Mota and Sampaio [8] who defined a strategy to convert a CSP_Z specification into an equivalent CSP_M process in order to be able to use the FDR tool [36] to analyse CSP_Z as well. Naturally, the translation raised some questions about embedding the Z language into CSP:

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 and the Z parts of a CSP_Z specification?

These questions have been carefully discussed in [8]. In this work, we give only a brief explanation about the conversion in a step-by-step manner and apply the strategy to Example 2.1. Moreover, we present a theorem given by Mota [7] stating that the generated CSP_M is equivalent to the original CSP_Z specification.

The first task is converting the synchronisation interface: CSP_Z channels produce CSP_M channels. Therefore, *chan tick,tack* produces

```
channel tick, tack.
```

If the CSP_Z specification includes local channels, they are also introduced as CSP channels, but are hidden in the top level CSP process translated.

All process definitions from the CSP part are rewritten to CSP_M. This is achieved by a one-to-one syntactic transformation (see Table 2.1).

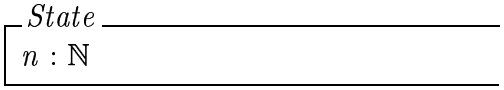
In the adopted example, $main = tick \rightarrow tack \rightarrow main$ produces

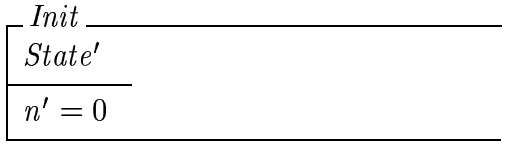
```
main = tick -> tack -> main.
```

The Z part conversion takes into account types, state-space, initialisation and operations. Although there is no type definition in the example, it is worth commenting that usually a Z type cannot be translated into a CSP_M structure automatically; the target language (CSP_M), unlike Z, does not offer, for example, maps, relations and bags as pre-defined abstract types, and, therefore, many steps of data refinement might be required. These conversions are, nevertheless, standard, and discussed, for example, in [65].

As we have already seen, the state is defined by a schema whose declarations denote its components and the predicate establishes a condition over the state itself (the invariant). Hence, the state is represented by a set comprehension including a tuple of variables representing the state components, and a predicate establishing the constraints over them.

Applying this transformation to the example, we have both state and initialisation translated into two set comprehensions as follows:



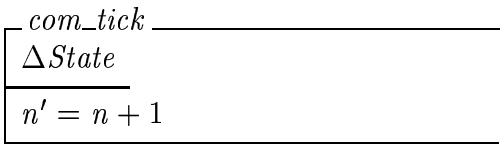


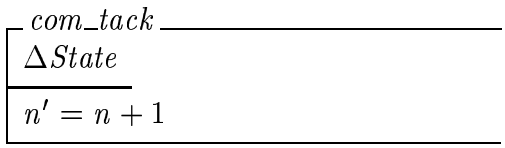
produces, respectively:

```
State = { n | n <- Int, n >= 0} and
Init = { n' | n' <- Int, n' == 0 }
```

The CSP_M type `Int` represents all integers. As the state component is a natural number, the set comprehension defining the state has a new constraint which imposes that `n` must be a non-negative integer (`n <- Int` and `n >= 0`).

In terms of operations, the idea is converting schemas into functions that possibly change the state. We must observe, however, that a Z operation actually defines a relation between input variables (and before states) and output variables (and after states). Therefore, its translation into a function requires that, for each state and input value, the output be a set of possible after states and output values. In order to include pre- and postconditions, such functions are defined by using set comprehension. For instance, recall from Example 2.1 the two operations, *com_tick* and *com_tack*:





Their translations produce, respectively:

```
com(n, tick) = {n' | n' <- State, n' == n + 1 } and
com(n, tack) = {n' | n' <- State, n' == n + 1 }
```

The strategy used to represent the Z part as a process consists of combining all *com_* functions into an external choice in a recursive way, such that all enabled operations are offered all the time. When the process engages into one of the events, the state is updated accordingly (the corresponding schema is executed). Furthermore, the next value `State'` is chosen internally among the set `States` of all possible values.

```
Z(State) = ([] (States,Comm) :  { (com(State,c),c) | c <- Interface } @
               States != {} & |~| State':States @ Comm -> Z(State'))
           []terminate -> SKIP
```

Before starting the above process, we have to initialise the state:

```
Z_PART = let
   Z(State) = as defined before
within |~| iState:  Init @ Z(iState)
```

Once defined a CSP_M process for the CSP and the Z parts, we have to synchronise them. The CSP part only performs events, whereas the Z one (possibly) changes the state. It is worth recalling that when an event is performed by the CSP part, its corresponding schema *com_* is executed by the Z one. This establishes a complete agreement between them which is expressed by the following generalised parallelism: `main [|Interface|] Z_PART`.

Now, converting the entire example we obtain:

```
channel tick, tack

Clock = let

   --The Interface
   Channels = {|tick,tack|}
   lChannels = {}
   Interface = union(Channels,lChannels)

   -- The CSP part
   main = tick -> tack -> main

   -- The Z part
   com(n, tick) = {n' | n' <- State, n' == n + 1 }
   com(n, tack) = {n' | n' <- State, n' == n + 1 }

   Z_PART = let
     Z(State) = ([](States,Comm) :  { (com(State,c),c) | c <- Interface } @
                    States != {} & |~| State':  States @ Comm -> Z(State'))
                []terminate -> SKIP
   within |~| iState:  Init @ Z(iState)
within (main [|Interface|] Z_PART)\lChannels
```

Note that local channels (`lChannels`) are hidden at the end. Therefore, events happening on `lChannels` are visible only within the scope of the `Clock` process definition.

After translation, a CSP_Z process is viewed as a synchronisation of two smaller processes, such that one captures the Z part (P_Z) and another captures the CSP one (P_{CSP}). Hence, a CSP_Z process has a standard form which is established by the following definition.

Definition 2.4 (Normal Form of CSP_Z Processes) *Let P_{CSP_Z} be a CSP_Z specification and P_{CSP_M} be the corresponding CSP_M process resulting from the translation. Let*

P_Z and P_{CSP} be processes which capture the Z part and the CSP one, respectively, after translation. Then, P_{CSP_M} can be viewed as a synchronisation between two smaller processes:

$$P_{CSP_M} = (P_Z \underset{I}{||} P_{CSP}) \setminus L,$$

where $I = \alpha P_Z = \alpha P_{CSP}$ (I is the synchronisation interface), L is the set of local channels declared on the original specification ($L \subseteq I$), and P_Z is a parameterised process which has the form:

$$P_Z(State) = \left(\begin{array}{l} \quad pre\ com\_chan_1\ \&\ chan_1 \rightarrow\ P_Z(com\_chan_1(State)) \\ \Box\ pre\ com\_chan_2\ \&\ chan_2 \rightarrow\ P_Z(com\_chan_2(State)) \\ \ldots \\ \Box\ pre\ com\_chan_n\ \&\ chan_n \rightarrow\ P_Z(com\_chan_n(State)) \\ \Box\ terminate \rightarrow SKIP \end{array} \right)$$

♢

It is worth noting that the processd resulting from the translation of a CSP_Z one can be viewed as a black box containing a synchronisation between two inner processes (see Figure 2.3). In the rest of this work, we assume that CSP_Z processes are in this normal form.

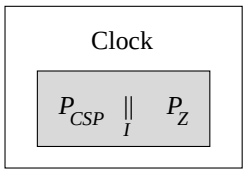


Figure 2.3: Clock process as a black box

Furthermore, the normal form also deals with termination. The Z part can terminate by behaving like $terminate \rightarrow SKIP$, and the translation of the CSP part replaces all occurrences of $SKIP$ with $terminate \rightarrow SKIP$. This causes the synchronisation of both parts upon termination as required.

The translation makes the CSP_Z model checking possible because the generated CSP_M process is equivalent to the original specification. Such an equivalence is according to the CSP denotational semantics ($[\![.]\!]^{\mathcal{D}}$) defined by Scattergood [16]. The following theorem was established by Mota [7] and states the equivalence between a CSP_Z specification and the CSP_M process resulting from the translation strategy defined in [8].

Theorem 2.1 (CSP_Z Translation) *Let P be the CSP_Z specification*

spec P
I; main*; State; Init; Z*
end_spec P

provided that P_{CSP} captures the CSP part and P_Z captures the Z part. If P' is the CSP_M process resulting from the translation approach, then $[\![P]\!]^{\mathcal{D}} = [\![P']\!]^{\mathcal{D}}$ ♢

It is worth pointing out that the equivalence between a CSP_Z process and a CSP_M one is defined in terms of CSP. Fischer [23] has established the semantics of CSP_Z in terms of CSP through the function $[\![.]\!]^{\mathcal{D}}$. Similarly, Scattergood [16] has established the semantics of a CSP_M process through the function $[\![.]\!]^{\mathcal{D}}$ and the correspondence between a CSP and a CSP_M representations, for a same process. As the semantics of CSP_Z and CSP_M are defined in the same model, processes written in those languages can be compared.

2.5 CSP_Z Behaviour

Roscoe [13] described the *CSP* semantics by using different formalisms: operational, denotational and algebraic semantics. The operational approach was presented by using *labelled transition system* (LTS) and traditional rules of operational semantics, which permits to look at them as a logical inference system.

Recall that the model checking of CSP_Z was defined in terms of CSP. Therefore, the behaviour of CSP_Z can be explained in terms of CSP. In this section, we give an overview of the blocking view of CSP_Z and its behaviour in terms of transition systems—a graphic representation where the progress of a process is easily represented.

In CSP, a LTS is a set of nodes and, for each event a in some set, a relation $\xrightarrow{a}$ between the nodes. It is a directed graph with a label on each edge representing what happens when we take the action which the edge represents. Figure 2.4 illustrates some examples of LTS for CSP. The process *STOP* stands for a canonical deadlock and it is represented by a black node (a). The process *SKIP* is a grey node and it leads the system to an end state Ω (b) by performing the special event $\surd$. The process $a \rightarrow b \rightarrow c \rightarrow STOP$ is represented by a linear LTS (c). The LTS for the external choice $a \rightarrow STOP \,\Box\, b \rightarrow c \rightarrow STOP$ presents two possibilities of behaviour (d), whereas the internal choice $a \rightarrow STOP \sqcap b \rightarrow c \rightarrow STOP$ performs an internal action (τ) and then decides the following behaviour (e). In the conditional choice ($a \rightarrow STOP \lhd b \rhd d \rightarrow e \rightarrow STOP$), the following behaviour is determined by a condition evaluation (indicated between square brackets).

2.5.1 The Blocking View of CSP_Z

A CSP_Z specification is defined as a parallel combination of its CSP and its Z parts via the interface, such that if an event *ev* occurs on the CSP part, then its related schema *com_ev* is executed [20, 23]. Recall from Section 2.2.3 that, when the precondition of a Z schema is false, then the execution of such a schema produces an unpredictable result. As the semantics of CSP_Z is based on the standard model of CSP, we should explain what happens when an event *ev* can be performed by the CSP part of a CSP_Z process, but its related schema is disabled (its precondition is false).

In the *blocking* view of CSP_Z, a schema is executed by the Z part if and only if its precondition is true. This is obtained by using the precondition as a guard of an operation [40], that is, $pre\ com\_chan\ \&\ chan \rightarrow P_Z(com\_chan(S))$. Note that, if *pre com_chan* is false, then this expression is similar to *STOP*. Figure 2.5 illustrates the blocking view

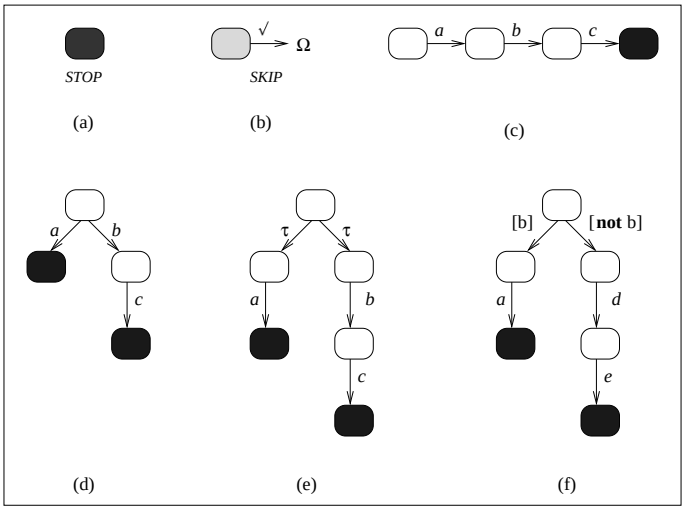


Figure 2.4: LTSs for CSP

of CSP_Z considering one operation (com_a). Although the CSP part offers the event a, it is not performed because *pre* com_a is false. Therefore, the whole process behaves like deadlock and the state remains unaltered.

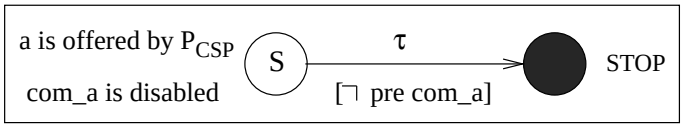


Figure 2.5: The blocking view of CSP_Z

2.5.2 LTS for CSP_Z

Representing the parallelism of both parts of a CSP_Z specification, the transitions of a CSP_Z LTS have two arrows [8]: a filled one denoting an event performance, and a dotted one indicating the corresponding schema execution. The information about state is kept in each node of the LTS. Figure 2.6 shows the LTS for a single transition in CSP_Z and Figure 2.7 shows the LTS for the specification presented by Example 2.1 (the *Clock* process).

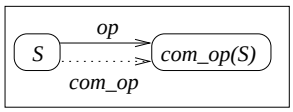


Figure 2.6: LTS for a CSP_Z transition

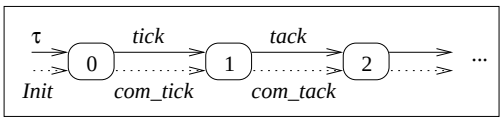


Figure 2.7: LTS for the infinite clock

At initialisation, the CSP part performs a τ-transition, whereas the Z part executes the *Init* schema. Afterwards, both parts synchronise their executions.

The acceptances of the whole process can be calculated by taking the events from the CSP part whose corresponding schemas are enabled. This is formalised in the following lemma, whose proof is in Appendix B.

Lemma 2.1 (Initials of a CSP_Z Process) *Let P_{CSP_Z} be a CSP_Z process whose synchronisation interface is I. Let P_{CSP} be the process representing the CSP part of P_{CSP_Z}. Then,*

$$initials(P_{CSP_Z}) = \bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i \wedge a_i \in initials(P_{CSP})\}. \qquad \diamondsuit$$

2.6 Conclusions

Linking theories and tools has been an effective effort for improving expressiveness of concurrent system modelling. Most integrated languages use a process algebra to describe behavioural aspects and a model-based or algebraic language to describe data aspects. In this chapter we presented CSP_Z [20, 23], a combination of CSP [13] and Z [62], which reuses the expressivenesses of both notations to give support for describing both aspects of concurrent systems: behavioural and data.

We also discussed the models under which CSP processes are analysed: $traces(\mathcal{T})$, $failures(\mathcal{F})$ and $failures\text{-}divergences(\mathcal{FD})$, as well as the definitions of refinements under these three models (Section 2.1.5).

This chapter has considered the specification language CSP_Z both from the analysis and the verification viewpoints. From the analysis viewpoint, we have briefly presented its syntax and behaviour in terms of labelled transition systems. From the verification viewpoint, we presented its model checking as an extension of the model checking of CSP, following the translation approach presented by Mota and Sampaio [8]. Moreover, the interesting normal form (Definition 2.4) generated after translation is useful for abstracting CSP_Z processes, as well as for using FDR [36] in CSP_Z model checking.

To formalise the idea of the acceptances of a CSP_Z process, we also presented a lemma (Lemma 2.1) stating that the acceptances of the whole process depend on the acceptances of both parts (CSP and Z). This complete agreement between those parts is a result of the blocking view of CSP_Z.

The information given in this chapter is essential for understanding next chapter, which deals with model checking of infinite CSP_Z processes.

Chapter 3

CSP_Z Data Abstraction

Although model checking has advanced substantially in the last decade, some systems cannot be automatically analysed due to their large size—the *state explosion* problem. In an attempt to overcome such a problem, several techniques of compression have been proposed to be used before applying model checking [64, 13]: local analysis, symmetry elimination, data independence, partial order methods, abstract interpretation, integration of model checkers with theorem provers, etc. These techniques reduce the number of states while still preserving most of the systems properties. The combination of theorem proving and model checking [87] is another approach which presents effective results in concurrent systems verification. Recall from Section 2.4 that model checking consists of finding a model M satisfying a given formula p. Moreover, the technique can be applied to finite-state systems without any user assistance. On the other hand, theorem proving is suitable for data-dominated verification where the state spaces can be large or unbounded [82].

Another recent trend is avoiding the state explosion problem by combining individual techniques. Cleaveland and Riely [73] presented a framework for generating *abstractions* of communicating systems based on the abstractions of the values exchanged by processes. Sifakis *et al* [28] presented a technique where the notion of abstraction is generally defined in terms of variants of simulation [78] and bisimulation [77]. The work of Wehrheim [42] followed an approach similar to [73] and [28] to deal with CSP_{OZ}; Mota [9, 7] extended and mechanised Wehrheim's approach for CSP_Z by applying the work of Lazić [76] on the CSP part, and data abstraction on the Z one. Furthermore, Mota used subtype abstraction[1] which makes the process of rewriting predicates (post-conditions of schemas) relatively less complex than the other approaches. Broadly, his approach uses the notion of stability (the Z part can infinitely execute a specific sequence of operations) to rewrite the operations, such that the stable sequence no longer causes infinite expansions. When considering only the data (Z) part, the algorithm analyses all possibilities, including those ones rejected by the CSP part, which acts as a controller process.

This chapter further explores data abstraction for CSP_Z. Our work extends Mota's approach by also considering the CSP part of a CSP_Z specification. This has resulted in

[1] The abstract domain is a subset of the original one.

improvements in the search for abstractions because our algorithm successfully terminates at least as often as Mota's one, and possibly more often, as the CSP part reduces the possible observable behaviour of the entire CSP_Z process. Our algorithm only analyses possibilities allowed by the CSP part. Naturally, our approach requires a more elaborate execution model to compute the abstractions; however, when using our approach, some abstractions do not have to be generated.

Moreover, some situations of the specifications, which might not be perceived by Mota's algorithm, are captured by our approach: deadlock, termination and divergence. In particular, deadlocks in CSP_Z specifications can happen when the CSP part really deadlocks, when it rejects the events offered by the Z part or when it offers events whose corresponding schemas are disabled.

The CSP_Z process resulting from the application of our data abstraction approach presents the same properties of the original process, in the failures-divergences model (see Corollary 3.2). Therefore, any property under that model can be verified after abstracting the original specification.

We believe that our approach not only contributes to the technique of data abstraction for CSP_Z specifications, but also for data abstraction of integrated notations whose component languages are a process algebra and a model-based language. Of course, adaptations of the idea presented here may be necessary when considering another integrated language.

This chapter is organised as follows. Section 3.1 presents the notion of data independence which serves to impose some restrictions on the CSP part. Sections 3.2 and 3.3 present the idea of *abstract interpretations* and data abstraction of CSP_Z processes, respectively. The algorithm implementing our approach is presented in Section 3.4. Section 3.5 gives some examples in order to contrast our approach and that of Mota. Finally, Section 3.7 presents our conclusions about the present chapter and some topics for future work .

3.1 Data Independence

Data independence is a property which guarantees that a system works similarly over any data type. That is, it does not matter what type the specification manipulates, its behaviour is the same. This property imposes some constraints to the specification in the sense that it must be free of using operations which work over a specific data type.

Lazić [76] deals with data independence of CSP specifications which manipulate a data type X (a parameter of the system). The main question around this issue is: given a specification *Spec* and an implementation *Impl* which have parameters, is *Spec* satisfied by *Impl* for all instantiations of the parameters?

Such a problem—known as the *Parameterised Verification Problem* (PVP)—was shown to be undecidable in general [53]; however, some constraints can be defined in order to make the problem treatable. Lazić developed studies on *data independence* which establish the necessary conditions to deal with this problem symbolically.

Informally, a system P is said to be data independent (with respect to a data type

X) if and only if it does not perform any operation involving values of type X; it can only input such values, store them, output them, and compare them for equality. In that case, the behaviour of P is preserved if any concrete data type (which admits equality) is substituted for X (a symbolic type). The formalisation of this property in syntactical terms is given by the following definition [76]:

Definition 3.1 (Data Independence) *A system P is data independent in a type X if and only if:*

1. *Constants do not appear in P, only variables appear, and*
2. *If operations are used then they must be polymorphic, or*
3. *If comparisons are done then only equality tests can be used, or*
4. *If used, complex functions and predicates must originate from 2 and 3, or*
5. *If replicated operators are used then only nondeterministic choices over X may appear in P.* ♢

The usefulness of the above definition is that if the processes *Spec* and *Impl* are infinite and data independent with respect to a type X, then it is possible to find out the minimum *threshold* (cardinality) of X, such that the relation $Spec \sqsubseteq_M Impl$ can be checked by only considering any subset of X with such a specific cardinality. Based on this idea, Lazić defined how to find out that cardinality, that is, $\#X \geq N$, for $N \in \mathbb{N}_1$. Therefore, the problem becomes decidable by making *Spec* and *Impl* dealing with a symbolic type X, provided $\#X$ is a finite number greater than or equal to the minimum cardinality required by *Spec* and *Impl*.

Based on the previous definition, Mota [7] presented a more restricted definition of data independence and then extended that notion to CSP_Z:

Definition 3.2 (Trivial Data Independence) *A trivially data independent CSP process is a data independent process which has no equality tests, nor polymorphic operations. Therefore, it satisfies $\#X \geq 1$ for all data independent type X.* ♢

Definition 3.3 (Partial Data Independence) *A CSP_Z specification is partially data independent if its CSP part is trivially data independent.* ♢

Data independence is enough to deal with the CSP part of a CSP_Z specification; however, it does not support the data dependence aspects of the Z part. Therefore, the following section presents a stronger theory to take care of the Z part.

3.2 Abstract Interpretation

According to the Cousots [66, 67], abstract interpretation of programs consists of describing computations in another universe of abstract objects, so that the results of abstract execution give some information on the actual computations. It is a theory widely used in most program analysis techniques (symbolic evaluation, data abstraction, program performance analysis, formalisation of program semantics, verification of correctness) and compiler optimisation, type verification, type discovery etc.

The motivation for using abstract interpretation with model checking is reducing state explosion. This is achieved by replacing infinite data types with finite ones, while still preserving most of the properties. The only drawback of this approach is that deterministic operations may become nondeterministic and specific communications may not be observed anymore. This is related to the precision of the considered abstraction. Hence, the more precise is the abstraction, the more properties about the original system are preserved [7].

In order to give an overview of abstract interpretation, we need to present some basic definitions.

Definition 3.4 (Partially Ordered Set) *A poset (**p**artially **o**rdered **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).* ♢

Definition 3.5 (Upper Bounds and Lower Bounds) *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 $s \in S$. S^u denotes the set containing all upper bounds of S:*

$$S^u = \{x \in L \mid \forall\, s \in S \bullet s \sqsubseteq x\}$$

The set containing all lower bounds of S (S^l) is defined dually:

$$S^l = \{x \in L \mid \forall\, s \in S \bullet x \sqsubseteq s\}$$ ♢

If S^u has a least element, then it is called the **lub** (least upper bound) of S. It is common to use **lub**(x,y)—the least element of {x,y}—as $x \sqcup y$ ("x join y"). Dually, if S^l has the greatest element, then it is called **glb** (greatest lower bound) of S, commonly represented by **glb**(x,y)—the greatest element of {x,y}—or $x \sqcap y$ ("x meet y").

These operations can also be extended to sets originating their distributed versions: $\bigsqcup S$ and $\bigsqcap S$, respectively.

Definition 3.6 (Lattice) *Let $\langle L, \sqsubseteq \rangle$ be a poset with L a non-empty set. Then,*

- *If there exist $x \sqcup y$ and $x \sqcap y$ for all $x,y \in L$ then $\langle L, \sqsubseteq \rangle$ is called a lattice;*
- *If there exist $\bigsqcup S$ and $\bigsqcap S$ for all $S \subseteq L$ then $\langle L, \sqsubseteq \rangle$ is called a complete lattice.* $\diamondsuit$

Example 3.1 *The structure $\langle \mathbb{P}\{a, b, c\}, \subseteq \rangle$ is a complete lattice. Figure 3.1 is an example of a Hasse diagram which shows the subset order on $\mathbb{P}\{a, b, c\}$.*

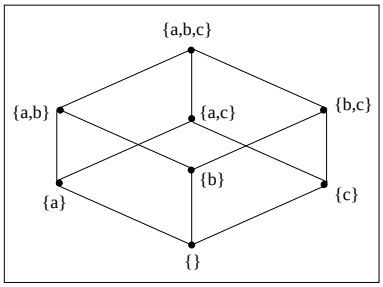


Figure 3.1: Hasse diagram for a complete lattice

Definition 3.7 (Monotonic Map) *Let $\langle L, \sqsubseteq_L \rangle$ and $\langle M, \sqsubseteq_M \rangle$ be posets. Let $f : L \to M$ be a function. Then f is said to be monotonic if and only if*

$$\forall x, y \in L \bullet x \sqsubseteq_L y \Rightarrow f(x) \sqsubseteq_M f(y)$$ $\diamondsuit$

Definition 3.8 (Galois Connection) *Let $\langle A, \sqsubseteq_A \rangle$ and $\langle C, \sqsubseteq_C \rangle$ be two lattices. If there exist monotonic maps $\alpha : C \to A$ (abstraction function) and $\gamma : A \to C$ (concretisation function) such that*

- $\forall a \in A : \alpha \circ \gamma(a) \sqsubseteq_A a$;
- $\forall c \in C : c \sqsubseteq_C \gamma \circ \alpha(c)$;

then the representation $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$ is said to be a Galois Connection. The maps α and γ are also called adjunctions. Further, from the above conditions we have

$$\alpha = \lambda X : C \bullet \bigsqcap \{Y \in A \bullet X \sqsubseteq \gamma(Y)\}$$
$$\gamma = \lambda Y : A \bullet \bigsqcup \{X \in C \bullet \alpha(X) \sqsubseteq Y\}$$ $\diamondsuit$

It is worth noting that, in the abstract interpretation terminology, $x \sqsubseteq y$ means "x is more precise than y" (or "y has less information than x"). 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 $\gamma \circ \alpha$ does not add information to c. A Galois Connection establishes a mapping between $\langle A, \sqsubseteq_A \rangle$ (the

lattice of properties) and $\langle C, \sqsubseteq_C \rangle$ (the original semantic domain), such that the *abstraction* (α) does not add information and the *concretisation* (γ) preserves information. Therefore, properties of the concrete semantics can be extracted from an abstract structure.

