Advances in Space and Time Efficient Model Checking of Finite State Systems

Atanas Nikolaev Parashkevov

Thesis submitted for the degree of
Doctor of Philosophy
in
The University of Adelaide
(Faculty of Engineering, Computer and Mathematical Sciences)

November 2002

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The complexity of today's computer and electronic systems has reached a point where traditional approaches for design verification—simulation, testing, and reviews—are no longer sufficient. The cost of design defects slipping into production has also sharply increased.

Formal verification holds the promise of a more robust, complete and hopefully automated approach to ensuring that these systems implement their desired behaviour. However, a phenomenon known as state space explosion limits the size of systems that can be handled successfully. This thesis studies automated formal verification techniques and their associated space and time implementation complexity when applied to finite state concurrent systems. In particular, the focus is on concurrent systems expressed in the Communicating Sequential Processes (CSP) framework. The large body of work on CSP includes a process description language, a set of semantic models—traces, failures, and failures-divergences, as well as formal notions of process refinement and equivalence.

An approach to the compilation of CSP system descriptions into boolean formulae in the form of Ordered Binary Decision Diagrams (OBDD) is presented. This representation of CSP semantics is further utilised by a basic algorithm that checks a refinement or equivalence relation between a pair of processes in any of the three CSP semantic models. The performance of this algorithm is studied on a set of benchmark examples and a comparative analysis of its strengths and weaknesses in relation to explicit on-the-fly model checking is performed. This analysis identifies the major factors affecting the performance of OBDD-based refinement checking.

The performance bottlenecks of the basic refinement checking algorithms are identified and addressed with the introduction of a number of novel techniques and algorithms. Hierarchical partitioned transition relations provide an alternative to monolithic transition relations which may grow too large for certain problems. A pseudo-root state approach is developed to reduce the number of states that need to be stored during reachability analysis. A partial order technique—the sleep set method—is applied to OBDD-based refinement checking, which significantly reduces the verification run-time. The pseudo-root state

and sleep set methods are synergistically combined to obtain lower space and time requirements for reachability analysis than either of these techniques alone can provide. We utilise the full abstraction property of the CSP semantics to reduce a CSP refinement checking problem into a reachability problem that can be solved efficiently using OBDDs and enables the application of SAT-based and semi-formal verification techniques.

The algorithms described in this thesis are implemented in the Adelaide Refinement Checking tool.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

Should this thesis be accepted for the award of the Degree, I hereby consent to this copy, when deposited in the University Library, being made available for loan and photocopying.

Signed: **Date:** 19/11/2002

Acknowledgments

Looking back at the time that has passed since I embarked on the long journey of pursuing my PhD degree, I am amazed at the influence this has had on my professional and personal life. While I enjoyed most of the changes along the way, I have to acknowledge the fact that none of this would be possible without the help and encouragement of many people.

It would be very difficult to overstate the important role of my supervisor Dr Jay Yantchev in this endeavour. He has been instrumental right from the very first steps of applying for a PhD course at the University of Adelaide and getting a scholarship support for it. His advice and inspiration in the early days of my research have had a profound effect on my career and positioned myself in the rather interesting area of formal verification applications. I am grateful for his guidance, optimism, support, and friendship throughout my course.

I have spent the better part of the first three years of my course with the friendly staff and graduate students at the Department of Computer Science at the University of Adelaide. Of particular importance was the unlimited support of the Head of Department Professor Chris Barter and Tracey Young. Cheryl Pope offered her help and friendship literally from day one of my stay in Australia and cheered me up when I needed it most.

Professor Bill Roscoe provided inspiration and encouragement both directly in his communications with me and indirectly by supporting the CSP community and advancing the state-of-the-art in formal verification by leading the development of the FDR tool. His sabbatical in Adelaide in late 1997 allowed unique access to his wealth of knowledge and experience in the subject and deepened my understanding of CSP.

I would like to use this unique opportunity to thank my school teachers and university lecturers and assistants I have had back in Bulgaria for developing my ability to think and solve problems. Of all these, I would like to honour a few special ones: Svetla Madjarova, Georgi Demirev, Rositca Stefanova, Dr Anatoli Antonov, and Irena Nicheva.

Last, but not least, I have been fortunate enough to have the understanding and support of my wife Antoaneta, who had to put up with me not being around

on a great number of occasions.

My research was partly funded by an Overseas Postgraduate Research Scholarship generously provided by the Australian Government, a University of Adelaide Scholarship, and a Small Australian Research Council Grant.

Chapter 1

Introduction

Overview

This chapter provides an introduction to the notions of formal verification, model checking, and concurrent systems, followed by a discussion of the motivation, aims and achievements of the research presented in this thesis.

1.1 Formal verification in context

Today, computer and electronic systems have become pervasive and ubiquitous. They can be found in a wide range of devices and appliances from satellites to personal computers to breadmakers. The need to provide functionality for more and more sophisticated tasks results in an ever increasing complexity of both hardware and software. Complexity, on the other hand, brings the potential of a higher number of defects, or, as they are informally called, bugs.

In general, a defect is a failure of a system or product to perform or behave as expected. However, not all failures are equally undesirable—one may go unnoticed whereas another may have serious consequences. Also, not all applications are equally safety-critical—for example, a failure of the ABS system in a car is much more likely to lead to a catastrophe and endanger human lives (not to mention the negative publicity and financial repercussions for the car manufacturer) than a failure of a home appliance.

The above is not to say that defects should only be a consideration in safety- and life-critical applications. Economic factors are perhaps equally important and much wider aspects of the need to minimise product defects. In many cases, fixing a defect in a system after it is built is quite costly—for example, for hardware such as discrete chips or mobile phones. It is also well known that the earlier a

defect is found, the less expensive it is to fix [McC98, p. 28–30]. Thus, to ensure that a product is developed within the required time frame and budget, it makes a lot of sense to adopt defect counter-measures from the very early stages of its development.

In any case, defects cost money and sometimes even human life may be at stake. Thus, every effort is necessary to prevent bugs from slipping into production hardware and software systems.

In general, there are two ways of reducing defects: by using defect *prevention* or defect *detection* methods. Defect prevention measures are aimed at decreasing the probability of bugs occurring in the first place. In a commercial setup, for example, the company's culture, policies and practices as well as the employees' attitude and motivation may have a profound effect on the quality of its products. More specifically, adopting an appropriate production *process*, such as the Cleanroom Software Engineering Process [Hea94], the Capability Maturity Model [Dym95] and the Personal Software Process [Hum97], has been shown to lower, in some cases significantly, the average number of defects per unit of output.

Defect detection techniques target identifying the problems in order to eliminate them. Testing and simulation is one of the most widely used methods for defect detection. In this approach, the system (or, early in the development process, its prototype or simulation model) is subjected to a series of tests for which the expected behaviour is known in advance. The set of test cases is derived from the system specification; it is expected that each requirement in that specification is tested by at least one such test case. When a test uncovers a defect, "de-bugging" is performed to find the root cause of the problem and eliminate the bug.

Despite the wide application of testing and simulation, this approach has a major drawback—passing all tests does not imply that the system is defect free, because not all possible behaviours may have been exercised by the tests. How much testing or simulation is enough? Clearly, it is impractical and maybe even impossible to exhaustively test a moderately complex system under all possible scenarios. Even if exhaustive testing is feasible, it is non-trivial to ensure that all scenarios are indeed covered. Therefore, testing and simulation are incomplete methods for defect detection.

There are many practical cases demonstrating this deficiency of testing and simulation. One such case that has had most publicity and discussion is the floating point division bug in the early revisions of the Intel Pentium processor [MNS95]. It is safe to assume that Intel engineers had run hundreds of billions of simulation cycles and yet a sizable defect has slipped into production silicon costing the company an estimated half a billion US dollars in rev-

enues. Ironically, quite a few papers appeared after the incident which describe how it could have been prevented had another emerging methodology that provides complete coverage—*formal verification*—been used instead of simulation [LO95, CGZ96, RSS96, Bry96].

Formal verification [CJMW96, CK96] employs mathematical analysis techniques to prove system properties and correctness as opposed to the inherently inconclusive result of that system simply passing a set of tests. In the context of sequential programs, its roots can be traced back to the 1960's and the work of Floyd [Flo66] and Hoare [Hoa69], who introduced a technique that would later become widely known as *theorem proving*. This method utilises some form of mathematical logic to describe the system and its desired properties, as well as a set of inference rules to derive new knowledge (lemmas) about the system from its basic properties (axioms) and existing lemmas. A proof in this context is a sequence of inference steps (implications) starting with the description of the system and ending with the desired property.

Model checking[1] is another formal verification method first introduced by Clarke and Emerson [CGP99]. Its basic idea is to go a step further than informal verification by exhaustively enumerating all possible states in which a system can be and checking, on a case-by-case basis, whether the desired properties hold for every state. If the latter is true, then the system as a whole satisfies the property being checked; if it is false, this method can provide a *counterexample*—a sequence of state transitions leading to a state that violates the property. An extensive reference text on temporal logic model checking can be found elsewhere [CGP99].

Theorem-proving is the more general and thus the more powerful of the two formal verification techniques. Model checking is generally restricted to the domain of finite systems[2] whereas theorem-proving can handle (through induction or otherwise) infinitely rich behaviours. However, for theorem-proving to be successful in finding the right chain of implications (which involves a search in an enormous tree of possibilities), assistance from a highly skilled user fluent in both mathematics and the target domain area is imperative. Therefore, despite the large number of reported successes in academic papers [CJMW96], the circle of potential users of this technique appears to be somewhat limited. Moreover, if a user of a theorem-proving tool is unable to prove a certain property, it may mean any of the following:

- The user has not spent enough time on the problem;

[1] There appears to be some confusion in the literature as to what exactly the term "model checking" encompasses; we use it to mean any formal verification technique based on state space exploration of a system's model.

[2] There are cases when model checking can handle arbitrarily large systems—see [CGL92, RGG+95].

- The user does not have sufficient expertise to properly guide the proof process;
- The system contains a bug and thus the property is not true.

and there is hardly any indication as to which of the above is the case, how much more effort is required to complete the proof, or even what the chances are of finding one.

In contrast, model checking is a completely automated approach—its users need only supply descriptions of the system and its desired properties, run an automated tool utilising the appropriate algorithms, and analyze the results. In case complete verification is impossible due to resource constraints (space or time), probabilistic information about the correctness of the system can be gathered [Hol88]. The ability to provide a counterexample when a property check fails is a useful aid in debugging a problem. All this makes model checking attractive for a wide range of potential users and applications.

The advantages of formal verification in comparison to the traditional verification techniques are already acknowledged by the industry [Kur97]. Commercial tools targeted at electronic design, communication and cryptographic protocol verification and even small embedded systems software verification are being introduced at an ever increasing pace.

1.2 Verification of concurrent systems

Concurrent systems are composed of a number of autonomous sequential components which may interact with each other in order to fulfill a common purpose. They span a wide variety of practical applications, such as communication networks and protocols, distributed databases, control systems, operating systems, and many others.

Concurrent systems differ from sequential ones in a number of ways. Firstly, there are peculiar concurrent phenomena such as communication and synchronisation, deadlock, and others that do not exist in systems with a single control thread. Secondly, due to the interactive nature of concurrent systems they can rarely be considered outside of their intended environment which can itself exhibit complex and unpredictable (nondeterministic) behaviour. Thirdly, the more complex behaviour of concurrent systems results in a much larger state space that can, in the worst case, grow at an exponential rate to the number of autonomous sequential components.

The potentially enormous state space size is not the only measure of the complexity of concurrent systems. Even systems of moderate size may con-

tain major design flaws and defects that are hard to spot and expensive to fix. Holzmann [Hol92] presents an anecdotal example—a mutual exclusion algorithm of just 223 distinct states—which contains multiple defects despite being inspected and recommended to a customer by a "major US computer company". Lowe [Low96] has used the FDR2 model checker [For97] to analyse the Needham-Schroeder public key protocol and uncovered a security attack which, quite surprisingly, involves only six message exchanges. This flaw in the cryptographic protocol has gone unnoticed for 17 years since its introduction and, in the meantime, has been "proven" correct in the literature by several independent researchers [Low96].

This inherent complexity of concurrent systems makes their design much more difficult and error prone than that of sequential ones. The development of formal methods for concurrent systems has tried to alleviate these difficulties by putting specification, design and verification on a solid mathematical ground. As a part of this development, a number of process algebras, among them CCS [Mil89], CSP [Hoa85], and ACP [BK85], have evolved as a basis for concurrent process calculi.

Because model checking relies on an exhaustive enumeration of the state space of a model in order to verify a given property, its application to concurrent systems is complicated by a phenomenon widely known as a *state space explosion*. It arises from the fact that models of concurrent systems may grow in size and generation time exponentially with the number and complexity of their autonomous sequential components. As an example, consider a system with n components each with m possible states. Depending on the pattern of interaction, such a system could have anywhere between a single state (when the composition of the sequential components deadlocks immediately) to m^n distinct states (when each process executes with little or no synchronisation with the others).

1.3 State space and model checking complexity

In his seminal work on the CSP framework, Tony Hoare presents the following argument on concurrent system complexity and the ability of a computer to check for deadlock using the dining philosophers problem as an example [Hoa85, pages 79–80]:

> Let us consider an alternative proof method: program a computer to explore all possible behaviours of the system to look for deadlock. [...] In the case of a finite-state system like the *COLLEGE* it is sufficient to consider only those traces whose length does not exceed a known upper bound on the number of states. [...] Since each philosopher has

> six states and each fork has three states, the total number of states of the *COLLEGE* does not exceed $6^5 \times 3^5$, or approximately 1.8 million. [...] Since in nearly every state there are two or more possible events, the number of traces that must be examined will exceed two raised to the power of 1.8 million. *There is no hope that a computer will ever be able to explore all these possibilities.* Proof of the absence of deadlock, even for quite simple finite processes, will remain the responsibility of the designer of concurrent systems.

Hoare has concisely captured the fact that no matter how fast and advanced computers become, there will always be interesting problems that will not be amenable to automated verification techniques (such as model checking) because of their enormous state spaces. Some problems intractable by exhaustive verification may have an analytical solution. For example, it is relatively easy to come up with a counterexample that demonstrates the presence of a deadlock in a dining philosophers example with any number of philosophers.

On the other hand, the example that Hoare used to demonstrate his point—the five dining philosophers problem—nowadays is not being considered challenging at all. ARC—the tool developed as a result of this work—handles this example in less than a minute even with the basic pair-by-pair refinement checking algorithm (see Section 3.3). Furthermore, it has been shown that this particular problem can be solved for an arbitrary large number of philosophers in logarithmic time using a technique known as *state compression* [RGG+95]. Undoubtedly, the state-of-the-art in formal verification has advanced a lot since the early 1980s, however, this factor alone hardly explains why Hoare has been so over-pessimistic regarding the complexity of his example.

Our explanation is that Hoare has not taken into account the difference between the *apparent* size of the state space and its *actual* size. By apparent size we mean the size obtained by multiplying the sizes of the state spaces of the sequential components in a concurrent system. This size, as Hoare points out, is the upper bound of the actual state space size. As we have already pointed out in the previous section, the actual state space size can be anywhere between one state (when the system immediately gets into a deadlock) and the apparent size of the state space. The actual problem size can be smaller than the apparent one for many reasons, for example because of tight coupling of concurrent processes or the use of behaviour abstractions. In the case of dining philosophers, the actual problem size is constant with respect to the number of philosophers and forks once all unimportant events are hidden. Arguably, this is a rare and most convenient property that is successfully exploited by the state compression techniques present in the FDR2 tool [For97].

For many practical problems, the actual state space size is hard determine analytically, but can be obtained experimentally by reachability analysis. Thus, the *reachable* state space size is the more frequently used term in the literature.

To compare the capacity of the tools and techniques devised, researchers almost exclusively use a comparison in terms of the number of states that these tools can handle. However, few of them specify if these numbers are in terms of the apparent complexity of the problem at hand or the actual reachable size of its model. For example, the use of models that abstract away the irrelevant details of the actual problem can alone achieve many orders of magnitude reduction in the number of reachable states. There have been success reports for a wide range of problem complexity, from the relatively modest 10^{20} states [BCM+92] to the startling 10^{1300} states [CGL92]. Such a disparity of claims is quite likely to suggest that, in many cases, the apparent complexity of the initial problem being solved is cited instead of the reachable states of the model on which the technique actually operates.

There are also factors other than the size of the reachable state space of the model which make the comparison of different techniques applied to different problems hard to compare. One such factor is the complexity of the interaction pattern between the components in a system. The Synthetic example from Appendix A with its 2^{128} states is much easier to verify than the Solitaire example which has less than 200 million reachable states, or even the Monkey Puzzle example with its 23,000 reachable states. The latter two examples have a rather complex pattern of interaction between their components which models puzzle game rules, while Synthetic consists of a large number of processes with very simple behaviour and no interaction with the rest of the system.

One can make the logical conclusion that a realistic comparison of different techniques would require that they be applied to the same examples and within a similar context. Unfortunately, this is rarely possible, because:

- Research papers do not generally include a description of the examples used in sufficient detail as to reproduce them. Even when enough information is available, the effort in reproducing the examples may be quite significant;

- The input languages of the available formal verification tools are, in general, incompatible with each other. Although most tools are distributed with a set of examples demonstrating their use and capabilities, translating these into a different language may be non-trivial and requires expertise with both tools and languages.

The research results presented in this thesis represent a nice exception from the above problem, because there are two tools—one commercial (FDR2 [For97]) and

one academic (MRC [BMTV94])—that are based on the same language (CSP) and support a sufficiently close input syntax to that of ARC as to allow a comparative analysis of the new techniques proposed in the thesis. We have made a further attempt at avoiding the performance comparison pitfalls listed above by providing all examples referred to in the text in Appendix A. This includes, for each example, both a brief description of the problem at hand and appropriately annotated CSP scripts.

1.4 Research motivation and objectives

The work presented in this thesis was inspired by a demonstration of the FDR1 tool by Formal Systems (Europe) Ltd. [For93] in late 1994. Being able to relate two pieces of software was a genuinely fascinating concept to the author. A literature search on the subject that followed showed that automated formal verification, although only an emerging technology with little industrial application, was a very active research area.

The promises of formal verification as well as the challenges that were laying ahead in making it a viable option for solving real world problems presented just the right kind of mixture of significance of results, difficulty in problematics, and personal interest and curiosity of the author. Fortunately, there were and still are many significant opportunities for improvement, in both space and time, of the efficiency of state-of-the-art verification algorithms and tools.

At the time, temporal logic model checking techniques [BCM+92] using Ordered Binary Decision Diagrams (OBDDs [Bry86]) as a concise representation of the model were the focus of the research in the hardware verification community but had not received the same amount of interest in the concurrent systems verification area. It was a natural step to investigate the application of these techniques to the domain of concurrent systems. The CSP framework with its underlying algebraic language, semantic models and notion of process refinement was chosen as a foundation for this work due to a number of factors—the background of the author, the availability of a commercial tool (initially FDR1, then FDR2 as it became available in 1996) to be used as a reference point, and the large existing body of research in CSP.

Again in 1994, Barrett *et al.* published a paper presenting a tool called Milano Refinement Checker (MRC) [BMTV94], detailing an early attempt at applying OBDDs for refinement checking of CSP processes[3]. We have been able to obtain a copy of MRC from its developers. However, our initial excitement at the opportunity to observe the advantages that OBDDs have to offer quickly subsided after

[3]According to the authors of the MRC tool, it is a precursor to the FDR1 tool.

MRC had shown performance far below expectations [PY95]. It became clear that the use of OBDDs alone is no guarantee for success and that a more sophisticated application of the general OBDD-based model checking approach [BCM+92] is required to obtain the desired efficiency.

Consequently, the aims of this research are:

- To establish the viability of OBDD-based formal verification for a concurrent process algebra (CSP);
- To compare the OBDD-based refinement checking techniques with state-of-the-art explicit state space representation approaches that make use of advanced methods such as on-the-fly verification and state compression (both are available in the FDR2 tool);
- To provide a criterion that can reasonably predict the expected performance of the refinement checking algorithm developed based on properties of the problem models;
- To further develop reachability analysis and formal verification techniques that can handle larger systems than previously possible by developing a set of algorithms addressing the potential bottlenecks in model checking;
- To develop experimental implementations for the devised techniques in a research tool.

1.5 Achievements and contribution

This thesis presents the results obtained by the author in pursuing the aims outlined in the previous section. This section offers a summary of the thesis contents emphasising the relevant achievements and contribution.

Chapter 2 presents the background for this work. It outlines the CSP framework, defines the subset of the CSP algebra that is considered in the thesis, and introduces the reader to the notion of refinement and equivalence in three CSP semantic models. A definition of Labelled Transition Systems (LTS) and operations on them is followed by an introduction to the Ordered Binary Decision Diagrams (OBDD) data structure and algorithms. The chapter concludes with brief descriptions of several academic and commercial formal verification tools relevant to this work.

Chapter 3 is concerned with techniques and algorithms for deriving an LTS representation of CSP terms and compiling them into an OBDD form using a syntax-driven approach. This translation is shown to be more efficient than

existing methods and supports advanced features such as parameterised recursion, identifiers, expressions and channel input/output. A technique for re-encoding of the OBDDs of sequential components often reduces considerably the size of the final transition relation and is computationally inexpensive. OBDD-based algorithms for computing process refusals and divergences are presented. To overcome early OBDD size blow-up for the refusals relation the latter is kept in an implicitly conjunctive form. A basic pair-by-pair procedure for refinement and equivalence checking of compiled process semantics in any of the three CSP models is developed, and its performance on a range of examples presented in Appendix A is studied. We discuss the strengths and weaknesses of this algorithm compared to state-of-the-art explicit model checking, and elaborate on the most likely bottlenecks of the overall verification approach developed in the chapter. Each of these bottlenecks is addressed separately in the following chapters.

Chapter 4 addresses the danger of OBDD blow-up when compiling very large process descriptions. A non-monolithic form of OBDD-based LTS representation referred to as Hierarchical Partitioned Transition Relation (HPTR) is developed. The structure of the latter closely reflects the synchronisation patterns of the individual components (leaf processes) in the concurrent system. To operate on HPTR, the pair-by-pair refinement checking algorithm from Chapter 3 is enhanced with an algorithm performing recursive image computation over HPTRs. The trade-offs between time and space requirements offered by this approach are studied experimentally using its implementation in the ARC tool.

The need to store all reached states in a reachability analysis algorithm is alleviated in Chapter 5. This is achieved by selectively discarding states that cannot be reached from outside of the reached state space. Such states are called Pseudo-Root States (PRS). A modified reachability analysis algorithm that identifies and discards PRSs from the set of reached states as early as possible is developed, and its theoretical worst-, best- and average-case complexity is analysed. Again, the PRS method is implemented in ARC and is shown experimentally to provide a more than 16 fold reduction in reached state storage requirements.

Chapter 6 explores a different avenue to performance improvement—that offered by partial order methods. It introduces a method for the computation of the event dependency relation that is then used to implement the sleep set technique for reachability analysis which can greatly reduce the number of transitions to be traversed in state space exploration. Experiments with the ARC tool confirm the positive effect of using sleep sets—the total run-time for verification checks can be reduced by a factor of two to three. A reachability algorithm that combines the advantages of both pseudo-root states and sleep sets is developed and is experimentally shown to deliver much greater space savings than the one presented in Chapter 5. For one of the examples used, the peak number of stored states

grows linearly with the size of the problem (number of processes).

Most of the algorithms and techniques presented throughout this work are implemented in a research tool called Adelaide Refinement Checker (ARC). Chapter 7 offers a brief overview of the tool's design and architecture details, user interface provided in terms of command-line options, as well as a concise user guide.