When establishing adjunctions between domains such that they form a Galois Connection, both objects and operations in the concrete domain have to be mapped into their abstract versions (compatible in some sense). This compatibility originates the notions of soundness (safety) and completeness (optimality) [73, 75] of the abstraction. For example, let $\langle C, \sqsubseteq_C \rangle \underset{\alpha_{C,A}}{\overset{\gamma_{A,C}}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$ and $\langle D, \sqsubseteq_D \rangle \underset{\alpha_{D,B}}{\overset{\gamma_{B,D}}{\leftrightarrows}} \langle B, \sqsubseteq_B \rangle$ be Galois Connections. Let $f_C : C \to D$ be a function defined over concrete domains and $f_A : A \to B$ be its abstract version. The following expressions define, respectively, the soundness and completeness of f_A:

- $\forall\, c \in C : (\alpha_{D,B} \circ f_C)(c) \sqsubseteq (f_A \circ \alpha_{C,A})(c) \Rightarrow f_A$ is sound for f_C
- $\forall\, c \in C : (\alpha_{D,B} \circ f_C)(c) = (f_A \circ \alpha_{C,A})(c) \Rightarrow f_A$ is complete for f_C

The first assertion states that f_A is a safe approximation of f_C if it can never "add information" to the results produced by f_C; that is, $f_A \circ \alpha_{C,A}$ produces a result with less information than $\alpha_{D,B} \circ f_C$. The optimality condition states that f_A is said to be optimal for f_C if f_A is the most precise safe approximation of f_C; that is, $f_A \circ \alpha_{C,A} \sqsubseteq \alpha_{D,B} \circ f_C$ and $\alpha_{D,B} \circ f_C \sqsubseteq f_A \circ \alpha_{C,A}$ (denoted by $f_A \circ \alpha_{C,A} =_A \alpha_{D,B} \circ f_C$). Figure 3.2 gives a graphical view of the correspondence between structures of a concrete domain and an abstract one.

The Cousots [66, 67] showed how to find the best approximation for an operation, if it exists: if f_A is the best correct approximation of an operation f_C, then $f_A =_A \alpha_{D,B} \circ f_C \circ \gamma_{AC}$. Based on this idea, Mota [7, 9] defined an algorithm which finds such an approximation considering only the Z part of a CSP_Z specification.

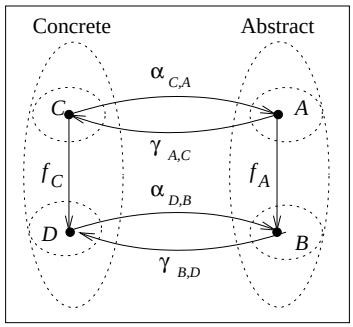


Figure 3.2: Operations mapping

3.3 CSP_Z Data Abstraction

The idea behind CSP_Z data abstraction is using abstract interpretation to build finite systems, and then overcoming the state explosion problem.

Wehrheim [42] was the first to investigate data abstraction for CSP_{OZ}. Then, Mota [7, 9] extended her work by providing mechanised support to generate abstractions for CSP_Z as well as fixing some weaknesses in the original work by also considering data independence constraints.

In this section, we present the notion of safe and optimal abstractions for CSP_Z as well as definitions and laws necessary to give support for the application of data abstraction. Furthermore, we present a theorem stating that, when also considering the CSP part, the LTS representation for a CSP_Z process can be substantially smaller than one which considers only the Z part, as investigated in [7, 9]. We also present a complete example of CSP_Z data abstraction to give the idea about its practical application.

Recall from Theorem 2.1 that, from a CSP_Z specification we can generate an equivalent CSP_M process by applying the strategy defined in [8]. From this, the state schema is represented by a tuple (e.g. $(v_1, ..., v_n)$) and all operations as functions with the following signature:

$$com\_c : D \rightarrow \mathbb{P}\, D,$$

where D is the whole state domain (i.e. $D = D_1 \times ... \times D_n$).

Similarly, the abstract version of an operation—denoted by $\{|\ .\ |\}$—has the following signature:

$$\{|\ com\_c\ |\} : D^A \rightarrow \mathbb{P}\, D^A,$$

where D^A is the abstract state domain, obtained by abstracting each original component domain, that is, $D^A = D_1^A \times ... \times D_n^A$.

The abstraction function h is a mapping from D (concrete domain) to D^A (abstract domain), that is, $h : D \rightarrow D^A$, such that:

$$\begin{array}{rcl} h(d) & = & h(d_1, ..., d_n) \\ & = & (h_1(d_1), ..., h_n(d_n)) \\ & = & (d_1^A, ..., d_n^A) \\ & = & d^A \end{array}$$

Note that the state abstraction process consists of abstracting each domain component by applying the respective abstraction function h_i. Moreover, the construction of $\{|\ com\_c\ |\}$ is compositional in the sense that it is obtained by abstracting its inner operations. For example, let s, s_1, s_2 be variables of type set and com_c be a concrete operation which has the inner operation $s = s_1 \cup s_2$ ($\cup$ is the usual union). Thus, in $\{|\ com\_c\ |\}$ we have abstract versions for variables and inner operations (e.g. $s^A = s_1^A \cup^A s_2^A$).

In order to deal with powersets, we define the function $H : \mathbb{P}\, D \rightarrow \mathbb{P}\, D^A$ as follows:

$$H(S) = \{x : S \bullet h(x)\}$$

Operations can have many abstract versions. In this work we are interested in abstractions which preserve some desirable properties. Such abstractions are considered as "good" approximations and can be classified into safe and optimal [67, 75] as follows.

Definition 3.9 (Safe Abstraction) *An abstract interpretation* $\{\!|\ .\ |\!\}$ *of an operation* com_c *is safe according to h iff:*

$$\forall\, d : D \bullet \{\!|\ com\_c\ |\!\}(d) = \bigcup_d (H \circ com\_c)(d). \qquad \diamondsuit$$

The above definition states that a safe abstraction captures more information than the original operation because it can insert nondeterminism (provided by the distributed union applied to the abstraction of the original interpretation). On the other hand, the optimal abstraction of an operation, if it exists, is the best approximation for that operation. Furthermore, it is unique and defined by strengthening the definition of safe abstraction.

Definition 3.10 (Optimal Abstraction) *An abstract interpretation* $\{\!|\ .\ |\!\}$ *of an operation* com_c *is optimal according to h iff:*

$$\forall\, d : D \bullet (\{\!|\ com\_c\ |\!\} \circ h)(d) = (h \circ com\_c)(d). \qquad \diamondsuit$$

In the following, we reproduce some definitions from [7] and then extend such definitions to our approach.

Lemma 3.1 *Let P be a partially data independent CSP_Z process and P^A its abstract version, defined by a safe abstract interpretation* $\{\!|\ .\ |\!\}$ *with interface abstraction given by the renaming R. Then $P^A \sqsubseteq_{\mathcal{T}} P[\![R]\!]$.* $\diamondsuit$

Lemma 3.2 *Let P be a partially data independent CSP_Z process and P^A its abstract version, defined by optimal abstract interpretation* $\{\!|\ .\ |\!\}$ *with interface abstraction given by the renaming R. Then $P^A =_{\mathcal{FD}} P[\![R]\!]$.* $\diamondsuit$

Lemma 3.1 states that, when abstracting the interface of the original CSP_Z process (P) by applying the renaming R, we obtain a process which refines, in the traces model, the abstract version of P (P^A), obtained by a safe abstraction. Analogously, Lemma 3.2 states the equality (in the failures-divergences model) between the original process with the renaming R ($P[\![R]\!]$) and the abstract version (P^A), obtained by optimal abstraction.

According to Mota [7], interface abstraction means applying a renaming relation to channels, such that concrete types are replaced with abstract ones, and replacing all occurrences of communicated values (concrete) with their abstract ones. Recall from Section 2.1 that a renaming consists of replacing an event with another one by applying a mapping. For example, let R be the renaming $\{a \mapsto b\}$ and let P be the process $a \rightarrow SKIP$. Applying R to P we obtain:

$$P[\![b/a]\!] = b \rightarrow SKIP.$$

Note that, if R is the identity (that is, $id_{\alpha P} : \alpha P \rightarrow \alpha P$ and $id_{\alpha P}(c) = c$), then the original process remains unaltered.

As our approach does not deal with communicated values, we adopt the identity as the renaming, and use the following corollaries of Lemmas 3.1 and 3.2.

Corollary 3.1 *Let P be a partially data independent CSP_Z process and P^A its abstract version, defined by a safe abstract interpretation $\{\!|\ .\ |\!\}$. Then $P^A \sqsubseteq_{\mathcal{T}} P$.* ♢

Proof. *The proof follows direct from considering the renaming R of Lemma 3.1 to be the identity map.*

□

Corollary 3.2 *Let P be a partially data independent CSP_Z process and P^A its abstract version, defined by optimal abstract interpretation $\{\!|\ .\ |\!\}$. Then $P^A =_{\mathcal{FD}} P$.* ♢

Proof. *The proof follows direct from considering the renaming R of Lemma 3.2 to be the identity map.*

□

Corollary 3.1 states that the original process does not perform any trace different from its abstract version, whereas Corollary 3.2 states that the original process and its abstract version are equivalent in the failures-divergences model, that is, they have the same failures and divergences.

An important contribution of Mota [7, 9] was a mechanised strategy of calculating the optimal abstraction for CSP_Z processes by achieving data abstraction on the Z part. The strategy consists of expanding the Z part, avoiding infinite repetitions of a same trace. In our approach, we also consider the CSP part during the expansion. This potentially causes a substantial reduction of the nodes in the LTS, in the sense that the whole process will typically perform less traces than its Z part. In Section 3.5 we present some examples in order to compare both approaches.

The expansion of a process is related to the traces performed by it. The more traces a process performs, the more nodes are observed on its LTS. Before we present a theorem establishing that the traces of a CSP_Z process is a subset of those ones produced by its Z part (Theorem 3.1), we reproduce a useful CSP law (Law 3.1) from [13], in order to derive an important corollary (Corollary 3.3) for CSP_Z. We also present an auxiliary lemma (Lemma 3.3) stating the traces of a CSP_Z process.

Considering two CSP processes, $P = ?x : A \rightarrow P'$ and $Q = ?x : B \rightarrow Q'$, the initial events of $P \underset{X}{\parallel} Q$ are $C = (X \cap A \cap B) \cup (A \setminus B) \cup (B \setminus A)$. The following step law shows the possible behaviour for the generalised parallelism: an event may be synchronised, unsynchronised but ambiguous, or from one side only.

Law 3.1 ($\underset{X}{\parallel}$ - step)

$$
\begin{array}{lll}
P \underset{X}{\parallel} Q = ?x : C & \rightarrow & (P' \underset{X}{\parallel} Q') \ \ \not< x \in X \not> \\
 & & (((P' \underset{X}{\parallel} Q) \sqcap (P \underset{X}{\parallel} Q')) \ \ \not< x \in A \cap B \not> \\
 & & ((P' \underset{X}{\parallel} Q) \not< x \in A \not> (P \underset{X}{\parallel} Q')))
\end{array}
$$

♢

Now, we can state how a translated CSP_Z process can evolve:

Corollary 3.3 ($\underset{I}{||}$ - step for CSP_Z) *Let P_Z and P_{CSP} be CSP processes that represent the Z and CSP parts of a CSP_Z specification, respectively. Then,*

$$P_Z \underset{I}{||} P_{CSP} = x : I \rightarrow (P'_Z \underset{I}{||} P'_{CSP}),$$

where $P'_Z = P_Z/\langle x \rangle$ and $P'_{CSP} = P_{CSP}/\langle x \rangle$. ♢

Proof. *Recall from Definition 2.4 that $I = \alpha P_Z = \alpha P_{CSP}$. Using simple laws of set theory we can trivially conclude that $C = I$, and that the event performed by the whole process always belongs to the interface. Therefore, the condition $\nleftarrow x \in X \nrightarrow$ of Law 3.1 is always valid (because $X = I$) and the processes P_Z and P_{CSP} synchronise on all events.* □

According to the above corollary, a CSP_Z process performs an event if and only if its parts synchronise on it. Otherwise, it behaves like *STOP* (deadlock). Therefore, it is reasonable to affirm that the traces of the whole process can be obtained by the intersection of the traces of its CSP and Z parts. In the following, we formalise this fact by presenting a lemma. For details about the proof, refer to Appendix B.

Lemma 3.3 (Traces of a CSP_Z Process) *Let P_Z and P_{CSP} be processes representing the CSP and Z parts of a CSP_Z process, respectively. Let I be the interface. Then,*

$$traces(P_Z \underset{I}{||} P_{CSP}) = traces(P_Z) \cap traces(P_{CSP}).$$ ♢

Now, we present a theorem stating that the expansion caused by the execution of a CSP_Z process, considering its two parts, is smaller than that considering only its Z part. To accomplish this, we examine the traces produced by the internal parallelism of a CSP_Z process.

Theorem 3.1 ($P_Z \underset{I}{||} P_{CSP}$ refines P_Z) *Let P_{CSP} and P_Z be CSP processes representing the CSP and the Z parts of a CSP_Z process, respectively, after translation. Then,*

$$P_Z \sqsubseteq_{\mathcal{T}} P_Z \underset{I}{||} P_{CSP}.$$ ♢

Proof. *The proof follows direct from traces refinement.*

1. $P_Z \sqsubseteq_{\mathcal{T}} P_Z \underset{I}{||} P_{CSP}$ [*by hypothesis*]

2. $[\Leftrightarrow] traces(P_Z) \supseteq traces(P_Z \underset{I}{||} P_{CSP})$ [*by Definition 2.1*]

3. $[\Leftrightarrow] traces(P_Z) \supseteq traces(P_Z) \cap traces(P_{CSP})$ [*by Lemma 3.3*]

4. $[\Leftrightarrow] true$ [*by set theory*]

□

Note that Theorem 3.1 also holds when the CSP part (P_{CSP}) terminates successfully, deadlocks or diverges. These situations arise when we consider the interaction between the CSP and the Z parts. Furthermore, the approach of Mota [7, 9] does not capture these situations because it considers only the Z part (P_Z).

In the following we present a complete example including the original specification and a detailed explanation about the underlying idea of achieving data abstraction.

Example 3.2 *Consider the following specification which performs two events (*a *and* b*) alternately and infinitely.*

spec P

$chan\ a,b$

$main = a \rightarrow b \rightarrow main$

$State \mathrel{\widehat=} [c : \mathbb{Z}]$

$Init \mathrel{\widehat=} [State' \mid c' = -1]$

$com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = -c]$

$com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = -(c * 2)]$

end_spec P

The behaviour of the above specification can be understood by looking at its LTS (Figure 3.3): in terms of events, it infinitely performs a and b and, in terms of state change, the value of the variable alternates between positive and negative values.

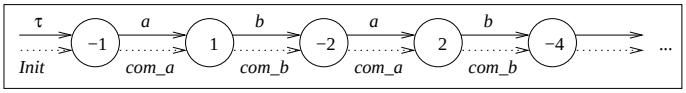


Figure 3.3: An infinite LTS of a CSP_Z process

Ignoring the state variable value, this preliminary analysis permits us to infer that the alternation between positive and negative values is the only relevant fact for abstracting the data type c. Therefore, adopting the set $\mathcal{A} = \{pos, neg\}$ as being the abstract domain, we can define the abstract map h and the abstract versions for the operations as follows:

$$h(x) = \begin{cases} pos, & if\ x > -1 \\ neg, & otherwise \end{cases}$$

$$-^{\mathcal{A}}(x) = \begin{cases} pos, & if\ x = neg \\ neg, & otherwise \end{cases}$$

$$x *^{\mathcal{A}} y = \begin{cases} pos, & if\ x = y = pos \lor x = y = neg \\ neg, & otherwise \end{cases}$$

$$x >^{A} y \quad = \quad \begin{cases} true, & if\ x = pos \wedge y = neg \\ false, & otherwise \end{cases}$$

$$x \leq^{A} y \quad = \quad \begin{cases} true, & if\ x = neg \\ false, & otherwise \end{cases}$$

Applying the above functions to the original specification, we produce its abstract version:

$spec\ P^A$

$chan\ a,b$

$main = a \rightarrow b \rightarrow main$

$State \mathrel{\widehat=} [c : \mathcal{A}]$ $\qquad$ $com\_a \mathrel{\widehat=} [\Delta State \mid c \leq^{A} neg \wedge c' = -^{A}(c)]$

$Init \mathrel{\widehat=} [State' \mid c' = neg]$ $\qquad$ $com\_b \mathrel{\widehat=} [\Delta State \mid c >^{A} neg \wedge c' = -^{A}(c *^{A} pos)]$

$end\_spec\ P^A$

Figure 3.4 shows the new LTS for the system after the abstraction technique has been applied. The behaviour (in terms of performed events) was not affected because it continues to perform a and b forever; however, it becomes finite and then can be verified by FDR [36]. Note that this abstraction is possible because the process has a predictable infinite behaviour (that is, the infinite repetition of the trace $\langle a, b \rangle$). As we are interested in preserving such a behaviour, we can find out an abstraction that reduces the system's state-space by replacing the original data domain with a finite one and giving new meanings to the operators.

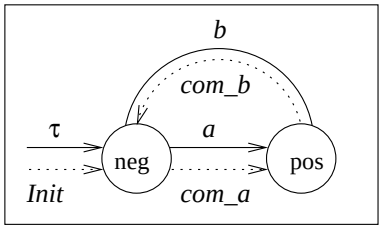


Figure 3.4: Abstracted system

The work of Mota [7, 9] gives a mechanised way of abstracting CSP_Z processes. He defined an algorithm which finds out the optimal abstraction (if it exists) based on a mathematical partitioning. Broadly, the algorithm identifies a class of values (possibly infinite) of the original type with a single value that represents the entire class (partition). This is achieved by showing that the result produced by symbolic execution of the process under analysis is the same for every value in the partition.

His approach applies that idea by considering only the data part of a CSP_Z specification, whereas our approach also considers the behavioural description. In the following, we present the algorithm implementing the data abstraction approach for CSP_Z.

3.4 Algorithm

The previous section gave an overview of data abstraction for CSP_Z, which consists of an inverse refinement on the Z part of a CSP_Z specification (a concrete data type is replaced with an abstract one). Looking at Figures 3.3 and 3.4, one can see that both versions (concrete and abstract) have the same behaviour. The difference is that the LTS representation of the second version is finite, whereas the LTS for the first one is not. In this section, we present two algorithms for data abstraction of infinite CSP_Z specifications. The first algorithm was proposed by Mota [7, 9], considering only the Z part of the whole specification and assuming that the CSP part is trivially data independent (see Definition 3.2). The second algorithm is an extensional improvement which also considers the CSP part. This algorithm constitutes the central contribution of our work.

Both approaches are based on identifying a particular behaviour of a CSP_Z process: the infinite repetition of a property. Such a behaviour, also referred as *infinite and stable* or *periodic*, can be represented by a finite LTS. In terms of performed traces, this means that the process can infinitely perform a specific trace. Furthermore, each approach has a different execution model and property representation, as explained in the rest of this section.

3.4.1 Considering the Z Part

The algorithm proposed by Mota [7, 9] tries to abstract a CSP_Z process by looking at its Z part. Therefore, it deals with properties concerning only data aspects. Instead of the functional presentation given by Mota, we present his algorithm in an imperative style due to our interest in an implementation using Java [27].

Lattice of Properties

As the algorithm is based on identifying a periodic behaviour, one has to think about representing such a behaviour as a concrete property. Considering only the data part, this property can be expressed by a conjunction of Z schema preconditions. For instance, the process presented in Example 3.2 performs $\langle a, b \rangle$ forever. Note that, before and after performing such a trace, the Z part is ready to execute only com_a. From this observation, the property is expressed by the following predicate: $pre\ com\_a \land \neg pre\ com\_b$. The Z part is ready to execute the schema com_a (uniquely) if and only if all other schemas are disabled (their preconditions are false). Analogously, if the Z part is ready to execute com_a or com_b at the same time, then their preconditions are valid at that time, and so on. Considering all possibilities (combinations) of executing schemas, such properties can be ordered in a *lattice of properties* where the bottom element represents deadlock (there is no enabled schema) and the top element is the full nondeterminism of the Z part (all schemas are enabled). Figure 3.1 shows an example of that lattice for a system with three operations (com_a_1, com_a_2 and com_a_3).

Recall from Definition 3.8 that a Galois Connection contains two mappings between a concrete structure (lattice) and an abstract one. The construction of the lattice of properties is necessary to show what properties we are interested in preserving.

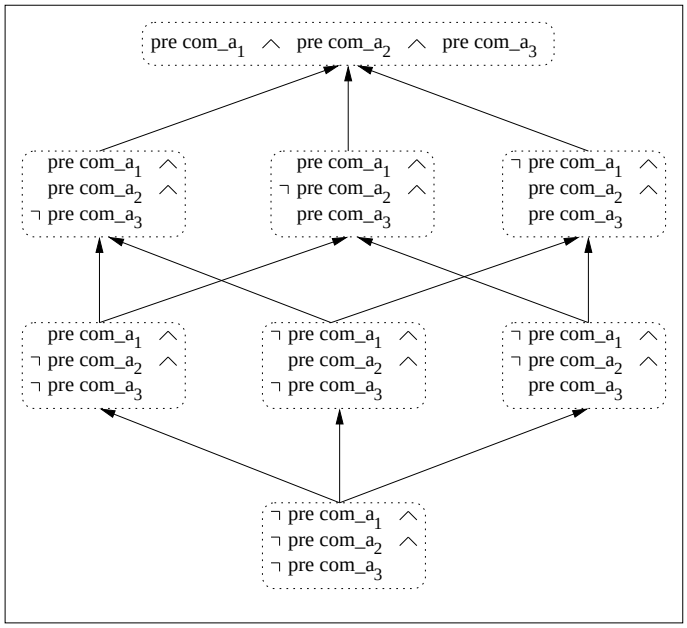


Figure 3.5: Lattice of preconditions

Stable Behaviour

The periodic behaviour of a CSP_Z process means that its Z part repeats a property forever (for some trace), that is, before and after performing some trace t, the Z part is ready to execute the same schemas again. Considering again Example 3.2, the property before and after performing $\langle a, b \rangle$ is $pre\ com\_a \wedge \neg pre\ com\_b$. Therefore, the strategy for determining its infinite repetition is described by the following steps:

1. Find the trace causing a property repetition. For example, $\langle a, b \rangle$.
2. Build the property which repeats before and after performing the trace found and call it by $conj$. For example, $conj = pre\ com\_a \wedge \neg pre\ com\_b$.
3. Recall that, in CSP_Z, a trace corresponds to a Z schema composition. Therefore, the trace $\langle a, b \rangle$ represents the composition $com\_a \fatsemi com\_b$. Call such a composition by $comp$ (i.e. $comp = com\_a \fatsemi com\_b$).
4. Build the following theorem describing the infinite stable behaviour of the Z part[2]: $\forall State, State' \mid conj \bullet comp \Rightarrow conj'$

[2]This strategy was used by Mota [7, 9] to interact with theorem provers like Z-Eves [60] or ACL2 [59].

According to the above theorem, for all state where *conj* is valid, if the execution of *comp* validates *conj* in the following state, then the system is periodic. Therefore, the algorithm no longer expands such a branch and continues analysing the other possibilities. After the construction of the LTS finishes, the algorithm builds the abstract CSP_Z process from it.

In our example, the Z part becomes stable after performing $\langle a, b \rangle$ for the first time. Hence, there is no need to generate the original LTS (Figure 3.6) if only behavioural aspects matter. If we find out equivalent nodes (that is, nodes with the same property), we can build a finite LTS representation for the same process (see Figure 3.7).

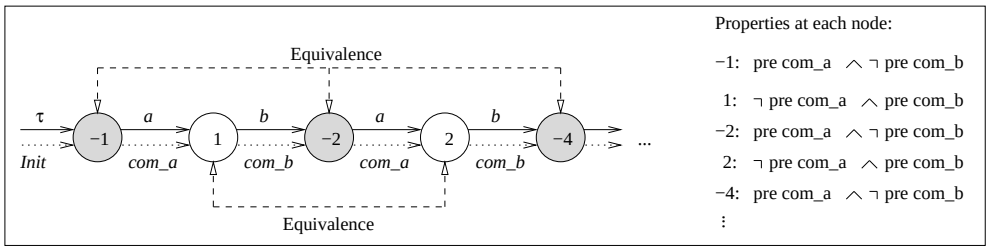


Figure 3.6: Equivalence between LTS nodes

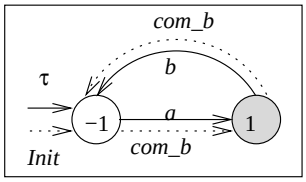


Figure 3.7: Finite LTS produced by the algorithm

Execution Model

The execution of the algorithm is based on LTS representation. Recall from Section 2.5.2 that a CSP_Z transition is denoted by two arrows indicating an event performance and its corresponding schema execution, and that the node of a CSP_Z contains information about the state and the performed trace. Furthermore, as the algorithm is based on searching property repetition, the node contains a property and the sequence of nodes produced up to it. Table 3.1 introduces the structures manipulated by the algorithm.