Chapter 8 covers a number of topics for future research extending the work presented in this thesis. An application of full abstraction in CSP that allows the reduction of a refinement checking problem to a reachability problem is presented. This addresses one of the bottlenecks of the basic pair-by-pair refinement checking algorithm from Chapter 3—the fact that it operates on a single pair of states at a time. This technique relies on obtaining an observer process from the specification process in the refinement check and composing the observer with the implementation process to obtain a composite process with an unlabelled transition relation for which breadth-first OBDD-based reachability analysis is quite efficient. Other incremental extensions of our work such as algorithms for OBDD-based state compression similar to those implemented in the FDR2 tool [For97] and other partial order reduction techniques are discussed. Two new promising verification research areas—SAT-based formal verification and semi-formal verification—are also introduced in the context of the work presented in this thesis.

Chapter 9 reflects upon the major results of this research and lists other threads of research and development spawned from our work.

Chapter 2

Background

Overview

This chapter provides the necessary background information for the thesis. It outlines the CSP framework with its language, semantic models, and refinement and equivalence relations between processes expressed in CSP. This chapter presents a definition for nondeterministic LTSs and operations on them as well as an introduction to the OBDD data structure and algorithms. It concludes with brief descriptions of several verification tools relevant to our research.

2.1 The CSP framework

2.1.1 Overview

Communicating Sequential Processes (CSP) is an algebraic framework for concurrent systems originally conceived by Tony Hoare [Hoa78, Hoa85]. Since its inception, a considerable amount of research, refinements, extensions, and experience with CSP and its applications has been accumulated. Today, the CSP framework entails a well developed algebraic notation with a corresponding set of laws [Hoa85], a hierarchy of semantic models [Hoa85], a standard text-based notation [Sca92, Sca98], a formal theory of refinement, equivalence, and compositional verification [Ros94], and a commercial refinement checking tool [For97].

The simplicity, elegance, and notational power of CSP have had influence well beyond the boundaries of the concurrency research community. Examples of that are too many to mention exhaustively. The *occam* programming language [Inm88b] and its hardware implementation—the transputer machine instruction set [Inm88a]—evolved from the fundamentals of CSP. An asynchronous

hardware design language called the Delay Insensitive Algebra (DIA) [JU91] is also heavily influenced on CSP. The semantic models of CSP have been used to describe the operators of the industrial SDL language [EHS97].

A book by Roscoe [Ros97] provides a comprehensive view of the current status of CSP, covering twelve years of research since the definitive introduction of CSP [Hoa85]. Most of the CSP notation used throughout this thesis is adopted from Roscoe's book.

The notions of an *event* and *process* are central in CSP. Events are instantaneous, that is, it is assumed that they take no time to execute, and atomic, i.e. they have no structure. Events can be observed and their occurrence recorded as a sequence of events—a *trace*. Traces do not capture the potential simultaneity of event occurrence; if two or more events happen at the same time they can be recorded in any sequence (interleaving).

A process in CSP is an abstraction of *behaviour* (patterns of event occurrences). The externally observable behaviour of a process is the occurrence of a sequence of events. The set of all events that a process may ever perform constitutes the *alphabet* of that process.

There are two special events in CSP—τ denotes the invisible (silent) event, and $\checkmark$ (pronounced *tick*) is used to signal process termination. Although τ events are not externally observable, they play an important role in the operational semantics of CSP and specifically in the semantic definition of the hiding operator (see below). The $\checkmark$ event, on the other hand, is externally observable and its purpose is the communication of the successful termination of a process. To acknowledge the special nature of τ and $\checkmark$, neither of these events can be a member of a process alphabet. In other words, alphabets contain only visible events different from $\checkmark$ and τ.

Processes in CSP execute concurrently and can interact by participating jointly in the same event. An event can only occur if all processes which have to participate in it agree to perform it. Thus, CSP provides a synchronous (blocking) model of concurrency. A distinctive feature of CSP not present in many other algebras, e.g. CCS [Mil89] and ACP [BK85], is that it provides for more than two processes to participate in the same event.

Concurrency of process execution is captured through interleaving. As a consequence, any composition of concurrent processes can be reduced to a single nondeterministic process that exhibits the same behaviour as that set of processes.

The set of processes with which a process communicates constitutes its *environment*. For example, the environment of a vending machine is its customer; the environment of a dining philosopher are the forks on both sides [Hoa85]. In many cases, the environment of a process restricts the set of possible behaviours

that can be exhibited by that process, taking control over the choices offered by it. This is discussed in more detail in the next section.

2.1.2 Syntax and operators

Before we discuss the syntactical form of CSP processes, we need to introduce the following notations:

Notation 1 Events are denoted by $a, b, c, \ldots$, and sets of events are denoted by $A, B, C, \ldots$.

Notation 2 Processes are denoted by $P, Q, R, \ldots$.

Notation 3 The alphabet of a process P is denoted by $\alpha(P)$.

Notation 4 The universal alphabet is denoted by Σ. It does not include τ or $\checkmark$.

Syntactically, processes in CSP are formed from events and a set of basic processes and operators. This work considers the value-passing subset of CSP with some extensions, but not all derived operators such as $\triangle$ (interrupt). The syntactic constructs of this subset are defined as follows:

$$\begin{aligned} P \;=\; & \mathit{Stop} \mid \mathit{Skip} \mid a \rightarrow P \mid P \,\Box\, Q \mid P \sqcap Q \mid P \,;\, Q \mid \\ & P \setminus A \mid \mathit{ch}?x \rightarrow P(x) \mid \mathit{ch}!\mathit{expr} \rightarrow P \mid P \underset{A}{\parallel} Q \mid P \,|||\, Q \mid \\ & f[P] \mid \mu\, P \bullet F(P) \mid \text{if } \mathit{Cond} \text{ then } P \text{ else } Q \end{aligned} \tag{2.1}$$

where $P(x)$ denotes process P in which all occurrences of x are substituted by the value of x, $f : \Sigma \rightarrow \Sigma$ is an event renaming function, F is a continuous function between processes, and *Cond* is a boolean condition.

A process description formed by the above productions is called a *process term*. The replicated versions of operators $\Box$, $\sqcap$, and $\underset{A}{\parallel}$ are not included in Equation (2.1), because these operators provide merely a shortcut notation, i.e. syntactic sugar facilitating concise CSP descriptions. In what follows we provide a brief description of the basic processes and operators that form process terms:

Stop — This basic process can perform no event whatsoever. It is used as a model of *deadlock*—a process in a state in which no further progress is possible.

Skip	This basic process is used to model successful termination. It performs the special event ✓, and is normally used in conjunction with the sequential composition operator (see below).
$a \rightarrow P$	Prefix operator. This process performs event a and then behaves as P.
$P \,\Box\, Q$	External (deterministic) choice operator. This process behaves as either P or Q, depending on the first event offered by the environment. If the first event can be performed by P only, then P is chosen for execution. If the first event can be performed by Q only, then $P \,\Box\, Q$ behaves as Q. If both P and Q can perform the first event provided by the environment, the choice between them is nondeterministic.
$P \sqcap Q$	Internal (nondeterministic) choice operator. This process behaves as either P or Q with the choice between them made internally (invisibly) without the control, participation or knowledge of the environment.
$P \,;\, Q$	Sequential composition operator. This process behaves as P until P terminates (performs the ✓ event), and as Q afterwards.
$P \setminus A$	Hiding operator. This process behaves as P but with the events from A are concealed (hidden) from its environment, and cannot be observed when they occur. Note that ✓ cannot be hidden. This operator abstracts away the events in A from the observable behaviour of P. Thus, $P \setminus A$ generally has a simpler observable behaviour than P.
$ch?x \rightarrow P(x)$	Channel input. This process waits for data on channel ch, binds the identifier x with the input value, and then behaves as $P(x)$, e.g. as process P in which x is substituted with the input value.
$ch!expr \rightarrow P$	Channel output. This process outputs the value of the expression $expr$ on channel ch and then behaves as P.

$P \underset{A}{\parallel} Q$	Parallel composition operator. In this process, P and Q execute concurrently while synchronising over the events in A.
$P \mid\mid\mid Q$	Interleaving operator. In this process, P and Q execute concurrently without synchronisation. For all intents and purposes this is the same as $P \underset{\emptyset}{\parallel} Q$.
$f[P]$	Renaming operator. This process executes the event $f(a)$ whenever the process P can engage in the event a. The renaming function does not affect $\checkmark$ events. In most practical uses of renaming, $f : \Sigma \rightarrow \Sigma$ is an injective (one-to-one) function [Ros97, p. 87]. Note that the hiding operator is a special case of process renaming.
$\mu P \bullet F(P)$	Recursion operator. This process is the solution to the equation $P = F(P)$, where $F(P)$ is a process term using P in its definition. In practice, process definitions often make use of parameters (further discussed in Section 3.2.2), but we ignore this here for conciseness of presentation.
if *Cond* then P else Q	Conditional statement, in which *Cond* is a boolean expression over variables whose value has been set either through channel input or recursion. This process behaves as P if *Cond* is true, or as Q if *Cond* is false.

Constructing a model of a realistic concurrent system in CSP usually involves the formulation of a number of processes that are later combined to form a single system. For this reason, CSP tools like FDR [For97] read CSP *scripts* which contain a collection of possibly recursive process definitions in the form[1]:

$$P_i \mathrel{\widehat{=}} F_i(P_i)$$

Example 1 A one-place buffer which inputs a value from channel *in* and outputs it on channel *out* can be defined as:

$$COPY(in, out) \mathrel{\widehat{=}} in?x \rightarrow out!x \rightarrow COPY(in, out)$$

[1] Of course, actual CSP scripts contain other elements, such as channel definitions, but these are beyond the scope of this introduction to CSP.

2.1.3 Semantic models

There are three semantic models of CSP processes which form a hierarchy with an increasing degree of expressiveness and detail. In other words, the more expressive model captures all semantic distinctions made by the less expressive one while introducing an additional semantic distinction. As a result, two processes indistinguishable in the less expressive model may be recognised as being different in the more refined model.

Traces model

As discussed in Section 2.1.1, a trace of a process is a sequence of visible events in which it may engage.

Notation 5 Traces are denoted by $s, t, \ldots$; we write $s = \langle a, b, c \rangle$ to mean a trace of events a, b, and c, in this order. The empty trace is denoted by $\langle\rangle$. The length of a trace s is denoted as $\#s$. The concatenation of two traces s and t is denoted as $s^\frown t$. If $0 < i \leq \#s$, then s_i denotes the i-th event in s.

A process can be associated with a number of traces observed at different times and even different "runs" of the process. Traces consist of observable events including termination as denoted by $\checkmark$. Note, however, that $\checkmark$ can only be the last event in a trace of a process.

Notation 6 The behaviour of process P after it has engaged in trace s is denoted by P/s.

Notation 7 A^* denotes the set of all finite sequences over A.

Notation 8 A^+ denotes the set of all non-empty finite sequences over A.

Notation 9 A^n denotes the set of all sequences of length n over A.

Notation 10 The set of all traces of a process P is denoted by:

$$traces(P) \subseteq \Sigma^* \cup \{s^\frown\langle\checkmark\rangle \mid s \in \Sigma^*\}$$

Definition 1 The *state* of process P after it has engaged in trace $s \in traces(P)$ captures the behaviour of P/s.

In the basic traces model $\mathcal{T}$ the semantics of a process P is given by $traces(P)$. In other words, the semantics of a process is characterised entirely by its externally observable behaviour. The traces model allows the formulation (and, as we shall further show, proof) of some important properties of a process, for example:

- A certain event can never happen;
- Event b can only occur after event a, etc.

On the other hand, externally observable behaviour is not sufficient to recognise nondeterminism which is an internal process property that cannot be influenced by the process environment. The traces model cannot distinguish between deterministic and nondeterministic processes since [Hoa85]:

$$traces(P \,\Box\, Q) = traces(P \sqcap Q) = traces(P) \cup traces(Q)$$

Clearly, a more expressive semantic model is required to capture nondeterminism.

Stable failures model

To distinguish between deterministic and nondeterministic processes, the notion of *refusal sets* is introduced in CSP [Hoa85]. A refusal is a set of events a process may refuse to engage in after a particular trace no matter how long these events are offered by its environment. Thus, refusal sets carry information about behaviours that a process may refuse to engage in. By definition, a state of a process from which the internal event τ can be performed does not contribute to the refusals of the process, because the process will eventually engage in τ and leave that state.

To understand how refusals help distinguish nondeterministic from deterministic behaviour, consider the processes $R = P \,\Box\, Q$ and $R' = P \sqcap Q$. Process R cannot refuse to engage in any event that either P or Q can participate in. Process R', on the other hand, may refuse any event from the refusal set of P if it chooses to behave as P, and can similarly refuse anything that Q can if R' behaves as Q. Thus, the initial refusal set (that is, the refusal set of a process after trace $\langle\rangle$) of R' is the union of the initial refusal sets of P and Q and is different from the initial refusal set of R.

Clearly, if a process may refuse a set of events A, it will also refuse any $B \subseteq A$. A *maximal refusal* is the set of the largest sets of events a process may refuse after a particular trace. As will become clear in the next chapter, maximal refusals are much more convenient to manipulate in a computer program for the purposes of automated refinement checking [Ros94]. Since refinement checking is the main focus of this work, for convenience and conciseness maximal refusals are used throughout the thesis instead of the standard definition of refusals from previous work [Hoa85, Ros97].

The process termination event $\checkmark$ plays a special role with respect to the refusals of a process. A process that is ready to terminate (i.e. perform $\checkmark$)

can refuse to engage in any event other than ✓. Also, a process that has already terminated cannot perform any more events and, therefore, has a maximal refusal of $\{\Sigma\}$.

Notation 11 The powerset of a set X is denoted as:

$$\mathbb{P}(X) = \{Y \mid Y \subseteq X\}$$

Notation 12 The maximal refusals of a process P after trace s are denoted by $\mathit{refusals}(P/s) \subseteq \mathbb{P}(\Sigma)$.

This allows the *stable failures* of a process P to be defined as:

$$\mathit{failures}(P) = \{(s, \mathit{refusals}(P/s)) \mid s \in \mathit{traces}(P)\}$$

The semantics of a process P in the stable failures model $\mathcal{F}$ is given by $\mathit{failures}(P)$. This provides a formal treatment of nondeterminism and deadlock in a very elegant way. A deterministic process has a unique maximal refusal after any trace. In contrast, a process P that is nondeterministic after a trace s may contain more than one set of events in $\mathit{refusals}(P/s)$. A deadlocked process refuses to engage in any event from Σ, therefore, its maximal refusal is $\{\Sigma\}$.

Failures-divergences model

A *divergent* process is one which may perform an infinite unbroken loop of invisible events. Such a process is totally out of the control of its environment—a behaviour that is most likely unwanted in practice. More importantly, a divergent process can refuse to engage in any event, much like the deadlocked process. Thus, $\mathcal{F}$ fails to differentiate between divergent and deadlocked processes.

CSP does not model the behaviour of a process after it diverges [Ros97]. Instead, any two processes that can diverge are assumed to be equivalent, and any process that can diverge after trace s may choose to do so after any trace prefixed by s.

Notation 13 The set of traces after which a process P may diverge are denoted by $\mathit{divergences}(P)$. This set is closed under the prefix operator, i.e.:

$$s \in \mathit{divergences}(P) \Rightarrow \{s^\frown t \mid s^\frown t \in \mathit{traces}(P)\} \subseteq \mathit{divergences}(P)$$

In the failures-divergences model the definition of failures for a divergent trace differs from the stable failures model and needs to be extended (closed up under divergence) in order to reflect the choice of not being able to see beyond divergence:

$$\mathit{failures}_\perp(P) = \mathit{failures}(P) \cup \{(s, \Gamma) \mid s \in \mathit{divergences}(P)\}$$

A process P in the failures-divergences model $\mathcal{N}$ is characterised by the pair:

$$(\mathit{failures}_{\perp}(P), \mathit{divergences}(P))$$

Thus, $\mathcal{N}$ augments the failures as used in $\mathcal{F}$ with explicit and complete information about the traces after which a process diverges. Consequently, the failures-divergences model distinguishes between deadlocked and divergent behaviour.

2.1.4 Refinement and equivalence relations

The algebraic framework of CSP provides means for comparing (or *relating*) two syntactically different descriptions of a concurrent system. Traditionally, most process algebras define an *equivalence relation* between processes. In this context, equivalence means having same behaviour in some well defined sense within a semantic model. Such equivalence relations are often based on bisimulation [Par81, Mil89, FM91].

CSP formalises the notion of *refinement* or *implementation* between processes. In general, it is said that process Q refines process P (denoted as $P \sqsubseteq Q$) if and only if all possible behaviours of Q are also possible behaviours of P. In this definition, P plays the role of the *specification*, and Q is the *implementation* process.

As the semantic models of CSP define process behaviour in three different ways, there are three different refinement relations defined as follows:

Definition 2 Traces refinement, denoted as $P \sqsubseteq_{\mathcal{T}} Q$, is defined as follows:

$$P \sqsubseteq_{\mathcal{T}} Q \mathrel{\widehat{=}} \mathit{traces}(Q) \subseteq \mathit{traces}(P)$$

Definition 3 Failures refinement, denoted as $P \sqsubseteq_{\mathcal{F}} Q$, is defined as follows:

$$P \sqsubseteq_{\mathcal{F}} Q \mathrel{\widehat{=}} P \sqsubseteq_{\mathcal{T}} Q \wedge \forall s \in \mathit{traces}(Q) : \mathit{refusals}(Q/s) \subseteq \mathit{refusals}(P/s)$$

Definition 4 Failures-divergences refinement, denoted as $P \sqsubseteq_{\mathcal{N}} Q$, is defined as follows:

$$P \sqsubseteq_{\mathcal{N}} Q \mathrel{\widehat{=}} \mathit{failures}_{\perp}(Q) \subseteq \mathit{failures}_{\perp}(P) \wedge \mathit{divergences}(Q) \subseteq \mathit{divergences}(P)$$

The definitions of refinement in $\mathcal{F}$ and $\mathcal{N}$ models presented above differ slightly in appearance from those given elsewhere [Hoa85, Ros97]. Our definitions make use of maximal refusals and it is trivial to show that these definitions are equivalent to previous ones [Hoa85, Ros97].

CSP also defines an equivalence relation between processes that is based on refinement rather than bisimulation. It is said that process P is equivalent to process Q (denoted as $P = Q$) if and only if both $P \sqsubseteq Q$ and $Q \sqsubseteq P$ hold. Again, the equivalence relation can be applied in any of the three semantic models $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{N}$. As equivalence in CSP is defined in terms of refinement, any result obtained for the latter would also be applicable to equivalence.

2.1.5 Refinement checking

Now that the notions of refinement and equivalence in CSP are formalised, a valid question is: Given a pair of processes P and Q, how to perform refinement checking (i.e. check whether $P \sqsubseteq Q$)? Another related important question is: can refinement checking be automated, to what extent, and how efficient would the relevant algorithms be in terms of space and time?

One possible approach is to use the algebraic laws of CSP [Hoa85, Ros97] to transform P and Q syntactically until refinement can be decided. The main attraction of this method is that the proof can be handled using the equational (axiomatic) laws purely at the syntactic level. De Nicola *et al.* [NIN89] utilise this idea in the context of the CCS algebra [Mil89] using term rewriting techniques. This approach has also been demonstrated by Hoare throughout his book [Hoa85]; however, it has been only used to show equivalence, not refinement. Moreover, for a lack of a clear strategy in applying the algebraic laws, an automated implementation of this method would likely require an automated theorem prover and the assistance of an experienced user for any but the simplest processes [NIN89].

Another approach to refinement checking is to derive the denotational semantics of the two processes and then compare them. Clearly, this method can be completely automated; however, it has two major drawbacks. Firstly, the traces or divergences of a process may be infinite, and secondly, the space requirements for representing *all* traces, refusals, and divergences for both processes may be prohibitive.

A third approach is to exploit the operational semantics of CSP [JH93, Ros97] and represent the semantics of the CSP processes in the form of a labelled transition system. This method, previously suggested and proven in the context of temporal logic model checking [CES86], has been also acknowledged as most viable for checking refinement in CSP [Ros94]. Despite the fact that this approach is, in general, only applicable to finite systems, there are cases when it can be used on arbitrarily large examples [CGL92, RGG+95]. Not surprisingly, this approach forms also the basis of the work presented in this thesis.

2.2 Labelled Transition Systems (LTS)

Labelled Transition Systems (LTS) are widely used to model both hardware and software systems. One of the earliest attempts to use LTSs for the verification of concurrent systems can be traced back to the work of Keller [Kel76], whose main ideas are still valid and applicable. In general, an LTS contains a set of states, a set of labels, and a set of transitions between states. Each transition may be associated with one or more labels as defined by a labelled transition relation between states. Thus, an LTS can be represented as a directed graph the vertices of which are the states of the LTS, and the labelled edges of the graph correspond to the labelled transitions of the LTS.

The semantic interpretation of an LTS is as follows: starting from one or more initial states, the outgoing transitions are followed and the actions implied by the labels of those transitions are performed. An LTS is *deterministic* if there is no state in the LTS that has two or more outgoing transitions that lead to different states but are labelled identically, and the set of initial states contains only one state. If either of these conditions is not met, the LTS is *nondeterministic*. In a deterministic LTS, the sequence of labels of the followed transitions uniquely identifies the state from which the next transition will be chosen; this property does not hold for nondeterministic LTS. An LTS is *finite* if the set of states and labels in it is finite.

The basic definition of LTSs described so far can be further extended in a number of ways to better suit a particular purpose. For example, finite state automata [HU69] include the notions of input and output events as well as final (accepting) states. One specific instance of finite state automata applicable to temporal logic model checking is the Büchi automaton [Buc62]. Finite state automata have a variety of applications ranging from computation theory, languages and regular expressions to digital logic design.

Roscoe *et al.* [RGG+95] have defined a so called Generalised LTS (GLTS) with application to state compression and LTS semantics representation of CSP processes in the FDR2 tool [For97]. In a GLTS states may be labelled with additional CSP-specific semantic information, such as maximal refusals and a divergence flag. It has been shown that a GLTS often provides a more concise representation of CSP semantics than an LTS. However, process semantics are by necessity first obtained in LTS form; the translation to a GLTS is a computationally intensive step that is acceptable only for LTS of a relatively small size or when run-time advantages are expected to offset the translation costs (e.g. for the purposes of state compression).

Another specialised LTS derivative called Labelled Formal Concurrent Systems (LFCS) [God94] has been developed as a modeling framework that can

capture a wide variety of concurrent systems and communication mechanisms. LFCSs are general enough to distinguish between processes and objects on which these processes operate as well as differentiate between control and data states. Processes are allowed to communicate synchronously or asynchronously by means of message passing or shared objects.

In this thesis, we use a generic version of finite nondeterministic LTSs for the representation of the semantics of CSP processes. Our choice is based on a number of considerations. Firstly, the operational semantics of CSP—an essential element in the automated refinement checking approach—is provided in terms of labelled transition systems [JH93, Ros97]. Secondly, we choose to use finite LTSs because a central topic of this thesis—formal verification through model checking—requires the construction of a complete model of the concurrent system under consideration. Thirdly, a nondeterministic LTS can be encoded and traversed efficiently using OBDDs (see Section 2.3), while an LTS with a more complex structure of states and/or labels (such as GLTS or LFCS) may not be as easily represented and manipulated in OBDD form. Thus, we have found finite nondeterministic LTS ideally suited for our purposes. In our opinion, based on extensive experience and experimentation, there are no substantial advantages to be had by adopting a more elaborate formal semantic model such as GLTS or LFCS.

In what follows we present a formal definition of finite nondeterministic LTSs and a set of operations that are used throughout this thesis.

Definition 5 A finite nondeterministic LTS $\mathcal{L}$ is a tuple $\mathcal{L} = (\mathcal{S}, \mathcal{E}, \mathcal{R}, \mathcal{I})$, where:

- $\mathcal{S}$ is a set of states;
- $\mathcal{E}$ is a set of events (transition labels). $\mathcal{E}$ may contain the special events τ and $\checkmark$;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ is a labelled transition relation;
- $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states.

In the rest of this thesis, the term LTS is used to mean finite nondeterministic LTS unless otherwise noted.

Notation 14 We use $\gamma, \lambda, \theta \in \mathcal{S}$ to denote states in $\mathcal{L}$, and $\Gamma, \Lambda, \Theta \subseteq \mathcal{S}$ to denote sets of states in $\mathcal{L}$.

Definition 6 An event $a \in \mathcal{E}$ is said to be *enabled* in the state $\lambda \in \mathcal{S}$ if and only if there exists a state $\gamma \in \mathcal{S}$ such that $\lambda \xrightarrow{a} \gamma \in \mathcal{R}$. Otherwise, a is said to be *disabled* in λ.

Definition 7 Given a transition relation $\mathcal{R}$, we define its transitive closure $\mathcal{R}^+$ as:

$$\mathcal{R}^+ = \{\gamma_1 \xrightarrow{a_1 a_2 \ldots a_n} \gamma_{n+1} \mid \exists \{\gamma_i\}_{i=2}^{n+1} : \gamma_{i-1} \xrightarrow{a_{i-1}} \gamma_i \in \mathcal{R}\}$$

Definition 8 Let a^* denote the possibly empty sequence of a events and $\Gamma \subseteq \mathcal{S}$. We define the following functions:

$$\begin{aligned}
\mathit{Image}(\Gamma, \mathcal{L}) &= \{\lambda \mid \exists \gamma \in \Gamma, a \in \mathcal{E} : \gamma \xrightarrow{a} \lambda \in \mathcal{R}\} \\
\mathit{BackImage}(\Gamma, \mathcal{L}) &= \{\lambda \mid \exists \gamma \in \Gamma, a \in \mathcal{E} : \lambda \xrightarrow{a} \gamma \in \mathcal{R}\} \\
\mathit{TauExpand}(\Gamma, \mathcal{L}) &= \{\lambda \mid \exists \gamma \in \Gamma : \gamma \xrightarrow{\tau^*} \lambda \in \mathcal{R}^+ \vee \gamma \xrightarrow{\tau^* \checkmark} \lambda \in \mathcal{R}^+\} \\
\mathit{BackTauExpand}(\Gamma, \mathcal{L}) &= \{\lambda \mid \exists \gamma \in \Gamma : \lambda \xrightarrow{\tau^*} \gamma \in \mathcal{R}^+\} \\
\mathit{NextEvents}(\Gamma, \mathcal{L}) &= \{a \mid \exists \lambda \in \mathit{TauExpand}(\Gamma, \mathcal{L}), \theta \in \mathcal{S} : \\
&\quad \lambda \xrightarrow{a} \theta \in \mathcal{R} \wedge a \neq \tau\} \\
\mathit{NextStates}(\Gamma, A, \mathcal{L}) &= \{\theta \mid \exists \lambda \in \mathit{TauExpand}(\Gamma, \mathcal{L}), a \in A : \lambda \xrightarrow{a} \theta \in \mathcal{R}\}
\end{aligned}$$

Let us briefly describe the meaning of the functions defined above. *Image* computes the set of states reachable from a given set of states Γ via a single transition. This operation is the basis for one of the fundamental algorithms used in formal verification: reachability analysis (see next section). The states computed by *Image* are often referred to as *successors* of Γ. Similarly, *BackImage* computes the set of states (the *predecessors*) from which a given set of states Γ can be reached through a single transition.

The other four functions are specific to the refinement checking algorithms presented later in this thesis. The function *TauExpand* computes the set of states reachable from a given set of states through a possibly empty sequence of invisible (τ) transitions followed by at most one $\checkmark$ transition. Its purpose is to compute all possible states in which a process may be after a particular trace, including the possibility for a process to terminate (see Section 3.3). *BackTauExpand*, on the other hand, computes the set of states from which a given set of states Γ can be reached through zero or more invisible transitions, and is used to compute the divergences of a process (refer to Section 3.3.1). *NextEvents* computes the set of visible events possible from the set of states Γ, whereas *NextStates* computes the set of states reachable from a set of states Γ through a single event from the set A.

2.2.1 Reachability analysis

In order to check a global property of a system modelled as an LTS, the set of all *reachable* states has to be explored. A state is reachable if there exists a

sequence of transitions that connects an initial state of the LTS with that state. The process of exhaustive state space exploration of a finite model is frequently referred to as *reachability analysis*, *LTS traversal*, or *state space traversal*.

In general, reachability analysis starts from the initial set of states in a LTS and follows all possible transitions. This approach is sometimes referred to as *forward* reachability analysis. Any state that is reached is added to the set of reachable states. If the LTS is finite its traversal is guaranteed to complete. Transitions can be explored in any sequence, however, two strategies—*depth-first* and *breadth-first* search (DFS and BFS for short)—have gained particular popularity.

If one merely needs to check if a particular set of states Γ are reachable from the initial LTS states, another form of state space traversal—*backwards* reachability analysis—can be applied. In this approach, state space exploration starts from Γ and proceeds backwards in the reachability graph, keeping track of all states reachable in this fashion. If any of the initial states of the LTS are encountered during the backwards state space traversal, then at least one of the states in Γ is reachable from the initial states.

Algorithm 1 computes the reachable states of an LTS using the breadth-first exploration technique. Two sets of states are used: *Frontier* contains the most recently reached states the successors of which are yet to be explored, and *Reached* which accumulates all reached states. The LTS traversal starts from the set of initial states $\mathcal{I}$ and proceeds iteratively within the **while** loop. The n-th iteration of this loop computes all states that are reachable from the initial ones through a sequence of exactly n transitions.

Algorithm 1 Reachability analysis using breadth-first search

Require: An LTS $\mathcal{L}$
Ensure: *Reached* contains the set of all reachable states in $\mathcal{L}$
  $Frontier \leftarrow \mathcal{I}$
  $Reached \leftarrow \mathcal{I}$
  while $Frontier \neq \emptyset$ **do**
    $Frontier \leftarrow Image(Frontier, \mathcal{L}) - Reached$
    $Reached \leftarrow Reached \cup Frontier$
  end while

Algorithm 2 computes the reachable states of an LTS using the depth-first exploration technique. In this algorithm, *Stack* contains the set of states to be further explored, and *Reached* again accumulates all reached states. Similarly to Algorithm 1, DFS-based reachability analysis starts from the set of initial states $\mathcal{I}$, however, a recursive rather than iterative approach is used for the exploration

of the state space. The recursion is implemented by the procedure DFS. Unlike the BFS strategy, the DFS algorithm does not explore new reachable states in the order they were reached, instead, a last-in-first-out (LIFO) order is used.

Algorithm 2 Reachability analysis using depth-first search

Require: An LTS $\mathcal{L}$
Ensure: *Reached* contains the set of all reachable states in $\mathcal{L}$

```
procedure DFS
Pop γ from Stack
Reached ← Reached ∪ {γ}
for all λ ∈ Image({γ}, L) do
  if λ ∉ Stack ∪ Reached then
    Push λ onto Stack
    call DFS
  end if
end for
end procedure
{Start of the Algorithm}
Push I onto Stack in any sequence
Reached ← ∅
call DFS
```

The theoretical complexity of reachability analysis with DFS and BFS exploration strategies is the same, since every reachable state and transition is visited exactly once. From the practical point of view, however, one may be chosen over the other depending on other considerations. In refinement checking CSP, for example, it is advantageous to find the shortest trace that shows the difference in behaviour between two processes in order to aid the analysis of the problem. This is easily achievable with BFS, while computing the shortest trace with DFS requires the complete exploration of the LTS. On the other hand, if reachability analysis has to maintain additional information on a per state basis (as is the case with sleep sets in Chapter 6) DFS may offer substantial space benefits compared to BFS.

For a further discussion of various reachability analysis techniques and the enhancements proposed by the author refer to Chapters 5 and 6.

2.3 Ordered Binary Decision Diagrams (OBDD)

Binary Decision Diagrams (BDDs) are rooted acyclic directed graphs that have been independently introduced by Lee [Lee59] and Akers [Ake78] as a represen-

tation of boolean functions used to model digital circuits' behaviour. It has been shown that BDDs may potentially be much more concise than classic representations of boolean functions, such as truth tables or Karnaugh maps [Ake78].

Bryant [Bry86] further refined the work of Lee and Akers. He suggested a restricted form of BDDs commonly referred to as *Ordered* BDDs (OBDDs)[2]. OBDDs impose a strict total order of variable occurrence in their graph and contain no identical subgraphs [Bry86]. This brings about two important properties of OBDDs:

- They are canonical representations, that is, two boolean functions will be represented by the same (isomorphic) OBDD graphs if and only if they are equal. Consequently, testing of equivalence of two boolean formulae is reduced to checking graph isomorphism of their OBDD representations;
- There are efficient graph-based algorithms for OBDD manipulation, which cover the full set of boolean operators from predicate logic. The time complexity of all binary operations is polynomial.

Essentially, an OBDD represents a boolean function $f : \mathcal{B}^n \rightarrow \mathcal{B}$, where $\mathcal{B}$ is the boolean domain: $\mathcal{B} = \{0, 1\}$ [Bry86]. The underlying data structure is a rooted directed acyclic graph the nodes of which are labelled with a single boolean variable, and have two outgoing edges marked with 0 and 1 that represent the boolean constants *true* and *false*, respectively. There are also two terminal nodes labelled with 0 and 1, which represent the value assumed by the boolean function for a given assignment to its variables. As an example, the OBDD graph representation of the function:

$$f(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$$

assuming a variable ordering $x \prec y \prec z$, is shown in Figure 2.1. To evaluate a boolean function given its OBDD and a particular variable assignment, one has to start from the root node and follow the edge labelled with 1 or 0, depending on the value of the variable at the current node. When a terminal node is reached, its label contains the value assumed by the function for the chosen variable assignment.

Variable renaming and existential quantification are two important operations for traversing labelled transition relations (see Section 2.3.3). Variable renaming

[2]In the literature, these are sometimes also referred to as *Reduced Ordered* BDDs (ROBDDs). There is also a large number of other BDD-based data structures [Bry95], but OBDDs are still the most widely used in practical applications because of the maturity of their algorithms and abundance of available OBDD packages.

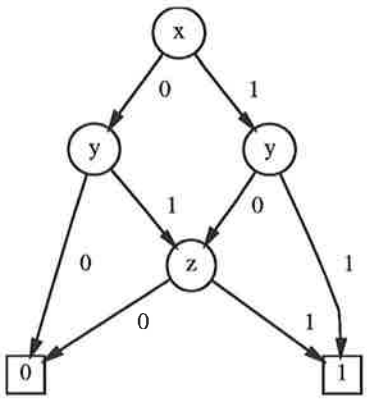


Figure 2.1: OBDD for function $(x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$

substitutes occurrences of certain variables in a boolean function f with others according to a mapping and is denoted as $f\,\big|_{[mapping]}$. Existential quantification over a single variable v in a boolean function f is defined as:

$$\exists_v f = f\,\big|_{v=0} \vee f\,\big|_{v=1}$$

and can be trivially extended to a vector of boolean variables V:

$$\exists_V f = \exists_{v_1} \exists_{v_2} \ldots \exists_{v_n} f$$

The *size* of an OBDD is measured by the number of nodes in its graph. OBDD size for a given boolean function is very sensitive to the chosen variable ordering [Bry86, Bry92], and in the worst case it can be exponential to the number of its variables. Although finding an optimal variable ordering is shown to be a coNP-complete problem [Bry86], some simple heuristics combined with empirical data available in the literature provide very good results for several classes of boolean functions including those used for encoding labelled transition relations [Bry92, ATB94].

A number of fundamental mathematical objects such as sets, tuples, relations, and transition systems can be compactly represented and efficiently manipulated as OBDDs. A mapping (encoding) is required to translate these objects from their original domains into the domain of boolean functions. In the next few sections we discuss how this is achieved in practice.

2.3.1 Representing sets as OBDDs

Cerny and Marin [CM77] have suggested using so called *characteristic functions* in order to represent and manipulate finite sets in terms of boolean functions. In

this section, a slightly modified version of their approach is presented.

Let D be a finite domain with m elements: $D = \{d_i\}_{i=1}^{m}$. Let V_D be a vector of $n \geq \lceil \log_2(m) \rceil$ boolean variables $(v_1^D, v_2^D, \ldots, v_n^D)$. Let a literal be a boolean variable or its negation. It is possible to assign a literal from V_D to each element of D by means of a mapping function $\xi_D : D \to (\mathcal{B}^n \to \mathcal{B})$ such that:

$$\forall x, y \in D : x \neq y \Rightarrow \xi_D(x) \wedge \xi_D(y) = 0$$

Definition 9 The characteristic function $\nu_D : \mathbb{P}(D) \to (\mathcal{B}^n \to \mathcal{B})$ of a set $S \subseteq D$ is defined as:

$$\nu_D(S) = \bigvee_{x \in S} \xi(x)$$

The variables from the vector V_D, in terms of which the characteristic function ν_D is encoded, are called *support* variables, and the vector V_D itself is called the *support vector* of the encoding of the domain D. Intuitively, the characteristic function evaluates to 1 for all variable assignments corresponding to members of S, and is 0 for all other assignments. From this definition of ν_D it is easy to see that for $S_1, S_2 \subseteq D$, the following equations hold [Bry92]:

$$\begin{aligned} \nu_D(\emptyset) &= 0 \\ \nu_D(S_1 \cup S_2) &= \nu_D(S_1) \vee \nu_D(S_2) \\ \nu_D(S_1 \cap S_2) &= \nu_D(S_1) \wedge \nu_D(S_2) \\ \nu_D(S_1 - S_2) &= \nu_D(S_1) \wedge \neg\, \nu_D(S_2) \end{aligned}$$

2.3.2 Representing LTSs as OBDDs

In a given LTS $\mathcal{L} = (\mathcal{S}, \mathcal{E}, \mathcal{R}, \mathcal{I})$, we can think of $\mathcal{S}$ and $\mathcal{E}$ as domains of all states and all events, respectively. Thus, any subset of states $\Gamma \subseteq \mathcal{S}$ (including $\mathcal{I}$) can be encoded as $\nu_{\mathcal{S}}(\Gamma)$ and any set of events $A \subseteq \mathcal{E}$ can be represented by $\nu_{\mathcal{E}}(A)$. To find a suitable representation for $\mathcal{R}$, recall that $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$. Clearly, the two $\mathcal{S}$ domains from the relation should be encoded using a different set of variables. It is possible to use two completely different characteristic functions for that, however, for the purposes of traversing (exploring) the LTS $\mathcal{L}$ (described in Section 2.3.3) it is quite convenient to derive the mapping $\nu'_{\mathcal{S}}$ for the "next" states in $\mathcal{R}$ from $\nu_{\mathcal{S}}$ via simple variable renaming. Then, the characteristic function for a single transition can be expressed as:

$$\nu_{\mathcal{R}}(\gamma \xrightarrow{a} \lambda) = \nu_{\mathcal{S}}(\{\gamma\}) \wedge \nu_{\mathcal{E}}(\{a\}) \wedge \nu'_{\mathcal{S}}(\{\lambda\})$$

and thus, the transition relation $\mathcal{R}$ is encoded as:

$$\nu_{\mathcal{R}}(\mathcal{R}) = \nu_{\mathcal{R}}(\bigcup_{i=1}^{n} \{\gamma_i \xrightarrow{a_i} \lambda_i\}) = \bigvee_{i=1}^{n} \nu_{\mathcal{S}}(\{\gamma_i\}) \wedge \nu_{\mathcal{E}}(\{a_i\}) \wedge \nu'_{\mathcal{S}}(\{\lambda_i\})$$

Definition 10 We define a mapping function ϕ converting an LTS $\mathcal{L}$ into a four-tuple of boolean functions as:

$$\phi(\mathcal{L}) = (\nu_{\mathcal{S}}(\mathcal{S}), \nu_{\mathcal{E}}(\mathcal{E}), \nu_{\mathcal{R}}(\mathcal{R}), \nu_{\mathcal{S}}(\mathcal{I}))$$

2.3.3 Traversing an OBDD-based transition relation

Having described how to encode a complete LTS in the previous section, we also have to provide means of computing the basic functions on LTS given in Definition 8. As an example, let us consider the function *Image*. Its OBDD-based counterpart $Image_{OBDD}$ has to compute a boolean function that is exactly the characteristic function of *Image*:

$$Image_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L})) = \nu_{\mathcal{S}}(Image(\Gamma, \mathcal{L}))$$

Intuitively, to compute the *Image* function one has to find all transitions in $\mathcal{L}$ starting from a state in Γ, and thus derive the set of states that are reachable via a single transition. This can be translated into the following definition of $Image_{OBDD}$ [TSL+90]:

$$Image_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L})) = (\exists_{V_{\mathcal{S}}, V_{\mathcal{E}}}(\nu_{\mathcal{S}}(\Gamma) \wedge \nu_{\mathcal{R}}(\mathcal{R}))) \Big|_{[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]} \tag{2.2}$$

where $[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]$ denotes substitution (renaming) of each variable in the support vector $V'_{\mathcal{S}}$ with the corresponding variable in the support vector $V_{\mathcal{S}}$, and $\exists_{V_{\mathcal{S}}, V_{\mathcal{E}}}$ denotes existential quantification over the variables in $V_{\mathcal{S}}$ and $V_{\mathcal{E}}$.

Equation (2.2) requires some explanation. Computing $Image_{OBDD}$ requires three operations: conjunction, existential quantification, and variable substitution. The conjunction $\nu_{\mathcal{S}}(\Gamma) \wedge \nu_{\mathcal{R}}(\mathcal{R})$ derives the set of all transitions in $\mathcal{R}$ starting from a state in Γ. Since only the ending states of the transitions are required, existential quantification is used to abstract away the starting states and the events labeling the transitions. Thus, the set of transitions is reduced to the set of all next states. However, they are encoded in terms of the variable vector $V'_{\mathcal{S}}$ and not $V_{\mathcal{S}}$ as $\nu_{\mathcal{S}}(\Gamma)$ is. This is taken care of by the variable substitution operation, which effectively re-encodes the set of all next states in terms of the vector $V_{\mathcal{S}}$ as required.

Computing the set of next states is a fundamental operation in reachability analysis, therefore the space and time overheads of the $Image_{OBDD}$ function should be kept to a minimum. The major contributors to the complexity of this function are the conjunction and the existential quantification operators, because the variable substitution is a constant time operation [Bry86]. To speed up the computation of these two operations applied in sequence, Burch *et al.* suggest to

Algorithm 3 Computation of $TauExpand_{OBDD}$

Require: An LTS $\mathcal{L}$ in OBDD form $\phi(\mathcal{L})$
Require: A set of states $\Gamma \subseteq \mathcal{S}$ encoded as $\nu_{\mathcal{S}}(\Gamma)$
Ensure: $States = \nu_{\mathcal{S}}(TauExpand_{OBDD}(\Gamma, \mathcal{L}))$
  $States \leftarrow \nu_{\mathcal{S}}(\Gamma)$
  $OldStates \leftarrow 0$
  while $States \neq OldStates$ **do**
    $OldStates \leftarrow States$
    $States \leftarrow States \cup (\exists_{V_{\mathcal{S}}, V_{\mathcal{E}}}(States \wedge \nu_{\mathcal{E}}(\tau) \wedge \nu_{\mathcal{R}}(\mathcal{R}))) \Big|_{[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]}$
  end while
  $States \leftarrow States \cup (\exists_{V_{\mathcal{S}}, V_{\mathcal{E}}}(States \wedge \nu_{\mathcal{E}}(\checkmark) \wedge \nu_{\mathcal{R}}(\mathcal{R}))) \Big|_{[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]}$

combine them into a single operation called *relational product* [BCL+94], which computes the end result without constructing the intermediate OBDD for the conjunction.

Function $BackImage_{OBDD}$ can similarly be derived as:

$$BackImage_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L})) = \exists_{V'_{\mathcal{S}}, V_{\mathcal{E}}}(\nu_{\mathcal{S}}(\Gamma) \Big|_{[V'_{\mathcal{S}} \leftarrow V_{\mathcal{S}}]} \wedge \nu_{\mathcal{R}}(\mathcal{R}))$$

Algorithm 4 Computation of $BackTauExpand_{OBDD}$

Require: An LTS $\mathcal{L}$ in OBDD form $\phi(\mathcal{L})$
Require: A set of states $\Gamma \subseteq \mathcal{S}$ encoded as $\nu_{\mathcal{S}}(\Gamma)$
Ensure: $States = \nu_{\mathcal{S}}(BackTauExpand_{OBDD}(\Gamma, \mathcal{L}))$
  $States \leftarrow \nu_{\mathcal{S}}(\Gamma)$
  $OldStates \leftarrow 0$
  while $States \neq OldStates$ **do**
    $OldStates \leftarrow States$
    $States \leftarrow States \cup \exists_{V'_{\mathcal{S}}, V_{\mathcal{E}}}(States \Big|_{[V'_{\mathcal{S}} \leftarrow V_{\mathcal{S}}]} \wedge \nu_{\mathcal{E}}(\tau) \wedge \nu_{\mathcal{R}}(\mathcal{R}))$
  end while

Functions $TauExpand_{OBDD}$ and $BackTauExpand_{OBDD}$ are computed iteratively until the largest set of states reachable through τ transitions is derived. These are presented as Algorithms 3 and 4. In both algorithms, the set of states *States* grows monotonically starting from $\nu_{\mathcal{S}}(\Gamma)$ until the fixed point of the **while** loop is reached. $TauExpand_{OBDD}$ also includes an extra step to identify if the process in its current state can terminate and add the termination states to the result.

Finally, functions $NextEvents_{OBDD}$ and $NextStates_{OBDD}$ can be computed as:

$$NextEvents_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L})) = \exists_{V_{\mathcal{S}}, V'_{\mathcal{S}}}(TauExpand_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L})) \wedge \nu_{\mathcal{R}}(\mathcal{R}))$$

$$NextStates_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \nu_{\mathcal{E}}(A), \phi(\mathcal{L})) = (\exists_{V_{\mathcal{S}}, V_{\mathcal{E}}}(TauExpand_{OBDD}(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L})) \wedge \nu_{\mathcal{E}}(A) \wedge \nu_{\mathcal{R}}(\mathcal{R}))) \Big|_{[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]}$$

2.4 Tools for automated verification

A number of tools for automated verification, both academic and commercial, have been developed over the last decade. In this section, we briefly present those that are most relevant to this work.

2.4.1 FDR

FDR (standing for *Failures-Divergence Refinement*) by Formal Systems (Europe) Ltd is a commercially available tool with a graphical front-end and an extensive debugging facility [For93, For97]. It accepts a machine-readable version of the CSP language extended with functional language constructs that simplify the definition of complex systems. The tool has been successfully applied to a number of examples of practical significance and has been used, for example, in verifying (and breaking) cryptographic protocols [Low96].

The FDR tool makes use of the operational semantics of CSP to derive LTS models of the relevant algebraic process descriptions. The primary application of the tool is refinement checking between two concurrent process descriptions in CSP.

The specification process is compiled into an explicit finite model in a so called *normal form* [Ros94], which is then used by the refinement check that follows. The normal form LTS has an important advantage: the current state can be uniquely identified given a valid process trace. However, LTS normalisation is an expensive operation and can significantly slow down refinement checking when the specification process contains a large number of states. This is rarely an issue when the specification process is a simple description of an abstract property (e.g. deadlock freedom) [Ros94], but a specification that is more complex (e.g. an abstract description of a system design) can, in the author's experience, be problematic to normalise.

The implementation process, on the other hand, is not compiled into an explicit finite model as this would be prohibitively expensive in many cases. Instead,

an approach very similar to that of on-the-fly model checking [FM91] is used[3]. The process description is translated into a set of explicitly represented (but hopefully small) LTSs for each so called *low-level* (sequential) components. The exploration of the full LTS of the implementation process uses a set of rules that allow the computation of the transitions and next states[4] of the whole system (i.e. the *Image* operation).

During refinement checking, the FDR tool stores the reached states explicitly as a list of binary vectors, each of them uniquely identifying a pair of states—one for the specification LTS and one for the implementation LTS. A clever reached state space management technique ensures that the performance of the tool does not degrade significantly when the available physical memory is exhausted [Ros94].