Description

At initialisation, the schema *Init* is executed and all variables from Table 3.1 are initialised. Figure 3.8 shows the algorithm that implements Mota's approach. The initialisation is described from line 1 up to line 7. Afterwards, the expansions follow as a Breadth First Search and, while there are nodes to be analysed (line 8), the algorithm takes all enabled schemas (line 11) and generates new children nodes (lines 13, 14, 15, 16 and 17) as long as such an expansion does not lead the system to a stable behaviour (line 12). The function

Definition	Explanation
`Node`	`Node=(State,Trace,Property,NodeSequence)`
`CurrentNode`	Denotes the node being processed.
`CurrentTrace`	The trace used to reach the current node.
`CurrentProperty`	The property of the current node.
`CurNodeSequence`	The sequence of nodes used to reach current node.
`VisitedNodes`	All nodes generated by the algorithm.
`CurrentStage`	A set of nodes being processed.
`NextStage`	A set of nodes which are children of the current nodes. They are processed after all parent nodes have been considered.
`first, second, third, fourth`	Return the respective element of a node. For example, `first(CurrentNode)` returns `CurrentState`, `second(CurrentNode)` returns `CurrentTrace`, and so on.
`extractProperty`	Gives the property of a node based on a specific state
`extractIndex`	Gives the index of a node on a sequence of nodes
`extractTrace`	

Table 3.1: Structures manipulated by the algorithm

`checkStable()` implements the idea introduced in Section 3.4.1 where the stability is defined by a Z-theorem. First, a new node is generated and then the sequence of nodes is analysed in order to find repetitions of such a property. Is so, the trace causing the repetition is extracted and then the theorem is built. Furthermore, the proof of such a theorem is only point of this algorithm which requires user assistance (because not always is possible proving theorems automatically).

In the following we present function `checkStable()`. For a detailed view of the auxiliary functions see Appendix B.

```
checkStable(com_op,Node){
  isZCycle := false
  State' := com_op(first(Node))
  Property' := extractProperty(State')
  indexOfRepetition := extractIndex(Property', fourth(Node))
  stableTrace := extractTrace(third(Node), indexOfRepetition)
  if (index != 0)
    conj := Property'
    comp := the composition corresponding to stableTrace
    z-theorem := ∀ State; State' | conj • comp ⇒ conj'
    isZCycle := (ask z-theorem to a theorem prover)
  fi
  return isZCycle
}
```

```
1.  CurrentState := Init
2.  CurrentTrace := ⟨⟩
3.  CurrentProperty := extractProperty(CurrentState)
4.  CurrentNode := (CurrentState,CurrentTrace,CurrentProperty)
5.  VisitedNodes := {CurrentNode}
6.  CurrentStage := {CurrentNode}
7.  NextStage := {}

8.  while (CurrentStage has more nodes)
9.    ∀node ∈ CurrentStage
10.      CurrentNode := node
11.      ∀com_e enabled
12.        if not checkStable(com_e,CurrentNode)
13.          NewState := com_e(CurrentState)
14.          NewTrace := CurrentTrace⌢⟨e⟩
15.          NewProperty := extractProperty(NewState)
16.          NewNodeSeq := CurNodeSequence⌢⟨CurrentNode⟩
17.          NewNode := (NewState,NewTrace,NewProperty,NewNodeSeq)
18.          if NewNode ∉ VisitedNodes
19.            VisitedNodes := VisitedNodes ∪ {NewNode}
20.            NextStage := NextStage ∪ {NewNode}
21.          fi
22.        fi
23.      ∀-end
24.   ∀-end
25.   CurrentStage := NextStage
26.   NextStage := {}
27. while-end
```

Figure 3.8: Algorithm expanding the Z part

Example 3.3 *Applying the algorithm to Example 3.2, the stability point is found after performing* $\langle a, b\rangle$ *(Figure 3.9.a) and the repeated property is pre com_a* $\wedge \neg$ *pre com_b. Therefore,* `checkStable(com_b)` *builds the Z-theorem*

$$\forall\, State, State' \mid conj \bullet comp \Rightarrow conj',$$

where $conj =$ *pre com_a* $\wedge \neg$ *pre com_b and* $comp = com\_a \fatsemi com\_b$*; and this can be submitted to a theorem prover to check if it holds. After finding out a periodic behaviour, the algorithm stops the expansion of that branch, which is represented by a finite LTS (Figure 3.9.b), and continues analysing the remaining nodes.*

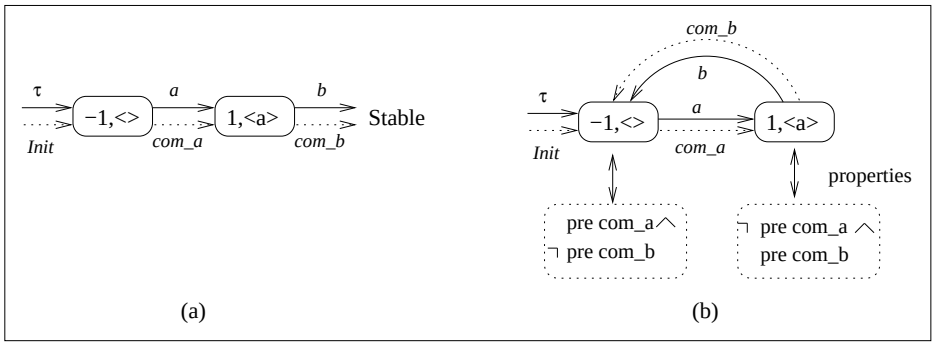


Figure 3.9: Expansions up to the stable point

Calculating the Abstraction

After avoiding infinite expansions by observing a property repetition, the algorithm builds an abstraction which is finite and captures the original behaviour. Mota [7, 9] defined a strategy for building such an abstraction by using subtype abstraction: the abstract domain is a subset of the original one. Furthermore, the abstraction process consists of abstracting schemas compositionally: inner operations are also abstracted.

An important result of the Cousots' work [66, 67] concerns the construction of the optimal abstraction. Recall from Section 3.2 that the best approximation (f_A) for a concrete operation (f_C) can be found by construction, that is, $f_A = \alpha \circ f_C \circ \gamma$, where α is the abstraction function, and γ is the concretisation function. Mota used this idea and set α to be a map h, and γ to be the identity (id). Therefore, the abstraction of an operation $\lambda\, x : \mathbb{N} \bullet x + 1$, for example, is:

$$\{\!|\ \lambda\, x : \mathbb{N} \bullet x + 1\ |\!\} = \alpha \circ (\lambda\, x : \mathbb{N} \bullet x + 1) \circ \gamma = \lambda\, x^A : \mathcal{A} \bullet h(x^A + 1),$$

where $\mathcal{A}$ is the abstract domain.

Let us illustrate the strategy by its application to Example 3.2:

1. Set the abstraction data domain to be the concrete one, that is, $\mathcal{A} = \mathbb{Z}$.

2. Set the abstraction map h to be the identity map, that is, $h : \mathbb{Z} \to \mathcal{A}$, such that $h(x) = x$.

3. Expand (symbolically) the Z part, avoiding infinite expansions (Figure 3.9.a). In our example, there is only one periodic behaviour: the infinite performance of $\langle a, b \rangle$.

4. Check whether all repetitions detected in the previous step are infinite. In our example, the property $pre\ com\_a \wedge \neg pre\ com\_b$ always repeats when the trace $\langle a, b \rangle$ is performed infinitely. Therefore, the idea is building equivalence classes for the state values which cause this periodic behaviour: each node occurring along the trace $\langle a, b \rangle$ is a strong candidate to represent an entire class of values. Nodes whose property is $pre\ com\_a \wedge \neg pre\ com\_b$, are on the equivalence relation:

$$E_a = \{c : \mathbb{Z} \mid c \leq -1 \bullet c \mapsto c * 2\}^*$$

It is worth noting that $c \leq -1$ is the reduced form of $pre\ com\_a \wedge \neg pre\ com\_b$, and the operator * is the usual reflexive-transitive closure operator.

5. Update the abstraction map h according to the equivalence class found as follows:

$$h(x) = \begin{cases} -1, & if\ (-1, x) \in E_a \\ x, & otherwise \end{cases}$$

The value representing the entire equivalence class can be any value from the domain of E_a. Is this example we have chosen -1.

6. Set $\mathcal{A}$ to be the range of h, that is $\mathcal{A} = \{-1\} \cup \{x \mid (-1, x) \notin E_a\}$.

7. Repeat steps 4, 5 and 6 until all nodes along the stable trace have been processed. For example, after performing the event a, the following node has the property $\neg pre\ com\_a \wedge pre\ com\_b$. Therefore, its equivalence relation is:

$$E_b = \{c : \mathbb{Z} \mid c > -1 \bullet c \mapsto c * 2\}^*$$

Again, the mapping h is updated to:

$$h(x) = \begin{cases} -1, & if\ (-1, x) \in E_a \\ 1, & if\ (1, x) \in E_b \end{cases}$$

and $\mathcal{A}$ becomes $\{-1, 1\}$.

8. Define the abstract state by replacing the original domain with $\mathcal{A}$:

$State \mathrel{\widehat=} [c : \mathcal{A}]$.

9. Abstract the initialisation by applying h to the predicate of $Init$ as follows:

$$\begin{array}{rcl} Init & \mathrel{\widehat=} & [State' \mid c' = h(-1)] \\ & = & [State' \mid c' = -1] \end{array}$$

10. Abstract all schemas by applying h to their post-conditions and preserving their pre-conditions:

$$\begin{array}{rcl} com\_a & \mathrel{\widehat=} & [\Delta State \mid c \leq -1 \wedge c' = h(-c)\,] \\ & = & [\Delta State \mid c \leq -1 \wedge c' = h(-h(c))\,] \\ com\_b & \mathrel{\widehat=} & [\Delta State \mid c > -1 \wedge c' = h(-(c * 2))]) \\ & = & [\Delta State \mid c > -1 \wedge c' = h(-(h(c) * h(2)))] \\ & = & [\Delta State \mid c \leq -1 \wedge c' = h(-(h(c) * 1))] \\ & = & [\Delta State \mid c \leq -1 \wedge c' = h(-(h(c)))] \end{array}$$

Building the abstracted specification based on the above steps we have:

$spec\ P^A$
$chan\ a,b$
$main = a \rightarrow b \rightarrow main$
$State \mathrel{\widehat=} [c : \mathcal{A}]$
$Init \mathrel{\widehat=} [State' \mid c' = -1]$
$com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = h(-h(c))]$
$com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = h(-(h(c)))]$
$end\_spec\ P^A$

Note that the LTS for the above specification (see Figure 3.9.b) is equivalent to the LTS of the same specification found in an *ad hoc* manner (see Figure 3.4). The main differences are: the current abstract domain is $\{-1, 1\}$ whereas the previous was $\{pos, neg\}$, and, most importantly, the current abstraction is built by a guided process while the previous had followed an heuristic approach.

3.4.2 Considering the CSP Part

As we have already explained, the approach of Mota does not consider the influence of the CSP part over the Z one. Therefore, some behaviour such as successful termination, deadlock and divergence are not captured by his algorithm. On the other hand, in our approach we also consider the CSP part, which acts as a controller process. This viewpoint has some impacts over the data abstraction technique because the CSP part filters the combinations taken into account when considering only the Z part.

The idea is expanding only the possibilities (traces) accepted by the CSP part. Its LTS establishes what traces can be performed by it. In this section, we investigate this strong relation between the CSP part and the abstraction process.

Lattice of Properties

The previous approach considered as stable a CSP_Z process whose Z part repeated infinitely some property, expressed by a conjunction of preconditions of schemas (enabled and disabled). When considering the CSP part, the stability is defined in terms of CSP and Z.

Recall from Lemma 2.1 that the initial acceptances of a CSP_Z process are obtained by capturing the events accepted by the CSP part, whose corresponding schemas are enabled. That is,

$$initials(P_{CSP_Z}) = \bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i \land a_i \in initials(P_{CSP})\},$$

where I is the synchronisation interface.

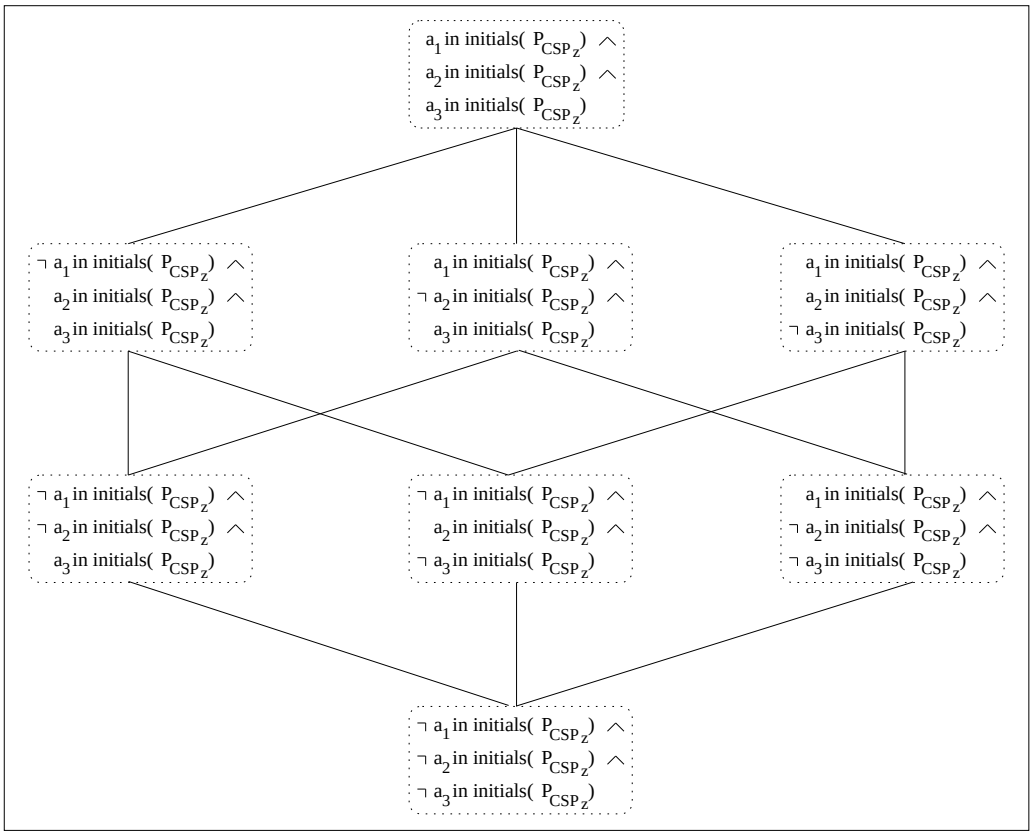


Figure 3.10: Lattice of properties

The new property is expressed in terms of $initials(P_{CSP_Z})$. For example, consider a CSP_Z processes whose synchronisation interface is $\{a, b, c\}$ and that, at a specific moment, the whole process accepts performing only the event a. The property at that moment is

$$a \in initials(P_{CSP_Z}) \land b \notin initials(P_{CSP_Z}) \land c \notin initials(P_{CSP_Z}).$$

Figure 3.10 illustrates the new lattice of properties for such a process. The bottom element represents deadlock of both parts, whereas the top element represents the full nondeterminism of the whole process.

It is worth noting that a deadlock of a CSP_Z process can occur in many situations: the CSP part is really deadlocked, the CSP part accepts a set of events whose corresponding schemas are disabled, and the Z part offers schemas whose corresponding events are rejected by the CSP part. Such situations typically concern synchronisation problems between the component processes (they cannot progress independently). On the other hand, the divergence of a CSP_Z process happens when its CSP part diverges (it does not offer any visible event to synchronise with the Z part). These situations are correctly captured by the property because $initials(P_{CSP_Z})$ is defined in terms of $initials(P_{CSP})$. If the CSP part does not have initial acceptances, then the whole process accepts performing no visible event.

Stable Behaviour

By considering the CSP part as well, the stability criterion becomes stronger: both the CSP and the Z parts have to be periodic and perform the same cycle. For CSP, this means that there is a cycle which performs a trace infinitely. For Z, it means that before and after performing a specific trace, the corresponding schema composition (*comp*) is enabled (i.e. *pre comp* is true) again.

Although the algorithm focuses on finding a repetition of a CSP_Z property, when such a repetition is found, the stability checks are performed separately. First of all, the trace causing the repetition is extracted. Then the CSP part is analysed in order to find out a cycle performing such a trace. If a cycle is found, then the CSP part is stable. Afterwards, the stability of the Z part is performed by determining if the composition corresponding to the periodic trace can always be executed by the Z part.

Note that, with this approach, one can find many possibilities of property repetition, however, the algorithm automatically discards those ones not allowed by the CSP and the Z parts.

Regarding the stability of the Z part, it is worth explaining how this is achieved in our approach. Recall from Section 3.4.1 that the stability of the Z part was determined by

$$\forall State, State' \mid conj \bullet comp \Rightarrow conj',$$

where *conj* is a conjunction of preconditions (enabled and disabled) and *comp* is a schema composition. In our approach, *conj* is replaced with *pre comp* because we are interested in analysing the behaviour of the Z part in respect to a specific composition (*comp*) rather than analysing all possible compositions enabled when *conj* is valid. Therefore, the above formula becomes

$$\forall State, State' \mid pre\ comp \bullet comp \Rightarrow (pre\ comp)' ,$$

where *comp* is the composition representing the probable stable trace.

We have also noticed that there is a subtle situation which is not captured by the above formula. If one schema of *comp* is disabled, then *pre comp* is not valid and, therefore, no change occurs (that is, $(pre\ comp)'$ is false as well). This situation can be reproduced by rewriting the above predicate, using simple laws from First-Order Logic, and replacing *pre comp*, *comp*, and $(pre\ comp)'$ with *false*: $false \Rightarrow (false \Rightarrow false)$, which produces *true* when *false* is expected (the system is not stable). This weakness[3] has motivated us to adopt a stronger formulae

$$\forall State, State' \mid (pre\ comp \Rightarrow comp) \bullet (pre\ comp)'$$

as the stability predicate of the Z part. It correctly captures the subtlety mentioned and produces the expected result; that is, $(false \Rightarrow false) \Rightarrow false$ evaluates to *false*. Furthermore, it is worth pointing out that this is the only situation where

$$\forall State, State' \mid pre\ comp \bullet comp \Rightarrow (pre\ comp)' \text{ and}$$
$$\forall State, State' \mid (pre\ comp \Rightarrow comp) \bullet (pre\ comp)'$$

[3]This happens because $\Rightarrow$ is not associative.

produce different results. Table 3.2 illustrates the comparison between the formulae. Lines 3, 4, 5 and 6 are not analysed because it does not make sense when *pre comp* and *comp* have different values. Moreover, line 7 does not happen because (*pre comp*)' must be *false* when *pre comp* and *comp* are so.

	pre comp	*comp*	*pre comp'*	*pre comp* $\Rightarrow$ (*comp* $\Rightarrow$ (*pre comp*)$'$)	(*pre comp* $\Rightarrow$ *comp*) $\Rightarrow$ (*pre comp*)$'$
1	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
2	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
3	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{T}$	-	-
4	$\mathcal{T}$	$\mathcal{F}$	$\mathcal{F}$	-	-
5	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{T}$	-	-
6	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{F}$	-	-
7	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	-	-
8	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{T}$	$\mathcal{F}$

Table 3.2: Values produced by the stability theorems

Execution Model

Our approach requires a more elaborate execution model than that adopted by Mota. As the CSP part is taken into account, the new algorithm has to deal with its representation (a LTS). Therefore, a node becomes the following structure:

```
Node = (State,LTS,Trace,Property,NodeSequence).
```

A new variable (`CurrentLTS`) represents the CSP part of the node being processed and, at initialisation, the function `buildInitialLTS()` builds its initial LTS representation. Moreover, we introduce a new function– `fifth(Node)`–which gives the fifth component of a node. Figure 3.11 gives a graphical view of the creation of the initial node.

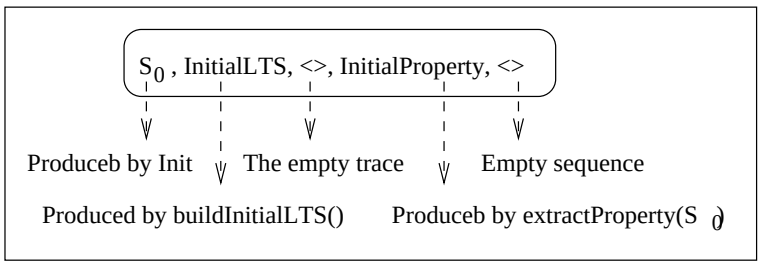


Figure 3.11: The structure of `Node` and its initialisation

Description

Now the algorithm has to execute both parts (CSP and Z) together, considering the CSP part as a master process.

Figure 3.12 shows the algorithm implementing our approach. The initialisation is described from line 1 up to line 10. As Mota's algorithm, while there are nodes to be analysed (line 11), the algorithm takes all enabled schemas (line 14) and then checks if their corresponding events are accepted by the CSP part (line 15). If so, a new child node is generated (from line 17 up to line 23) as long as such an expansion does not lead the system to a stable behaviour (line 16).

It is worth mentioning that, in this version of the algorithm, we deal with CSP_Z processes whose CSP part produces only one following behaviour when performing an event, that is, it must be deterministic. Therefore, nondeterministic processes, as for example, $(a \rightarrow P \,\Box\, a \rightarrow Q)$ cannot be dealt by our algorithm because it presents two possible behaviour (P or Q) after performing the event a.

The new version of the function `checkStable()` determines if an expansion causes some property repetition. Now, before checking the stability of the Z part it checks if the CSP part has a cycle performing the probable stable trace. The auxiliary function `extractProperty` now returns the property of a CSP_Z process (see Appendix B for its new version).

```
checkStable(com_op, Node){
  isCspzCycle := false
  State' := com_op(first(Node))
  LTS' := second(Node)/⟨op⟩
  Property' := extractProperty(State',LTS')
  indexOfRepetition := extractIndex(Property', fifth(Node))
  stableTrace := extractTrace(third(Node), indexOfRepetition)
  if (index != 0)
    if (checkCycles(stableTrace))
      comp := the composition corresponding to stableTrace
      z-theorem := ∀ State; State' | (pre comp ⇒ comp) • pre comp'
      isCspzCycle := (ask z-theorem to a theorem prover)
    fi
  fi
  return isCspzCycle
}
```

```
1.  CurrentState := Init
2.  CurrentTrace := ⟨⟩
3.  CurrentLTS := buildInitialLTS()
4.  CurrentProperty := extractProperty(CurrentState)
5.  CurNodeSequence := ⟨⟩
6.  CurrentNode := (CurrentState,CurrentLTS,CurrentTrace,
7.                  CurrentProperty,CurNodeSequence
8.  VisitedNodes := {CurrentNode}
9.  CurrentStage := {CurrentNode}
10. NextStage := {}

11. while (CurrentStage has more nodes)
12.   ∀node ∈ CurrentStage
13.      CurrentNode := node
14.      ∀com_e enabled
15.        if e ∈ initials(CurrentLTS)
16.          if not checkStable(com_e,CurrentNode)
17.            NewState := com_e(CurrentState)
18.            NewLTS := CurrentLTS/⟨e⟩
19.            NewTrace := CurrentTrace⌢⟨e⟩
20.            NewProperty := estractProperty(NewState)
21.            NewNodeSequence := CurNodeSequence⌢⟨CurrentNode⟩
22.            NewNode := (NewState,NewLTS,NewTrace,
23.                        NewProperty,NewNodeSequence)
24.            if NewNode ∉ VisitedNodes
25.              VisitedNodes := VisitedNodes ∪ {NewNode}
26.              NextStage := NextStage ∪ {NewNode}
27.            fi
28.          fi
29.        fi
30.      ∀-end
31.   ∀-end
32.   CurrentStage := NextStage
33.   NextStage := {}
34. while-end
```

Figure 3.12: Algorithm expanding for the whole CSP_Z process

Applying this approach to Example 3.2, one can see that $\langle a, b \rangle$ is performed by a cycle on the CSP part and the composition $com\_a \fatsemi com\_b$ can always be performed by the Z part (as in the previous approach). Figure 3.13 shows the expansion according to the new execution model.

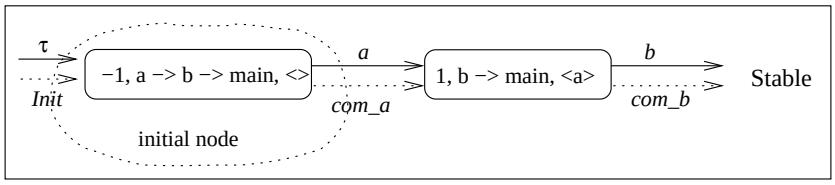


Figure 3.13: Expansions according to the new execution model

Calculating the Abstraction

In our approach, the abstraction technique is a simplification of the one adopted by Mota's approach. This happens because the CSP part filters the Z execution by rejecting some events, even when their corresponding schemas are enabled. Moreover, the process for calculating the abstraction is quite different from the one used in [7, 9]. Instead of building the abstraction function h, the abstract specification is extracted from the LTS. Comparing the original and the abstracted process, one can infer the abstraction function (see Example 3.5).

Consider, for example, the LTS depicted in Figure 3.13. After performing b the system becomes stable and accepts a again. Like the previous approach, the following nodes are equivalent to some previous one. Here, the meaning of equivalence takes into account a new property: the conjunction of acceptances of the whole CSP_Z process.