There exist two major revisions of the tool. The earlier version is referred to as FDR1 [For93] and the latest is known as FDR2 [For97]. The front-end and the compiler of FDR1 are written in the SML language which, in the experience of the author, adds noticeable albeit relatively constant overhead when loading CSP scripts. Also, the division of processes to low- and high-level in FDR1 is restrictive—for example, terms like $(P \setminus A) \,\Box\, (Q \setminus B)$ are not supported.

FDR2 improves on the previous revision of the tool by offering a rewritten front-end and a new implementation of the CSP operational semantics which relaxes the division between high- and low-level processes. Low-level operators applied to high-level process descriptions are enabled by keeping internal structures similar to syntax trees and interpreting those at run-time [Ros97]. More importantly, however, FDR2 features a set of sophisticated, semi-automated LTS reduction techniques known as *state compression* [RGG+95]. The general idea behind state compression is to build the LTSs of the parallel components in a system iteratively, deriving a semantically equivalent representation that is hopefully smaller on each iteration. This works very well for concurrent systems which exhibit locality of communication and repetitive structure. An example of such a system is the dining philosophers problem for which the size of the compressed LTS remains constant when successively adding a philosopher and a fork process and hiding the local events [RGG+95]. For some other concurrent systems, such as Solitaire and Monkey puzzle examples described in Appendix A, the state compression algorithms built into FDR2 do not bring about any benefits.

Future versions of FDR will likely include further state space reduction tech-

[3] On-the-fly model checking is a general explicit model checking technique that does not pre-compute the reachability graph prior to verification. Instead, an executable model of the system is built and its state space is traversed concurrently with the verification activity. This avoids the bottleneck of keeping the complete reachability graph in memory.

[4] A state of the full LTS in FDR is a tuple containing the local states of all low-level processes.

niques such as data independence [Ros97]. The latter aims at proving properties of concurrent systems using any data type based on demonstrating refinement for a particular (smallest) finite data type which is known to preserve the original properties of the system.

2.4.2 MRC

The *Milano Refinement Checker* (MRC) tool is an academic tool developed at the University of Milan as a Master's project [BMTV94]. Designed as an alternative to an early prototype of FDR built at Inmos, it supports only a very limited subset of the CSP language in a textual format that is not compatible with that of FDR. These factors make running CSP scripts suitable for FDR on MRC difficult.

Similarly to the ARC tool discussed in this thesis, MRC attempts to alleviate the state space explosion problem in refinement checking CSP by using OBDDs as an underlying representation of CSP semantics. However, some deficiencies in its OBDD encoding and refinement checking algorithms result in severe performance bottlenecks that prevent MRC in exploiting the full potential of the OBDD approach. The shortcomings of the MRC tool are further discussed in Chapter 3.

2.4.3 SMV

The SMV system is one of the earliest OBDD-based model checking tools developed by McMillan at CMU [McM92a, McM92b]. It checks finite system models derived from a description in the SMV language against specifications expressed in the temporal logic CTL (Computational Tree Logic) [CGP99]. The latter is sufficiently expressive to capture various properties such as safety, liveness, and fairness.

The SMV language, on the other hand, is specific to the tool and is targeted at describing primarily digital hardware systems, although it can also be used as a general language for describing transition systems. Consequently, most of the applications of SMV reported in the literature concern sequential digital circuit verification.

A case study on the use of SMV for the verification of communication protocols [BK95] pinpoints some deficiencies of the SMV language when used for modeling of concurrent systems. In particular, difficulties have been experienced in combining nondeterminism with simultaneous transitions, and in representing asynchronous process execution. These problems not only necessitate the modeling of concurrent systems to be done at the lowest abstraction level—that of

transition relations—but reportedly may also increase the size of the finite models obtained after the compilation into OBDDs.

2.4.4 SPIN

SPIN is one of the earliest on-the-fly model checking tools for concurrent systems [Hol97]. It has been successfully applied to a wide range of problems including a variety of protocols, operating systems code, concurrent algorithms, etc. The tool can check for specific properties, such as deadlock, dead (unreachable) code, and process invariants, and for general correctness requirements expressed in linear temporal logic (LTL) [CGP99].

The system model in SPIN is described in a tool-specific language called Promela. The language supports both synchronous communication through rendezvous and asynchronous communication by means of buffered channels and shared process memory. System descriptions can dynamically vary in the number of active processes, which can be seen as a unique feature of the tool.

What makes SPIN interesting from our point of view is the abundance of advanced reachability analysis and verification techniques built into the tool. The list of options includes scatter searching [Hol87], bit-state hashing [Hol88], state space caching [Hol87, GHP92], and partial order methods [HP94, God94]. These algorithms represent different methods for increasing the capacity of the tool, that is, the relative size of system models that can be handled when space and time limits are imposed. We further discuss some of these techniques and their applicability to our work in Chapters 5 and 6.

2.4.5 Others

A few other formal verification tools are also referenced throughout this thesis. The CCS tool [EFT91] is one of the first to use OBDDs in a process algebra setting and checks weak bisimulation equivalence. The Simple tool [DB95] uses a general method for deriving an LTS model from algebraic specifications applicable to a large class of process algebras whose operational semantics can be expressed in terms of a Simple system.

Chapter 3

Basic OBDD-based Refinement Checking

Overview

There is a considerable semantic gap between CSP process expressions and boolean functions. In order to fully exploit the potential advantages of using OBDDs, three key sets of algorithms are required: one for converting CSP terms into OBDDs, another for computing the CSP semantics (traces, maximal refusals, and divergences) in OBDD form, and a third for performing refinement checking on the OBDD representations of the specification and implementation processes. These representations and algorithms are the topic of this chapter. We also present experimental performance data for these algorithms based on their implementation in the ARC tool, contrast ARC's performance with that of several other tools, and analyse this data to identify strengths and potential bottlenecks of our approach.

3.1 Related work

The initial research in symbolic model checking [CBM89, BCMD90, TSL+90] emphasises primarily the algorithmic aspects of model checking with OBDDs while largely ignoring the issue of constructing the OBDDs on which the algorithms operate. This is because the area of application of this technique was initially restricted to digital hardware, which can be described through boolean functions in a rather straightforward way. Although Burch *et al.* [BCM+92] describe how their technique can be applied to checking strong and weak (observational) bisimulation as well, no method for deriving an OBDD representation given an LTS

or process algebra term is provided in the paper.

A central issue then is the derivation of a model (e.g. an LTS) from a syntactic description of a process on which the verification algorithm operates. The straightforward and naive approach of generating a transition relation by enumeration of (simply following the operational semantics or otherwise) all transitions $\gamma \xrightarrow{a} \theta$ is computationally hardly better than explicit state enumeration, since the number of transitions in a concurrent system is subject to the same explosion problem as the number of states. Thus, although the OBDD derived by such an approach may be as compact as with any other technique, the time overhead of constructing it may be prohibitive.

This problem has been addressed by Enders *et al.* [EFT91], who provide a method for generating OBDDs for the CCS algebra. Their approach is based on defining an operator on OBDDs for each CCS operator from a certain subset. The transition relation for an arbitrary process in that subset of CCS can thus be obtained in a syntax-driven manner—OBDDs for process sub-expressions are obtained while traversing the syntax tree of the input description. The OBDD encoding for the LTS representation of a CCS process is, therefore, derived in a compositional way. The computational complexity of the syntax-driven approach is determined by the syntactic complexity of the process expression as opposed to the semantic state space complexity. This basic idea has been widely accepted and developed for a number of process algebras, including CSP [BMTV94], Circal [MM95, CCFP95], and Simple systems [DB95].

The approach presented in this chapter is similarly based on syntax-driven computation of semantics and extends previous work done by the author [PY96a, PY96b]. The majority of the techniques have been implemented in the ARC tool [PY95, PY96b]. Our approach presents a considerable improvement over the existing background body of research, in the following ways:

- The process algebras on which previous methods were applied [EFT91, BMTV94, MM95, CCFP95], are relatively basic and most of them lack support for advanced features such as recursion, parameterised processes, channel input/output, sequential composition, identifiers, and expressions. In contrast to this, our semantic derivation technique supports all these features;

- We provide a correct and efficient compilation of the external choice operator ($\Box$). In comparison, the translation technique suggested by Barrett *et al.* [BMTV94] is not only less efficient, but also contains a semantic flaw as discussed in Section 3.2.5. Since external choice is used quite often in practice, our technique results in a much smaller OBDD for the transition relation. The complex semantics of this operator would also likely be

problematic to define in Simple [DB95];

- The re-encoding of sequential components provides inexpensive means to further reduce the final OBDD for a process term both in the number of boolean variables and in size (number of OBDD nodes);
- Process refusals are pre-computed in an efficient way and stored as an implicit conjunction of boolean functions;
- The input language of the ARC tool is the de-facto standard for Scattergood's text-based CSP notation [Sca92], which makes it largely compatible with the FDR tool[1] [For97]. This sets it apart from the MRC tool which uses a different syntax [BMTV94].

3.2 Compiling processes into OBDDs

Throughout this chapter, we refer to the process of deriving the semantics of a CSP term in OBDD form as compilation, due to the many similarities with the compilation of traditional programming languages. The task of translating a process term in the CSP algebra into what is essentially a set of boolean formulae is not trivial. The general background required for understanding the approach presented in this section has been provided in Chapter 2.

A syntax-driven compilation technique similar to that described by Enders *et al.* [EFT91] is used to build the process semantics while traversing the syntax trees of process definitions generated by a parser for the standard text-based CSP notation. As mentioned in the previous section, the parser used in the ARC tool is based on the public domain parser by Scattergood [Sca92], with a few minor modifications and enhancements—most importantly, some syntax changes introduced in FDR2 [For97] have been implemented. Throughout the rest of this thesis it is assumed that the reader is already familiar with the text-based CSP notation as described by Formal Methods (Europe) Ltd. [For97] and used by Roscoe [Ros97].

Conceptually, our compilation technique works in three steps:

1. The relevant text-based notation is parsed and a corresponding syntax tree is constructed;
2. A mapping function based on the operational semantics of CSP [JH93, Ros97] is used to compute the LTS process representation from the syntax tree;

[1] See Chapter 7 for a discussion of the input language ARC accepts.

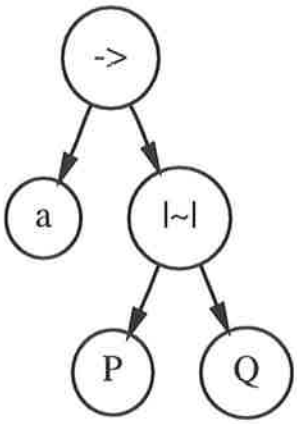


Figure 3.1: An example syntax tree

3. The LTS is encoded in OBDD form in preparation for the refinement check.

In practice, steps 2 and 3 can be combined into one to avoid the need for an explicit (graph-based) representation of the intermediate LTS from step 2. In this case, LTSs are stored and manipulated in OBDD form using the encoding function $\phi(\mathcal{L})$ introduced in Section 2.3.2.

A mapping function $\psi : \mathit{Process} \rightarrow \mathit{LTS}$, where *Process* is the domain of valid CSP process terms provided by Equation (2.1) is key to step 2 of our compilation method. The mapping operates on the syntax tree of a process term, and, for each CSP operator $\odot_{CSP}$ in the syntax tree, ψ defines a corresponding operator $\odot_{LTS}$ such that:

$$\psi[\![P \odot_{CSP} Q]\!] = \psi[\![P]\!] \odot_{LTS} \psi[\![Q]\!]$$

The application of ψ to the syntax tree involves the traversal of the tree. We describe this process informally on the syntax tree of the term $a \rightarrow (P \sqcap Q)$ shown in Figure 3.1. To construct the LTS corresponding to the root node, we descend to the node representing $P \sqcap Q$. Next, we construct the LTSs for P and Q (handling of process references and recursion in general is detailed in Section 3.2.2) in preparation for the construction of the LTS for $P \sqcap Q$. Once $\psi[\![P \sqcap Q]\!]$ is computed, we return to the root of the tree and compute the final LTS for that tree.

In Definition 10, Section 2.3.2, we have introduced ϕ as a mapping from LTS to a four-tuple of boolean functions. The composite mapping $\phi \circ \psi$ implements the translation of CSP semantics into boolean formulae, thus completing steps 2 and 3 described above.

Before going into the details of the actual compilation, we need to introduce the notions of identifiers, expressions, recursion, and OBDD encoding of domains.

3.2.1 Identifiers and expressions

Commonly throughout the literature, the CSP language is extended to allow the use of identifiers in a way similar to functional programming languages. Identifiers can be assigned a value (bound) only once [Ros97, pp. 159–160], either through parameterised recursion (see Section 3.2.2) or channel input.

This interpretation of identifiers has the useful side effect of making the value of all expressions using identifiers known at compile-time. This considerably simplifies the compilation task at the expense of *iterative compilation* of certain CSP sub-terms, e.g. the channel input operator. By iterative compilation we mean the iterative translation of a given CSP term while changing the *identifier context* of its compilation. An identifier context, on the other hand, is simply a mapping from identifiers to values[2]. For conciseness, we assume that the domain of values is that of the integer numbers, and that expressions are formed using the basic arithmetic and comparison operators. In practice, at least one more domain for identifiers is generally required for constructing processes—that of event sets, as well as the corresponding set operators.

Notation 15 Identifier contexts are denoted by $\mathcal{C}_v : \mathit{IdName} \rightarrow (\mathit{Value} \cup \{\mathit{error}\})$. The semantic value of a process P under the identifier context $\mathcal{C}_v$ is denoted by $[\![P]\!]_{\mathcal{C}_v}$.

The special value *error* introduced above can be obtained, for example, when an unknown (that is, unbound) identifier is encountered in an expression. Any occurrence of *error* during the compilation stage in ARC causes the tool to abort compilation and exit with an error message provided to the user.

As an introductory example of how identifier contexts affect compilation, consider the following CSP script:

$$\mathit{COMPARE}(a, b) \mathrel{\widehat{=}} \text{if } (a < b) \text{ then } \mathit{less} \rightarrow \mathit{STOP} \text{ else } \mathit{notless} \rightarrow \mathit{STOP}$$
$$\mathit{MAIN} \mathrel{\widehat{=}} \mathit{COMPARE}(3, 4)$$

in which *COMPARE* is a process with two parameters (a and b), and engages in the event *less* if the first parameter is less than the second one, or performs *notless* otherwise. Process *MAIN* uses *COMPARE* to compare the constants 3

[2]Sometimes an identifier context is also referred to as *environment* [Sca98], but this term could create a potential ambiguity with process environments.

and 4. During the compilation of *MAIN*, the parameter a is bound to 3, and b is bound to 4. Process *MAIN* can be rewritten as:

$$[\![MAIN]\!] = [\![\text{ if } (a < b) \text{ then } less \rightarrow STOP \text{ else } notless \rightarrow STOP]\!]_{\{a \mapsto 3, b \mapsto 4\}}$$

which after substitution of a and b with the corresponding values reduces to:

$$[\![MAIN]\!] = [\![less \rightarrow STOP]\!]$$

In the above example, the mapping $\{a \mapsto 3, b \mapsto 4\}$ represents the identifier context for the compilation of the process *MAIN*.

There are several important properties of identifier contexts that have to be taken into account when compiling CSP scripts:

1. Identifier contexts are valid only throughout the compilation of a single process definition. It is meaningless to relate two identifiers with a common name belonging to two distinct process definitions (see Roscoe's related discussion [Ros97, pp. 136–137]);

2. Mappings *IdName* $\mapsto$ *Value* can be added to identifier contexts in two ways:

   - When a formal parameter in a process definition acquires the value of an actual parameter in another process term that refers to that definition (like in process *COMPARE* above), or
   - When the channel input operator is used;

3. An attempt to add a new mapping *IdName* $\mapsto$ *Value* to a identifier context in which *IdName* is already bound to a value represents an error—identifiers can be assigned values only once;

4. As previously discussed, evaluating an expression containing an identifier not present in the current identifier context is an error that is detected at compile time.

Note that the last two properties above essentially limit the syntactic constructs that are considered to be semantically correct CSP terms. For example, the following process description is illegal because it attempts two consequtive channel inputs into the same identifier:

$$ch?x \rightarrow ch?x \rightarrow P$$

As an example of an identifier bound to a value via channel input, consider process $MAIN2$ from the following script[3]:

$$domain = \{1..9\}$$
$$channel\ in : domain$$
$$MAIN2 \mathrel{\widehat{=}} in?x \rightarrow COMPARE(x, 5)$$

The environment of $MAIN2$ can communicate any value from the set $\{1..9\}$ through the channel in, and that value is passed on to $COMPARE$ as a first parameter. However, the identifier x, which is bound to the input value, can only be assigned a value once. This is resolved by using iterative compilation of a CSP term with a somewhat different identifier context. This allows the channel input operator to be converted into what is essentially an equivalent indexed event:

$$MAIN2 \mathrel{\widehat{=}} \Box_{i \in \{1..9\}}\ in.i \rightarrow [\![COMPARE(x, 5)]\!]_{\{x \mapsto i\}}$$

Iterative compilation can be applied in a similar fashion to more complex process definitions containing, for example, more than one channel input operator, the replicated versions of operators $\Box$, $\sqcap$ and $\underset{A}{\|}$, etc. It is interesting to note that the number of iterations required is proportional to the size of the data types used, and this factor can be a major contributor to the complexity of the generated LTS.

3.2.2 Process references and recursion

Using a reference to a process definition on the right-hand side of another process definition is a common way of constructing non-trivial processes. Processes $MAIN$ and $COMPARE$ defined in Section 3.2.1 provide examples of the use of process references.

A chain of references forming a closed loop is generally referred to as *recursion* [Ros97], because of the similarities with recursion in common procedural programming languages [WG84]. Recursion may involve one or more process definitions and their parameters. Clearly, the compilation of an arbitrary CSP term requires a mechanism to deal with recursion in order to resolve the cyclic dependencies between process definitions and produce a finite LTS that correctly reflects the semantics of the process term[4].

[3]This style of channel definition is specific to the ARC tool. For a further discussion refer to Chapter 7.

[4]In general, the finiteness of the LTS cannot be guaranteed, because recursion combined with parallel or sequential operators may result in a process with an infinite state space.

A key to our approach for compiling recursive process definitions is keeping track of so called *process signatures*. Process signatures consist of process definition names and actual parameter values used in a particular reference to that definition, if any. Note that in CSP all process signatures are known at compile-time. Process *COMPARE* from the previous section, for example, is used by *MAIN* with the process signature $COMPARE(3, 4)$.

In general, the compilation of a CSP term goes through a sequence of process signatures—one for every process reference. If there is a process signature that appears more than once in the sequence, there is a cyclic dependency (recursion) among process definitions used by that CSP term. This solves the problem of detecting recursion leaving us with the issue of resolving it in order to derive the operational semantics of processes that use recursion.

Compilers for conventional procedural programming languages associate each function and procedure with an entry point of the corresponding thread of control [WG84]. Similarly, a process signature can be associated with the set of initial states in the LTS which would be derived by the compilation of the corresponding process reference. As an example, consider the following CSP script:

$$COUNTER(x) \mathrel{\widehat{=}} \text{if } (x == 0) \text{ then } \mathit{Skip} \text{ else } \mathit{down} \rightarrow COUNTER(x-1)$$
$$MAIN3 \mathrel{\widehat{=}} COUNTER(2) \sqcap COUNTER(1)$$

Assuming left-to-right order of compilation of the internal choice operator in the process *MAIN*3, the derivation of its LTS would first construct the LTS for the process signature $COUNTER(2)$, which eventually constructs the LTS shown in Figure 3.2. The initial state is marked with a double circle, and each state is labelled with the corresponding process signature, if any. Next, the right-hand side of the internal choice operator is being compiled, at which stage the process signature $COUNTER(1)$ is encountered. This process signature has been already processed during the compilation of $COUNTER(2)$, and there is a state labelled with this signature. At this point, there are two options to be considered:

- To compile $COUNTER(1)$ similarly to $COUNTER(2)$ and then to appropriately combine the two partial LTSs into one for *MAIN*3;
- To reuse the LTS from Figure 3.2, since the LTS of $COUNTER(1)$ is contained in that of $COUNTER(2)$.

The first option appears to be a safer choice, however, if adopted as is, it would be too restrictive as it would make impossible the compilation of recursive process definitions. The second option appears more attractive, for it enables the compilation of such process definitions. Attractive as it is, the second option cannot be applied unconditionally. Even if a process signature is encountered

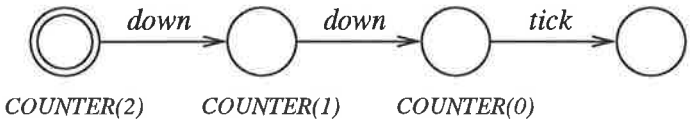


Figure 3.2: The LTS after compiling the process signature $COUNTER(2)$

more than once during the compilation of a process definition, we may still be forced to construct separate LTS instances for that process signature. This can be demonstrated using process $MAIN4$ defined as:

$$MAIN4 \triangleq COUNTER(1);\ COUNTER(1)$$

In this example, the semantics of sequential composition require that when the left-hand-side process $COUNTER(1)$ terminates, the right-hand-side process, which has the same process signature, is executed. To reflect this in the LTS for $MAIN4$ the compilation of sequential composition has to modify the LTS of the left-hand-side process instance and connect it to the initial states of the right-hand-side process instance (see Section 3.2.5). Because of this modification, the two occurrences of the process signature $COUNTER(1)$ in $MAIN4$ need two separate instances of their LTSs.

It is evident that the proper handling of recurrent process signatures requires an analysis of the syntactic context in which these signatures appear. In particular, one has to distinguish between CSP operators $\odot_{LTS}$ which modify the LTSs of their operands from those operators which do not and merely build on the LTSs of the processes they compose together. We call the latter group of operators *preserving* and the rest *non-preserving*; the classification of each operator can be easily made by examining its operational semantics (more on this in Section 3.2.5).

In general, when two identical process signatures occur in the context of preserving operators, the LTS derived for the first occurrence of the signature can be reused. This is the case with process $MAIN3$—as it is demonstrated in Section 3.2.5, the internal choice operator is preserving. The final LTS for that process is shown in Figure 3.3. In contrast to this, the sequential composition operator used in $MAIN4$ is non-preserving, and thus the process signature $COUNTER(1)$ has to be compiled twice in order to derive its semantics correctly.

Next, our approach to compiling recursive process definitions is detailed. The main idea is to maintain a so-called *recursion context* throughout the syntax-driven compilation process. A recursion context is simply a mapping from process signatures to sets of states—the initial states of the LTS corresponding to these process signatures.

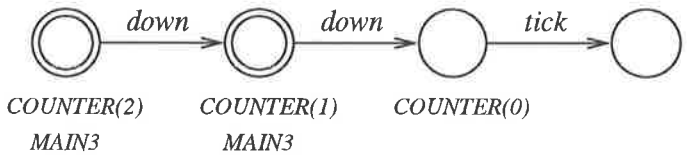


Figure 3.3: The LTS of the process definition $MAIN3$

Notation 16 Recursion contexts are denoted by $\mathcal{C}_r : ProcSignature \rightarrow \mathcal{S}$. The semantic value of a process P given a recursion context $\mathcal{C}_r$ is denoted by $[\![P]\!]_{\mathcal{C}_r}$.

Every process signature in a given recursion context is unique. Recursion context information propagates through preserving CSP operators $\odot_{LTS}$ and does not propagate through non-preserving ones. Because of this, it is guaranteed that, whenever a process signature in the current recursion context is encountered during compilation, the corresponding LTS recorded in the recursion context can be reused. This mechanism also makes possible the detection of infinite recursion.

Our approach to handling parameterised recursion, which is later formalised as a part of the syntax-driven process compilation presented in Section 3.2.5, is further illustrated by the following example:

$$C = up \rightarrow O \ ; \ C$$
$$O = up \rightarrow down \rightarrow O \ \Box \ down \rightarrow SKIP$$

The compilation of process C starts with an empty recursion context. Since the process identifier C is not found in the context, a new LTS representation for C needs to be built, and an initial state for C is created to update the recursion context (Figure 3.4a). The next step is to compile the left operand of the sequential process composition $(O \ ; \ C)$, which is a process reference that is not found in the recursion context, therefore, O is compiled separately obtaining the LTS shown in Figure 3.4b. The right operand of the sequential composition is compiled next, however, since C is already in the recursion context, no further action is taken, and the LTS for $(O \ ; \ C)$ is given in Figure 3.4c. Finally, the transition labelled with the event up is created to obtain the complete LTS representation of the process C (Figure 3.4d).

It has to be noted that the treatment of recursive process definitions suggested by Barrett *et al.* [BMTV94] is not only much more limited in scope to the one presented here (it only considers self-recursion, and has no support for parameterised process definitions), but also fails to make the important distinction between preserving and non-preserving operators in CSP. As a result, the LTS semantic representation derived by the MRC tool for non-preserving operators is

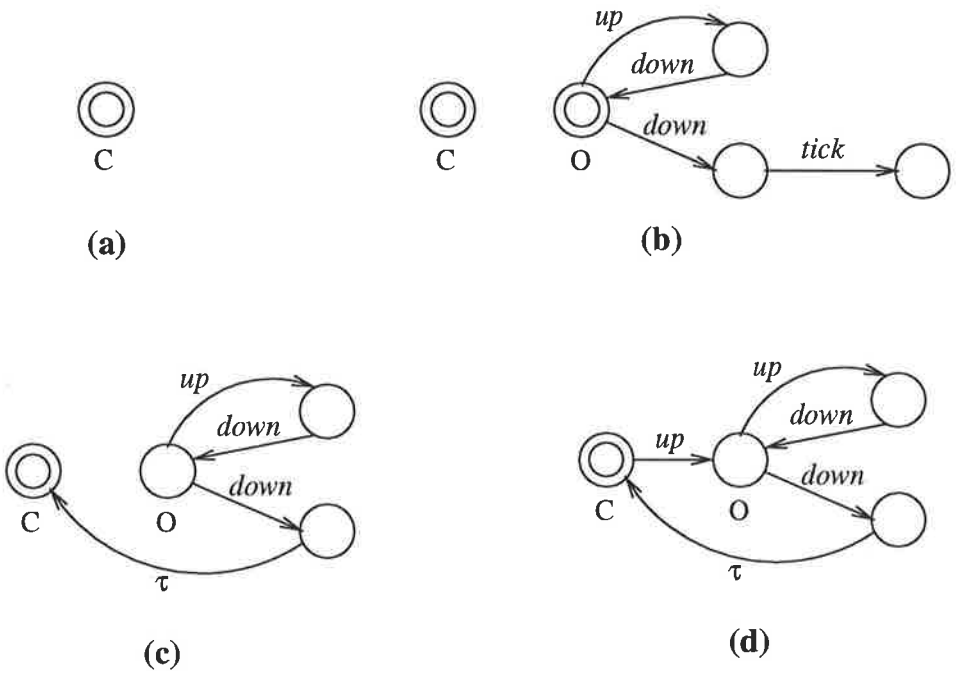


Figure 3.4: Buidling the LTS for a more complex recursion

incorrect. This has been confirmed experimentally on a copy of the tool by the author.

The MRC tool is not the only one not providing full support for recursion in its input language. Enders *et al.* [EFT91] do not present their approach to recursion, although they provide an example (Milner's schedulers [Mil89]) which uses recursion without parameters. The Circal [MM95] and Simple [DB95] systems and the Concurrency Workbench [CPS93] provide support for recursion but do not appear to allow for process parameters. Procedures in the Concurrency Factory [CGL+94], which correspond to process definitions in our framework, cannot be mutually or self-recursive, although they make use of parameters [Sok96, p. 36–37].

3.2.3 OBDD encodings of states and events

As discussed in Section 2.3, the encoding of a given LTS into a boolean formula requires:

- A set of boolean variables, split into three vectors (V_S, $V_\mathcal{E}$, and V'_S) for the encoding of the three separate domains—starting states, events (transition

Element of D	Index of the element	Binary number	Conjunction
d_0	0	00	$\neg\, v_D^1 \wedge \neg\, v_D^0$
d_1	1	01	$\neg\, v_D^1 \wedge v_D^0$
d_2	2	10	$v_D^1 \wedge \neg\, v_D^0$
d_3	3	11	$v_D^1 \wedge v_D^0$

Table 3.1: Construction of unique conjunctions for a four element domain

labels), and ending states—used in the representation of the LTS, and

- A set of mapping functions from domain elements to unique conjunctions of boolean variables, one for each of the domains.

A rather straightforward and commonly used approach to generating a unique conjunction for domain elements $d_i \in D$ (see Section 2.3.1) is to use the binary representation of i, consisting of exactly $\lceil \log_2(\mid D \mid) \rceil$ digits (trailing zeroes are added as appropriate). A boolean variable from the appropriate vector is assigned to each of the digits in the binary representation. The conjunction for d_i is then constructed from those variables, where variables that correspond to 0 are negated, and the others are not. As an example, consider a domain D with four elements—d_0, d_1, d_2, and d_3. We need two boolean variables v_D^0 and v_D^1 for the mapping ξ, the construction of which is given in Table 3.1.

To use this encoding scheme in practice, a mapping from events and states to natural numbers is required. The text-based CSP language enforces the up-front declaration of all events and channels in a CSP script via the channel keyword [For97]. Therefore, the size of the universal domain of events Σ is known before process compilation starts and it is straightforward to come up with a mapping from events to natural numbers.

In contrast, the compilation of a process definition may require an arbitrary (that is, unknown in advance) number of states, yet conjunctions for new unique states have to be generated for the construction of the LTS representation before the size of $\mathcal{S}$ is known. In the ARC tool, this problem is resolved by assigning a sufficiently large number of boolean variables to each sequential component P_i. The assignment of integers to states uses a simple counter, which starts from 0 and is incremented by 1 for each new unique state required by the compilation. It has been shown that, on the average, using successive natural numbers for adjacent state conjunctions is more efficient in terms of OBDD size of the final LTS than alternative encoding schemes such as Gray code [TM96]. When the full LTS for P_i is constructed, the size of $\mathcal{S}_{P_i}$ becomes known, and it is likely that more variables have been used in the encoding than actually needed for the states in P_i. Since those extra variables do not contribute to the uniqueness of

each state's encoding, they are removed by existential quantification and may be reused for the compilation of subsequent sequential components.

A related but, in our opinion, insufficiently explored issue is how the variable vector $V_{\mathcal{S}}$ (and, respectively, $V'_{\mathcal{S}}$) is used when compiling a process definition and how new states are taken from the available state space when required. The size and complexity of the OBDD representation of the LTS representation of a process semantics can be very sensitive to the effectiveness of the particular variable assignment scheme.

As mentioned above, our approach splits potentially large and complex process definitions into sequential components P_i. This occurs naturally within the syntax-driven translation technique described in 3.2. In a similar way, the variable vector $V_{\mathcal{S}}$ is split into sub-vectors $V_{\mathcal{S}_i}$, one for each P_i. The number of variables in each $V_{\mathcal{S}_i}$ is exactly $\lceil \log_2(\mid \mathcal{S}_{P_i} \mid) \rceil$, i.e. the encoding of each sequential component uses the minimum number of OBDD variables possible. The sub-vectors $V_{\mathcal{S}_i}$ are gradually merged by the parallel composition operators in the definition of P to form the vector $V_{\mathcal{S}}$ (see Section 3.2.5).

Even more important to the size of the OBDD that represents the final LTS is the choice of global ordering of boolean variables [Bry92]. The ordering that has been chosen for the ARC tool is[5]:

$$v_1^{\mathcal{S}} \prec v_1'^{\mathcal{S}} \prec v_2^{\mathcal{S}} \prec v_2'^{\mathcal{S}} \prec \ldots \prec v_k^{\mathcal{S}} \prec v_k'^{\mathcal{S}} \prec v_1^{\mathcal{E}} \prec v_2^{\mathcal{E}} \prec \ldots \prec v_l^{\mathcal{E}} \prec v_1^{\mathcal{U}} \prec v_2^{\mathcal{U}} \prec \ldots \prec v_{2^l-2}^{\mathcal{U}}$$

where variables $v_i^{\mathcal{U}}$ belong to the vector $V_{\mathcal{U}}$ which is used to compute and store maximal refusals (see Section 3.2.4). The number of variables in $V_{\mathcal{U}}$ is $2^l - 2$, where l is the number of boolean variables in $V_{\mathcal{E}}$, because the special events τ and $\checkmark$ are ignored for the computation of maximal refusals. The exact ordering of the variables from $V_{\mathcal{U}}$ is not too critical because implicit conjunction is used for storing process refusals.

Various researchers have reported that interleaving the variables $v_i^{\mathcal{S}}$ and $v_i'^{\mathcal{S}}$ provides best results for encoding transition relations [EFT91, Bry92, ATB94]. Another popular heuristic is to keep the state variables corresponding to interacting sequential components as close as possible [Hu95]. The variable assignment technique presented in this section achieves that by composing the sub-vectors $V_{\mathcal{S}_i}$ and $V'_{\mathcal{S}_i}$ in a syntax-driven manner thus forming a sub-vector ordering that closely reflects the synchronisation patterns of the system.

Our experience with the ARC tool so far has strongly indicated that putting variables from the vector $V_{\mathcal{E}}$ after those in vectors $V_{\mathcal{S}}$ and $V'_{\mathcal{S}}$ consistently leads

[5]Actually, this is the default variable ordering for the ARC tool which allows for flexible, user-defined ordering controlled by a command-line option (see Chapter 7).

to smaller OBDD sizes and faster verification times compared to putting $V_{\mathcal{E}}$ in front of $V_{\mathcal{S}}$ and $V'_{\mathcal{S}}$. Note that this differs from Enders *et al.*'s proposed ordering [EFT91], which is most likely due to the different OBDD encoding strategy and the disparate mix of operations on the final LTS by the respective verification algorithms.

3.2.4 Computing CSP semantics on LTS representations

Given a process term compiled into an LTS form, a method of computing its semantics is required in order to check the properties of that process and its relationship to other processes. In other words, we have to define the semantic functions $traces_L$, $failures_L$, and $divergences_L$ which operate on an LTS $\mathcal{L} = (\mathcal{S}, \mathcal{E}, \mathcal{R}, \mathcal{I})$ of a compiled process P, instead of its syntactical representation.

It should be noted that, for the purposes of refinement checking, one does not need separate computations to obtain $failures(P)$ and $failures_\perp(P)$. By definition, these two functions differ only for divergent traces, and our refinement checking algorithm does not perform reachability analysis beyond the traces at which a process diverges (see Section 3.3.2). Therefore, one only needs to compute the semantic function for stable refusals $failures_L$ as long as it is not used on traces for which $divergences_L$ computes to true.

It is important to make a distinction between an LTS state and a state of a process that is translated to an LTS form. An LTS state γ is merely a single member of the set of all states: $\gamma \in \mathcal{S}$. A process, on the other hand, may exhibit nondeterministic behaviour; thus a process state Γ is represented as a set of LTS states: $\Gamma \subseteq \mathcal{S}$.

The traces of P can be computed by traversing the graph of $\mathcal{L}$ and recording the sequence of visible events (that is, all events excluding τ) labeling the transitions being taken:

$$traces(P) = traces_L(\mathcal{I}, \mathcal{L}) = \bigcup_{\gamma \in \mathcal{I}} traces_L(\gamma, \mathcal{L}) = \{\langle\rangle\} \cup$$
$$\bigcup_{\gamma \in \mathcal{I}} \bigcup_{a \in NextEvents(\{\gamma\}, \mathcal{L})} \langle a \rangle \hat{} traces_L(NextStates(\{\gamma\}, \{a\}, \mathcal{L}), \mathcal{L}) \tag{3.1}$$

The above equation is clearly valid for any process state and is useful when computing $traces(P/s)$. The computation of failures involves computing the refusals of a process after a certain trace (or at a certain process state). Because of the nondeterminism inherent in the process state, we have:

$$refusals_L(\Gamma, \mathcal{L}) = \bigcup_{\gamma \in \Gamma} refusals_L(\{\gamma\}, \mathcal{L}) \tag{3.2}$$

and thus we need a way to compute the refusals of a single LTS state. To accomplish this, we consider three possibilities:

1. The LTS state γ has no outgoing transitions labelled with τ or $\checkmark$ (such states are called *stable* [Ros97]), and the set of labels of the outgoing transitions is A_γ. A process in this state will accept any event from A_γ and refuse any other event:

$$refusals_L(\{\gamma\}, \mathcal{L}) = \{\Sigma - A_\gamma\}$$

2. The LTS state γ has an outgoing transition labelled with $\checkmark$. A process in this state may refuse any subset of Σ:

$$refusals_L(\{\gamma\}, \mathcal{L}) = \{\Sigma\}$$

since the environment of that process cannot stop it from terminating by following the $\checkmark$ transition. Note that after termination the process enters an LTS state with no outgoing transitions and continues to refuse $\{\Sigma\}$;

3. There is a single outgoing τ-transition from γ to another state λ, where λ is stable and has outgoing transitions labelled with events from the set A_λ. While in state γ, a process may accept any event from the set $A_\gamma \cup A_\lambda$, because the invisible transition is bound to eventually happen [Ros97]. On the other hand, the τ-transition may overtake the offering of any visible event from A_γ, moving the process into the state λ. Whether or not the τ-transition is going to "wait" for a transition on event from A_γ to happen is nondeterministic, and in this case we have:

$$refusals_L(\{\gamma\}, \mathcal{L}) = \{\Sigma - A_\gamma - A_\lambda, \Sigma - A_\lambda\} = \{\Sigma - A_\lambda\}$$

since we are interested in the maximal refusals only;

4. The LTS state γ has n τ-transitions to the stable LTS states $\{\lambda_i\}_{i=1}^n$. The set of visible transition labels in γ is A_γ, and that of λ_i is A_{λ_i}. While in state γ, a process may accept any event from A_γ and any event from one of the sets A_{λ_i} (the choice of which λ_i the process will be in after a τ-transition is nondeterministic). Similarly to the previous case, we have:

$$refusals_L(\{\gamma\}, \mathcal{L}) = \bigcup_i\{\Sigma - A_\gamma - A_{\lambda_i}\} \cup \bigcup_i\{\Sigma - A_{\lambda_i}\} = \bigcup_i\{\Sigma - A_{\lambda_i}\}$$

as we require maximal refusals.

The results from the three cases above can be combined to derive the function $refusals_L$ in the general case using induction. Recall that the set of all LTS states reachable from a given LTS state γ by a sequence of τ transitions and at most one transition labelled with $\surd$ is computed by the function $TauExpand(\{\gamma\}, \mathcal{L})$ (defined in Section 2.2). By unfolding $refusals_L$ until all states reachable via τ-transitions are covered, we obtain:

$$refusals_L(\{\gamma\}, \mathcal{L}) = \bigcup_{\lambda \in TauExpand(\{\gamma\}, \mathcal{L})} \{\Sigma - A_\lambda\} = \bigcup_{\lambda \in TauExpand(\{\gamma\}, \mathcal{L}) \wedge \nexists \theta : \lambda \xrightarrow{\tau} \theta \in \mathcal{R}} \{\Sigma - NextEvents(\{\lambda\}, \mathcal{L})\} \tag{3.3}$$

By combining Equations (3.2) and (3.3), we derive:

$$refusals_L(\Gamma, \mathcal{L}) = \bigcup_{\lambda \in TauExpand(\Gamma, \mathcal{L}) \wedge \nexists \theta : \lambda \xrightarrow{\tau} \theta \in \mathcal{R}} \{\Sigma - NextEvents(\{\lambda\}, \mathcal{L})\} \tag{3.4}$$

We are now prepared to formulate the equation for $failures_L$, using equations (3.1) and (3.4):

$$failures_L(\Gamma, \mathcal{L}) = \{(t, r) \mid t \in traces_L(\Gamma, \mathcal{L}), r = refusals_L(\Pi(\Gamma, t, \mathcal{L}), \mathcal{L})\} \tag{3.5}$$

where

$$\Pi(\Gamma, t, \mathcal{L}) = \{\pi_{\#t} \mid \exists \{\pi_i\}_{i=0}^{\#t} : \pi_0 \in \Gamma \wedge \forall\, i \in \{1, 2, \ldots, \#t\} : \pi_{i-1} \xrightarrow{\tau^* t_i} \pi_i \in \mathcal{R}^+\}$$

is the set of LTS states reachable from Γ via a sequence of visible transitions t.

The divergent states of $\mathcal{L}$ are those which [Ros94]:

- Are part of a τ-loop, that is, states γ such that $\gamma \xrightarrow{\tau^*} \gamma \in \mathcal{R}^+$, and
- Have a sequence of τ-transitions to states γ as above.

Therefore, the equation for $divergences_L$ can be expressed as:

$$divergences_L(\mathcal{L}) = \{\gamma \mid \exists\, \lambda \in \mathcal{S} : \gamma \xrightarrow{\tau^*} \lambda \in \mathcal{R}^+ \wedge \lambda \xrightarrow{\tau^*} \lambda \in \mathcal{R}^+\} \tag{3.6}$$

and a process in state Γ is divergent if and only if:

$$\Gamma \cap divergences_L(\mathcal{L}) \neq \{\} \tag{3.7}$$

3.2.5 Syntax-driven transformation rules

Having presented the underlying issues of handling identifiers, expressions, recursion, and state and event encoding, we are ready to present the syntax-driven transformation rules controlling the compilation of the semantics of a process into OBDDs. In this section, we only consider the CSP subset defined by Equation (2.1), although rules for other CSP operators and predefined processes, such as the chain operator $\gg$ or the *Chaos* process, may also be constructed in a similar way [Ros97, p. 164–165].

We assume that the process term to be compiled is converted from its textual representation into a syntax tree form, the non-leaf nodes of which contain CSP operators. We proceed to define the mapping:

$$\psi : CSPTerm \rightarrow LTS \cup \{error\}$$

for each of the CSP operators from Equation (2.1), making use of an identifier context $\mathcal{C}_v$ and a recursion context $\mathcal{C}_r$. The special value *error* is obtained when compilation does not succeed. $\mathcal{C}_v$ contains the identifier values of the process definition being compiled, while $\mathcal{C}_r$ is used to pass on process signatures from previously compiled CSP terms and is updated with process signatures encountered while compiling the terms associated with the current operator. In defining ψ, we closely follow the operational semantics of CSP provided elsewhere [JH93, Ros97].

For the rest of this section, it is assumed that $\psi[\![P]\!] = (\mathcal{S}_P, \mathcal{E}_P, \mathcal{R}_P, \mathcal{I}_P)$ with a recursion context $\mathcal{C}_r^P$, $\psi[\![Q]\!] = (\mathcal{S}_Q, \mathcal{E}_Q, \mathcal{R}_Q, \mathcal{I}_Q)$ with a recursion context $\mathcal{C}_r^Q$, and γ, λ and θ are distinct LTS states. Also, the notation $\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \mathcal{C}_r^X$ is used to denote updates to the recursion context that occur during process compilation.

Process *Stop* has only one state and no transitions (Figure 3.5a):

$$\psi[\![Stop]\!]_{\mathcal{C}_v, \mathcal{C}_r \leftarrow \mathcal{C}_r \cup \{Stop \mapsto \{\gamma\}\}} = (\{\gamma\}, \{\}, \{\}, \{\gamma\})$$

As *Stop* has the same semantics in any recursion context, it is added as a separate process signature to $\mathcal{C}_r$. This allows other occurrences of *Stop* to reuse the state γ. Process *Skip* also has a rather simple LTS semantic representation having two states and a single transition between them, labelled with the special event $\checkmark$ (Figure 3.5b):

$$\psi[\![Skip]\!]_{\mathcal{C}_v, \mathcal{C}_r \leftarrow \mathcal{C}_r \cup \{Skip \mapsto \{\gamma\}\}} = (\{\gamma, \lambda\}, \{\checkmark\}, \{\gamma \xrightarrow{\checkmark} \lambda\}, \{\gamma\})$$

As with process *Stop*, *Skip* is also added to $\mathcal{C}_r$ with the intent to reuse its LTS whenever possible. In general, this decision helps to reduce the final LTS for certain processes (see Section 3.2.6). The mapping for the prefix operator is

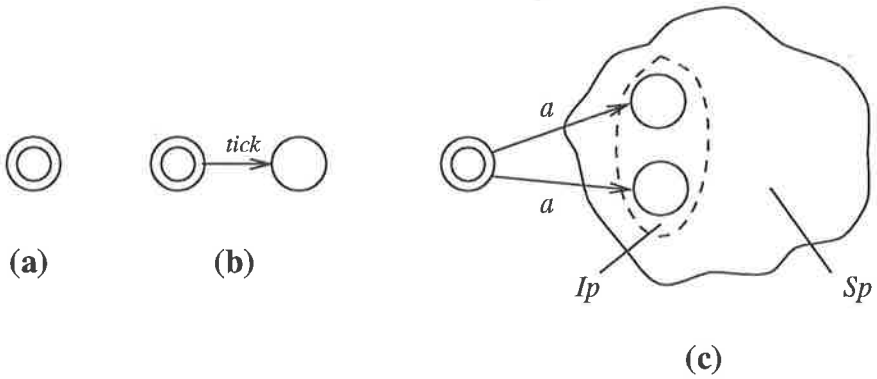


Figure 3.5: The mapping ψ for *Stop*, *Skip*, and prefix operators

defined as (Figure 3.5c):

$$\begin{aligned}\psi[\![a \rightarrow P]\!]_{\mathcal{C}_v,\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \mathcal{C}_r^P} \quad = \quad & (\{\gamma\} \cup \mathcal{S}_P, \\ & \{a\} \cup \mathcal{E}_P, \\ & \{\gamma \xrightarrow{a} \lambda \mid \lambda \in \mathcal{I}_P\} \cup \mathcal{R}_P, \\ & \{\gamma\})\end{aligned}$$

where $\gamma \notin \mathcal{S}_P$ and P is compiled with the recursion context $\mathcal{C}_r$, because prefix is a preserving operator. This can be seen by examining the resulting transition relation—the LTS of P is not modified in any way other than by adding transitions to its initial states. The internal choice operator is also a preserving one and essentially joins all the elements in the LTS tuples of P and Q (Figure 3.6a):

$$\psi[\![P \sqcap Q]\!]_{\mathcal{C}_v,\mathcal{C}_r \leftarrow (\mathcal{C}_r \cup \mathcal{C}_r^P \cup \mathcal{C}_r^Q)} = (\mathcal{S}_P \cup \mathcal{S}_Q, \mathcal{E}_P \cup \mathcal{E}_Q, \mathcal{R}_P \cup \mathcal{R}_Q, \mathcal{I}_P \cup \mathcal{I}_Q)$$

however, the relationship between the recursion contexts $\mathcal{C}_r$, $\mathcal{C}_r^P$, and $\mathcal{C}_r^Q$ is more complicated. Assuming a left-to-right translation sequence[6], process P is compiled first with the recursion context $\mathcal{C}_r$, followed by the compilation of Q with the recursion context $\mathcal{C}_r \cup \mathcal{C}_r^P$.

The renaming operator $f[P]$ is a typical example of a non-preserving operator. It does not add or remove states or transitions in the LTS of P, but rather modifies the transition labels (events) of $\mathcal{R}_P$. Because of this, no recursion context is

[6]Either a left-to-right or a right-to-left translation sequence could be used, because internal choice is a symmetric operator: $((P) \sqcap Q) = (P \sqcap (Q))$.