The abstraction technique is achieved by executing the following steps:

1. Expand (symbolically) the LTS of the whole process, avoiding infinite expansions;
2. The abstract domain is determined by the union of the state variable values observed on the nodes of the LTS. For example, $\mathcal{A} = \{-1\} \cup \{1\} = \{-1, 1\}$;
3. The state abstraction is obtained by replacing the infinite type with $\mathcal{A}$: $State \mathrel{\widehat=} [c : \mathcal{A}]$;
4. The initialisation produces the state value observed in the initial node, that is: $Init \mathrel{\widehat=} [State' \mid c' = -1]$;
5. The abstractions of the operations are obtained by changing their post-conditions when necessary: if a schema belongs to a cycle, then its post-condition produces the values observed in the LTS after its execution. Otherwise, it remains unaltered. In the example, the execution of $com\_a \fatsemi com\_b$ is periodic. Therefore, their abstractions are, respectively:

$$com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = 1]$$
$$com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = -1]$$

Building the entire abstraction, we obtain:

$spec\ P^A$

$chan\ a,b$

$main = a \rightarrow b \rightarrow main$

$\mathcal{A} == \{-1, 1\}$

$State \mathrel{\widehat=} [c : \mathcal{A}]$

$Init \mathrel{\widehat=} [State' \mid c' = -1]$

$com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = -1]$

$end\_spec\ P^A$

Figure 3.14 shows the LTS for the above specification. Note that it is equivalent to the one found by Mota's approach (see Figure 3.9.b).

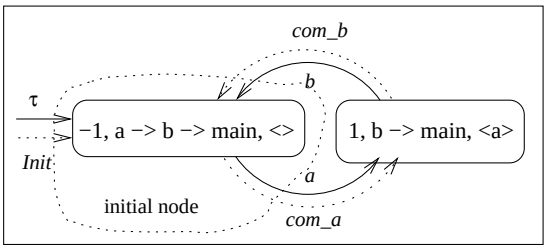


Figure 3.14: LTS of the process after abstracted

Note that, comparing the above specification with the original one, the abstraction function and the equivalence classes are implicit.

$$E_a = \{c : \mathbb{Z} \mid c \leq -1 \bullet c \mapsto c * 2\}^*$$
$$E_b = \{c : \mathbb{Z} \mid c > -1 \bullet c \mapsto c * 2\}^*$$
$$h(x) = \begin{cases} -1, & if\ (-1, x) \in E_a \\ 1, & if\ (1, x) \in E_b \end{cases}$$

The function $h(x)$ maps state variable values to -1 or 1, depending on the equivalence class they belong to. Although the abstract specification produced by our approach has the same behaviour than that produced by using the definitions of h and the equivalence classes, one can perceive we did not have to build them explicitly.

In addition, it is worth pointing out some characteristics on the Z part which simplify the calculation. For example, when all preconditions are trivially true, the abstraction can be extracted from the LTS of the CSP part because the Z part becomes infinitely stable with any trace. In other words, we are free of analysing the Z part to infer the abstraction.

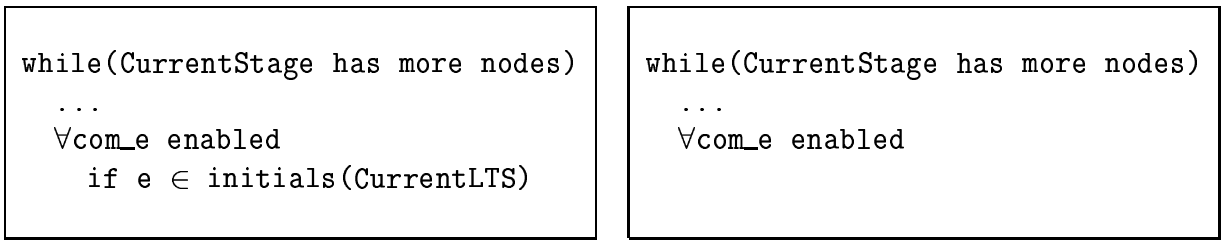


Figure 3.15: Differences between the algorithms

Another important point concerns the superior performance of our algorithm. Figure 3.15 illustrates the main difference between our approach (left) and Mota's (right). Following a CSP-driven execution, our algorithm converges faster than Mota's. This can be perceived by observing the main difference between them. For all enabled schema, our algorithm checks whether the corresponding event is accepted by the CSP part, whereas Mota's approach does not. In addition, it is worth pointing out that both algorithms might never stop when the process never becomes stable. This is regarded by the line `while(CurrentStage has more nodes)`. In fact, if the process is not stable, new nodes are generated continuously. Therefore, there will always be nodes to be processed and the loop never terminates. The limitations of our algorithm are explained in Section 3.6.

3.5 Examples and Comparisons

Because of considering the CSP part in the data abstraction approach, we can deal with a wider class of CSP_Z processes. In this section we present more examples of the application of our approach, and make a brief comparison with Mota's.

Example 3.4 *Consider the following specification whose Z part is always ready to execute all schemas, that is, all preconditions are true:*

spec P

chan a,b

$main = a \rightarrow b \rightarrow main$

$State \mathrel{\widehat=} [c : \mathbb{Z}]$ $\qquad$ $com\_a \mathrel{\widehat=} [\Delta State \mid c' = -c]$

$Init \mathrel{\widehat=} [State' \mid c' = -1]$ $\qquad$ $com\_b \mathrel{\widehat=} [\Delta State \mid c' = -(c * 2)]$

end_spec P

Figure 3.16 illustrates the expansions of the above process according to both abstraction approaches. The first one is produced by Mota's approach (a), whereas our approach (b) recognises $\langle a, b \rangle$ *as the only stable trace.*

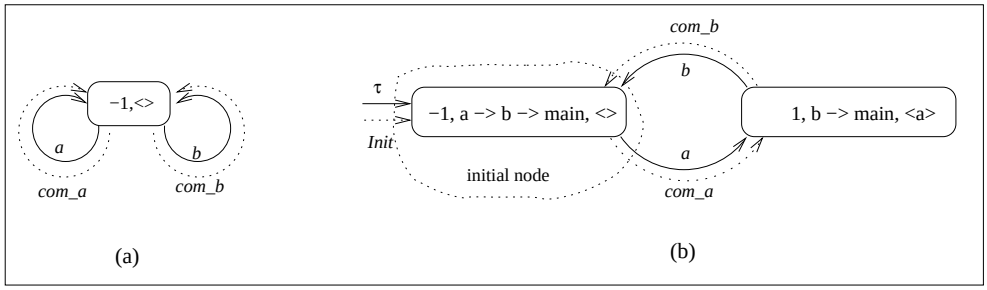


Figure 3.16: Two LTSs for the same process

At the end, Mota's approach produces the abstraction on the left and ours produces the abstraction on the right. Note that, although the abstractions are syntactically different, they have the same behaviour when considering the process as a whole.

$spec\ P^A$
$chan\ a,b$
$main = a \rightarrow b \rightarrow main$
$State \mathrel{\widehat=} [c : \{-1\}]$
$Init \mathrel{\widehat=} [State' \mid c' = -1]$
$com\_a \mathrel{\widehat=} [\Delta State \mid c' = -1]$
$com\_b \mathrel{\widehat=} [\Delta State \mid c' = -1]$
$end\_spec\ P^A$

$spec\ P^A$
$chan\ a,b$
$main = a \rightarrow b \rightarrow main$
$State \mathrel{\widehat=} [c : \{-1, 1\}]$
$Init \mathrel{\widehat=} [State' \mid c' = -1]$
$com\_a \mathrel{\widehat=} [\Delta State \mid c' = 1]$
$com\_b \mathrel{\widehat=} [\Delta State \mid c' = -1]$
$end\_spec\ P^A$

Example 3.5 *This example is a CSP_Z specification whose CSP part can perform the traces $\langle a, b\rangle^n$ and $\langle a\rangle^\frown\langle b, a\rangle^{*\frown}\langle c\rangle^n$ infinitely (* denotes 0 or more occurrences). However, the Z part does not allow com_a ⨟ com_b to be performed forever.*

$spec\ P$
$chan\ a,b,c$
$main = a \rightarrow (b \rightarrow main \,\Box\, \mu X.(c \rightarrow X))$
$State \mathrel{\widehat=} [n : \mathbb{N}]$
$Init \mathrel{\widehat=} [State' \mid n' = 0]$
$com\_a \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$
$com\_b \mathrel{\widehat=} [\Delta State \mid n \leq 5 \land n' = n + 2]$
$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \land n' = n + 1]$
$end\_spec\ P$

The expansion according to Mota's approach (Figure 3.17.a) detects stability at the point where the state variable value is 7. From that point, any sequence of com_a's and com_c's

maintain the system property (pre com_a ∧ ¬pre com_b ∧ pre com_c). On the other hand, the LTS produced by our approach (Figure 3.17.b) is a simplification of that depicted in Figure 3.17.a.

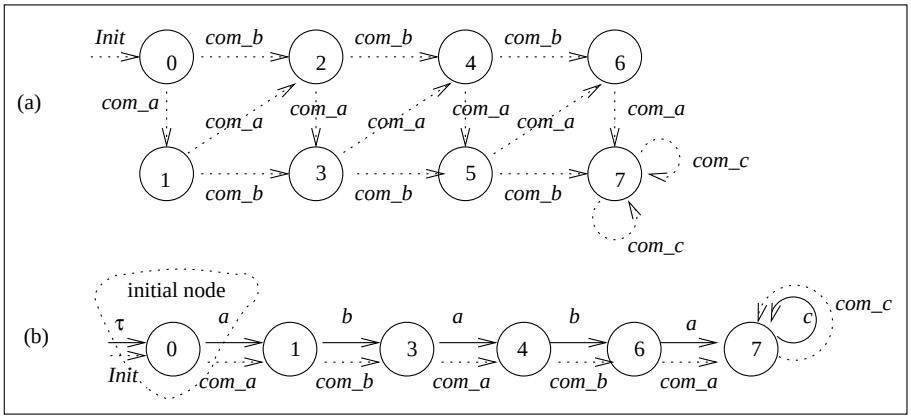


Figure 3.17: LTS produced by the two approaches

Mota's approach generates the abstraction function h and the abstract specification as follows:

$$h(x) = \begin{cases} x & \text{if } x < 7 \\ 7 & \text{otherwise} \end{cases}$$

spec P^A

chan a,b,c

$main = a \rightarrow (b \rightarrow main \,\Box\, \mu X.(c \rightarrow X))$

$\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6, 7\}$

$State \mathrel{\widehat=} [n : \mathcal{A}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid n' = h(h(n) + 1)]$

$com\_b \mathrel{\widehat=} [\Delta State \mid n \leq 5 \wedge n' = h(h(n) + 2)]$

$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \wedge n' = h(h(n) + 1)]$

end_spec P^A

In our approach, we extract the abstract domain from the LTS, that is $\mathcal{A} = \{0, 1, 3, 4, 6, 7\}$. *Afterwards, the state, initialisation and operations are abstracted. Note that, only* com_c *is abstracted (*com_a *and* com_b *do not belong to a cycle).*

$spec\ P^A$

$chan\ a,b,c$

$main = a \rightarrow (b \rightarrow main \,\Box\, \mu X.(c \rightarrow X))$

$\mathcal{A} = \{0, 1, 3, 4, 6, 7\}$

$State \mathrel{\widehat=} [n : \mathcal{A}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid n \leq 5 \land n' = n + 2]$

$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \land n' = 7]$

$end\_spec\ P^A$

Example 3.6 *This example shows a specification whose CSP part terminates:*

$spec\ P$

$chan\ a,b$

$main = a \rightarrow b \rightarrow SKIP$

$State \mathrel{\widehat=} [c : \mathbb{Z}]$ $\qquad$ $com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = -c]$

$Init \mathrel{\widehat=} [State' \mid c' : -1]$ $\qquad$ $com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = -(c * 2)]$

$end\_spec\ P$

Recall from Definition 2.4 that the Z part is a recursive process which can terminate. Therefore, there is no need to abstract the specification. Figure 3.18 illustrates the difference between applying the two approaches: in Mota's approach (see Figure 3.18.a) the specification is recognised as infinite and the system is abstracted, whereas in our approach the termination is correctly captured (see Figure 3.18.b).

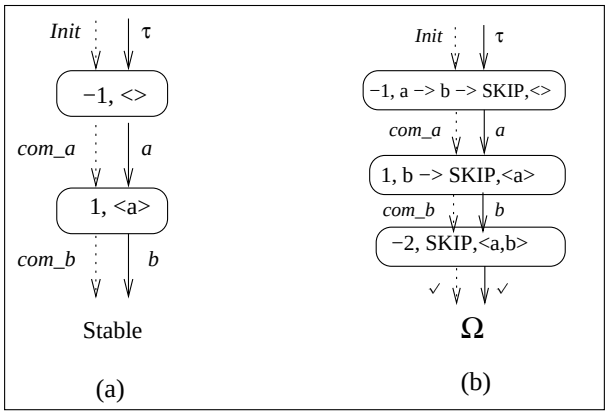


Figure 3.18: Example with termination

Example 3.7 *This example is a specification whose CSP part presents a deadlock:*

$spec\ P$
$chan\ a,b$
$main = a \rightarrow b \rightarrow STOP$
$State \mathrel{\widehat=} [c : \mathbb{Z}]$ $\quad com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = -c]$
$Init \mathrel{\widehat=} [State' \mid c' : -1]$ $\quad com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = -(c * 2)]$
$end\_spec\ P$

Mota's approach does not capture deadlocks from the CSP part (see Figure 3.19.a), whereas our approach does so (see Figure 3.19.b).

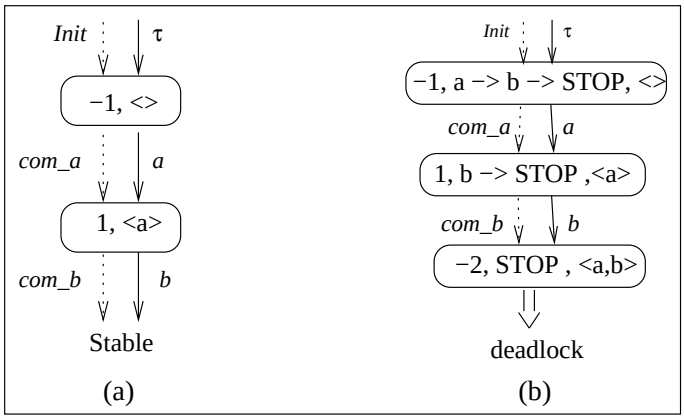


Figure 3.19: Example with deadlock

Example 3.8 *This example is a divergent process because its CSP part executes an infinite sequence of internal actions. After performing the trace $\langle a, b \rangle$ its Z part waits for events which never occur (see Figure 3.20.b). Again, Mota's approach does not capture such a situation (see Figure 3.20.a):*

$spec\ P$
$chan\ a,b$
$main = a \rightarrow b \rightarrow \mathbf{div}$
$State \mathrel{\widehat=} [c : \mathbb{Z}]$ $\quad com\_a \mathrel{\widehat=} [\Delta State \mid c \leq 0 \land c' = -c]$
$Init \mathrel{\widehat=} [State' \mid c' : -1]$ $\quad com\_b \mathrel{\widehat=} [\Delta State \mid c > 0 \land c' = -(c * 2)]$
$end\_spec\ P$

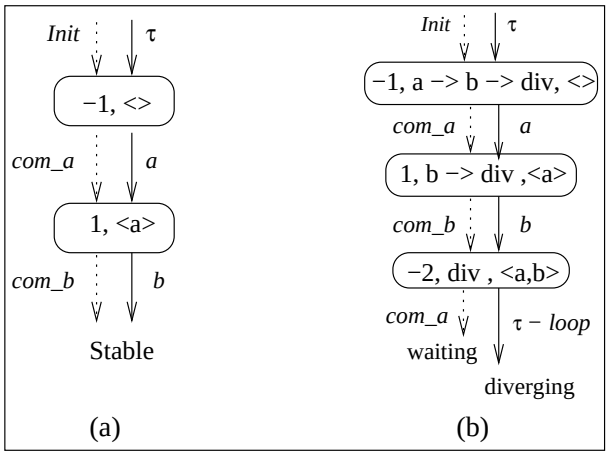


Figure 3.20: Example with divergence

Example 3.9 *This example presents a CSP_Z process whose parts have different acceptances at some point:*

$spec\ P$

$chan\ a,b$

$main = a \rightarrow a \rightarrow a \rightarrow b \rightarrow main$

$State \mathrel{\widehat=} [c : \mathbb{Z}]$ $\qquad com\_a \mathrel{\widehat=} [\Delta State \mid c < 10^6 \land c' = c + 1]$

$Init \mathrel{\widehat=} [State' \mid c' : 0]$ $\qquad com\_b \mathrel{\widehat=} [\Delta State \mid c \geq 10^6 \land c' = c + 2]$

$end\_spec\ P$

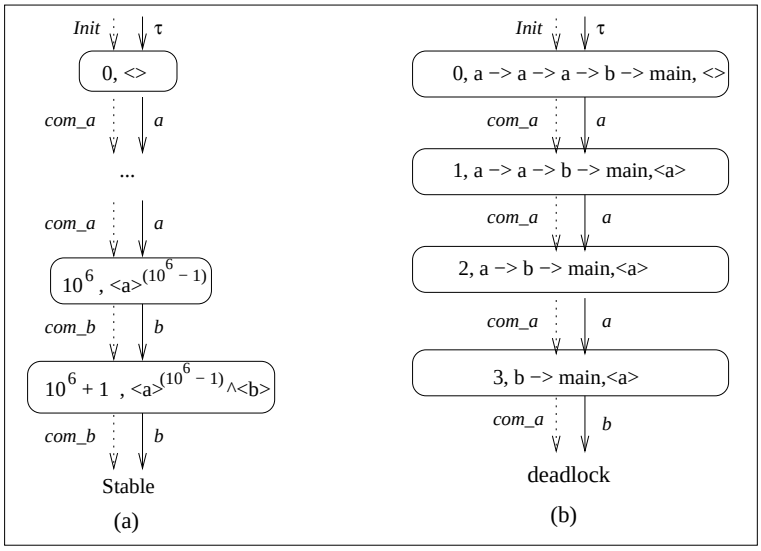


Figure 3.21: Different acceptances of both parts

Note that, after performing $\langle a, a, a \rangle$ *the Z part is ready to execute* com_a, *while the CSP part accepts* b. *In Mota's approach* $10^6 + 1$ *expansions are executed before finding out a stable behaviour (see Figure 3.21.a), whereas in our approach 4 steps are sufficient to detect different acceptances of both parts and, consequently, deadlock (see Figure 3.21.b).*

Example 3.10 *This example is a specification with a cycle containing a same event* (a) *two times. Therefore, the abstraction of the corresponding schema* (com_a) *has a subtlety.*

spec P

$chan\ a,b,c$

$main = a \rightarrow b \rightarrow a \rightarrow c \rightarrow main$

$State \mathrel{\widehat=} [c : \mathbb{N}]$

$Init \mathrel{\widehat=} [State' \mid c' : 0]$ $\qquad$ $com\_a \mathrel{\widehat=} [\Delta State \mid c' = c + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid c' = c + 2]$ $\qquad$ $com\_c \mathrel{\widehat=} [\Delta State \mid c' = c + 3]$

end_spec P

As depicted in Figure 3.22, all schemas belong to the cycle. Note that, as the event a *happens two times in the cycle, the effect of* com_a *depends on the previous value of the state variable. For example, when the state variable value is 0,* com_a *produces* $c' = 1$ *and, when its value is 3,* com_a *produces* $c' = 4$.

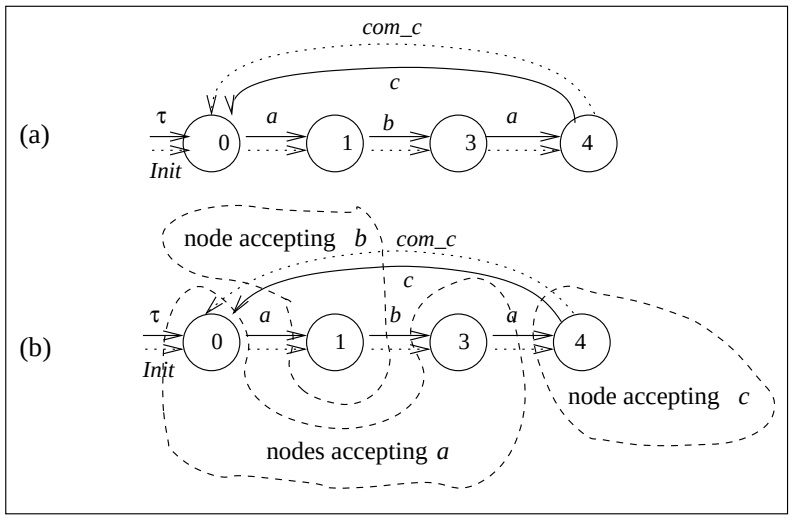


Figure 3.22: An event occurring twice in a cycle

At the end, the abstract specification produced by our approach is:

$$
\begin{array}{l}
\mathit{spec}\ P^A \\
\quad \mathit{chan}\ a,b,c \\
\quad \mathit{main} = a \rightarrow b \rightarrow a \rightarrow c \rightarrow \mathit{main} \\
\quad \mathcal{A} = \{0,1,3,4\} \\
\quad \mathit{State} \mathrel{\widehat=} [c : \mathbb{N}] \\
\quad \mathit{Init} \mathrel{\widehat=} [\mathit{State}' \mid c' : 0] \\
\quad \mathit{com\_a} \mathrel{\widehat=} [\Delta \mathit{State} \mid (c = 0 \land c' = 1) \lor (c = 3 \land c' = 4) \\
\quad \mathit{com\_b} \mathrel{\widehat=} [\Delta \mathit{State} \mid c' = 3] \\
\quad \mathit{com\_c} \mathrel{\widehat=} [\Delta \mathit{State} \mid c' = 0] \\
\mathit{end\_spec}\ P^A
\end{array}
$$

It is worth mentioning that, in all examples where the abstraction process was performed, the abstracted operations had their postconditions replaced with an assignment producing the value observed in a node from the LTS. In fact, when considering a cycle to be abstracted, it is only required that the operation closing such a cycle must be abstracted. In the last example, the operation *com_c* closes the cycle $\langle a, b, a, c \rangle$. Therefore, the following specification also represents the optimal abstraction for the original process:

$$
\begin{array}{l}
\mathit{spec}\ P^A \\
\quad \mathit{chan}\ a,b,c \\
\quad \mathit{main} = a \rightarrow b \rightarrow a \rightarrow c \rightarrow \mathit{main} \\
\quad \mathcal{A} = \{0,1,3,4\}] \\
\quad \mathit{State} \mathrel{\widehat=} [c : \mathcal{A}] \\
\quad \mathit{Init} \mathrel{\widehat=} [\mathit{State}' \mid c' : 0] \\
\quad \mathit{com\_a} \mathrel{\widehat=} [\Delta \mathit{State} \mid c' = c + 1] \\
\quad \mathit{com\_b} \mathrel{\widehat=} [\Delta \mathit{State} \mid c' = c + 2] \\
\quad \mathit{com\_c} \mathrel{\widehat=} [\Delta \mathit{State} \mid c' = 0] \\
\mathit{end\_spec}\ P^A
\end{array}
$$

Note that, as the two last abstract specifications have the same abstract domain (that is $\mathcal{A} = \{0,1,3,4\}$), their execution produce (exactly) the same behaviour. We have decided adopting the approach to abstract all operations from a cycle for questions of implementation.

3.6 Limitations

In this section we show some limitations of our data abstraction approach through examples. The first limitation concerns cycles (periodic behaviour) that can be broken by

performing an event outside it (Example 3.11). The second limitation reveals a situation where our algorithm does not produce any abstraction because the system never becomes stable (Example 3.12).