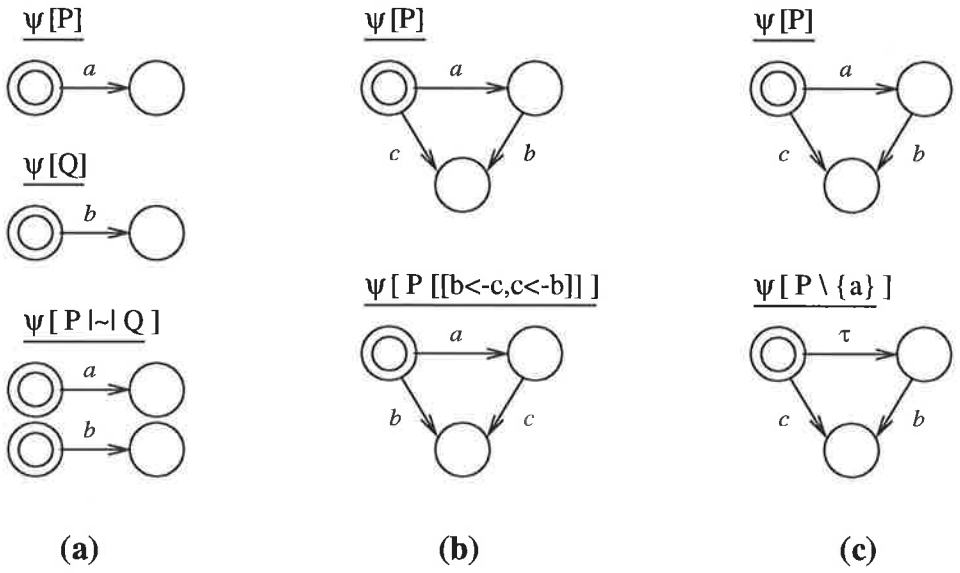


Figure 3.6: Example mappings for internal choice, renaming and hiding

passed from the environment of $f[P]$ to P and vice versa (Figure 3.6b):

$$\begin{aligned}\psi[\![f[P]]\!]_{c_v,c_r\leftarrow c_r} = \; & (\mathcal{S}_P, \\ & \{f(a) \mid a \in \mathcal{E}_P\}, \\ & \{\gamma \xrightarrow{f(a)} \lambda \mid \gamma \xrightarrow{a} \lambda \in \mathcal{R}_P\}, \\ & \mathcal{I}_P)\end{aligned}$$

The OBDD encoding for the renaming operator requires further clarification. The boolean formula $\phi(\psi[\![f[P]]\!])$ can be constructed in two different ways. Firstly, it is possible to use as many iterations as the number of events renamed by f and change the transition labels in $\mathcal{E}_P$ and $\mathcal{R}_P$ one event at a time. Secondly, it is possible to create an OBDD representation of f and apply renaming as a composition of relational product and variable renaming. While the second approach is clearly more efficient, either of these techniques is much less computationally intensive than iterating through all transitions in $\mathcal{R}_P$ one transition at a time that an explicit LTS building approach would require.

Hiding can be regarded as a special case of renaming, wherein a certain set of

event labels A is substituted by the internal event τ (Figure 3.6c):

$$\begin{aligned}\psi[\![P \setminus A]\!]_{\mathcal{C}_v,\mathcal{C}_r \leftarrow \mathcal{C}_r} \;=\; & (\mathcal{S}_P, \\ & \mathcal{E}_P - A, \\ & \{\gamma \xrightarrow{a} \lambda \mid \gamma \xrightarrow{a} \lambda \in \mathcal{R}_P \wedge a \notin A\} \cup \\ & \{\gamma \xrightarrow{\tau} \lambda \mid \gamma \xrightarrow{a} \lambda \in \mathcal{R}_P \wedge a \in A\}, \\ & \mathcal{I}_P)\end{aligned}$$

The computation of $\phi(\psi[\![P \setminus A]\!])$ requires a constant number of boolean operations, and thus is very efficient. Another observation of practical importance is that $\phi(\psi[\![P \setminus A]\!])$ contains at most as many OBDD nodes as $\phi(\psi[\![P]\!])$ due to the simplification of the boolean function for the transition relation after hiding. Although the LTS of $P \setminus A$ has exactly the same number of states and transitions as P, the performance of the pair-by-pair refinement checking algorithm benefits a lot from hiding (see Sections 3.3 and 3.3.3). In contrast, most explicit model checking techniques[7] do not utilise the potential advantages of hiding as a means for reducing the complexity of verification.

Whereas the internal choice operator can be thought of as introducing *explicit* nondeterminism in a process and its LTS representation by increasing the number of initial states of that LTS, hiding leads to *implicit* nondeterminism by introducing internal events, the execution of which cannot be controlled or even registered by the environment of the process. Distinguishing between the two forms of nondeterminism in an LTS is important for the correct definition and efficient implementation of ψ for the external choice operator. The MRC tool fails to acknowledge that external choice cannot be resolved by a sequence of internal events (cf. [BMTV94]), while a superseded version of the FDR tool (version 1.42) used hiding as a high-level operator and external choice as a low-level one, effectively disallowing CSP terms like $(P \setminus A) \,\Box\, (Q \setminus B)$ [For93].

The mapping function for external choice is indeed quite complicated. Therefore, it is presented in two steps:

- Firstly, we present a simpler definition that overcomes the semantic inaccuracy of the MRC tool but is not as efficient in terms of size of the OBDD representation of the computed LTS as it could be;
- Secondly, we elaborate on a more complicated algorithmic approach which is, however, more efficient in terms of OBDD size of the resulting LTS.

[7] Excluding those which use some form of post-processing of the LTS targeted at minimizing it while preserving its semantics with respect to a certain relation. However, LTS minimisation (or compression, as used in the context of the FDR2 tool [For97]) can be computationally expensive and is, in general, only applicable to relatively small LTS.

Our simpler version of the definition of the mapping function for external choice overcomes the problem of the mapping used by Barrett *et al.* [BMTV94] by adding two more terms (marked with (*)) to the equation:

$$
\begin{aligned}
\psi[\![P \,\Box\, Q]\!] \;=\; & ((\mathcal{S}_P \cup \{\xi_P\}) \times (\mathcal{S}_Q \cup \{\xi_Q\}), \\
& \mathcal{E}_P \cup \mathcal{E}_Q, \\
& \{(\lambda,\theta) \xrightarrow{\tau} (\lambda',\theta) \mid \theta \in \mathcal{S}_Q \wedge \lambda \xrightarrow{\tau} \lambda' \in \mathcal{R}_P\} \cup && (*) \\
& \{(\lambda,\theta) \xrightarrow{\tau} (\lambda,\theta') \mid \lambda \in \mathcal{S}_P \wedge \theta \xrightarrow{\tau} \theta' \in \mathcal{R}_Q\} \cup && (*) \\
& \{(\lambda,\theta) \xrightarrow{a} (\lambda',\xi_Q) \mid a \neq \tau \wedge \theta \in \mathcal{S}_Q \wedge \lambda \xrightarrow{a} \lambda' \in \mathcal{R}_P\} \cup && (\dagger) \\
& \{(\lambda,\theta) \xrightarrow{a} (\xi_P,\theta') \mid a \neq \tau \wedge \lambda \in \mathcal{S}_P \wedge \theta \xrightarrow{a} \theta' \in \mathcal{R}_Q\} \cup && (\dagger) \\
& \{(\lambda,\xi_Q) \xrightarrow{a} (\lambda',\xi_Q) \mid \lambda \xrightarrow{a} \lambda' \in \mathcal{R}_P\} \cup && (\ddagger) \\
& \{(\xi_P,\theta) \xrightarrow{a} (\xi_P,\theta') \mid \theta \xrightarrow{a} \theta' \in \mathcal{R}_Q\}, && (\ddagger) \\
& I_P \times I_Q)
\end{aligned}
$$

The above equation requires some explanation. There are two new states $\xi_P, \xi_Q \notin \mathcal{S}_P \cup \mathcal{S}_Q$ added to the state spaces of, respectively, processes P and Q. The terms marked with (*) capture the property of the external choice operator that it cannot be resolved by a sequence of internal transitions (i.e. neither P nor Q will be chosen after τ^*). The terms marked with (†) stand for the choice between P and Q; when this happens the process which has not been chosen makes a transition to its respective ξ state. Finally, the terms marked with (‡) represent the behaviour of $P \,\Box\, Q$ after one of the processes has been already chosen.

As far as recursion contexts are concerned, P and Q are compiled with empty ones because they are translated into separate state domains. This is rather restrictive, as it means that processes such as:

$$P \mathrel{\widehat{=}} a \rightarrow P \,\Box\, b \rightarrow P$$

which are quite common in practice (see Appendix A) cannot be successfully compiled as the compilation would result in infinite recursion. Furthermore, this definition of ψ for external choice is quite inefficient, as it requires building the product of the state spaces of P and Q. Although this operation is relatively inexpensive in terms of time when using OBDDs, it certainly increases the number of OBDD variables and OBDD nodes in the final LTS. Considering that external choice is an operator used frequently in CSP, any inefficiency in its LTS implementation is likely to become a serious bottleneck when compiling and checking larger processes.

To overcome the shortcomings of the above compilation approach for external choice, a new algorithmic approach has been developed by the author [PY96a]. In contrast to building the product of two state spaces, it adds only a small number

of additional states to the union of the original state spaces. Our algorithm works in two steps.

The first step transforms $\psi[\![P]\!]$ and $\psi[\![Q]\!]$ into $\psi[\![P]\!]'$ and $\psi[\![Q]\!]'$ by removing the potential implicit nondeterminism in $\mathcal{I}_P$ and $\mathcal{I}_Q$ while preserving process semantics in $\mathcal{F}$ and $\mathcal{N}$. The initial states of the derived LTSs may have a τ transition to themselves in order to preserve divergences, but do not have internal transitions to any other states. This is achieved by the application of the LTS transformation function $TauTransform : LTS \rightarrow LTS$, which works as follows (Figure 3.7a):

1. It derives the set of states Γ reachable from any LTS state in $\mathcal{I}$ via a sequence of invisible transitions that have at least one possible visible transition or no transitions at all (a *Stop* state):

$$\begin{aligned}\Gamma \;=\; & \{\gamma \mid \gamma \in TauExpand(\mathcal{I}, \mathcal{L}) \wedge \\ & (\exists\, a \neq \tau : \gamma \xrightarrow{a} \lambda \in \mathcal{R} \vee \nexists\, a : \gamma \xrightarrow{a} \lambda \in \mathcal{R})\}\end{aligned}$$

2. If $\Gamma = \{\gamma_i\}_{i=1}^{n}$, a new set of states $\Lambda = \{\lambda_i\}_{i=1}^{n}$ is created, such that $\Lambda \cap \mathcal{S} = \{\}$, and there is a one-to-one mapping function $f_{map} : \Lambda \rightarrow \Gamma$;

3. The new LTS $\mathcal{L}'$ is then constructed as follows:

$$\begin{aligned}\mathcal{L}' \;=\; & (\mathcal{S} \cup \Lambda, \\ & \mathcal{E}, \\ & \mathcal{R} \cup & (*) \\ & \{\lambda \xrightarrow{a} \theta \mid \exists\, a \in \Sigma : \lambda \in \Lambda \wedge \theta \in NextStates(\{f_{map}(\lambda)\}, \{a\}, \mathcal{L})\} \cup & (\dagger) \\ & \{\lambda \xrightarrow{\tau} \lambda \mid f_{map}(\lambda) \in divergences_L(\mathcal{L})\}, & (\ddagger) \\ & \Lambda)\end{aligned}$$

   where the function *NextStates*, introduced in Section 2.2, is used to compute the set of states reachable from $f_{map}(\lambda)$ via a sequence of transitions from the set $\tau^* a$. In the above equation, the term marked with $(*)$ preserves all transitions from $\mathcal{L}$, the term marked with $(\dagger)$ adds transitions from Λ to $\mathcal{S}$, and the term marked with $(\ddagger$ is introduced to preserve the potential divergencies.

Theorem 1 *TauTransform* preserves the failures semantics, that is:

$$failures_L(\mathcal{I}, \mathcal{L}) = failures_L(\mathcal{I}', \mathcal{L}')$$

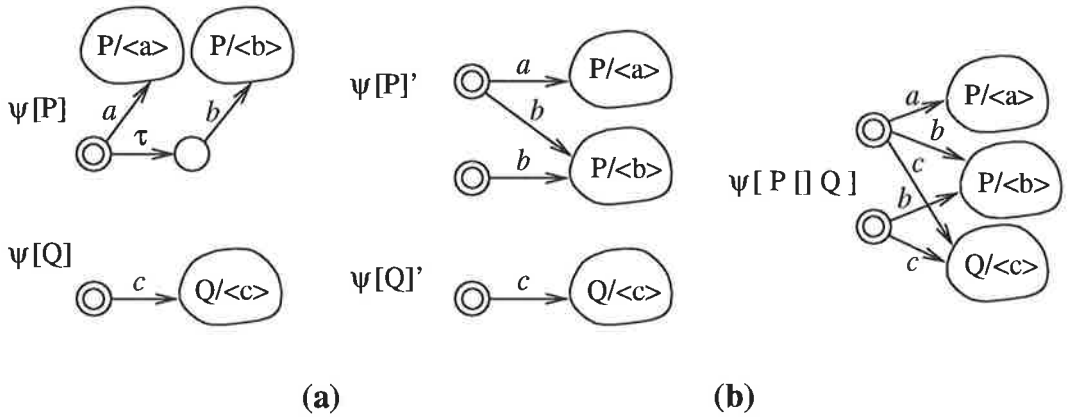


Figure 3.7: The mapping for external choice (some states are omitted)

Proof: To prove this theorem, we use Equations (3.1), (3.4), and (3.5) from Section 3.2.4. From the definition of *TauTransform* we conclude that:

$$NextEvents(\mathcal{I},\mathcal{L}) = NextEvents(\mathcal{I}',\mathcal{L}') \quad (3.8)$$

$$NextStates(\mathcal{I},\mathcal{L}) = NextStates(\mathcal{I}',\mathcal{L}') \quad (3.9)$$

Another property of $\mathcal{L}'$ that follows from the definition of *TauTransform* is that the states $\mathcal{I}'$ are unreachable after the first transition, and therefore:

$$\mathcal{R}' \triangleright \{\mathcal{S}' - \mathcal{I}'\} = \mathcal{R}$$

where the $\triangleright$ operator restricts a labelled transition relation. From this and Equations (3.1), (3.8), and (3.9) it follows that:

$$traces_L(\mathcal{I},\mathcal{L}) = traces_L(\mathcal{I}',\mathcal{L}') \quad (3.10)$$

$$\forall\, t \in traces_L(\mathcal{I},\mathcal{L}), t \neq \langle\rangle \;:\; \Pi(\mathcal{I},t,\mathcal{L}) = \Pi(\mathcal{I}',t,\mathcal{L}') \quad (3.11)$$

From Equations (3.5), (3.10), and (3.11) we derive:

$$failures_L(\mathcal{I},\mathcal{L}) = failures_L(\mathcal{I}',\mathcal{L}') \Leftrightarrow refusals_L(\mathcal{I},\mathcal{L}) = refusals_L(\mathcal{I}',\mathcal{L}') \quad (3.12)$$

Applying Equation (3.4) to $\mathcal{L}$ and $\mathcal{L}'$, we have:

$$refusals_L(\mathcal{I},\mathcal{L}) = \bigcup_{\gamma \in TauExpand(\mathcal{I},\mathcal{L}) \wedge \nexists \lambda : \gamma \xrightarrow{\tau} \lambda \in \mathcal{R}} \{\{\Sigma - NextEvents(\{\gamma\},\mathcal{L})\}\} \quad (3.13)$$

and

$$refusals_L(\mathcal{I}',\mathcal{L}') = \bigcup_{\gamma \in \mathcal{I}' \wedge \gamma \xrightarrow{\tau} \gamma \notin \mathcal{R}'} \{\{\Sigma - NextEvents(\{\gamma\},\mathcal{L}')\}\} \quad (3.14)$$

The definition of *TauTransform* implies that for any $\gamma \in \mathcal{I}'$:

$$NextEvents(\{\gamma\}, \mathcal{L}') = NextEvents(\{f_{map}(\gamma)\}, \mathcal{L}) \tag{3.15}$$

and that:

$$\{\gamma \mid \gamma \in TauExpand(\mathcal{I}, \mathcal{L}) \wedge \nexists \lambda : \gamma \xrightarrow{\tau} \lambda \in \mathcal{R}\} \subseteq \bigcup_{\gamma \in \mathcal{I}' \wedge \gamma \xrightarrow{\tau} \gamma \notin \mathcal{R}'} f_{map}(\gamma) \subseteq TauExpand(\mathcal{I}, \mathcal{L}) \tag{3.16}$$

From Equations (3.13), (3.14), (3.15), and (3.16) we conclude that:

$$refusals_L(\mathcal{I}, \mathcal{L}) = refusals_L(\mathcal{I}', \mathcal{L}')$$

and applying Equation (3.12) proves the theorem. ■

In general, the space cost of applying *TauTransform* is the addition of n new states constituting the set Γ. In practice, however, some of the states from $TauExpand(\mathcal{I}, \mathcal{L})$ may become unreachable in $\mathcal{L}'$ and, therefore, can be safely removed from the state space of $\mathcal{L}'$. Thus, the number of additional states actually required by *TauTransform* is potentially fewer than n.

The second step in our algorithmic approach to the translation of the external choice operator is combining $TauTransform(\psi[\![P]\!])$ and $TauTransform(\psi[\![Q]\!])$ (Figure 3.7b):

$$\begin{aligned}
\psi[\![P \,\Box\, Q]\!]_{\mathcal{C}_v, \mathcal{C}_r \leftarrow \mathcal{C}_r \cup \mathcal{C}_r^P \cup \mathcal{C}_r^Q} \;=\; & (\mathcal{S}'_P \cup \mathcal{S}'_Q \cup (\mathcal{I}'_P \times \mathcal{I}'_Q) - \mathcal{I}'_P - \mathcal{I}'_Q, \\
& \mathcal{E}'_P \cup \mathcal{E}'_Q, \\
& \mathcal{R}'_P \cup \mathcal{R}'_Q \cup \\
& \{(\gamma_P^i, \gamma_Q^j) \xrightarrow{a} \lambda \mid \\
& \gamma_P^i \xrightarrow{a} \lambda \in \mathcal{R}'_P \vee \gamma_Q^j \xrightarrow{a} \lambda \in \mathcal{R}'_Q\} \cup \\
& \{(\gamma_P^i, \gamma_Q^j) \xrightarrow{\tau} (\gamma_P^i, \gamma_Q^j) \mid \qquad (*) \\
& f_{map}^P(\gamma_P^i) \in divergences_L(P) \vee \qquad (*) \\
& f_{map}^Q(\gamma_Q^j) \in divergences_L(Q)\}, \qquad (*) \\
& (\mathcal{I}'_P \times \mathcal{I}'_Q))
\end{aligned}$$

where $\mathcal{I}'_P = \{\gamma_P^i\}_{i=1}^m$ and $\mathcal{I}'_Q = \{\gamma_Q^j\}_{j=1}^n$. The term marked with (*) is used to preserve the divergence in $P \,\Box\, Q$ after the trace $\langle\rangle$ whenever P or Q are divergent after the trace $\langle\rangle$.

A definite advantage of this algorithmic approach is that P and Q are compiled in the same state space domain—that of the sequential component to which they belong. Also, the algorithmic transformations preserve the initial transition relations of both P and Q; the added states and transitions are clearly unreachable from the original state spaces. This makes the external choice operator a

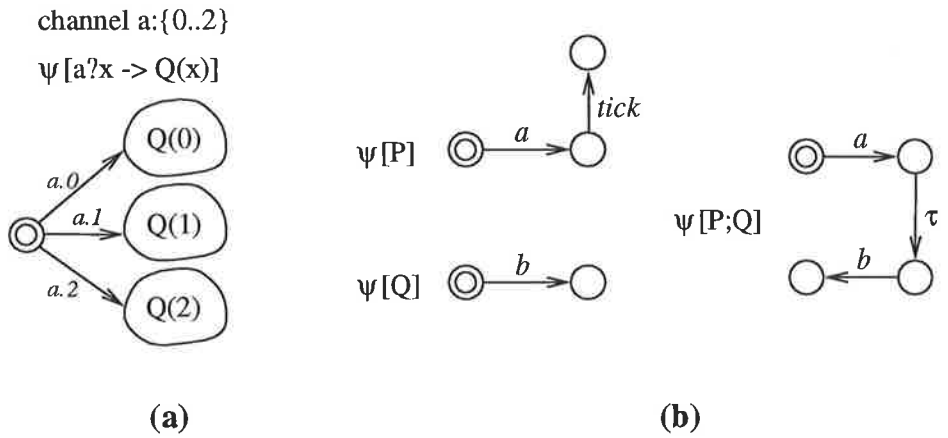


Figure 3.8: The mapping for channel input and sequential composition

preserving one. As a consequence, assuming left-to-right order of compilation[8], P is compiled with the recursion context $\mathcal{C}_r$, and Q with the recursion context $\mathcal{C}_r \cup \mathcal{C}_r^P$.

The product $(\mathcal{I}'_P \times \mathcal{I}'_Q)$ used above does not introduce substantial overhead; it merely means that mn new LTS states are created. Since $\mathcal{L}'_P$ and $\mathcal{L}'_Q$ are derived using *TauTransform*, $\mathcal{I}'_P$ and $\mathcal{I}'_Q$ become unreachable in $\psi[\![P \,\Box\, Q]\!]$ and can be removed from it. Thus, the second step of our algorithm requires $mn - m - n$ new states. The total number of states in $\psi[\![P \,\Box\, Q]\!]$ then becomes:

$$(\mid \mathcal{S}_P \mid + m) + (\mid \mathcal{S}_Q \mid + n) + (mn - m - n) = \mid \mathcal{S}_P \mid + \mid \mathcal{S}_Q \mid + mn$$

which compares quite favourably to the cost of the initial translation of external choice $\mid \mathcal{S}_P \mid \times \mid \mathcal{S}_Q \mid$. In most practical CSP examples, neither P nor Q have internal transitions possible from their initial states, in which case at most one new state has to be added.

There are three syntactical constructs from the CSP subset defined in Equation (2.1) which do not require a separate definition of ψ, but rather can be implemented using the existing translation rules. Firstly, channel output can be thought of as a form of the prefix operator, where the potentially complex channel communication has to be converted into a concrete event. Secondly, channel input is a form of external choice, which can be compiled iteratively as discussed in Section 3.2.1. Unlike the general form of external choice, however, this can be implemented as a successive application of the prefix operator for all values of the

[8]Again, either a left-to-right or a right-to-left translation sequence could be used, because external choice is a symmetric operator: $((P) \,\Box\, Q) = (P \,\Box\, (Q))$.

input identifiers on a common initial state (Figure 3.8a), because the labels of the outgoing transitions of the initial state are clearly unique. Finally, the construct:

$$\text{if } \mathit{Cond} \text{ then } P \text{ else } Q$$

merely requires the evaluation of the logical expression *Cond* in the current identifier context and then the compilation of either P or Q depending on the value of *Cond*.

The translation rule of the sequential composition operator is (Figure 3.8b):

$$\begin{aligned}\psi[\![P;\ Q]\!]_{\mathcal{C}_v,\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \mathcal{C}_r^Q} \quad = \quad & (\mathcal{S}_P \cup \mathcal{S}_Q, \\ & \mathcal{E}_P \cup \mathcal{E}_Q, \\ & \{\gamma \xrightarrow{a} \lambda \mid \gamma \xrightarrow{a} \lambda \in \mathcal{R}_P \wedge a \neq \checkmark\} \cup \\ & \{\gamma \xrightarrow{\tau} \lambda \mid \gamma \in \mathcal{S}_P \wedge \lambda \in \mathcal{I}_Q \wedge \\ & \exists \theta \in \mathcal{S}_P : \gamma \xrightarrow{\checkmark} \theta \in \mathcal{R}_P\} \cup \mathcal{R}_Q, \\ & \mathcal{I}_P)\end{aligned}$$

where P is compiled with an empty recursion context, and Q with the recursion context $\mathcal{C}_r$. The transition relation of $\psi(P)$ is modified in order to have all $\checkmark$-transitions in P become invisible τ-transitions ending at the initial states of $\psi(Q)$. Thus, sequential composition is preserving with respect to process Q but non-preserving with respect to P.

Before presenting the translation rules for parallel composition and interleaving operators, it should be noted that there are two different treatments of the operational semantics for these operators. The difference arises in the handling of process termination (and therefore the special event $\checkmark$) within concurrent processes:

- In the "classic" treatment [Hoa85, JH93], it is assumed that the environment may prevent a process from terminating. Thus, the concurrent processes synchronise over $\checkmark$;

- In the "modern" treatment [Ros97], it is assumed that the environment cannot prevent a process from terminating when the latter is prepared to do so. However, a concurrent process terminates only when all of its components have terminated.

The above differences result in a subtle yet non-trivial difference in the CSP compilation rules. The classic operational semantics of the parallel composition

operator result in the mapping:

$$\begin{aligned}\psi[\![P \underset{A}{\parallel} Q]\!]_{\mathcal{C}_v,\mathcal{C}_r} = (&\mathcal{S}_P \times \mathcal{S}_Q,\\ &\mathcal{E}_P \cup \mathcal{E}_Q,\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{a} (\gamma'_P,\gamma_Q) \mid \gamma_Q \in \mathcal{S}_Q \wedge \gamma_P \xrightarrow{a} \gamma'_P \in \mathcal{R}_P \wedge\\ &a \notin A \cup \{\checkmark\}\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{a} (\gamma_P,\gamma'_Q) \mid \gamma_P \in \mathcal{S}_P \wedge \gamma_Q \xrightarrow{a} \gamma'_Q \in \mathcal{R}_Q \wedge\\ &a \notin A \cup \{\checkmark\}\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{a} (\gamma'_P,\gamma'_Q) \mid \gamma_P \xrightarrow{a} \gamma'_P \in \mathcal{R}_P \wedge\\ &\gamma_Q \xrightarrow{a} \gamma'_Q \in \mathcal{R}_Q \wedge a \in A \cup \{\checkmark\}\},\\ &\mathcal{I}_P \times \mathcal{I}_Q)\end{aligned}$$

where both P and Q are compiled with the empty recursion context in separate state domains. This allows the building of the product $\mathcal{S}_P \times \mathcal{S}_Q$ and the resulting LTS to be carried out on the OBDD representation implicitly using standard set operations and logical conjunction. The size of the OBDDs constructed in this way has been shown to grow polynomially to the number of the concurrent components [EFT91].

The modern treatment of the operational semantics of the parallel composition operator [Ros97, p. 164] calls for a more complicated compilation rule:

$$\begin{aligned}\psi[\![P \underset{A}{\parallel} Q]\!]_{\mathcal{C}_v,\mathcal{C}_r} = (&(\mathcal{S}_P \cup \{\xi_P\}) \times (\mathcal{S}_Q \cup \{\xi_Q\}),\\ &\mathcal{E}_P \cup \mathcal{E}_Q,\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{a} (\gamma'_P,\gamma_Q) \mid \gamma_Q \in \mathcal{S}_Q \wedge \gamma_P \xrightarrow{a} \gamma'_P \in \mathcal{R}_P \wedge\\ &a \notin A \cup \{\checkmark\}\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{\tau} (\gamma'_P,\gamma_Q) \mid \gamma_Q \in \mathcal{S}_Q \wedge \gamma_P \xrightarrow{\checkmark} \gamma'_P \in \mathcal{R}_P\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{a} (\gamma_P,\gamma'_Q) \mid \gamma_P \in \mathcal{S}_P \wedge \gamma_Q \xrightarrow{a} \gamma'_Q \in \mathcal{R}_Q \wedge\\ &a \notin A \cup \{\checkmark\}\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{\tau} (\gamma_P,\gamma'_Q) \mid \gamma_P \in \mathcal{S}_P \wedge \gamma_Q \xrightarrow{\checkmark} \gamma'_Q \in \mathcal{R}_Q\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{a} (\gamma'_P,\gamma'_Q) \mid \gamma_P \xrightarrow{a} \gamma'_P \in \mathcal{R}_P \wedge\\ &\gamma_Q \xrightarrow{a} \gamma'_Q \in \mathcal{R}_Q \wedge a \in A\} \cup\\ &\{(\gamma_P,\gamma_Q) \xrightarrow{\checkmark} (\xi_P,\xi_Q) \mid \exists \lambda_P : \lambda_P \xrightarrow{\checkmark} \gamma_P \in \mathcal{R}_P \wedge\\ &\exists \lambda_Q : \lambda_Q \xrightarrow{\checkmark} \gamma_Q \in \mathcal{R}_Q\},\\ &\mathcal{I}_P \times \mathcal{I}_Q)\end{aligned}$$

Again, P and Q are compiled with the empty recursion context in separate state domains. A new state ξ_P (ξ_Q) is added to the state space of P (Q) to allow for the introduced $\checkmark$ transition which becomes possible only after both P and Q have terminated themselves, while the $\checkmark$ events in P and Q are hidden from the environment.

The mapping ψ for the interleaving operator can be easily derived from that of parallel composition using the CSP law:

$$P \;|||\; Q \equiv P \underset{\emptyset}{\|} Q$$

and therefore is not discussed separately.

Finally, we consider the mapping for process references (recursion):

$$\psi[\![P(X)]\!]_{\mathcal{C}_v,\mathcal{C}_r \leftarrow \mathcal{C}_r} = \begin{cases} \textbf{if } \exists\, \Sigma : (P(X), \Sigma) \in \mathcal{C}_r & (\{\}, \{\}, \{\}, \Sigma) \\ \textbf{else if } \exists\, T : P(X) \mathrel{\widehat{=}} T & \psi[\![T]\!]_{\mathcal{C}_v^T,\mathcal{C}_r \leftarrow \mathcal{C}_r \cup \mathcal{C}_r^T} \\ \textbf{else} & \text{error} \end{cases}$$

where $P(X)$ is a process signature, T is a process term that is used to define the behaviour corresponding to $P(X)$ in the CSP description, $\mathcal{C}_v^T$ is the identifier context obtained by binding the formal parameters of the definition of $P(X)$ to the values of actual parameters in $P(X)$, and $\mathcal{C}_r^T$ denotes the updates to the recursion context during the compilation of the process term T.

Despite the apparent complexity of the above equation it expresses a relatively simple sequence of steps in handling process references that have been discussed to some extent in Section 3.2.2. Firstly, if the process reference $P(X)$ has been already compiled, the mapping is a simple LTS that contains the same initial states as used in the previous compilation of $P(X)$ (the other elements of the resulting LTS are empty because they are "reused"). Secondly, if $P(X)$ has not yet been compiled and there exists a definition that matches it in the form $P(X) \mathrel{\widehat{=}} T$, the mapping is the same as the mapping for T with the appropriate identifier context. Finally, if $P(X)$ has no definition, the result is the special value *error* which can be used as a signal to abort the compilation process.

3.2.6 Re-encoding of sequential components

For a number of reasons, the size of the LTS (its number of states and transitions) and its OBDD representation derived by applying the compilation rules presented in the previous section may not be optimal. Some of the states in the LTS may become unreachable and some of its transitions may never be followed, especially for definitions that contain general choice, recursion, and the *Stop* and *Skip* processes. LTS minimisation techniques aim to reduce the size of an LTS while preserving the semantics implemented by this LTS with respect to the chosen semantic model.

Roscoe *et al.* [RGG+95] present several state compression algorithms which have been incorporated into the FDR2 tool [For97]. Although the implementation of their techniques is based on explicit state space exploration as present in FDR2,

an OBDD-based implementation of those algorithms is certainly possible. For example, an OBDD-based strong bisimulation minimisation algorithm has been developed by Bouali and de Simone [BdS92].

We propose the use of a much simpler but nevertheless useful technique that re-encodes sequential components whose LTSs are already encoded in OBDD form as a result of the translation process described in the previous section. The idea is simple—the LTS that is to be minimised is traversed starting from its initial states, and an isomorphic LTS is constructed. The latter differs from the original in the OBDD encoding of its states, and does not encode any unreachable states and transitions that may be present in the original LTS.

To understand the usefulness of this approach, let us assume that a particular process with four reachable states requires nine states during its compilation. The compilation will derive an OBDD representation that uses four OBDD variables to encode just four unique states, which is clearly inefficient. The re-encoding approach suggested above reduces the number of required variables for the LTS of the process by four (two from the variable vector V_S and two from V'_S) and decreases the size of the OBDD as well. Since re-encoding of states occurs concurrently with the traversal of the initial LTS, two states of the resulting LTS connected by a transition have, in general, very similar encodings, which has the potential to further reduce the number of OBDD nodes representing the modified LTS.

Compared to most other LTS minimisation techniques this approach is computationally inexpensive, since its complexity is linear in the number of states and transitions in the LTS it is applied to. Thus, in cases where re-encoding does not reduce the OBDD representation of the LTS, it would not incur too much overhead. In the ARC tool, this technique is applied, when enabled by the user through a command-line option, to the sequential components of the process descriptions which generally do not have very complex behaviour—up to 100 states for most examples we have run.

It should be noted that a similar or potentially better result to that obtained by the re-encoding technique could be achieved if one compiles a process into an explicit LTS representation, and then performs OBDD encoding as a subsequent step. That way, the more complex and computationally intensive state compression algorithms can be applied before the OBDD encoding of the minimised LTS.

Two of the CSP examples presented in Appendix A benefit most from the re-encoding of their sequential components and in both cases it is the key to their successful verification. The first example is Monkey Puzzle for which re-encoding lowers the number of OBDD variables required for state encoding from 108 to 66, and the size of the OBDD representation of the final LTS from 35147 to

21449 nodes. For the second example—Solitaire Puzzle—the improvement is even greater. The number of OBDD variables is reduced from 133 to 34 and the size of the final OBDD from 18057 to 4619 nodes.

3.3 Refinement checking

3.3.1 Computing refusals and divergences

In this section, we introduce methods for computing the refusals and divergences of a process in OBDD form which serve as a prerequisite for the presentation of the refinement checking algorithm. The problem can be formulated as follows: given an LTS $\mathcal{L}$ and a process state Γ, how can one:

- Compute the refusals of Γ;
- Check whether any state in Γ is divergent?

In ARC the derivation of refusals during the refinement check is facilitated by a pre-computed relation $ref(\mathcal{L}) : \mathcal{S} \times \mathbb{P}(\Sigma) \rightarrow \mathcal{B}$, which encodes the refusal sets for every state of $\mathcal{L}$. The OBDD representation of *ref* uses the state vector $V_{\mathcal{S}}$ for encoding states and $\mid \Sigma \mid$ OBDD variables from $V_{\mathcal{U}}$ for encoding the refusals. Each event $a \in \Sigma$ has an assigned boolean variable $u_a \in V_{\mathcal{U}}$. Given $ref(\mathcal{L})$, the refusals of any process state $\Gamma \in \mathcal{S}$ can be derived using a single OBDD operation—relational product:

$$\nu_{\mathcal{U}}(refusals_L(\Gamma, \mathcal{L})) = \exists_{V_{\mathcal{S}}}(\nu_{\mathcal{S}}(\Gamma) \wedge ref(\mathcal{L})) \tag{3.17}$$

Obtaining $ref(\mathcal{L})$ involves computing the set of states $\Lambda_a(\mathcal{L}) \subseteq \mathcal{S}$ which accept the event a, for each $a \in \Sigma$ using:

$$\begin{aligned} \Lambda_a(\mathcal{L}) \;=\; & \{\gamma \mid \exists \theta \in \mathcal{S} : \gamma \xrightarrow{a} \theta \in \mathcal{R}\} \cup \\ & \{\gamma \mid \exists \theta \in \mathcal{S} : \gamma \xrightarrow{\tau^+ a} \theta \in \mathcal{R}^+\} \end{aligned} \tag{3.18}$$

Then, the relation $ref(\mathcal{L})$ is computed as:

$$ref(\mathcal{L}) = \bigwedge_{a \in \Sigma} (\nu_{\mathcal{S}}(\Lambda_a(\mathcal{L})) \wedge u_a \vee \neg\, \nu_{\mathcal{S}}(\Lambda_a(\mathcal{L}))) \tag{3.19}$$

where $\neg\, \nu_{\mathcal{S}}(\Lambda_a(\mathcal{L}))$ is used as a shortcut notation for $\nu_{\mathcal{S}}(\mathcal{S} - \Lambda_a(\mathcal{L}))$.

Equations (3.18) and (3.19) can be easily converted into an algorithm for the derivation of $ref(\mathcal{L})$ (Algorithm 5). Statements 3 and 4 are responsible for the computation of the first and second line of Equation (3.18) respectively, whereas the **for** loop and statement 5 iteratively derive $ref(\mathcal{L})$.

Algorithm 5 Computation of the refusal relation (initial version)

Require: An LTS $\mathcal{L}$ and its OBDD representation $\phi(\mathcal{L})$
Ensure: *Ref* contains the refusal relation $ref(\mathcal{L})$
1: $Ref \leftarrow 1$
2: **for all** $a \in \Sigma$ **do**
3: $\quad Lambda \leftarrow \exists_{V_\mathcal{E}, V'_\mathcal{S}}(\nu_\mathcal{R}(\mathcal{R}) \wedge \nu_\mathcal{E}(a))$
4: $\quad Lambda \leftarrow BackTauExpand_{OBDD}(Lambda, \phi(\mathcal{L}))$
5: $\quad Ref = Ref \wedge (Lambda \wedge \nu_\mathcal{U}(u_a) \vee \neg\, Lambda)$
6: **end for**

The implementation of Algorithm 5 has uncovered two inefficiencies. The first one is the presence of the LTS function *BackTauExpand* in statement 5, which can be an expensive operation for LTSs with a large proportion of τ transitions. The second problem is more subtle—experiments have shown that the OBDD representation of $ref(\mathcal{L})$ may require a very large number of OBDD nodes. In fact, it has been found to grow exponentially with the size of Σ and the complexity of the LTS, and changing the ordering of the variables in the vector $V_\mathcal{U}$ has little or no effect.

The first deficiency of Algorithm 5 is rather easy to overcome. Recall that Equation (3.4) has demonstrated that the refusals of a process state Γ can be derived by examining the stable states (those that do not have outgoing τ transitions) in Γ only. As the complete labelled transition relation is available in OBDD form, it is possible to compute the set of all stable states first and then combine statements 3 and 4 into one that does not contain *BackTauExpand*. This change alone has been found to improve the performance of Algorithm 5 by up to an order of magnitude for certain examples (e.g. Milner's Schedulers).

The solution to the second problem lies in the experimental observation that, although the OBDD representation of $ref(\mathcal{L})$ may be quite large, the OBDD representation of each sub-term of Equation (3.17) is usually very compact. Therefore, the number of OBDD nodes required can be drastically reduced if $ref(\mathcal{L})$ is stored as an implicit conjunction of relations:

$$ref_a(\mathcal{L}) = \nu_\mathcal{S}(\Lambda_a(\mathcal{L})) \wedge u_a \vee \neg\, \nu_\mathcal{S}(\Lambda_a(\mathcal{L}))$$

Equation (3.17) can then be rewritten as:

$$\nu_\mathcal{U}(refusals_L(\Gamma, \mathcal{L})) = \exists_{V_\mathcal{S}}(\nu_\mathcal{S}(\Gamma) \wedge \bigwedge_{a \in \Sigma} ref_a(\mathcal{L}))$$

Storing a complex boolean function as an implicit conjunction is not a new idea for reducing the size of an OBDD. This approach has been applied in a

different context, e.g. for reducing the size of an OBDD representing a large digital circuit [BCL91b, BCL+94]. In ARC this method has resulted in considerable space savings while incurring a negligible additional computational overhead.

Algorithm 6 Computation of the refusal relation (final version)

Require: An LTS $\mathcal{L}$ and its OBDD representation $\phi(\mathcal{L})$
Ensure: $Ref(a)$ contains the partial refusal relation $ref_a(\mathcal{L})$
1: $StableStates \leftarrow \neg\ \exists_{V_\mathcal{E}, V'_\mathcal{S}}(\nu_\mathcal{R}(\mathcal{R}) \wedge (\nu_\mathcal{E}(\tau) \vee \nu_\mathcal{E}(\checkmark)))$
2: $StableTR \leftarrow \nu_\mathcal{R}(\mathcal{R}) \wedge StableStates$
3: **for all** $a \in \Sigma$ **do**
4: $\quad Lambda \leftarrow \exists_{V_\mathcal{E}, V'_\mathcal{S}}(StableTR \wedge \nu_\mathcal{E}(a))$
5: $\quad Ref(a) \leftarrow Lambda \wedge \nu_\mathcal{U}(u_a) \vee \neg\ Lambda$
6: **end for**

Algorithm 6 incorporates the two modifications to Algorithm 5 discussed above. Statement 1 computes the stable states in the LTS from which *StableTR*—the subset of $\mathcal{R}$ which contains only transitions from stable states—is derived. Inside the **for** loop *Lambda* is computed as the set of stable states which have an outgoing transition labelled with the event a.

Algorithm 7 Computation of the divergence relation

Require: An LTS $\mathcal{L}$ and its OBDD representation $\phi(\mathcal{L})$
Ensure: *Div* contains the divergence relation $div(\mathcal{L})$
1: $TauTR \leftarrow \nu_\mathcal{R}(\mathcal{R}) \wedge \nu_\mathcal{E}(\tau)$
2: $StateSet1 \leftarrow \exists_{V_\mathcal{S}, V_\mathcal{E}}\ TauTR\ \big|_{[V_\mathcal{S} \leftarrow V'_\mathcal{S}]}$
3: $StateSet2 \leftarrow 0$
4: **while** $StateSet1 \neq StateSet2$ **do**
5: $\quad StateSet2 \leftarrow StateSet1$
6: $\quad StateSet1 \leftarrow \exists_{V_\mathcal{S}, V_\mathcal{E}}(StateSet1 \wedge TauTR)\ \big|_{[V_\mathcal{S} \leftarrow V'_\mathcal{S}]}$
7: **end while**
8: $Div \leftarrow BackTauExpand_{OBDD}(StateSet1, \phi(\mathcal{L}))$

Equation (3.6) is the key to constructing Algorithm 7, which pre-computes a relation $div : \mathcal{S} \rightarrow \mathcal{B}$ and is based on an approach discussed by Vaccari [Vac95]. Firstly, *TauTR* is computed as the OBDD representation of the set of invisible transitions in $\mathcal{R}$. The set of states *StateSet1* initially contains the set of all states in $\mathcal{S}$ with incoming invisible transitions, whereas *StateSet2* is set to empty in statement 3. The purpose of the **while** loop is to compute the set of states that form a τ-loop by iteratively excluding states whose predecessors in *TauTR* are not

members of *StateSet*1. Finally, *Div* is derived by applying the *BackTauExpand* function on the fixed point from the **while** loop.

Given $div(\mathcal{L})$, Equation (3.7) can be easily converted into an OBDD form:

$$divergent(\Gamma, \mathcal{L}) \Leftrightarrow \nu_{\mathcal{S}}(\Gamma) \wedge div(\mathcal{L}) \neq 0$$

3.3.2 Pair-by-pair refinement checking algorithm

Algorithm 8 represents the pair-by-pair version of the ARC procedure checking refinement and equivalence, which resembles the one used in MRC [BMTV94]. For brevity and conciseness, Algorithm 8 is presented in terms of LTS operations and functions whereas the actual implementation of the algorithm uses the OBDD-based counterparts discussed throughout this chapter. The **head**, **tail** and **cons** operations are the standard list functions.

The core functionality of the verification algorithm is contained within the **while** loop in statements 5–38. The purpose of the loop is to go through all pairs of process states $(\Gamma_P, \Gamma_Q) \subseteq \mathcal{S}_P \times \mathcal{S}_Q$ reachable from the initial pair $(TauExpand(\mathcal{I}_P, \mathcal{L}_P), TauExpand(\mathcal{I}_Q, \mathcal{L}_Q))$. To ensure that each pair of states is visited exactly once in the exploration, two lists of triples $(\Gamma_P, \Gamma_Q, Trace)$ are maintained—*Checked* for the checked pairs, and *Pending* for the pairs that have been reached but are yet to be checked. The last item in the triples (*Trace*) contains the trace of events which leads to the particular pair of states and is used for debugging purposes when a violation of the relation being checked is found.

Each iteration of the body of the **while** loop performs one step in the verification on a pair of process states. The conditions checked at each step of the loop in statements 10–27 depend on the model (*Model*) and type of relation (*Rel*) chosen for verification. These checks follow closely the definition of refinement and equivalence in the corresponding CSP models as presented in Section 2.1.4. The boolean flag *divFlag* is used to ensure that stable refusals are not computed and used when either the specification or the implementation process are found to be divergent at the current trace and the check is performed in the failures-divergences model.

If the check for a particular pair of states is successful, all pairs of next states reachable through a single visible transition from that pair of states and that have not been encountered yet are added to *Pending* unless *divFlag* computes to true (statements 29–37). If the relation being checked does not hold, the user is provided with a list of traces of increasing length after which the two processes behave differently. In other words, the verification analysis goes as far as possible, unless the user explicitly disables this feature through a command-line option (see Chapter 7). We believe that this may be more useful than reporting

Algorithm 8 The refinement/equivalence checking algorithm

Require: Process P and its LTS $\mathcal{L}_P$
Require: Process Q and its LTS $\mathcal{L}_Q$
Require: *Rel* is the relation to be checked
Require: *Model* is a CSP model for the check—one of $\{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$
Ensure: All violations of the relation being checked are reported

1: {Start from the pair of initial states}
2: $Pending \leftarrow \{(TauExpand(\mathcal{I}_P, \mathcal{L}_P), TauExpand(\mathcal{I}_Q, \mathcal{L}_Q), \langle\rangle)\}$
3: {Initially, we have checked nothing}
4: $Checked \leftarrow \emptyset$
5: **while** $Pending \neq \emptyset$ **do**
6: {Get the first pending pair of states}
7: $(\Gamma_P, \Gamma_Q, Trace) \leftarrow$ **head**$(Pending)$
8: $Pending \leftarrow$ **tail**$(Pending)$
9: $divFlag \leftarrow Model = \mathcal{N} \wedge (divergent(\Gamma_Q, \mathcal{L}_Q) \vee divergent(\Gamma_P, \mathcal{L}_P))$
10: **if** *Rel* is refinement **then**
11: **if** $NextEvents(\Gamma_Q, \mathcal{L}_Q) \not\subseteq NextEvents(\Gamma_P, \mathcal{L}_P)$ **then**
12: {Traces error after *Trace*}
13: **else if** $Model = \mathcal{N} \wedge divergent(\Gamma_Q, \mathcal{L}_Q) \wedge \neg\, divergent(\Gamma_P, \mathcal{L}_P)$ **then**
14: {Divergence error after *Trace*}
15: **else if** $\neg\, divFlag \wedge Model \in \{\mathcal{F}, \mathcal{N}\} \wedge refusals(\Gamma_Q, \mathcal{L}_Q) \not\subseteq refusals(\Gamma_P, \mathcal{L}_P)$ **then**
16: {Failures error after *Trace*}
17: **end if**
18: **else**
19: {*Rel* is equivalence}
20: **if** $NextEvents(\Gamma_Q, \mathcal{L}_Q) \neq NextEvents(\Gamma_P, \mathcal{L}_P)$ **then**
21: {Traces error after *Trace*}
22: **else if** $Model = \mathcal{N} \wedge divergent(\Gamma_Q, \mathcal{L}_Q) \neq divergent(\Gamma_P, \mathcal{L}_P)$ **then**
23: {Divergence error after *Trace*}
24: **else if** $\neg\, divFlag \wedge Model \in \{\mathcal{F}, \mathcal{N}\} \wedge refusals(\Gamma_Q, \mathcal{L}_Q) \neq refusals(\Gamma_P, \mathcal{L}_P)$ **then**
25: {Failures error after *Trace*}
26: **end if**
27: **end if**
28: $Checked \leftarrow Checked \cup (\Gamma_P, \Gamma_Q, Trace)$
29: **if** $\neg\, divFlag$ **then**
30: **for all** $a \in NextEvents(\Gamma_Q, \mathcal{L}_Q)$ **do**
31: $\Gamma'_P \leftarrow NextStates(\Gamma_P, \{a\}, \mathcal{L}_P)$
32: $\Gamma'_Q \leftarrow NextStates(\Gamma_Q, \{a\}, \mathcal{L}_Q)$
33: **if** $\nexists X : (\Gamma'_P, \Gamma'_Q, X) \in Checked \cup Pending$ **then**
34: $Pending \leftarrow$ **cons**$(Pending, (\Gamma'_P, \Gamma'_Q, Trace^\frown\langle a\rangle))$
35: **end if**
36: **end for**
37: **end if**
38: **end while**

only the first encountered error (as in FDR1). For example, if a refinement check reveals failures (liveness) errors but no traces (safety) ones (like in the dining philosophers example), it provides some insight into the nature of the problem and may hint as to what the possible solutions are. If the user is given only one error trace, she cannot be sure whether there are other such traces or not.