Example 3.11 *This example has two cycles and presents a subtle characteristic which is not dealt by our approach (nor Mota's one). However, one can overcome this problem by a more detailed analysis of the LTS:*

spec P

$chan\ a,b,c$

$main = a \rightarrow (b \rightarrow main \,\Box\, \mu X.(c \rightarrow X))$

$State \mathrel{\widehat=} [n : \mathbb{N}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid n' = n + 2]$

$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \wedge n' = n + 1]$

end_spec P

Figure 3.23.a shows the LTS. Note that, although $\langle a, b \rangle$ *is an infinite cycle, the algorithm cannot simply stop the expansions because there is a possibility of breaking such a cycle (by performing the event* c*). Therefore, the expansions continue up to the point where* com_c *is enabled. Afterwards, the expansions corresponding to* $\langle a, b \rangle$ *do not continue anymore.*

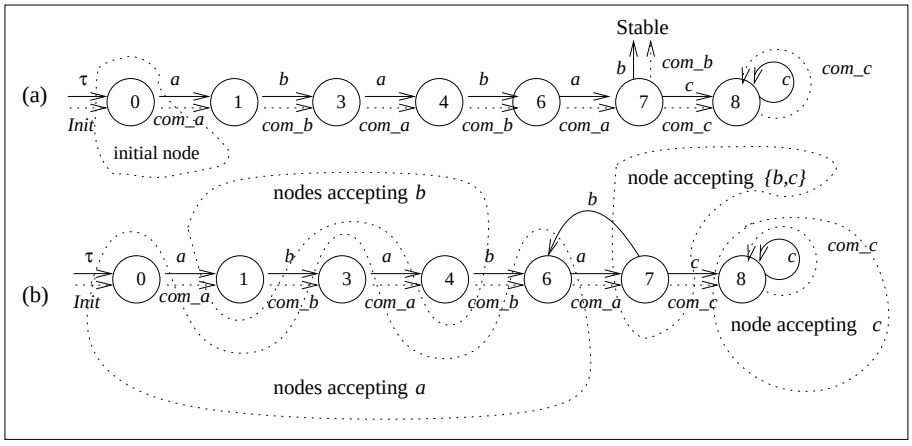


Figure 3.23: LTS with two cycles

At the point where $\langle a, b \rangle$ *is broken (the state variable value is 7), the next execution of* com_b *must lead the system to a state where* com_a *is enabled (0, 3 or 6). One must choose 6 because, after executing* com_a *at that state,* com_b *and* com_c *become enabled (this is a behaviour observed in the LTS of the original process). Therefore, the effect of*

com_b *depends on the current value of the state variable, that is, the deterministic choice* $(n \neq 7 \land n' = n + 1) \lor (n = 7 \land n' = 6)$ *represents its post-condition. At the end, the abstract specification is:*

spec P^A

chan a,b,c

$main = a \rightarrow (b \rightarrow main \,\Box\, \mu X.(c \rightarrow X))$

$\mathcal{A} = \{0, 1, 3, 4, 6, 7, 8\}$

$State \mathrel{\widehat=} [n : \mathcal{A}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid (n \neq 7 \land n' = n + 1) \lor (n = 7 \land n' = 6)]$

$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \land n' = 8]$

end_spec P^A

Example 3.12 *This example shows a situation where neither Mota's algorithm nor ours stop.*

spec P

chan a,b,c

$main = a \rightarrow main \,\Box\, b \rightarrow main \,\Box\, c \rightarrow main$

$State \mathrel{\widehat=} [x : \mathbb{N}]$

$Init \mathrel{\widehat=} [State' \mid x' : 1]$

$com\_a \mathrel{\widehat=} [\Delta State \mid x \leq 5 \land x' = x + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid x \geq 5 \land x' = x - 3]$

$com\_c \mathrel{\widehat=} [\Delta State \mid (x < 4 \lor x \geq 6) \land x' = 2 * x]$

end_spec P

Figure 3.24 illustrates the LTS of the above process (representing only the state variable and event occurrences). Note that, before reaching $x = 6$*, various stability predicates are unsuccessfully checked. From that point, we can find optimal abstraction via* com_c*, because the property* $a \notin initials(P_{CSP_Z}) \land b \in initials(P_{CSP_Z}) \land c \in initials(P_{CSP_Z})$ *is infinitely repeated. However, from the same point via* com_b *the property is not maintained. The execution of com_b—in finitely many steps—changes such a property. Therefore, the process never becomes stable and the algorithm is non-terminating.*

For this example, we can find a safe abstraction—in a manual way—by inserting non-determinism into com_b*. From the point where* $x = 6$*, its execution produces* $x = 3$*, which validates the property* $a \in initials(P_{CSP_Z}) \land b \notin initials(P_{CSP_Z}) \land c \in initials(P_{CSP_Z})$*, or* $x = 6$*, which validates the property* $a \notin initials(P_{CSP_Z}) \land b \in initials(P_{CSP_Z}) \land c \in initials(P_{CSP_Z})$ *again. This idea captures—abstractly—the behaviour of* com_b*, however*

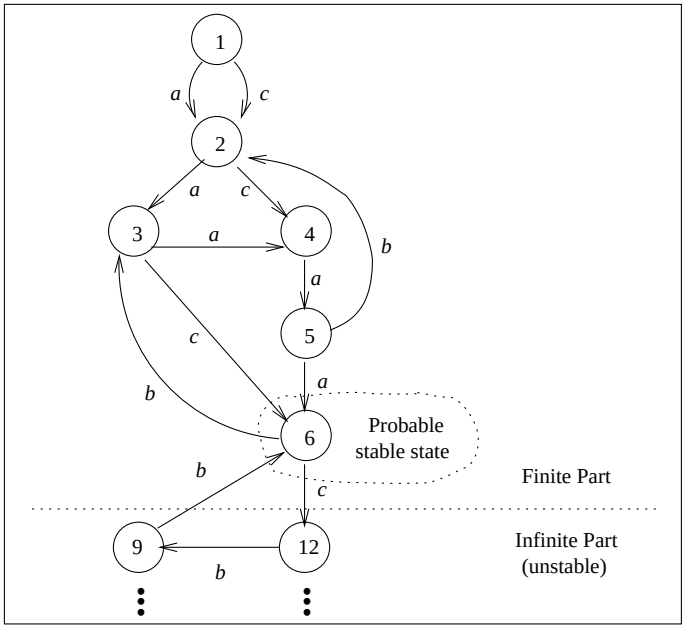


Figure 3.24: Example with divergence

the operation becomes non-deterministic. Operation com_a *does not need to be abstracted (it does not belong to a cycle), whereas* com_c *produces a stable value* ($x' = 6$) *only for all value greater or equal to* 6.

$$
\begin{array}{l}
spec\ P^A \\
\quad chan\ a,b,c \\
\quad main = a \rightarrow main \ \Box\ b \rightarrow main \ \Box\ c \rightarrow main \\
\quad State \mathrel{\widehat=} [x : \mathbb{N}] \\
\quad Init \mathrel{\widehat=} [State' \mid x' : 1] \\
\quad com\_a \mathrel{\widehat=} [\Delta State \mid x \leq 5 \land x' = x + 1] \\
\quad com\_b \mathrel{\widehat=} [\Delta State \mid x \geq 5 \land (x' = x - 3 \lor x' = 6)] \\
\quad com\_c \mathrel{\widehat=} [\Delta State \mid (x < 4 \land x' = 2 * x) \lor (x \geq 6 \land x' = 6)] \\
end\_spec\ P^A
\end{array}
$$

3.7 Conclusions

State explosion is a problem which emerges naturally in model checking of CSP_Z processes. The combination of abstraction and model checking has been used to overcome such a problem. Abstraction is used to transform an infinite process into a finite behavioural equivalent one. The work of Mota [7, 9] was an integration of the works of Lazić [76] and

Wehrheim [42] towards mechanisation of data abstraction by considering only the data part of a partially data independent CSP_Z specification (see Definition 3.3).

Synchronisation aspects between both parts of a CSP_Z specification have been studied in this work. In this sense, we believe to have provided the following contributions:

- Application of the data abstraction technique to a more general class of problems. Our approach deals with specifications whose CSP part has a more complex structure than that adopted in [7, 9]. The way the expansions are performed (considering only the Z part) is as if the CSP part accepted all events. Indeed, the only assumption concerns trivially data independence properties and unnecessary expansions are refused by the CSP part after combining both parts in parallel. As Mota's approach does not considers the CSP part, it is not able to capture situations like successful termination, deadlocks and divergences. In those situations, it does not make sense finding abstractions (see Examples 3.6, 3.7, 3.8 and 3.9);

- Reduction of the expansions caused by Mota's approach. In our approach, we do not have to expand all possibilities, just those which are accepted by the CSP part. This means that our algorithm converges more quickly and more often than that of Mota. Of course, our approach produces the same expansions when the CSP part is an external choice of all possible events. We have provided Theorem 3.1 to state this advantage by comparing the traces of the whole process with those produced by its Z part;

- Simplification of the stability predicate of the Z part. It considers specific compositions (*comp*) which are possible to happen (that is, when *pre comp* is true). Furthermore, we changed the Z stability theorem because the one adopted by Mota is not strong enough to capture the problem which arises when *pre comp* is false (see Table 3.2);

- Construction of the optimal abstraction directly from the LTS. The calculation of the abstraction is based on values observed on the LTS. This means that, differently from Mota's algorithm, we give the abstract specification as a whole without building neither the abstraction function nor any equivalence class. As explained at the end of Section 3.4.2, the abstraction function h is implicit. Comparing the abstract specification to the original one, we can infer it. We have decided to adopt this approach because the construction of h depends on the equivalence classes found out along the process, which are built by considering information about the Z part. In our case, we also need information about the CSP part. We have also discovered when all preconditions are always true, the LTS of the whole process overlaps that of its CSP part. This is an evidence of the role of the CSP part on a CSP_Z process;

- We also extended some definitions in order to supply the formalisation of our approach. Corollaries 3.1 and 3.2 are extensions of Lemmas 3.1 and 3.2 from Mota [7],

respectively. We also presented a corollary establishing the traces of a CSP_Z process (Corollary 3.3), and a theorem (Theorem 3.1) concerning the efficiency of our approach over Mota's one.

In terms of mechanisation, our approach contributes with an idea which can be implemented. The works of Cleaveland and Riely [73], Sifakis *et al* [28] and Wehrheim [42] are also devoted to treat the state explosion problem. Nevertheless, they do not give any mechanised way of doing it. In the next chapter we present a tool which mechanises our algorithm. In fact, there is no need of any user assistance with our data abstraction approach, except when proving the Z stability into a theorem prover. Moreover, the strategy for building abstractions can be extended to deal with problems of breaking infinite cycles during the expansions (as explained in Example 3.11).

We have also considered only simple examples, but we plan to deal with mode realistic case studies. In the literature, we could not find works which deal with mechanising data abstraction with real world problems. We also believe that this work can be extended to deal with communication events as well, that is, events which involve a transmission of data through channels. This extension will make the technique more general and applicable to a more satisfactory class of problems.

Chapter 4

Tool Support

In the previous chapter we extended an approach which implements a technique to overcome the *state explosion* problem. We proposed an algorithm which applies data abstraction to a CSP_Z process, considering its CSP and Z parts (see Figure 3.12). The idea behind avoiding infinite expansions focuses on finding out a periodic behaviour of the process. When a specific trace is infinitely performed, the corresponding sequence of properties is also repeated. At the moment of a property repetition, the algorithm stops the expansions and performs stability tests for both CSP and Z parts. If they are really periodic and stable (considering a specific trace) the algorithm abstracts that branch by building a cycle in the LTS. At the end, the abstract version of the specification is extracted from such a LTS.

The implementation of the algorithm is relatively simple. In this chapter, we present a tool which implements our approach to CSP_Z data abstraction. The tool is an extension of a previous work [1, 2], which simply translates a CSP_Z specification into an equivalent CSP_M process. We have decided using Java [27] because of our experience, the support for writing scanners and parsers, and also because we are interested in extending an existing implementation. Furthermore, because we have used design patterns [33] and the object-oriented paradigm, modularity, reusability, and extension facilities[1] were substantially improved.

This chapter is organised as follows. Section 4.1 presents the tool and its main modules: Parser, Translator, Data Independence and Data Abstraction. Section 4.2 concerns information to the user. It shows the screens, dialogs, tool bar, menus, and explains how each of them works. Section 4.3 gives an overview of design patterns and presents the patterns adopted in our implementation. In Section 4.4, we show details about configuring of the tool. Finally, in Section 4.5 we present our conclusions about this chapter and some topics for future work.

[1]Please, refer to [33] for details about the improvements obtained when adopting design patterns.

4.1 The Tool

In this section we present the tool which applies the data abstraction technique presented in the previous chapter. The tool is composed by four main modules: CSP_Z Parser, Translator, Data Independence and Data Abstraction (see Figure 4.1). It accepts a specification of a CSP_Z process (possibly infinite) and produces a finite one by applying the data abstraction technique. Furthermore, both the original specification and its abstract version can be translated into CSP_M [36].

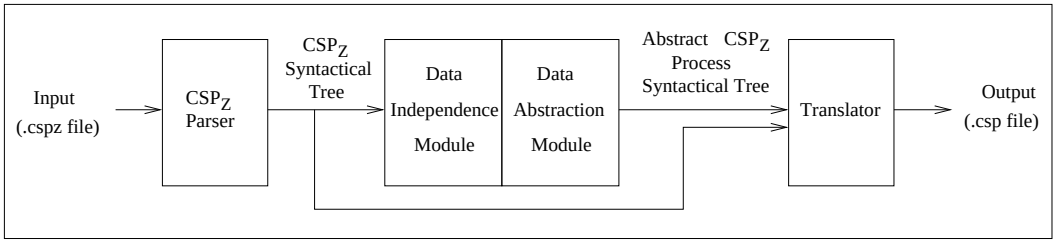


Figure 4.1: Modules of the tool

4.1.1 The CSP_Z Parser

As CSP_Z uses CSP and Z structures, its parser must accept terms of both notations. We have implemented a parser for CSP_Z by modularising it as two distinct parsers: one for each language. Figure 4.2 illustrates the structure of the parser. It accepts a CSP_Z specification written into a text file (with extension `.cspz`) and then splits the specification into two distinct and smaller specifications. Both CSP and Z parts are given to their corresponding parsers. Afterwards, the parsers make a syntactical analysis and give two syntactical trees, which are used to compose the whole CSP_Z syntactical tree. The inner parsers were implemented in Java [27] by using auxiliary libraries for scanner and parser generation: JLex [32] and JavaCUP [80], respectively. The Z parser is an implementation of a subset of the standard established in [43], and the CSP parser is an implementation of the grammar used by FDR. The original Z grammar was written for Lex and Yacc [50] and the CSP one was written for Flex [84] and Bison [19]. Like JLex and JavaCUP, those tools are useful to build scanners and parsers from a grammar description. Naturally, the files describing the grammars needed a few modifications to be processed by JLex and JavaCUP.

In this work we also made some changes on the parsers. Based on the syntax accepted by the tool [4], we removed some terms of the original Z grammar, like lambda and conditional expressions. From the CSP parser, we eliminated indexed constructions (e.g. $\Box_i P_i$) from the CSP and adjusted channel declarations to the CSP_Z syntax, where local channels are also permitted and the channel types have a different form from that in CSP. Those changes were necessary because there were no structures representing all terms of both languages. Furthermore, we also improved the analysis of CSP_Z files: the specification is first loaded

to the memory and then the parsing is performed. The previous version analyses the specification directly from the file system. Details about the syntax accepted by the tool can be found in [4].

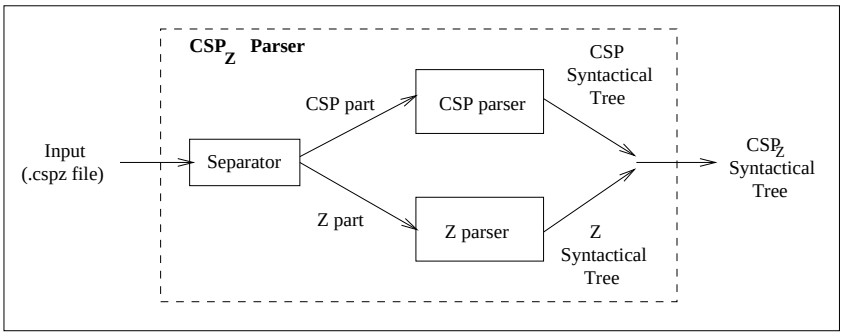


Figure 4.2: Structure of the CSP_Z parser

4.1.2 The Translator Module

The tool presented in [1, 2] implements the strategy for converting a CSP_Z specification into an equivalent CSP_M process [8]. The syntactical tree produced by the parser is actually an object representing a CSP_Z specification (see Figure 4.3).

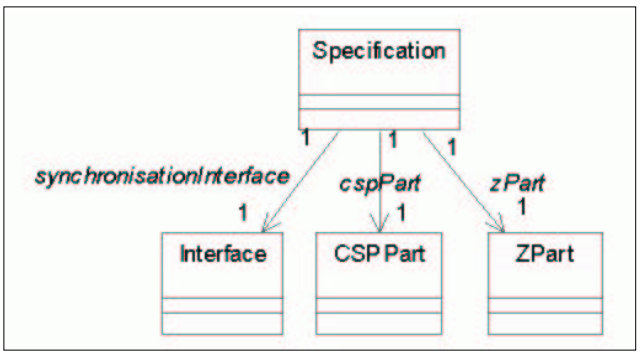


Figure 4.3: The Specification object

After receiving such a structure, the Translator module (see Figure 4.4) converts this structure into an equivalent CSP_M process in a semi-automatic manner. Such a conversion raises questions of decidability. For example, not all predicates can be transformed into CSP_M functions; the Z structure *Bag* does not have any corresponding CSP_M one. Therefore, some steps of refinement might be required before converting such a structure. These questions are dealt with, for example, by Borba and Meira [65].

The generated `.csp` file contains the description of a CSP_M process, which can be analysed by the FDR [36] tool. Furthermore, we have also implemented some functions

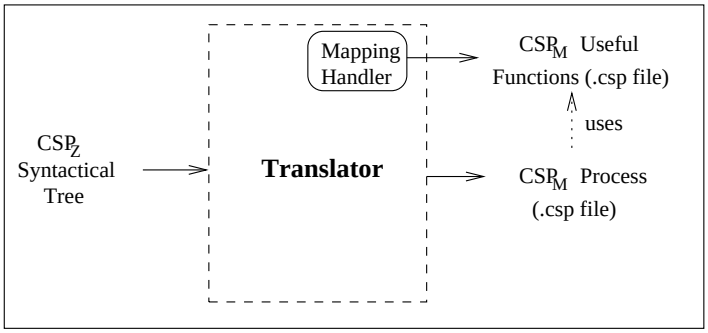


Figure 4.4: Translator module

representing the most used Z operators. For example, if the original specification contains the expression $\{s\} \subset S$, it is translated into `subset({s},S)`. The definition of such a function is put into an auxiliary file produced by the *MappingHandler* component. In this case, it is defined as follows:

```
subset(S1,S2) =
        if( (card(diff(S2,S1)) > 0) and (card(diff(S1,S2)) == 0) ) then
           true
        else
           false
```

The tool also provides support for editing such a definition.

4.1.3 The Data Independence Module

Recall from the previous chapter that the data abstraction technique proposed in this work can only be applied to partially data independent CSP_Z specifications (Definition 3.3), that is, specifications whose CSP part is trivially data independent (Definition 3.2). The abstract version of the original specification (possibly infinite) has the same properties and can be verified by using the FDR [36] tool.

In the implementation of Mota [7, 9] the CSP part is assumed to be trivially data independent. Therefore, no check is performed before applying data abstraction. The Data Independence module eliminates this assumption by verifying the accordance of the CSP part with Definition 3.2. Therefore, if such constraint is satisfied, this module assures that the input corresponds to a partially data independent CSP_Z process. Otherwise, it produces an error and the abstraction process is not performed (see Figure 4.5).

4.1.4 The Data Abstraction Module

After assuring that the original CSP_Z specification is partially data independent, the data abstraction technique can be applied. The Data Abstraction module—the most complex part of the tool—applies the algorithm proposed in this work.

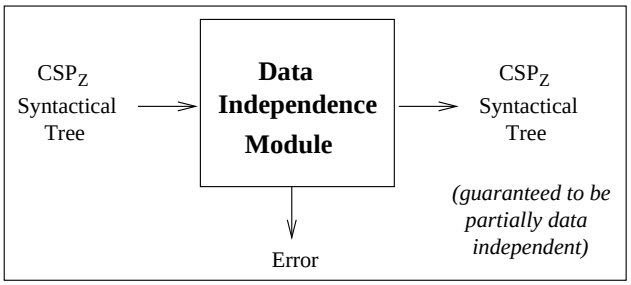


Figure 4.5: Data Independence module

Recall from Section 3.4.2 that our approach also considers the CSP part of a CSP_Z specification. Therefore, in our algorithm, both parts (CSP and Z) must execute together, that is, they must synchronise on all events from the interface. To accomplish this task, we had to implement two animators: one for Z and another for CSP. Before giving a detailed explanation about them, we present the general structure of the Data Abstraction module (see Figure 4.6).

The module contains a CSP_Z animator, which is composed by three components: Specification Abstractor, Stability Plugin Factory, and Expansion Engine. Based on a syntactical tree of a partially independent CSP_Z process, this module creates an execution tree as an input to a CSP_Z animator and then executes the algorithm presented by Figure 3.12.

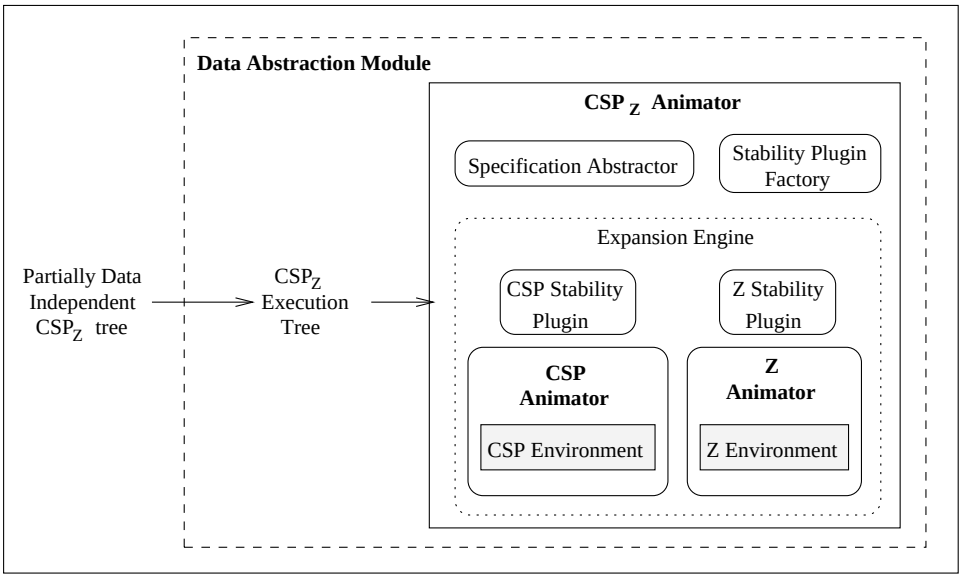


Figure 4.6: Data Abstraction module

After receiving an execution tree, the CSP_Z Animator component instantiates two internal animators, CSP and Z, and configures their environments (with definitions of processes and schemas, respectively) in order to start the execution of both parts. Afterwards, the initial

node is built and the expansion of the whole process takes place. The CSP_Z Animator acts as an execution controller, managing the expansions and analysing the stability of the whole process. When a property repetition is found, it tries to determine the stability property of both parts. This is achieved by obtaining two stability plugins from the Stability Plugin Factory component, and requesting to them such an analysis separately. If the repetition really occurs, then the animator does not generate a new node; otherwise, the expansion continues. After all nodes have been processed, the Specification Abstractor component analyses the resulting LTS, calculates the abstract domain, and builds the abstract versions of each schema. Afterwards, a new specification containing these abstract structures and the original CSP part is produced.

In the following, we give a brief explanation about the structure of the CSP_Z Animator component.

4.1.4.1 The Expansion Engine

This component is responsible to give support for executing the expansions. Both CSP and Z parts must be executed in a step-by-step manner. Therefore, each of its internal components has a specific purpose.

The Z Animator

This component gives support for executing Z schemas based on a state. First of all, its environment is configured with the definitions of the state, initialisation and operations. After that, it becomes ready to execute schemas based on a current state. Furthermore, this component is able to verify if the precondition of a specific schema is satisfied in a given state.

The main functions of this component are listed below:

- **configure** - receives all definitions of schemas and then configures the execution environment. This consists of putting all definitions into the environment.
- **initialise** - initialises the animator by performing the schema *Init*.
- **canExecute** - given a state and a schema name, determines whether its precondition is valid.
- **execute** - executes a given schema in a given state. Note that, according to the blocking view of CSP_Z (Section 2.5.1), if the precondition is not valid, the execution does not produce changes in the state.

It is worth mentioning that the tool is able to animate a subset of the Z language. Considering a state with one component defined as an integer, the following expressions can be evaluated by the animator:

- Relational expressions – expressions involving the operators $<, >, \leq, \geq, =$.

- Arithmetic expressions – expressions with the simple arithmetic operators $+, -, *, /, \%$ (modulus).
- Logic expressions – expressions involving the operators $\wedge, \vee, \neg$.
- Assignments – assignments applied to the state variable.

The CSP Animator

Analogously to the Z animator, this component gives support for executing CSP processes. First of all, the environment is configured by putting all process definitions into it. The component also contains the synchronisation interface and is able to make a process perform a specific event. Moreover, this component also gives the initial acceptances of a process. Below, we present its main functions:

- **configure** - receives all definitions of processes and put them into the environment. The processes are identified by a name.
- **initials** - returns the initial acceptances (*initials*) of a process.
- **canPerform** - determines if a given process can perform a given event.
- **perform** - returns the next behaviour of a given process by performing a given event. Note that the next behaviour of a process is also a process.

The CSP animator also presents some limitations. In this version, it is able to animate some constructs like prefixing, external choices, *STOP* and *SKIP*. Furthermore, external choices cannot produce nondeterministic behaviour, that is, the performance of an event in an LTS must produce only one following behaviour.

The CSP Stability Plugin

This component is responsible for determining if there is a cycle in the CSP part which repeats a specific property. For example, if before and after performing a trace s, a CSP_Z process has the same property, then this repetition can be a stable behaviour. Therefore, one has to check if there is a cycle on the CSP part whose transitions correspond to the trace s. To accomplish this task, we implemented a cycle analyser which performs such a checking for deterministic CSP processes. This component does not analyse non-deterministic processes because they require a more elaborate CSP execution engine. Indeed, in those processes, a single transition can produce more than one following behaviour, which must be analysed.

The Z Stability Plugin

As we have already mentioned in the previous chapter, the stability of the Z part is determined by theorem proving, such that a property representing the stable behaviour is built and then checked into an theorem prover. Recall from Section 3.4.2 that the stability property is described as the following logical formula:

$$\forall State, State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp',$$

where *comp* is the composition of schemas whose execution causes a CSP_Z property repetition.

In the current implementation this component simply builds the above theorem and requests to the user for checking it into the theorem prover Z-Eves [60].

4.1.4.2 Stability Plugin Factory

The role of this component is supplying the animator with specific implementations of components to analyse the stability of both parts (CSP and Z). Its implementation follows the pattern Abstract Factory (see Section 4.3) in order to provide flexibility for implementing different stability plugins. That is, our tool can interact with different theorem provers and model checkers. The default implementation of the Stability Plugin Factory component creates two default plugins that work as explained below:

- **Default CSP Stability Plugin** - is a component which receives a trace and then checks if the CSP part has a cycle performing it. The provided implementation only works with specifications whose CSP part is deterministic, in the sense that it cannot behave differently when performing the same event. For example, a processes like $(a \rightarrow P \,\Box\, a \rightarrow Q)$ cannot be analysed by this component. Furthermore, it is not able to deal with traces performed by combinations of cycles. For example, consider a CSP_Z process whose LTS for its CSP part is illustrated by Figure 4.7. Therefore, the performance of $\langle a, b, c \rangle$ is a stable behaviour of the CSP part.

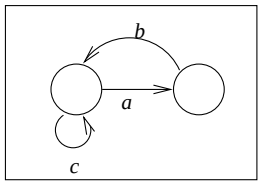


Figure 4.7: A CSP LTS with two cycles

- **Default Z Stability Plugin** - is a component which receives a trace and then checks if the Z part executes the corresponding schema composition forever. The default implementation of this plugin builds a temporary file containing the necessary schemas and the stability theorem ($\forall State, State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$). After that, the user has to check the validity of such a theorem into Z-Eves [60], and give the result (true or false) to the plugin.

Once we use the Abstract Factory pattern, the user can provide its implementation by supplying a new factory which instantiates two new stability plugins. For example, the stability of the Z part can be requested to another theorem prover like ACL2 [59] or PVS [63], and the stability of the CSP part can be analysed by interacting with the FDR [36] tool. Refer to [4] for details about implementing other stability factories.

4.1.4.3 Specification Abstractor

After the expansions have been done, the tool has to determine the abstract domain and build the abstract version for all operations.

Recall from Example 3.5 that the abstract domain is determined by taking the value of the variable of the state from all nodes of the LTS, and the abstract version of an operation—as explained in Section 3.4.2—is obtained by observing the following items:

- If the operation belongs to a cycle, then its post-condition must produce the value observed on the next node.
- If the operation does not belong to a cycle, there is no need to abstract it, that is, its abstract version is identical to the original one.

The Specification Abstractor component achieves this task. It analyses the LTS generated by the expansion engine, and then builds the abstract domain and the abstract versions for all operations.

4.2 Screens, Dialogs and Components

In this section, we present all screens and dialogs which the user can interact with. Moreover, we give an explanation about their functionalities.

Main Screen

It represents the first view of the tool. Three components permit to interact with the tool: a menu bar, a tool bar and an editor panel (see Figure 4.8). We explain each of them separately.

Menu Bar

The Menu Bar presents options for manipulating files, configuring the tool, and accessing help.

Menu *File* contains the following options:

- *New* – creates a new CSP_Z specification with the basic structure accepted by the tool. The specification contains an empty CSP part and the schemas State and Init.

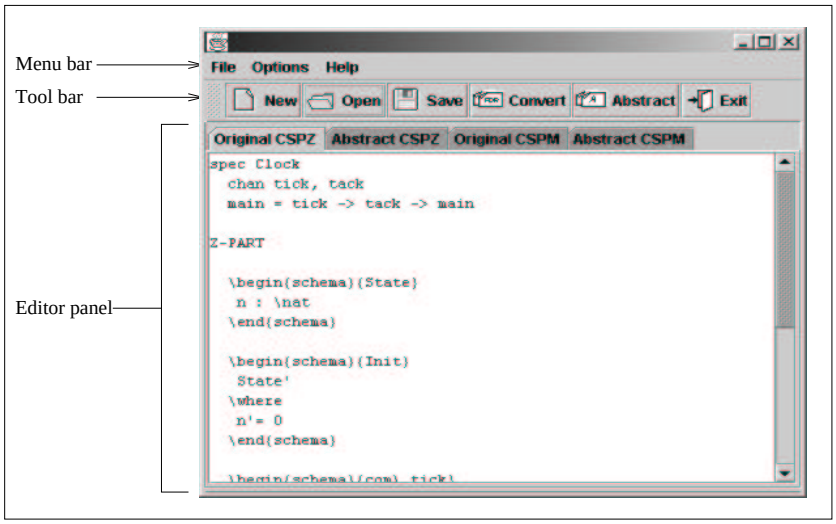
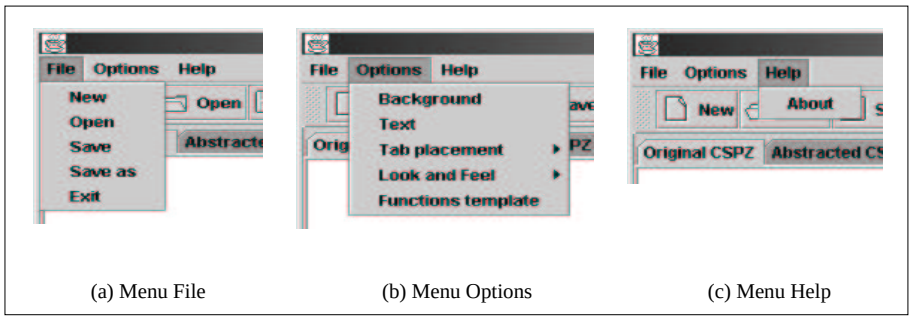


Figure 4.8: Main screen



(a) Menu File (b) Menu Options (c) Menu Help

Figure 4.9: Menu Bar

- *Open* – loads a specification from the file system.
- *Save* – saves the current specification into a file.
- *Save as* – saves the specification into a file chosen by the user.
- *Exit* – closes the application.

Menu *Options* presents the following functionalities:

- *Background* – permits the user to change the background colour of the editor panel.
- *Text* – permits the user to change the text colour of the editor panel.
- *Tab Placement* – permits the user to change the placement of the tabs of the editor panel. They can be: top, right, left or bottom.

- *Look and Feel* – permits the user to change the appearance of the graphical interface of the tool by changing the look and feel supported by the platform. The tool provides three defaults look and feel: metal, motif and windows.

- *Functions Templates* – exhibits a dialog which permits the user to write (or modify) simple functions that are used by the final CSP_M. Figure 4.10 shows such a dialog. To modify the definition of a function, the user must choose one item from the list on left and press the button *Edit*. The panel on right and the text fields on the top become editable. After the change is made, the user must press the button *Accept*.

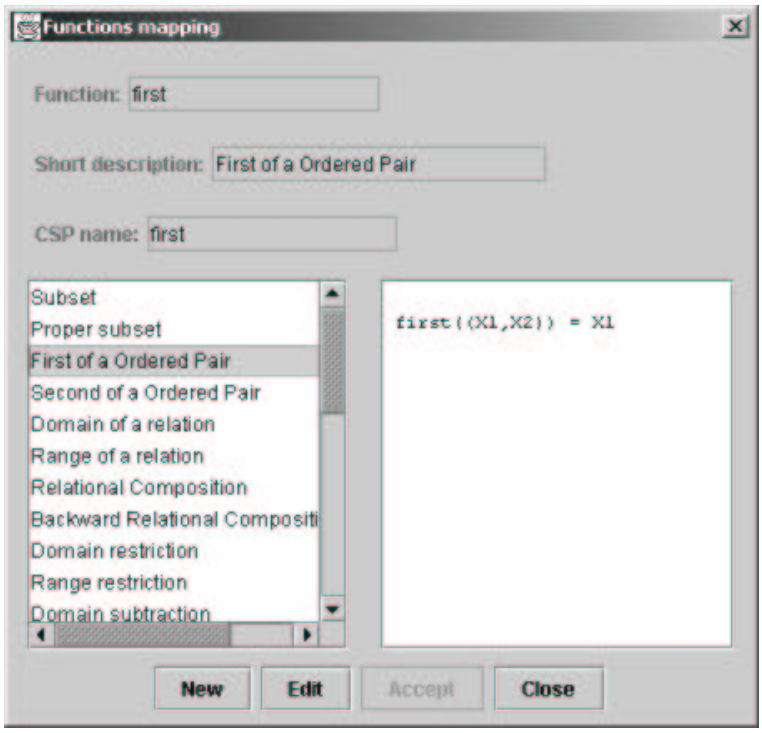


Figure 4.10: Dialog for editing functions

Menu *Help* presents permits the user access the system help:

- *About* – shows a dialog containing information about the tool.

Tool Bar

It is a component which permits the user to interact with the tool without accessing menus. Furthermore, some functionalities like abstracting a CSP_Z specification or generating its CSP_M code are not accessible from menus (see Figure 4.11).

The functionalities of the buttons are listed below:

- *New* – creates a new basic specification.

Figure 4.11: Tool Bar

- *Open* – opens a specification from the system file.
- *Save* – saves the current specification.
- *Convert* - converts the current CSP_Z specification (original or abstract) into CSP_M. Note that, if one of the CSP_Z tabs is selected, this button is enabled; otherwise it is disabled.
- *Abstract* – this option is enabled only for the original CSP_Z specification. It applies the data abstraction technique. First of all, the tool performs a syntactical analysis and then checks whether the specification is partially data independent or not. Afterwards, it abstracts the process by applying the technique explained in the previous chapter. Recall from Section 4.1.4 that, when determining the stability property, the tool uses two auxiliary plugins. Based on the trace causing the property repetition, the default plugin for the CSP part determines if there is a cycle performing it, whereas the default plugin for the Z part builds a theorem and the user has to analyse it into a theorem prover. Figure 4.12 shows the dialog box presented to the user, who has to copy and paste the content into a file, and then analyses the theorem named *ZStability* into Z-Eves [60]. Depending on the answer from the theorem prover, the user press *Yes* or *No*, informing whether the Z part is stable or not, respectively.

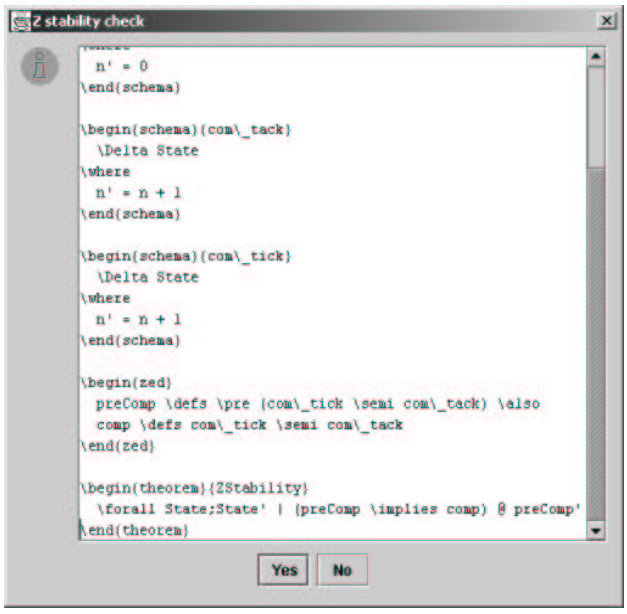


Figure 4.12: Dialog for the stability of the Z part

- *Exit* – closes the application.

Editor Panel

This component provides support for editing all files involved in the analysis of an infinite CSP_Z specification: the original specification, the abstract specification and their corresponding generated CSP_M codes (see Figure 4.13). It also provides popup menus for saving, checking the syntax, converting to CSP_M, abstracting, and editing specifications.

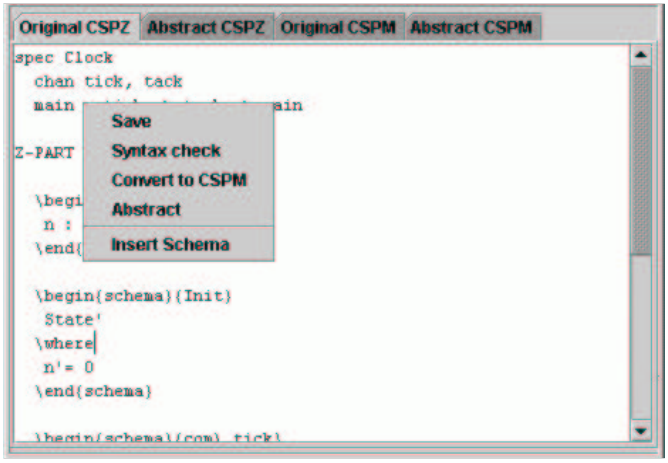


Figure 4.13: Editor panel

The options of Popup Menu are:

- *Save* – saves the current specification. This option is enabled for all tabs.
- *Syntax check* – parses a CSP_Z specification (original or abstract). This option is enabled only for CSP_Z specification. If some error occurs during this task, the editor panel shows a message panel on the bottom (see Figure 4.14), with an information about the error, and selects the corresponding line in the specification (if such an error concerns syntax).
- *Convert to* CSP_M – translates a CSP_Z specification into CSP_M. If the original specification is selected, this action sets the content of the *Original* CSP_M tab with the resulting CSP_M code. The conversion of the abstract specification sets the content of the *Abstract* CSP_M tab with its corresponding CSP_M code. This option is enabled only for CSP_Z specification tabs.
- *Abstract* – applies data abstraction technique similarly to the action of pressing the button *Abstract* on Tool Bar.
- *Insert Schema* – inserts a schema in the original specification from the point where the cursor is on. The definition of a schema, in LaTeX [58], is inserted.

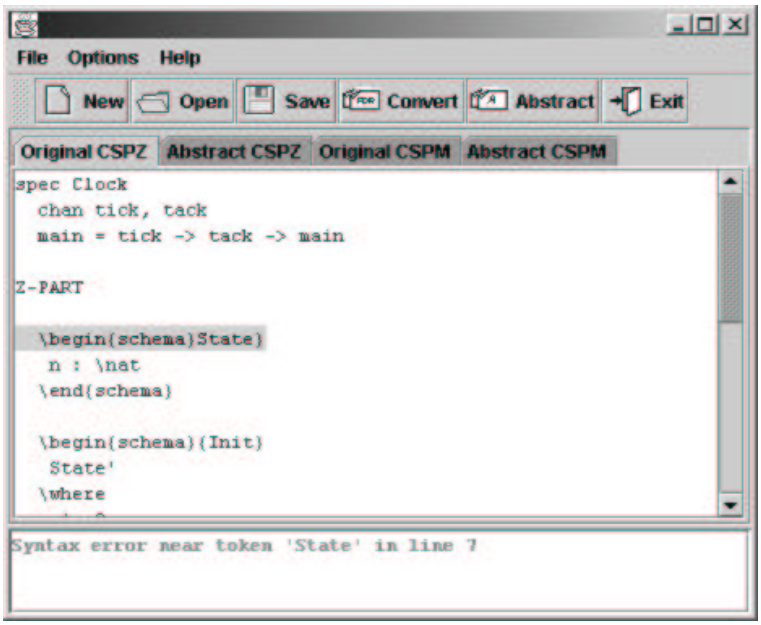


Figure 4.14: Error messages

4.3 Design Patterns

In order to provide a more reusable and flexible structure for the tool, we have adopted some design patterns [33]. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. In this section, we give an overview of the classification of design patterns and present those ones adopted in our implementation.

According to [33], design patterns for object-oriented structures are classified as follows:

- Creational patterns – abstract details about creation, composition and representation of objects. These patterns give flexibility in *what* gets created, *who* creates it, *how* it gets created, and *when*.
- Structural patterns – concerned with how classes and objects are composed to form larger structures.
- Behavioural patterns – concerned with algorithms and the assignment of responsibility between objects. They also describe patterns of communication between them. Moreover, they simplify the complex control flow, which is difficult to follow at runtime.

In the following we present the patterns used in our tool. For a detailed view of them and other patterns, refer to [33].

Abstract Factory

It is a Creational pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes. In other words, it makes implementations flexible by fixing its functionalities which will be accessed by clients. Moreover, it is applicable in the following cases:

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- Providing a class library of products by revealing just their interfaces, not their implementations.

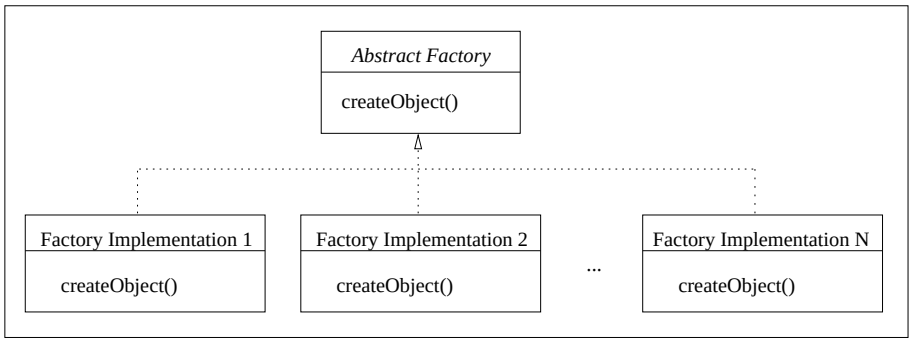


Figure 4.15: Abstract Factory pattern

Figure 4.15 illustrates this pattern. The Abstract Factory is a class containing the signature of the method to be implemented by any factory. The actual implementations must provide their specific manner of creating objects.

This pattern was used in our implementation to provide flexibility for interacting with different theorem provers. Recall from Section 4.1.4 that, when determining the stability property, the CSP_Z Animator component requests the Stability Plugin Factory to instantiate two plugins. As this component follows the Abstract Factory pattern, the user is free to implement different kinds of stability plugin factories. It is required only that such an implementation have to implement two methods: `createCSPStabilityPlugin()` and `createZStabilityPlugin()`. Furthermore, Section 4.4 explains how to choose a specific factory.

Mediator

Although partitioning a system into many objects generally enhances reusability, the large number of interconnections tends to reduce it again by causing functional dependence (one object requires the support of others). Worrying about concentrating these interactions

into one component, the pattern Mediator defines an object which encapsulates how a set of objects interact. Therefore, it promotes a loose coupling by keeping objects from referring to each other explicitly, and permits to vary their interaction independently. It is applicable in the following cases:

- A set of objects communicating in well defined but complex manners. The interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.

The Mediator pattern establishes a central component which achieves all communications when necessary (see Figure 4.16). Instead of making each component directly refer to the others, all interconnections are maintained into *Mediator*, which is responsible for notifying all components involved in the communication.

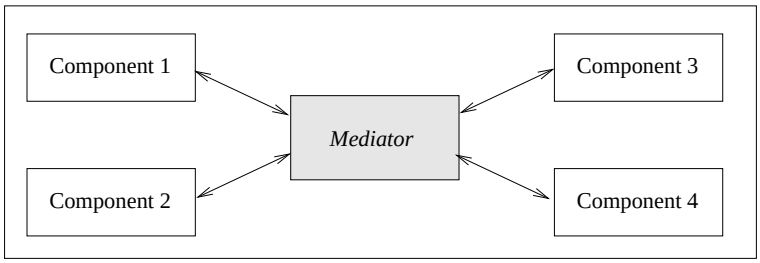


Figure 4.16: Mediator pattern

In our tool, this pattern was employed in a component called GUI Manager in order to avoid mutual references into some graphical components (Menu Bar, Tool Bar, Editor Panel and Popup Menu). For example, when editing the abstract CSP_Z specification, button *Abstract* from the Tool Bar and option *Abstract* from the Popup Menu must be disabled. GUI Manager holds all communications between components and changes their states when necessary (see Figure 4.17).

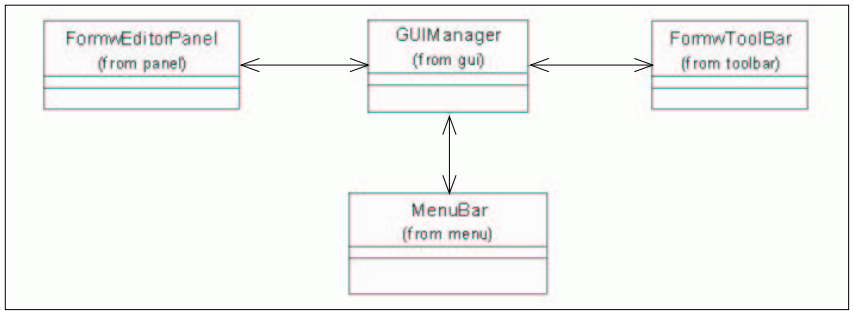


Figure 4.17: The use of the Mediator pattern

Observer

This pattern describes how to establish relationships such that objects (observers) have to be notified when some event or change happens in a source component (subject). Unlike the *Mediator* pattern, *Observer* defines a one-to-many dependency between objects so that when one object changes the state, all its dependents are notified and updated automatically. Figure 4.18 illustrates a simple example of a HTML editor, where two viewers of the same structure are provided: a HTML viewer and a source viewer. If the content changes by editing the source, the other viewer should be notified (and vice-versa).

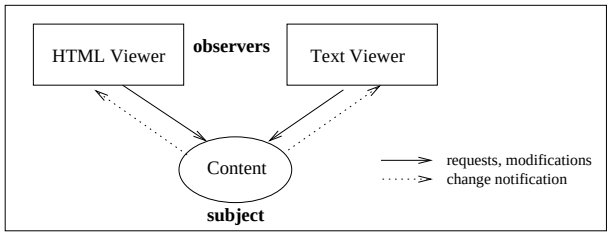


Figure 4.18: Observer pattern

The Observer pattern is applicable in the following situations:

- When a change to one component requires changing others, and we do not know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

The accordance of the Event Model of Java with this pattern made its use rather straightforward. It is employed between the GUI Manager, the main graphical components (Menu Bar, Tool Bar, Editor Panel and Popup Menu), and the File Manager component. We have created events for establishing relationships between these components. When an event happens in Editor Panel (for example, the tab *"Original CSP_Z"* is chosen), Tool Bar and Popup Menu must activate the option for abstracting and converting to CSP_M. As the communications are delegated to GUI Manager, it must be an observer of specific events happening in the Editor Panel. Figure 4.19 illustrates the use of this pattern. GUI Manager is a listener of events happening in several components. When such events happen, it is automatically notified.

Facade

This pattern is intended to provide a unified (higher level) interface to a set of interfaces in a subsystem, that is, instead of accessing a system through its several components, the facilities are grouped in one component which represents the whole system.

A good example of a system which makes use of this pattern is a compiler. It contains several components that provide the complex task of generating a program from a source

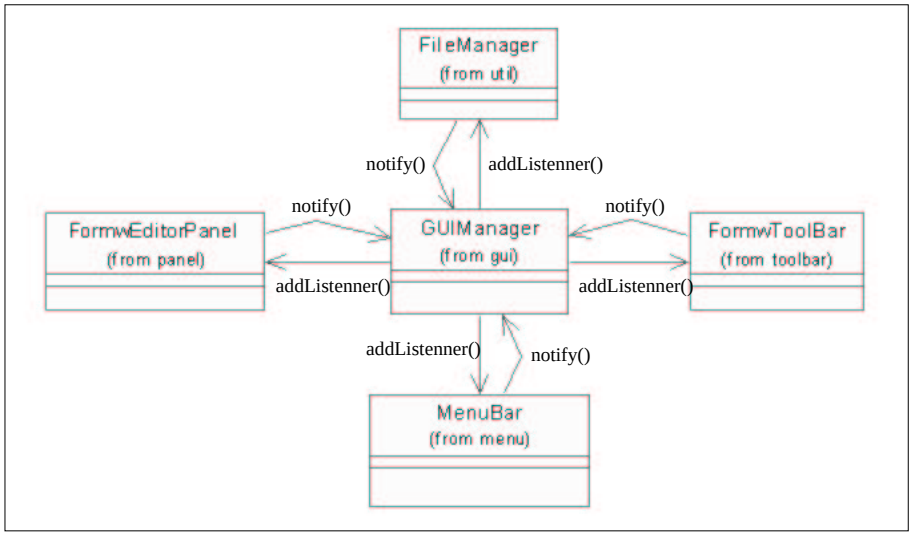


Figure 4.19: The use of the Observer pattern

code. Figure 4.20 illustrates the structure of a simple compiler. The Scanner and the Parser components are responsible for achieving the lexical and syntactical analysis, respectively. The Type Checker is useful to analyse semantical properties concerning type, and the Code Generator is used to produce a description (possibly a program) into a target language.

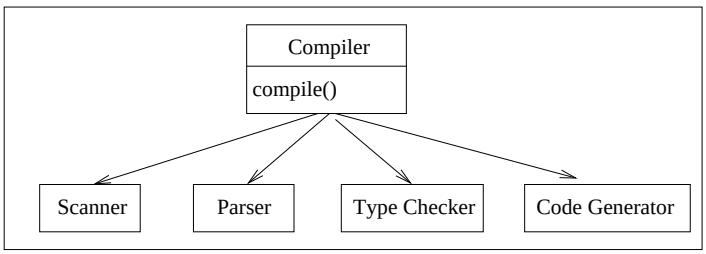


Figure 4.20: Facade pattern

The method *compile*() from the Compiler component executes the complex analysis which encompasses scanning, parsing, type checking and code generation. These facilities are not accessed separately. Instead, a component provides a method which simplifies the use of all components involved along the compilation.

The use of this pattern in our tool is straightforward. The facilities of animating and abstracting a CSP_Z specification are performed by a unique component, the CSP_Z Animator (see Figure 4.21).

Singleton

Some applications require the existence of exactly one instance of a class, providing a global access point to it. The *Singleton* pattern defines a manner of achieving this idea.

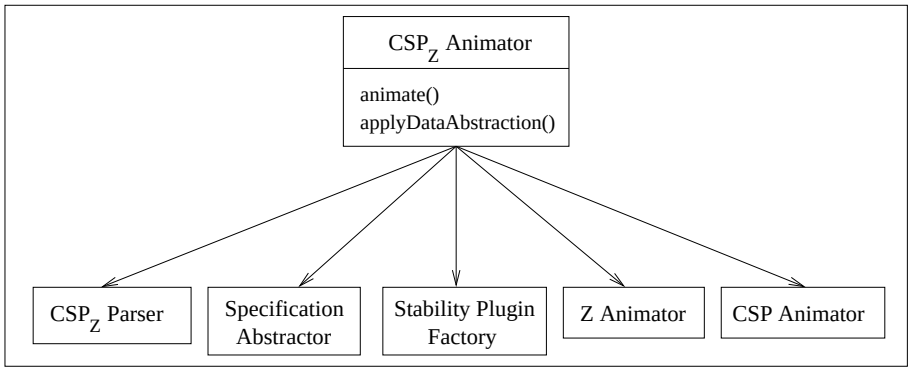


Figure 4.21: The use of the Facade pattern

Although it establishes a unique object to deal with many requests, it is suitable for implementing activities as concurrency control and consistency maintenance. Figure 4.22 shows an example. The component implementing the pattern must provide a manner to recover the unique instance of it (`getInstance()`). Clients that wish to interact with such an instance, must first request it from the singleton component.

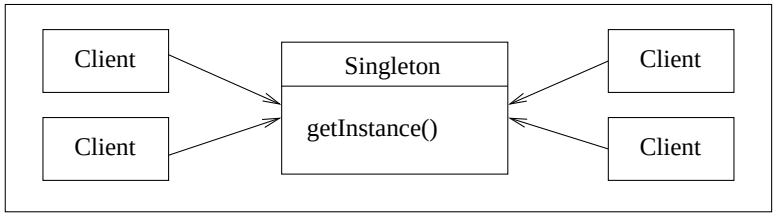


Figure 4.22: Singleton pattern

The use of this pattern is justified by defining a unique instance of the GUI Manager and File Manager components. Furthermore, the CSP_Z Animator also follows this pattern because the execution environment must be unique (see Figure 4.23). Instead of providing environments for different specifications loaded by the translator, the animator cleans its environment and configures it again. This makes the implementation simpler because it avoids the management of different animators and environments.

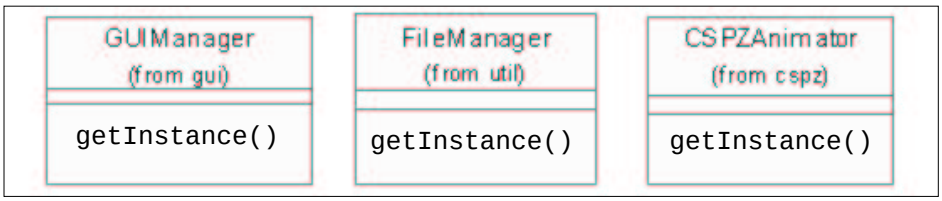


Figure 4.23: The use of Singleton pattern

4.4 Configuration

The tool also permits the user configuring some parameters. This is achieved keeping such parameters into a property file. Table 4.1 shows the properties used by the tool and gives a brief explanation about them.

The tool was designed using Rational Rose [71], and implemented with JBuilder [18]. Table 4.2 shows the number of classes for the main modules of the tool. Furthermore, the whole source code contains 34,982 lines. Excluding the parsers and scanners, it remains 20,582 lines.

Property Name	Description
mappingFileName	Indicates the name of the file containing the functions used by the generated CSP_M. Those functions are recorded using Java Serialisation [27]. The default value of this property is *mapping.serial*. We do not advise changing this property.
cspmFunctionsFileName	Indicates the name of the CSP_M functions library file. Its default value is *functions.csp*.
stabilityPluginFactory	The name of the stability plugin factory. The user can change this property as long as the other factory is provided. The property value must be the name of a Java class. Its default value is *formw.factory.DefaultStabilityPluginFactory* .
specificationTemplate	The path of a template file containing the basic content of a CSP_Z specification. Its default value is *resources/specification.tpl*. Its content is put into the editor panel when the button *New* is pressed. The user can provide another template file or change its content.
backgroundColorRed backgroundColorGreen backgroundColorBlue	Indicates the background colour of the Editor Panel. Each of them contains a number between 0 and 255 and can be configured directly from the menu *Options* → *Background*.
textColorRed textColorGreen textColorBlue	Indicates the text colour of the Editor Panel. Each of them contains a number between 0 and 255 and can be configured directly from the menu *Options* → *Text*.
tabPlacement	Indicates the position of the Editor Panel tab. It can be configured directly from the menu *Options* → *Tab Placement*.

Table 4.1: Properties of the tool

Module	Number of Classes
Data Independence	2
Data Abstraction	117
CSP_Z Parser	12
Translator	223
GUI components	24
Total	**378**

Table 4.2: Further information

4.5 Conclusions

In the previous chapter, we presented a mechanised manner of applying the data abstraction approach to infinite CSP_Z specifications, considering its behavioural part as well. Such an approach focuses on finding out a periodic behaviour of the whole process, that is, a behaviour in which the CSP and the Z parts perform the same sequence of events forever.

Providing supporting tools for formal techniques is a decisive contribution for their adoption in software development. Not only the supported technique must be well consolidated, it is crucial to provide a friendly user interface as well. In general, the user is interested in the results instead of the technique itself. In this chapter we presented a tool for our data abstraction approach[2]. The adoption of Design Patterns [33] during its development brought some advantages to the implementation: modularity, reusability, maintenance and extension facilities. We have adopted the patterns Factory, Mediator, Observer, Facade and Singleton. Particularly, the Factory pattern makes our implementation flexible to interact with several tools like FDR [36], Z-Eves [60], ACL2 [59], SMV [55] or PVS [63]. In this version we provide two components for determining the stability property: one for the CSP part and another for the Z one. The former analyses the LTS of the CSP part without interacting with FDR. In [3] we have proposed a manner of finding out a stable behaviour of the CSP part by using refinement between processes (verified into FDR). On the other hand, the stability component for the Z part interacts with Z-Eves by using a file-based strategy. Although the stability theorem is automatically built by the tool, the user has to save it into a file and then analyse it using the theorem prover. The tool also provides support for converting a CSP_Z specification into an equivalent CSP_M process. Furthermore, a configurable library of CSP_M functions is also provided.

Naturally, the tool presents some points to be improved. The most important among them concern the enrichment of the syntax accepted by the parsers and the execution of more complex terms of CSP_Z. In this version, the tool is able to execute only simple Z terms like relational and arithmetic expressions, and assignments. The CSP animator also presents some restrictions: it is not able to execute neither nondeterministic processes nor

[2]The tool is available for download at http://www.cin.ufpe.br/~acf.

indexed constructs. Furthermore, the tool is not able to deal with CSP_Z cycles which can be broken during the expansion by performing an event outside the cycle (see Example 3.11). As future work we intend to eliminate these limitations in order to make the tool able to support a larger class of specifications. Moreover, the treatment of problems like Example 3.11 requires a more sophisticated analysis of cycles and stability, which can be solved in future versions of the tool.

The main goal of the tool is CSP_Z data abstraction and model checking. The translation from CSP_Z into CSP_M is an elegant manner of extending the CSP model checking to CSP_Z, an integrated theory. However, some class of problems (infinite processes) cannot be dealt by this approach, because they present a large (or infinite) number of states—the *state explosion* problem. Therefore, Mota [7, 9] proposed applying data abstraction, a technique of compression based on inverse refinement[3], considering only the data part of a CSP_Z specification. We extended this approach by considering the behavioural part as well, and implemented a supporting tool. Therefore, we believe that the tool plays an important role in concurrent system development. Using a conversion strategy, it gives support for model checking; and using a compression technique (data abstraction) before the conversion, it permits to transform untreatable problems into feasible ones, in a mechanised manner.

Although the user assistance is required, the tool achieves CSP_Z data abstraction as automatically as possible. Furthermore, we have not found examples of tools implementing data abstraction for concurrent systems development. The work of Mota [7, 9] provided a prototype in Haskell [72] without a friendly user interface. The specification must be written using a format accepted by the prototype. There is also an intention of implementing the work of Lazić [76] into FDR. In general, the problem is abstracted by hand and then its abstract version is analysed by formal tools. Therefore, we believe that our tool represents an important practical contribution to concurrent systems development.

[3] A concrete type is replaced with an abstract one, while still preserving the properties of the system.

Chapter 5

Conclusions

The central contribution of this work is in the field of model checking [8, 10] and data abstraction [7, 9] for CSP_Z specifications. Model checking for CSP_Z processes originates from a strategy proposed by Mota and Sampaio [8] for converting a CSP_Z specification into a pure CSP [36, 13] process, thus allowing the use of FDR [36]—the standard CSP model checker—for analysing CSP_Z as well. In [1, 2] we report on an implementation of this strategy.

The *state explosion* problem—intrinsic to model checking—limits the class of problems to which the technique can be applied; several systems cannot be automatically verified because their state spaces are too large. Hence, impressive efforts focus on applying auxiliary compression techniques before applying model checking [7, 30, 75, 76, 34, 82, 28, 55, 69]: elimination of symmetry, abstraction, symbolic verification, partial order methods, local analysis, data independence, integration of tools. For CSP_Z, the approach used by Wehrheim [42] and Mota [7, 9] was data abstraction, a technique which permits to transform an infinite process into a finite one, while still preserving most of its properties. Wehrheim used this approach assuming informally that the CSP part cannot influence the Z one. However, Mota [7] observed—based on the work of Lazić [76]—that this assumption must be stated formally. The reason is rather simple: suppose that the CSP part is not completely free of data manipulation (that is, some data requirements are needed). Therefore, data abstraction on the Z part might affect the CSP one, and then some properties of the whole process are lost. Mota tackled this problem by using a combination of two theories: data independence [76] on the CSP part, and abstract interpretation [66, 67] on the Z one. The algorithm presented in [7] builds an abstract (finite) representation of an infinite CSP_Z process, based on the behaviour of the Z part. Further, Mota also cited the analysis of the CSP part as an important point for searching abstractions.

Our work has investigated this idea further and presented another data abstraction approach for CSP_Z. Recall from Chapter 3 (more precisely Definition 2.4) that the normal form of a CSP_Z process, after translation, suggests the CSP part as a master process—responsible for achieving control flow—and the Z part as a slave process—responsible for achieving data manipulation. This view permits to apply distinct techniques to these parts (see Figure 5.1). As the approach of Mota [7], ours abstracts a process by replacing

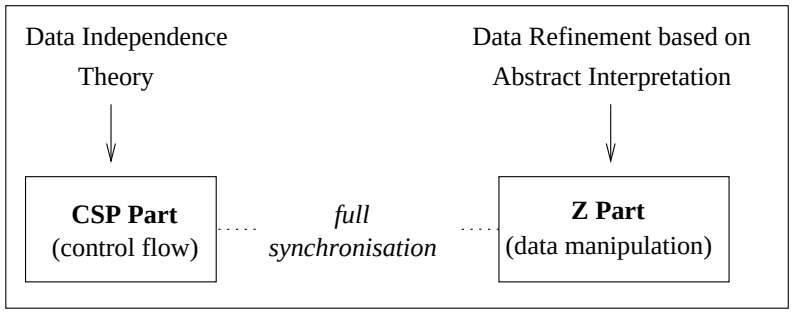


Figure 5.1: The use of distinct theories in CSP_Z data abstraction

infinite and stable (or periodic) behaviour with finite equivalent ones. The main difference concerns the CSP part. While our approach captures specific situations (like termination, deadlock and divergence), the other generates all possible stable traces of the Z part (even those refused by the CSP part), Therefore, our technique is more elaborate and thus can be applied to a larger class of problems; naturally, it requires a slightly more complex execution model.

In this work we have proposed an algorithm for our data abstraction approach. It is worth noting that our results coincide with those produced heuristically (see the *ad hoc* abstraction of Example 3.2). Like Mota's approach, the abstract domain produced by our approach is a subset of the original one, as well as the abstract process becomes easy to generate: only postconditions are modified (when necessary). However, our strategy for calculating schema abstraction is simpler because we build neither the abstraction function nor equivalence classes.

The way of building the equivalence classes used by Mota is not adequate in our approach. Because he uses a conjunction of preconditions, acceptances of the CSP part are not captured. Although we tried to use equivalence classes in our abstraction process, we reached a point where the user assistance was required to build them. Therefore, we adopted another notion of property of a CSP_Z process (see Section 3.4.2), which has brought the following advantages:

- **Reduction of the expansions**. As the CSP part is also considered, fewer possible abstraction are examined (instead of exploring all possibilities accepted by the Z part). The formal comparison is presented by Theorem 3.1.
- **Generalisation of the property**. We have observed that the property used by Mota is a simplified form of ours. Indeed, when the CSP part accepts all events from the Z one, our property reduces to Mota's one (see a detailed explanation in Appendix B.1).

Another contribution concerns convergence. Our algorithm is more efficient than Mota's one because it has a CSP-driven execution (only traces permitted by the CSP part are explored). Furthermore, it stops as soon as it discovers successful termination, divergence or deadlock on the CSP part.

Concerning periodic behaviour, our work follows the ideas of Pnueli [87] and Shankar [82] where model checking and theorem proving are viewed as complementary verification technologies. The former is effective for control-dominated systems, whereas the latter is suitable for data-dominated verification where the state spaces can be large or unbounded. During our data abstraction approach, theorem proving is used to guarantee the existence of periodic behaviour, whereas model checking is employed to analyse the resulting finite process.

It is worth pointing out that, although requiring user interaction to determine periodic behaviour through theorem proving[1], our data abstraction strategy discards any assistance when building the abstract process.

To give support for analysis of infinite CSP_Z specifications, we have developed a tool in Java [27] which implements our data abstraction approach. The tool is user friendly and also gives support for translating a CSP_Z specification into an equivalent CSP_M process [8]; it only requires knowledge of CSP_Z and user intervention when interacting with a theorem prover. The user writes the original specification and both its abstract version and its corresponding CSP_M code are generated automatically[2]. Furthermore, the adoption of Design Patterns [33] in the architecture of the tool provides flexibility for developers to implement components which interact with other tools (model checkers and theorem provers).

In summary, we have hopefully contributed to the state-of-the-art of the analysis of (possibly) infinite processes by extending a mechanised strategy, and to the application of formal techniques in concurrent systems development by providing tool support.

5.1 Related Work

In terms of techniques and formal approaches, there are several researchers focused on tackling this attractive challenge—controlling the state explosion problem. Many of them combine other techniques rather than abstraction and model checking. In the following, we list a few works that have similar directions to this one (under the viewpoint of abstraction).

- **Data abstraction for CSP_Z.** This work was strongly influenced by the work of Mota [7, 9], which proposes techniques for controlling the state explosion problem. The data abstraction approach presented in that work is based on two powerful theories—data independence [76] and abstract interpretation [66, 67]—and completely automatic. However, its data-driven characteristic (only the Z part is taken into account) limits its application to a specific class of problems, as already explained. Because our approach deals with the behavioural part as well, it is applicable to a larger class of problems.

- **Data abstraction for CSP_{OZ}.** Wehrheim [42] proposed a similar way for abstracting CSP_{OZ} specifications, however no consideration about data independence is made.

[1] When non-decidable logics [47] are used.

[2] During the conversion, the tool generates statements between $, indicating points of non-decidability. The user must edit those statements by hand.

Furthermore, her work does not present an algorithm to implement the strategy. The efforts of building the abstracted system is left to the user. In our approach we consider data independence aspects and present an automatic way of abstracting CSP_Z process.

- **Process abstraction based on value abstraction**. Cleaveland and Riely [73] have investigated a way of abstracting processes based on the abstraction of the values exchanged by them. To accomplish this, they defined a language similar to CSP [13] and presented their approach based on the semantics of that language. Although they mention their approach can be mechanised, neither algorithms nor tool support is provided.

- **Decomposition, abstraction and compositionality**. The work of Stahl [56] presents a strategy to overcome state explosion by using the data independence theory by Wolper [70], abstraction and compositionality. First, the original system—expressing *global* properties—is broken into subsystems—expressing *local* properties. Then, each subsystem is abstracted individually. Afterwards, these smaller parts are composed in order to build the whole abstraction. Although they use auxiliary tools (the SPIN model checker [38] and the PVS [63] theorem prover), the application of the strategy is not mechanised.

- **Predicate Abstraction**. A similar abstraction technique comes from the work of Graf [81]. With predicate abstraciton, the concrete states of a system are mapped to abstract states according to their evaluation under a finite set of predicates. The central idea of that work is building a *boolean program* abstraction, a program that has identical control structure to the original program, but contains only boolean variables. The results of this work have been used into the SLAM project[3], where a tool is employed to build abstractions of C programs.

Regarding tool support, we could not find, until the moment of writing, tools implementing techniques for abstracting processes. Although many works use auxiliary tools during the process, the user has to provide the inputs to them. Therefore, a great deal of experience and knowledge is strongly required from the user.

Mota [7] has presented a prototype in Haskell [72] implementing his approach. However, the user has to write the behavioural (CSP) and the data (Z) parts using a specific format accepted by the prototype. The implementation of a tool supporting the data abstraction strategy defined for CSP_{OZ} was mentioned by Wehrheim [42] as future work. However, no implementation seems to have been provided yet. We believe that our tool represents an important contribution concerning automatic support to model checking.

[3]Details about the project can be found at `http://research.microsoft.com/slam`.

5.2 Future Work

Providing a complete formalism and tool support for dealing with analysis of infinite systems requires an exhaustive and continuous effort. The automatic strategy and the tool support for CSP_Z presented in this work have some limitations. Therefore, we classify future work in two categories: improvements to the technique and enrichment of the tool.

Improvements to the Technique

Although our data abstraction approach has proved promising, it can be improved in several directions.

- **Treatment of multiple cycles**. As we have mentioned in Chapter 3 (more precisely in Example 3.11), when a cycle can be broken by performing an event outside it, the algorithm cannot abstract that cycle immediately. Instead, the expansions should continue until the sequence of properties along that cycle becomes stable. We have observed that this analysis requires a more elaborate study about the influence of the schemas inside the cycle upon schemas outside the cycle. Furthermore, we have also noticed influence of the initialisation: depending on the initial value of the state, a cycle can be broken or not. We therefore believe that this problem can be solved by theorem proving, where a stronger theorem should capture the possibilities of breaking cycles after initialisation.

- **Analysis of unstable processes**. The algorithm presented in Figure 3.12 achieves CSP_Z data abstraction for processes which exhibit periodic behaviour. For processes which never stabilises, the algorithm is non-terminating. We solved this problem by inserting (manually) non-determinism into the abstracted system. The natural extension of our approach in this case is to consider safe abstractions as well, in order to increase the class of analysable problems.

- **Communication**. The problem of dealing with communication events increases drastically the complexity of the data abstraction approach. For example, $c?x$ can represent an infinite set of events, depending on the type of x (see Figure 5.2). This requires new notions of property and equivalence between nodes in order to establish another concept of stability. This feature represents the biggest challenge for the technique because most of the real world problems involve communication.

Enrichment of the Tool

Apart from implementing the improvements suggested above, the support through the tool presented in Chapter 3 can be substantially improved. The following items certainly represent important contributions to make the tool more attractive to practical use.

Figure 5.2: Explosion of communication events

- **Syntax enrichment**. Both Z and CSP parsers implements a subset of those languages. Therefore, they can be extended to accept a larger range of constructs. In particular, the Z parser presents limitations concerning precedence of operators (for example, + and * have the same precedence). Further details about suggested extensions to both parsers can be found in [4].

- **Type checking and error handling**. The current version of the tool does not apply any strategy to recover from errors. It only shows them in a message panel. Furthermore, the type checking would increase substantially the quality of the tool because most of the semantic errors concern typing problems.

- **Animators enrichment**. In this version, it is possible to animate specific CSP processes and Z constructs. This improvement would permit the user to write specification with a richer structure (internal choice, domain/range restriction/subtraction, quantified expressions, lambda expressions etc).

- **Stability plugins library**. The flexibility for interacting with different formal tools is another aspect to be further explored. Instead of providing two plugins, the use of a library of plugins for interacting with model checkers (like FDR [36] and SPIN [38]) and other theorem provers (like ACL2 [59], SMV [55] and PVS [63]) would make the tool more attractive.

- **Debugging facilities and graphical viewer**. Debugging is an essential facility in systems development. It permits to inspect internal structures and discover subtle errors occurred during execution. This idea can be used to inspect the nodes generated during the expansion—in a step-by-step manner—of the process. Additionally, a graphical viewer can also be coupled to the tool in order to provide a more intuitive view of the expansion at runtime.

- **Support for dealing with local analysis**. Mota [7] has not only defined an automatic abstraction technique for CSP_Z processes, but also a strategy for decomposing large specifications into smaller ones, such that some properties like deadlock and livelock can be performed more efficiently. The support theory for such a strategy was investigated by Roscoe [25]. Martin [48] has also investigated techniques for analysing networks of processes interacting among themselves. Moreover, he has provided an

implementation in Java (the Deadlock Checker [49]) which interacts with FDR. The first change in our tool is to support many CSP_Z specifications. Afterwards, the integration with the Deadlock Checker can be made by a plugin which converts all (smaller) specifications into the notation accepted by that tool. Certainly, the possibility of dealing with smaller specifications (local analysis) is an important feature to be implemented.

Bibliography

[1] A. Farias, A. Mota and A. Sampaio. Um Conversor da Notação CSP_Z para CSP_M. *Revista Eletrônica de Iniciação Científica*, Brazilian Society of Computing (SBC), August 2001. Available at: *http://www.sbc.org.br/reic.*

[2] A. Farias, A. Mota and A. Sampaio. De CSP_Z para CSP_M: uma Ferramenta Transformacional Java. *In Proceedings of IV Workshop on Formal Methods* (*WMF2001*), pages 1 - 10.

[3] A. Farias, A. Mota and A. Sampaio. Efficient Analysis of Infinite CSP_Z Processes. *In Proceedings of V Workshop on formal Methods* (*WMF2002*), pages 113 – 128.

[4] A. Farias. *A Support Tool for CSP_Z Data Abstraction: reference guide.* Federal University of Pernambuco, Brazil, 2002. Available at: http://www.cin.ufpe.br/~acf.

[5] A. Galloway. *Integrated formal Methods with Richer Methodological Profiles for the Development of Multi-Perspective Systems.* PhD thesis, University of Teesside, School of Computing and Mathematics, 1996.

[6] A. Mota. *Formalização e Análise do SACI-1 em CSP-Z.* MSc dissertation. Federal University of Pernambuco, Brazil, 1997.

[7] A. Mota. *Model Checking CSP_Z: Techniques to Overcome State Explosion.* PhD thesis. Federal University of Pernambuco, Brazil, 2002.

[8] A. Mota and A. Sampaio. Model-Checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40: 59 – 96. Elsevier, 2001.

[9] A. Mota, A. Sampaio and P. Borba. Mechanical Abstraction of CSP_Z Processes. In L. Erikson and P.A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 163 – 183. Springer-Verlag, 2002.

[10] A. Mota and A. Sampaio. Model-Checking CSP-Z. In *Proceedings of the Europe Join Conference on Theory and Practice of Software*, volume 1382 of *LNCS*, pages 215–220. Springer-Verlag, 1998.

[11] *Annual Proceeding of the Computer-aided Verification (CAV) Conference*, Springer LNCS.

[12] *Annual Proceeding of the Computer-aided Deduction (CADE) Conference*, Springer LNAI.

[13] A. Roscoe. *The Theory and Practice of Concurrency.* Oxford University, 1998.

[14] A. Roscoe. Model Checking CSP. *In A.W. Roscoe (Ed.), A Classical Mind: Essays in Honour of C.A.R Hoare.* Prentice-Hall, 1994.

[15] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks, 1992.

[16] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP.* PhD thesis, University of Oxford, 1998.

[17] B. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[18] Borland. *JBuilder Reference Guide.* Available at: http://www.borland.com.

[19] C. Donnelly and R. Stallman. *Bison: the YACC-compatible Parser Generator.* Version 1.25. 1995. Available at: http://dinosaur.compilertools.net/bison/manpage.html.

[20] C. Fischer. Combining CSP and Z. Technical Report, University of Oldenburg, 1996.

[21] C. Fischer. CSP-OZ: a combination of object-Z and CSP. *In 2nd IFIP Internat. Conf. on Formal Methods for Open Object-based Distributed Systems* (*FMOODS'97*), Chapmam & Hall, London, 1997.

[22] C. Fischer. How to Combine Z with a process algebra. In A. Fett, J. Bowen, M. Hinchey, editors, *ZUM'98 The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 5 – 23. Springer-Verlag, 1998.

[23] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.

[24] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, K. Taguchi, editors, *Proc. 1st Internat. Conf. on Integrated Formal Methods* (*IFM*), pages 315 – 334. Springer-Verlag, 1999.

[25] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[26] C. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[27] C. Horstman and G. Cornell. *Core Java 2.* Sun Microsystems Press, volumes I and II, 2000.

[28] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. In Formal Methods in System Design, volume 6, pages 11 – 44. Kluwer Academic Publishers, Boston, 1995.

[29] D. A. Schmidt. Abstract interpretation of small steps semantics. *In Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 76 – 99. Springer-Verlag, 1997.

[30] D. Dams. *Abstract Interpretation and Partial Refinement for Model Checking.* PhD thesis. Faculteit der Wiskunde en Informatica, Technische Universiteit Eindhoven, 1996.

[31] D. Gries and F. B. Schneider. *A logical approach to discrete math.* Springer-Verlag, 1993.

[32] E. Berk. *JLex: A lexical analyser generator for Java.* Version 1.2. Departament of Computer Science, Princeton University, 1997.
Available at: www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html.

[33] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1998.

[34] F. Levi. A symbolic semantics for abstract model checking. *Science of Computer Programming*, 39: 93 – 123, 2001.

[35] F. Moller and P. Stevens. *The Edinburg Concurrency Workbench User Manual (7.1).* Laboratory for Foundations of Computer Science, University of Edinburgh, July 1999.

[36] Formal Systems (Europe). *FDR2 User's Manual*, 1997.

[37] Formal Systems (Europe). *PROBE User's Manual, version 1.25,* 1998.

[38] G. J. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5): 279–295, 1997.

[39] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1999.

[40] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C.B. Jones, P. Lucas, editors, *FME'97: Industrial Applications and Strenghtened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, Berlin, 1997.

[41] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83-01, Technische Universität Berlin, 1983.

[42] H. Wehrheim. Data Abstraction for CSP-OZ. In J. Woodcock and J. Wing, editors, *FM'99 World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1028–1047. Springer-Verlag, 1999.

[43] I. Toyn. *Z Notation: Consensus Work Draft 2.1.* Z Standards Panel, 1999.

[44] ISO. Information technology - Programming languages, their environments and system software interfaces - *Vienna Development Method - Specification Language.* International Standard ISO/IEC 13817-1, 1996.

[45] ISO. Information Processing Systems - Open Systems Interconnection - *LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* International Standard ISO/IEC 8807, 1989.

[46] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi and J. P. Jouannaud. *Introducing OBJ.* Software Engineering with OBJ: algebraic specification in action. Edited with Grant Malcolm, Kluwer, 2000, ISBN 0-7923-7757-5.

[47] J. G. Larrecq and I. Mackie. *Proof Theory and Automated Deduction.* Kluwer Academic Publishers, 1997.

[48] J. Martin. *The Design and Construction Of Deadlock-Free Concurrent Systems.* PhD thesis. University of Buckingham, 1996.

[49] J. Martin and S. A. Jassim. A Tool for Proving Deadlock Freedom. *WOTUG-20*, 1997.

[50] J. R. Levine, T. Mason and D. Brown. *Lex & Yacc.* O'Reilly & Associates, Paperback, 2nd edition, 1992.

[51] J. Woodcock and A. Cavalcanti. The Semantics of Circus. In Didier Ber, Jonathan P. Bowen, Martin C. Henson and Ken Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184 – 203. Springer-Verlag, 2002.

[52] J. Woodcock and J. Davies. *Using Z Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[53] K. Apt and D. Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *In Information Processing Letters*, 22 (6): 307–309, 1986.

[54] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[55] K. L. McMillan. *Getting started with SMV.* Cadence Berkeley Labs, 1999. Available at: http://www-cad.eecs.berkeley.edu/~kenmcmil.

[56] K. Stahl, K. Baukus, Y. Lakhnech and M. Steffen. Divide, Abstract and Model Check. *SPIN*, pages 57–76, 1999.

[57] L. Freitas. *JACK: a process algebra implementation in Java.* MSc dissertation. Federal University of Pernambuco, Brazil, 2002.

[58] L. Lamport. *LaTeX: a Document Preparation System. User's Guide and Reference Manual.* Addison-Wesley, 1995.

[59] M. Kaufmann and J. Moore. An Industrial Strenght Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4): 203 – 213, 1997.

[60] M. Saaltink. The Z-Eves System. *In ZUM'97: the Formal Specification Notation*, volume 1212, LNCS, Springer, 1992.

[61] M. Hennessy and H. Lin. *Symbolic Bissimulations.* Theoretical Computer Science 138 (2):353–389, 1995.

[62] M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 1992.

[63] N. Shankar, S. Owre, J.M. Rushby, D. W. J. Stringer-Calvert. *PVS Prover Guide.* Version 2.4, 2001. Available at: http://pvs.csl.sri.com.

[64] O. Grumberg, E. Clarke and D. Peled. *Model Checking.* The MIT Press, Cambrige, MA, 1999.

[65] P. Borba and S. Meira. From model-based specifications to functional prototypes. *IEEE TENCON'91 Session on Rapid Prototyping with Functional Programming Languages*, August 1991.

[66] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *In Conference Record of the 6th ACM Symp. on Principles of Programming Languages*, August, 1991.

[67] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4): 511 – 547, 1992.

[68] P. Kruchten. *The Rational Unified Process, An Introduction.* Addison Wesley, 2000.

[69] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proc. CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 233 – 146. Springer-Verlag, 1993.

[70] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Thrirteenth POPL*, pages 184–193. ACM, 1986.

[71] Rational Software. *Rational Rose Reference Guide.* Available at: http://www.rational.com.

[72] R. Bird. *Introduction to Functional Programming Using Haskell.* Second Edition. Prentice Hall, 1998.

[73] R. Cleaveland and J. Riely. Testing-based abstractions for value-passing systems. In J. P. B. Jonsson, editor, *CONCUR'94*, volume 836, pages 417 – 432. Spring-Verlag, Berlin, 1994.

[74] R. Duke, G. A. Rose and G. Smith. Object-Z: A specification language advocated for the description of standards. Computer Standards and Interfaces, 17: 511 – 533, 1995.

[75] R. Giacobazzi and F. Ranzato. Making abstract interpretations complete. *Journal of the ACM*, 47(2): 361 – 416, 2000.

[76] R. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking.* PhD thesis, Oxford University, 1999.

[77] R. Milner. A Calculus of Communicating Systems. In *Lecture Notes in Computer Science*, volume 92. Springer-Verlag, 1980.

[78] R. Milner. An Algebraic definition of simulation between programs. *In Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481 – 489. BCS, 1971.

[79] R. Pressman. *Software Engineering: a practitioner's approach.* 4^{th} edition. McGraw Hill, 1997.

[80] S. E. Hudson. *CUP User's Manual.* Georgia Institute of Technology, 1999.
Available at: http://www.cs.princeton.edu/~appel/modern/java/CUP.

[81] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[82] S. Rajan, N. Shankar and M. K. Srivas. An integration of model checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84 – 97. Springer-Verlag, 1995.

[83] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series, Prentice-Hall, 1995.

[84] V. Paxson. Flex: a faster scanner generator. Version 2.5. 1995.
Available at: http://dinosaur.compilertools.net/flex/manpage.html.

[85] W. J. Toetenal. *Model-Oriented Specification of Communicating Agents.* PhD thesis, Faculty of Mathematics and Informatics, 1992.

[86] W. Reisig. Petri Nets: An Introduction. In *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[87] Y. Kesten, A. Klein, A. Pnueli and G. Raanan. A Perfecto Verification: Combining Model Checking with Deductive Analysis to Verify Real-Life Software. In J. J. M. Wing and J. Davies editors, *FM'99-Formal Methods*, volume 1078 of *LNCS*, pages 173–194. Springer-Verlag, 1999.

[88] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.

Appendix A

CSP and CSP_M

In this section we put some important definitions used along this work and the main terms of CSP and CSP_M. For a detailed of this language, please refer to [13, 36].

A.1 Process Expressions

CSP	CSP_M	**Explanation**
$SKIP$	`SKIP`	Successful Termination
$STOP$	`STOP`	Deadlock
$a \rightarrow P$	`a -> P`	Simple Prefix
$a?x \rightarrow P$	`a?x -> P`	Input Prefix
$a!x \rightarrow P$	`a!x -> P`	Output Prefix
$a?x?y : A!v \rightarrow P$	`a?x?y:A!v -> P`	Multiprefix
$P \Box Q$	`P [] Q`	External Choice
$P \sqcap Q$	`P |~| Q`	Internal Choice
$P \backslash A$	`P \ A`	Hiding
$P \trianglelefteq b \trianglerighteq Q$	`if b then P else Q`	Conditional Choice
$P \trianglelefteq b \trianglerighteq STOP$	`b & P`	Boolean Guard
$P \parallel Q$	`P || Q`	Synchronous Parallelism
$P \mid\mid\mid Q$	`P ||| Q`	Interleaving
$P_X\|\|_Y Q$	`P [X||Y] Q`	Alphabetized Parallelism
$P \underset{X}{\parallel} Q$	`P [|X|] Q`	Generalized Parallelism
$P; Q$	`P ; Q`	Sequential Composition
$P >> Q$	`P [c <-> c'] Q`	Piping
$P(s)$	`P(s)`	Parameterisation
P_i	`P(i)`	Parameterisation
Chaos(A)	`CHAOS(A)`	Chaos Process
div	`div`	Divergent Process
P(f(s))	`let s'=f(s) whithin P(s')`	Local Declaration

A.2 Sets

`{1,2,3}`	Set literal
`{m..n}`	Closed ranges (from integer m to n inclusive)
`{m..}`	Open range (from integer m upwards)
`union(a,b)`	Set union
`inter(a,b)`	Set intersection
`diff(a,b)`	Set difference
`Union(A)`	Distributed union
`Inter(A)`	Distributed intersection (A must be non-empty)
`member(x,a)`	Membership test (x belongs to a)
`card(a)`	Cardinality
`empty(a)`	Check for empty set
`set(s)`	Convert a sequence to a set
`Set(a)`	Powerset of a set
`Seq(a)`	Set of sequences over a set (infinite if the set is not empty)
`{x1,...,xn | x<-a,b}`	Set comprehension

A.3 Sequences

`<>, <1,9>`	Sequence literals
`<m..n>`	Closed ranges (from integer m to n inclusive)
`<m..>`	Open range (from integer m upwards)
`s^t`	Concatenation
`#s,length(s)`	Length of a sequence
`null(s)`	Tests if a sequence is empty
`head(s)`	The first element of a non-empty sequence
`tail(s)`	`s = <head(s)>^tail(s)`
`concat(S)`	Join together a sequence of sequences
`elemen(x,s)`	Tests if an element occurs in a sequence
`<x1,...,xn |x<-s, b>`	Sequence comprehension

A.4 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.5 Extra

`(5,<1>,{3})`	Tuple
`let... within ...`	Local definitions
`\x1,...,xn @ e`	Lambda definition
`Int, Bool`	Simple types
`nametype n=e`	Abbreviation
`datatype n=e1|...|en`	Free type

A.6 Traces

The traces of a process is a set containing all possible sequence of events performed by a process. In the following we present some simple definitions of traces for the main CSP constructs.

A.6.1 $traces(STOP) = \{\langle\rangle\}$

A.6.2 $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in traces(P)\}$

A.6.3 $traces(P \,\Box\, Q) = traces(P) \cup traces(Q)$

A.6.4 $traces(P \sqcap Q) = traces(P) \cup traces(Q)$

A.6.5 $traces(P \trianglelefteq b \trianglerighteq Q) = \begin{cases} traces(P), & \textit{if } b \textit{ evaluates to true} \\ traces(Q), & \textit{otherwise} \end{cases}$

A.6.6 $traces(b \,\&\, P) = traces(P \trianglelefteq b \trianglerighteq STOP) = \begin{cases} traces(P), & \textit{if } b \textit{ evaluates to true} \\ \{\langle\rangle\}, & \textit{otherwise} \end{cases}$

A.6.7 $traces(\mu\, p.(a \rightarrow p)) = \{\langle a\rangle^n \mid n \in \mathbb{N}\}$

A.6.8 $traces(\mathbf{div}) = \{\langle\rangle\}$

The traces of a process present some important properties:

- $traces(P)$ is non-empty: it always contains the empty trace $\langle\rangle$
- $traces(P)$ is prefix-closed: if $s \hat{} t$ is a trace performed by P then at some earlier time, the trace s was also performed by P

A.7 Initials

The *initials* of a process is defined as the set of the events accepted by it in a specific context.

Definition A.1 (Initials of a CSP Process)

$$initials(P) = \{a \mid \langle a \rangle \in traces(P)\}$$

♢

The following items are results from the application of the of the above definition to the main CSP constructs:

A.7.1 $initials(STOP) = \{\}$

A.7.2 $initials(a \rightarrow P) = \{a\}$

A.7.3 $initials(P \,\Box\, Q) = initials(P \sqcap Q) = initials(P) \cup initials(Q)$

A.7.4 $initials(P \mid\mid Q) = initials(P) \cap initials(Q)$

A.7.5 $initials(P \lessdot b \gtrdot Q) = \begin{cases} initials(P), & \textit{if } b \textit{ evaluates to true} \\ initials(Q), & \textit{otherwise} \end{cases}$

A.7.6 $initials(b \,\&\, P) = initials(P \lessdot b \gtrdot STOP) = \begin{cases} initials(P), & \textit{if } b \textit{ evaluates to true} \\ \{\}, & \textit{otherwise} \end{cases}$

A.7.7 $initials(\mu\, p.F(p)) = initials(F(p)[F(p)/p])$

A.7.8 $initials(\mathbf{div}) = \{\}$

Appendix B

CSP_Z

This section contains proofs of some lemmas used in this work and a formal explanation about the comparison between the property of our approach and that one adopted by Mota. Moreover, it also contains the auxiliary functions used by our algorithm.

B.1 Property Generalisation

The property used by our approach is a generalisation of that proposed by Mota. To show this affirmation, we use the case where our approach produces the same expansions as that of Mota; that is, the CSP part accepts all events performed by the Z part. Therefore, $initials(P_{CSP}) = I$, where I is the synchronisation interface. Recall from Section 3.4.2 that the property of our approach is a conjunction of all acceptances and refusals of the whole process, that is

$$(\bigwedge_{ev_a \in acc} ev_a \in initials(P_{CSP_Z})) \wedge (\bigwedge_{ev_r \in ref} ev_r \notin initials(P_{CSP_Z})),$$

where acc and ref denote the acceptances and refusals of P_{CSP_Z}, respectively, $acc \cap ref = \varnothing$ and $acc \cup ref = I$.

Because $initials(P_{CSP_Z})$ includes information about the CSP and the Z parts (see Lemma 2.1), the expression $ev \in initials(P_{CSP_Z})$ can be rewritten to $ev \in initials(P_{CSP}) \wedge pre\ com\_ev$. Therefore, the whole expression becomes

$$(\bigwedge_{ev_a \in acc} (ev_a \in initials(P_{CSP}) \wedge pre\ com\_ev_a)) \wedge (\bigwedge_{ev_r \in ref} \neg(ev_r \in initials(P_{CSP}) \wedge pre\ com\_ev_r)).$$

Note that expressions $ev \in initials(P_{CSP})$ are trivially true because $initials(P_{CSP}) = I$. Then, simplifying the above formula we have

$$(\bigwedge_{ev_a \in acc} (true \wedge pre\ com\_ev_a)) \wedge (\bigwedge_{ev_r \in ref} \neg(true \wedge pre\ com\_ev_r)).$$

Now using simple rules from Propositional Logic the final result is

$$(\bigwedge_{ev_a \in acc} pre\ com\_ev_a) \wedge (\bigwedge_{ev_r \in ref} \neg pre\ com\_ev_r).$$

The above expression is exactly the property adopted by the previous approach, which considers only the Z part.

B.2 Proofs

Proof of Lemma 2.1

The proof is based on the normal form of a CSP_Z process. In order to simplify it, we present an auxiliary lemma stating the initial acceptances of the Z part of a CSP_Z process and then the initial acceptances of the whole process.

Lemma B.1 (Initials of the Z part of a CSP_Z Process) *Let P_Z be a process representing the Z part of a CSP_Z process whose synchronisation interface is I. Then,*

$$initials(P_Z) = \bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i\}. \qquad \diamondsuit$$

Proof. *The proof is based on the form of P_Z and definitions of initials presented in Appendix A.*

1. $$initials(P_Z) = initials \left(\begin{array}{l} pre\ com\_a_1\ \&\ a_1 \rightarrow\ P_Z^{com\_a_1} \\ \Box\ pre\ com\_a_2\ \&\ a_2 \rightarrow\ P_Z^{com\_a_2} \\ \ldots \\ \Box\ pre\ com\_a_n\ \&\ a_n \rightarrow\ P_Z^{com\_a_n} \end{array} \right) \qquad [by\ Definition\ 2.4]$$

The external choice can be rewritten to an indexed form:

2. $[\Leftrightarrow] initials(P_Z) = initials(\Box_{a_i \in I} pre\ com\_a_i\ \&\ a_i \rightarrow P_Z^{com\_a_i})$

Applying the definition of initials to the indexed form o P_Z we have:

3. $[\Leftrightarrow] initials(P_Z) = \bigcup_{a_i \in I} initials(pre\ com\_a_i\ \&\ a_i \rightarrow P_Z^{com\_a_i})$

According to Appendix A, $initials(b\&P) = initials(P)$, such that b evaluates to true, and $\{\}$ otherwise. Applying this definition to step 3 and adjusting the result to the set notation we have:

4. $[\Leftrightarrow] initials(P_Z) = \bigcup_{a_i \in I} initials(a_i \rightarrow P_Z^{com\_a_i})$, *such that* $pre\ com\_a_i$

5. $[\Leftrightarrow] initials(P_Z) = \bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i\}$

□

Now, we can really construct the proof of Lemma 2.1.

Proof. (Of Lemma 2.1)

1. $initials(P_{CSP_Z}) = initials(P_Z \underset{I}{||} P_{CSP})$ $[by\ Definition\ 2.4]$

2. $[\Leftrightarrow] initials(P_{CSP_Z}) = initials(P_Z) \cap initials(P_{CSP})$ $[by\ Lemma\ 3.3]$

3. $[\Leftrightarrow] initials(P_{CSP_Z}) = (\bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i\}) \cap initials(P_{CSP})$ [*by Lemma B.1*]

In the following steps we apply simple set operations:

4. $[\Leftrightarrow] initials(P_{CSP_Z}) = (\bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i\}) \cap initials(P_{CSP})$

5. $[\Leftrightarrow] initials(P_{CSP_Z}) = \bigcup_{a_i \in I} (\{a_i \mid pre\ com\_a_i\} \cap initials(P_{CSP_Z}))$

6. $[\Leftrightarrow] initials(P_{CSP_Z}) = \bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i \wedge a_i \in initials(P_{CSP})\}$

7. $[\Leftrightarrow] initials(P_{CSP_Z}) = \bigcup_{a_i \in I} \{a_i \mid pre\ com\_a_i \wedge a_i \in initials(P_{CSP})\}$

□

Proof of Lemma 3.3

Recall from Definition 2.4 that a P_{CSP_Z} process can be viewed as a generalised parallelism of two smaller processes. The process representing the CSP part (P_{CSP}) can be any (trivially data independent) CSP process and the process representing the Z part (P_Z) is an external choice of expressions, where each of them are guarded by its respective schema precondition. Moreover, recall from Corollary 3.3 that the progress of the whole process depends on the initial acceptances of P_Z and P_{CSP}. If they offer the same event (for example, e, such that $e \in initials(P_Z)$ and $e \in initials(P_{CSP})$), then both perform it. Otherwise, the whole process deadlocks.

We build the proof based on the analysis of P_{CSP} (it plays a major role and can be any CSP process). Instead of applying the proof on the structure of P_{CSP}, we apply induction on its initial acceptances as follows.

Proof. (Of Lemma 3.3)

Base Case: *$initials(P_{CSP}) = \varnothing$. This is the case when the whole process does not produce any visible event (that is, $traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\}$). In fact, we cannot distinguish if the CSP part has successfully terminated, is deadlocked or diverging (they have the same trace). We use one of these possibilities ($P_{CSP} = STOP$) to prove the base case.*

1. $traces(P_Z \underset{I}{||} P_{CSP}) = traces(STOP)$

2. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\}$ [*by definition of STOP in $\mathcal{T}$*]

From Section 2.1.4 we have that, for all process P, $traces(P)$ always contains the empty trace. Therefore, $traces(P) \cap \{\langle\rangle\} = \{\langle\rangle\}$. This permits us to rewrite the previous step to:

3. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = traces(P_Z) \cap \{\langle\rangle\}$ [*because $\langle\rangle \in traces(P_Z)$*]

5. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = traces(P_Z) \cap traces(P_{CSP})$ [*because $initials(P_{CSP}) = \varnothing$*]

Inductive Case: *let us consider* $initials(P_Z) \neq \varnothing$, $initials(P_{CSP}) \neq \varnothing$, $e_1 \in initials(P_{CSP})$, *and* $e_2 \in initials(P_Z)$. *Note that, if* $e_1 \neq e_2$, *the whole process deadlocks and, thus, the proof is similar to the base case. On the other hand, if* $e_1 = e_2$ *we need to assume that the lemma holds to processes* P'_Z *and* P'_{CSP}, *and try to prove for* P_Z *and* P_{CSP}, *such that* $P_Z = e_1 \rightarrow P'_Z$, $P_{CSP} = e_2 \rightarrow P'_{CSP}$.

1. $traces(P_Z \underset{I}{||} P_{CSP}) = traces(x : I \rightarrow P'_Z \underset{I}{||} P'_{CSP})$ [*by Corollary* 3.3]

2. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\} \cup \{x \frown s \mid s \in traces(P'_Z \underset{I}{||} P'_{CSP})\}$

[*by Definition* A.2.2]

3. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\} \cup \{x \frown s \mid s \in traces(P'_Z) \cap traces(P'_{CSP})\}$ [*inductive step*]

4. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\} \cup$
$\{x \frown s \mid s \in traces(P'_Z) \wedge s \in traces(P'_{CSP})\}$ [*by set theory*]

5. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\} \cup (\{x \frown s \mid s \in traces(P'_Z)\} \cap$
$\{x \frown s \mid s \in traces(P'_{CSP})\})$ [*by set theory*]

6. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\} \cup$
$(traces(x \rightarrow P'_Z) \cap traces(x \rightarrow P'_{CSP}))$ [*by Definition* A.2.2]

Because we have assumed that $P_Z = e_1 \rightarrow P'_Z$ *and* $P_{CSP} = e_2 \rightarrow P'_{CSP}$ *and* $e_1 = e_2$, *Step* **6** *can be rewritten to:*

7. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = \{\langle\rangle\} \cup (traces(P_Z) \cap traces(P_{CSP}))$ [*by assumption*]

8. $[\Leftrightarrow] traces(P_Z \underset{I}{||} P_{CSP}) = traces(P_Z) \cap traces(P_{CSP})$ [*by set theory*]

□

B.3 Auxiliary Functions

In this section we present some auxiliary functions used by the data abstraction algorithms (Figures 3.8 and 3.12).

extractProperty(State)

- State - the state used to evaluate the preconditions.

This function extracts the property used by the algorithm of Mota [7, 9], which considers as property a conjunction of enabled and dizabled preconditions. The function `eval` evaluates a precondition considering the given state.

```
extractProperty(State){
  result := true
  ∀com_op:Schema
    if(eval(pre com_op))
      result := result ∧ pre comp_op
    else
      result := result ∧ ¬pre comp_op
  ∀-end
  return result
}
```

extractProperty(State, LTS)

- State - the state used to evaluate the preconditions.
- LTS - a CSP LTS used to discover the acceptances of the CSP part.

This function extracts the property used by our algorithm, which considers as property a conjunction of the acceptances of the whole process in a context ($initials(P_{CSP_Z})$). The function `eval` evaluates a precondition considering the given state.

```
extractProperty(State, LTS){
  result := true
  ∀com_op:Schema
    if(eval(pre com_op) ∧ op ∈ initials(LTS) )
      result := result ∧ op ∈ initials(P_CSP_Z)
    else
      result := result ∧ op ∉ initials(P_CSP_Z)
  ∀-end
  return result
}
```

extractIndexOfRepetition(Property, NodeSequence)

- Property - a property.
- NodeSequence - a sequence of nodes.

This function returns the index of a given property in a sequence of nodes, where a node is the structure (State, LTS, Trace, Property, NodeSequence). The i^{th} element os a sequence s can be accessed by applying the sequence s to the integer i (that is, $s\ i$).

```
extractIndexOfRepetition(Property, NodeSequence){
  index := 0
  i := #NodeSequence
  notFound := true
  while (i > 0 ∧ notFound)
    if(fourth(NodeSequence i) = Property)
      index := i
      notFound := false
    fi
  while-end
  return index
}
```

extractTrace(Trace, i)

- Trace - a trace.
- index - the index from which the subtrace is extracted.

This function returns a subtrace of a trace from a specific index. As a trace is a sequence of events, the i^{th} element of a trace t is denoted by $t\ i$.

```
extractTrace(Trace, index){
  Trace result := ⟨⟩
  while (index < #Trace)
    result := result⌢⟨Trace index⟩
  while-end
  return result
}
```

Appendix C

Z-Eves Proofs

This section contains proofs of stability checked by Z-Eves [60]. The specifications correspond to the Z part of the examples from Chapter 3. Note that some predicates were not trivially reduced to true or false, however we give a comment in order to infer the final result.

Specification 1

The following specification corresponds to the Z part of Example 3.2, where $\langle a, b \rangle$ is infinitely performed by the whole process:

$State \mathrel{\widehat=} [c : \mathbb{Z} \setminus \{0\}]$
$Init \mathrel{\widehat=} [State' \mid c' = -1]$
$com\_a \mathrel{\widehat=} [\Delta State \mid c \leq -1 \land c' = -c]$
$com\_b \mathrel{\widehat=} [\Delta State \mid c > -1 \land c' = -(c * 2)]$

Stability: $\forall State;\ State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$, where $comp = com\_a \fatsemi com\_b$

Result: $c \in \mathbb{Z} \land c' \in \mathbb{Z} \land \neg c = 0 \land \neg c' = 0 \land (c \leq -1 \Rightarrow c' = 2 * c) \Rightarrow c' \leq -1$

Comments: Concentrating on the term $(c \leq -1 \Rightarrow c' = 2 * c)$, one can see that, when $c \leq -1$ the composition is enabled and its effect produces $c' = 2 * c$, that is, values on the range $c' \leq -2$. Therefore, $c' \leq -1$ is also (and ever) valid.

Specification 2

The following specification corresponds to the Z part of Example 3.4, where *com_a* and *com_b* are enabled all the time:

$State \mathrel{\widehat=} [c : \mathbb{Z} \setminus \{0\}]$
$Init \mathrel{\widehat=} [State' \mid c' = -1]$
$com\_a \mathrel{\widehat=} [\Delta State \mid c' = -c]$
$com\_b \mathrel{\widehat=} [\Delta State \mid c' = -(c * 2)]$

Stability: $\forall\, State;\ State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$, where $comp = com\_a \fatsemi com\_b$

Result: *true*

Comments: no comments.

Specification 3

The following specification corresponds to the Z part of Example 3.5, where $\langle a\ b \rangle$ is not a stable trace because of the Z part, and the trace $\langle c \rangle$ is stable:

$State \mathrel{\widehat=} [n : \mathbb{N}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid n \leq 5 \land n' = n + 2]$

$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \land n' = n + 1]$

Stability 1: $\forall\, State;\ State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$, where $comp = com\_a \fatsemi com\_b$.

Result 1: $n \in \mathbb{Z} \land n' \in \mathbb{Z} \land n \geq 0 \land c' \geq 0 \land (1 + n \leq 5 \Rightarrow n' = 3 + n) \Rightarrow 1 + n' \leq 5$.

Comments: Let us simplify the term $(1 + n \leq 5 \Rightarrow n' = 3 + n)$ to $(n \leq 4 \Rightarrow n' = 3 + n)$. Note that, when $n \leq 4$ the composition is enabled and its effect produces $n' = 3 + n$, that is, values on the range $n' \leq 7$. Comparing this result with $n' \leq 4$ (the simplified form of $1 + n' \leq 5$), one can see that $n' \leq 4$ is stronger than $n' \leq 7$. Therefore, the result is false.

Stability 2: $\forall\, State;\ State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$, where $comp = com\_c$.

Result 2: *true*

Comments: no comments.

Specification 4

The following specification corresponds to the Z part of Example 3.11, where both $\langle a, b \rangle$ and $\langle c \rangle$ are stable traces:

$State \mathrel{\widehat=} [n : \mathbb{N}]$

$Init \mathrel{\widehat=} [State' \mid n' = 0]$

$com\_a \mathrel{\widehat=} [\Delta State \mid n' = n + 1]$

$com\_b \mathrel{\widehat=} [\Delta State \mid n' = n + 2]$

$com\_c \mathrel{\widehat=} [\Delta State \mid n > 5 \land n' = n + 1]$

Stability 1: $\forall\, State;\ State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$, where $comp = com\_a \fatsemi com\_b$.

Result 1: *true*

Comments: no comments.

Stability 2: $\forall\, State;\ State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$, where $comp = com\_c$.

Result 2: *true*

Comments: no comments.