It is interesting to compare Algorithm 8 with the corresponding algorithm in the FDR tool. FDR employs an intermediate step between process compilation and refinement checking—normalisation of the LTS of the specification process [Ros94]. The normal form LTS is derived from the nondeterministic LTS generated by the CSP compiler by constructing the corresponding deterministic LTS with no hidden transitions where each state is labelled with its maximal refusals and divergence status. Then, states belonging to the same equivalence classes are joined into one. Each state of the CSP process corresponds to exactly one state of the normal form LTS, which considerably facilitates the refinement checking algorithm. On the other hand, the normalisation process can be expensive for larger processes.

The advantage of normalisation is that it prevents working with sets of states at the specification process end, as the latter can cause serious performance bottleneck when states are stored explicitly as they are in FDR. On the other hand, OBDDs do provide a compact representation for sets of states as well as efficient algorithms for their manipulation. This is why Algorithm 8 works directly on the nondeterministic LTS derived from CSP process descriptions and thus requires no intermediate normalisation step.

Another distinctive feature of the algorithm presented here is that it may check equivalence with virtually no additional overhead as compared to refinement. A naive equivalence check in FDR would involve checking refinement both ways, which would require the normalisation of the LTS of the potentially large implementation process. It is conceivable, of course, that FDR's refinement checking algorithm [Ros94] can be modified to allow equivalence checking as well.

3.3.3 Experimental results

The techniques and algorithms presented in this chapter have been implemented in the ARC tool[9]. This has enabled us to obtain experimental results regarding the performance of our algorithms in practice and also compare that to other tools. We refer to our implementation as ARC/PP to emphasize the fact that the results reported in this section are obtained with the pair-by-pair refinement checking algorithm described in Section 3.3.2 without any of the enhancements

[9]See Chapter 7 for more information on the tool itself.

presented later in this thesis. The benchmark examples used are described in detail in Appendix A.

Unless otherwise stated, all experimental results are obtained by executing the corresponding tool on a PC with a Cyrix 5x86/120 CPU and 48MB RAM running Linux RedHat 4.2 operating system. The execution times are measured with the Unix `time` and `ps` commands, as some of the tools used do not report correct CPU usage. The reported times include parsing, compilation, and refinement checking. For FDR2 version 2.28 used in the tests, we have ignored the overhead of the initial start-up of its graphical user interface. All tools are run with their default options.

In most cases, we provide a comparison of the run-times of the tools used, but not a comparison of their memory consumption. In our experience, it is difficult, if not impossible, to provide an accurate measurement and comparison of space requirements. The main reason for this is that different tools may have different strategies for memory management. Various techniques for trading space for speed, such as data caching and delayed garbage collection, may significantly skew experimental data. A more realistic measure for memory consumption of a given tool is the size of the largest problem instance that can be handled without resorting to virtual memory. This metric is reported whenever an example does reach the memory limit of the workstation used for the experiment.

Another interesting metric is the size of the OBDD (in number of nodes) representing the labelled transition relation for a process. This reflects primarily the effectiveness of the proposed compilation techniques and their implementation. Smaller size of the relation generally means faster OBDD operations [Bry86] and therefore results in faster verification.

We start with the run-time graphs of ARC/PP, MRC and FDR2 for the Synthetic example presented in Figure 3.9. Synthetic is an example with a very large number of states but simple behaviour of the sequential components. Two measurements for FDR2 have been taken: one without the application of state compression algorithms and the other with compression. As expected, the run-time of FDR2 without state compression on this example grows at a fast exponential rate with n (and respectively, the number of processes) and the tool is unable to complete the refinement check for $n > 4$ because of insufficient memory. It is a surprise, however, that the MRC tool, which is using an OBDD-based refinement checking engine, also exhibits a very sharp rise in run-time with growing n, and fails to complete the check for $n > 4$.

On the other hand, the performance of ARC/PP and FDR2 with state compression is comparable. FDR2 exhibits a relatively constant compression time overhead of about two seconds for $n < 4$, and then its run-time starts to grow at a slower rate than ARC/PP's run-time. Thus, although ARC/PP shows better

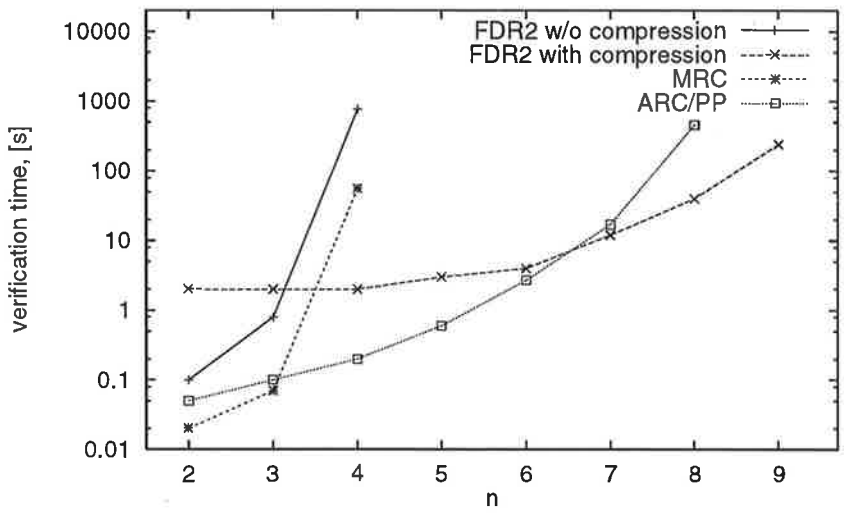


Figure 3.9: Synthetic example verification times

run-times for $n < 7$, it is not surprising that its advantage disappears for larger values of n, because ARC/PP actually requires 2^{n+1} OBDD variables for the encoding of the labelled transition relation of this example.

The run-time graphs obtained for variant 1 of Milner's Schedulers example are provided in Figure 3.10. This example is based on the well-known n-schedulers problem [Mil89] and contains a limited amount of hiding. Due to that, enabling state compression in FDR2 is of little benefit. Once again, the performance of the MRC tool is inferior compared with the other two tools and its run-time on this example grows at a faster rate. FDR2 demonstrates a slower rate of run-time increase than ARC/PP, which makes it the fastest tool[10] for $n > 7$. However, FDR2 does not complete the normalisation of the specification process for $n = 13$ in this example because it runs out of virtual memory. ARC/PP, on the other hand, comfortably finishes the refinement check for $n = 13$ with only 35MB memory required, albeit it takes over twenty minutes to complete.

The main difference between variant 1 and variant 2 of Milner's Schedulers example is that the latter checks a simpler property and thus more hiding is applied to the implementation process. Consequently, we examine tool performance on variant 2 of Milner's Schedulers for larger values of n (between 10 and 38) with the experimental results presented in Figure 3.11. The data for the CCS tool and Simple was provided by Dsouza and Bloom [DB95]. It should be noted that

[10]The enhanced verification techniques developed later in this thesis provide a considerable improvement for ARC/PP's run-times over the pair-by-pair approach used in this section.

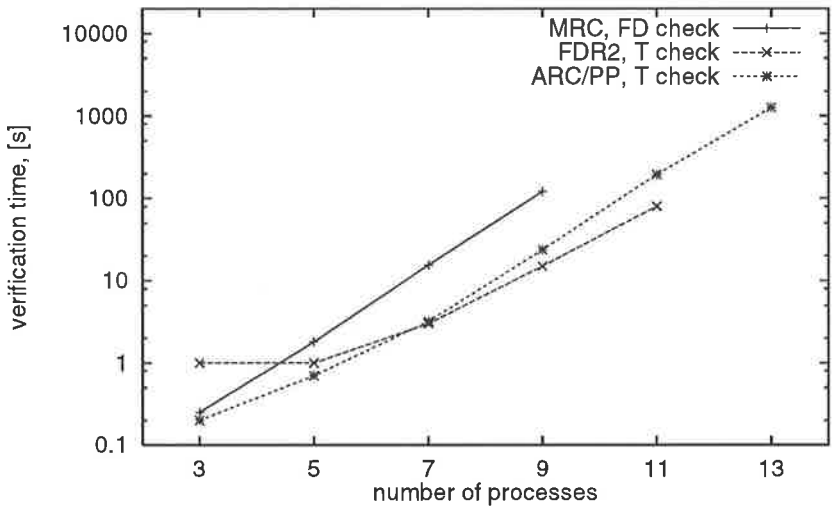


Figure 3.10: Milner's Schedulers (variant 1) example verification times

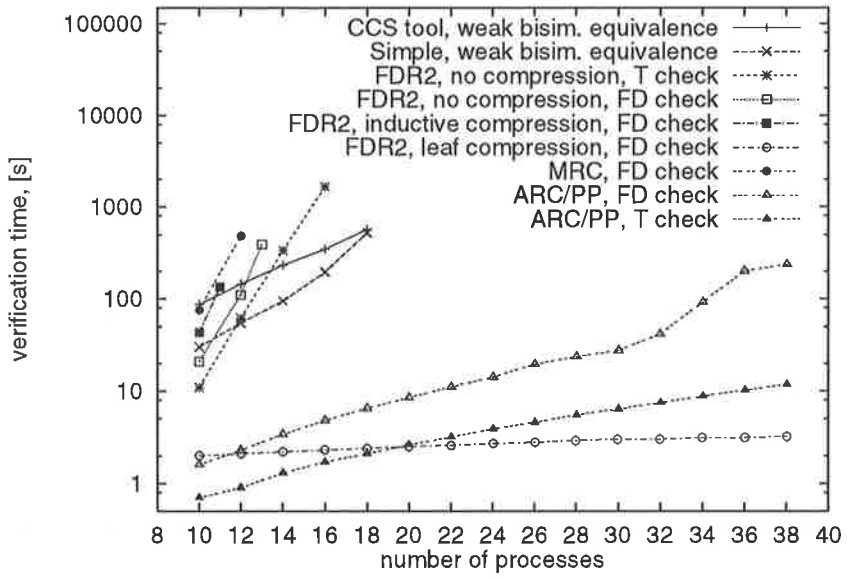


Figure 3.11: Milner's Schedulers (variant 2) example verification times

the results for these two tools are not directly comparable with the rest of the data for neither the property verified nor the experimental setup machines are the same. Nevertheless, it is interesting to observe that the rates of growth of the run-time of Simple and ARC/PP are comparable.

The rest of the tools used for the comparison in Figure 3.11 can be clearly divided into two groups with noticeable performance difference. FDR2 without compression, FDR2 with inductive compression[11] and MRC are one to two orders of magnitude slower than the second group, and quickly reach the memory limitations imposed by the workstation used for the experiments. We could not get a failure-divergence check for $n > 13$ nor a traces check with $n > 16$ to finish with FDR2 (no compression). FDR2 with inductive compression performed even worse, failing to complete a check for $n = 12$. MRC also exhibits exponential growth in memory requirements with this example and does not complete the check for $n = 14$.

As with the Synthetic example, the ARC/PP and FDR2 with leaf compression are noticeably faster than the first group of tools. Leaf compression brings remarkable results for this example, making the execution time virtually linear to the number of schedulers. This is due to the fact that leaf compression reduces the final LTS to just n states on which refinement checking is almost instantaneous. This is in contrast to ARC/PP which operates on the complete LTS of this example similarly to FDR2 without compression. Nevertheless, ARC/PP also demonstrates slow (but not linear) growth of tool run-time with n.

It is interesting to note that, for this example, the inductive form of state compression in FDR2 not only fails to bring any run-time improvement but actually reduces performance by a factor of more than two. Thus, we believe that the successful application of compression requires a certain amount of experience with FDR2 and an understanding of its underlying verification algorithms.

We compare the OBDD sizes for the derived LTS process representations for MRC, ARC/PP, the CCS tool and Simple in Table 3.2. The experimental data for the latter two tools is once again provided by Dsouza and Bloom [DB95]. ARC/PP produces an OBDD that is less than half the size of that of MRC and compares favourably with the other two tools (both using CCS as input language). Note that the CSP definition of this problem requires an extra process to insert a token into the scheduler ring that is not needed in the CCS description.

The experimental data for Dining Philosophers is presented in Figure 3.12. This example was defined by Hoare [Hoa85] and the system being verified contains a chain of sequential components modeling philosophers and forks. Again, the

[11] Roscoe presents a description of the difference between inductive and leaf compression and, in particular, the big difference observed for variant 2 of the Milner's Schedulers example [Ros97, Section C.2.2].

Number of processes	ARC/PP	Simple	CCS tool	MRC
12	1135	1167	1200	2488
14	1447	1488	1528	3242
16	1798	1861	1897	4092
18	2182	2254	2297	5038
20	2603	—	2810	6080
40	8793	—	—	crashed

Table 3.2: OBDD sizes for Milner's schedulers, variant 2

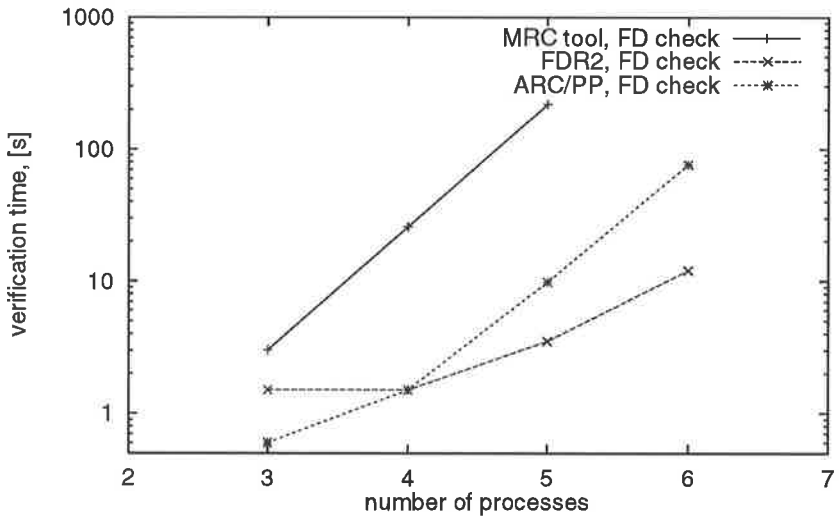


Figure 3.12: Dining Philosophers example verification times

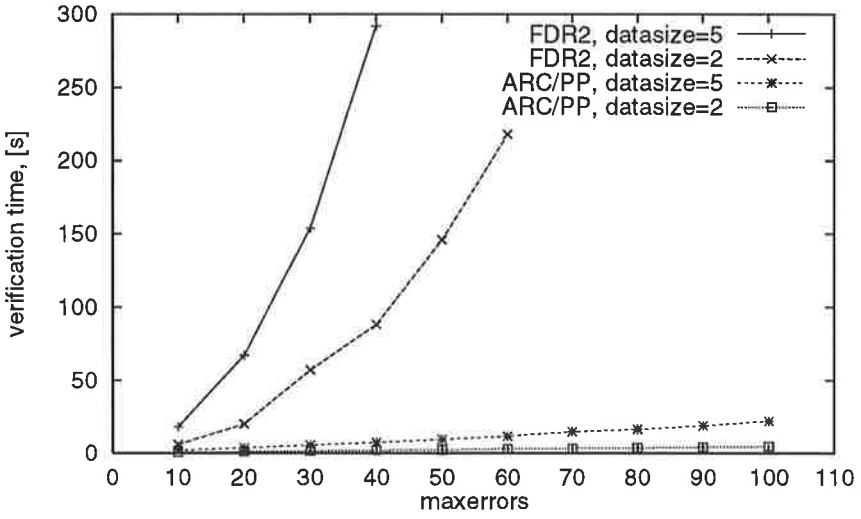


Figure 3.13: Alternating Bit Protocol example verification times

complexity of the example can be varied by specifying a different number of philosophers (n). For this example, FDR2 has a clear performance lead for $n > 4$ over both ARC/PP and MRC. Of the latter two tools, ARC/PP performs the verification faster, but its rate of run-time growth is not much better than that of MRC.

Next, we consider the Alternating Bit Protocol example modeling a very simple communication protocol over an unreliable media. Unlike the examples discussed so far, this one allows the complexity of its instances to be determined in two ways—by varying the size of the data type that is being sent and by varying the maximum number of times the medium can lose data. The size of the data type determines the amount of hiding (e.g. the number of events to hide) that is required, whereas data loss by the medium is modelled by the use of nondeterminism. Thus, one can vary the mixture of hiding and nondeterminism in the instances of this example by changing the values of the corresponding parameters (`datasize` and `maxerrors` in the CSP script from Figure A.5).

Interestingly, we have not been able to apply FDR2's state compression techniques successfully for this example. Neither inductive nor leaf compression work well as the explicated versions of the leaf processes contain a very large number of states. This confirms the earlier observation that the application of state compression in FDR2 requires substantial user experience.

Figure 3.13 summarises the experimental data collected for the Alternating Bit Protocol example. Two sets of runs have been performed: one with `datasize`=2

	ARC/PP	EUMC	FDR2
OBDD variables in LTS	66	80	—
OBDD nodes in LTS	21449	115903	—
Run-time, s	295	1607	36

Table 3.3: Experimental results for Monkey Puzzle

	ARC/PP	FDR 2.78
Peak memory usage, MB	130	5000 (approx.)
Run-time, s	24016	45000 (approx.)

Table 3.4: Experimental results for Solitaire

and one with `datasize`=5. In both cases, FDR2 exhibits exponential growth of run-time, whereas ARC/PP's run-time grows virtually linearly with respect to `maxerrors`.

The two remaining examples from Appendix A—Monkey Puzzle and Solitaire—contain no parameters and thus require only a single run of each tool. For Monkey Puzzle, we compare the performance of three tools on two models of the example. The first model is written by Geert Janssen in an IBM proprietary hardware description language and solved with the Eindhoven University OBDD-based model checker EUMC [Jan96]. The second model is written by Bill Roscoe for the FDR2 tool and adapted by the author for the ARC/PP tool.

Table 3.3 presents the data collected for Monkey Puzzle. The run-time of EUMC has been obtained on an HP 9000/755 machine with 256MB of RAM. The results clearly show FDR2 to be the fastest of the three tools, while ARC/PP performs better than EUMC both in terms of compactness of the labelled transition relation derived and verification speed. The small size of the transition relation can be largely attributed to the use of the re-encoding technique described in Section 3.2.6. Without its use, the size of the transition system derived by ARC/PP grows from 21449 to 35148 OBDD nodes using 108 instead of 66 OBDD variables for encoding the state space of this example.

The unconstrained Solitaire was, until very recently, impossible to solve with FDR2 [Ros97] due to its very large state space (2^{34} potential and 187,636,299 actual reachable states) and the inability to apply state compression techniques. To overcome this problem, we used a new version of the tool (2.78) which has an improved state storage mechanism that allows the utilisation of secondary storage (fixed disk file system) instead of main memory.

This challenging example also required us to run it on a Sun Ultra80 workstation with 1 GB RAM instead of the configuration used for the examples pre-

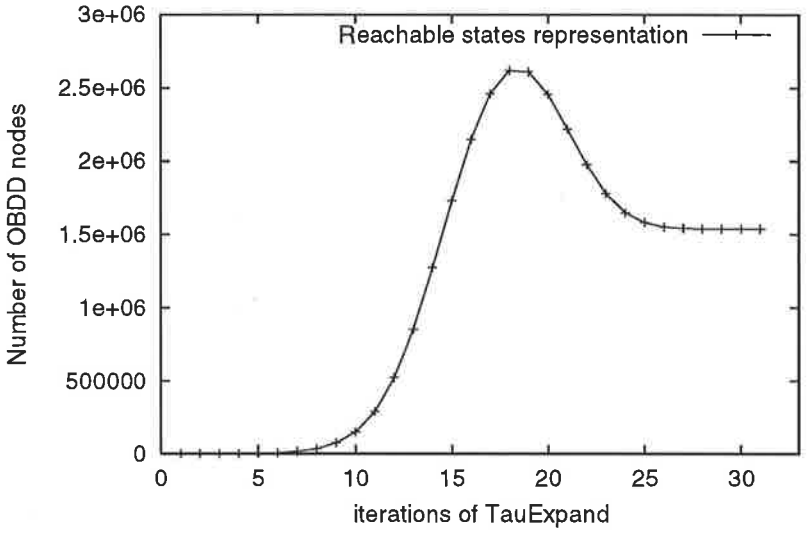


Figure 3.14: Change in OBDD size for the reachable states in Solitaire

viously discussed. The results are presented in Table 3.4. Unfortunately, we were unable to complete Solitaire with FDR2 on this machine due to the lack of sufficient amount of available free disk space, which was exhausted after exploring approximately 82,000,000 reachable states. The figures presented in the table are approximations of what the results for FDR2 should be assuming that the speed of state space exploration does not drop significantly later in the search, and that the disk space requirement grows linearly with the number of reached states.

Table 3.4 highlights a large difference in the space requirements of ARC and FDR2 for Solitaire. This can be attributed to a much more compact representation of the reachable state space of this example by OBDDs compared to an explicit state encoding. Due to the use of hiding in Solitaire the entire reachable space is computed in a single call to $TauExpand_{OBDD}$ that iterates through all possible moves in the puzzle. Figure 3.14 shows the change of the size of the OBDD encoding the reached state space for each iteration of $TauExpand_{OBDD}$. One can see the characteristic hump in the resulting curve exhibiting an exponential growth in the first fifteen iterations with a peak of more than 2,600,000 nodes required at iteration 18 of $TauExpand_{OBDD}$, after which the size of the OBDD gradually reduces to approximately 1,500,000 nodes. This means that the ratio of total reachable states to OBDD nodes in their encoding in ARC for the complete puzzle is more than 100:1, which easily explains the low space requirement for OBDD-based refinement checking of Solitaire.

Since Solitaire contains a lot of hiding, ARC requires almost half the time that

FDR2 would take to complete the example. In practice, the advantage of using OBDD-based refinement checking for this example is even greater, because of the following factors:

- The figure for the total time it would take FDR2 to go through the reachable state space is an under-approximation, because going through the second part of the reachable state space is likely to take more than the exploration of the first part, which was experimentally measured;
- The execution times reflected in Table 3.4 are based on pure CPU usage rather than wall (real) time as perceived by the user of the tool. While ARC is consistently utilising 95% or more of a CPU, FDR2's CPU utilisation has been measured to be about 50% on the average and sometimes drop to 30% due to continuous disk access. In practice, this doubles the perceived verification time by the tool's user;
- The time required by ARC can be further reduced by a factor of six when the computation of the $TauExpand_{OBDD}$ utilises the PRS-enhanced reachability analysis technique from Chapter 5 (see Section 5.4).

3.4 Discussion

Having presented our experimental results in the previous section, we consider two most interesting questions regarding the algorithms developed in this chapter:

1. What are the relative efficiency advantages and disadvantages of the two most popular model checking techniques—OBDD-based and explicit on-the-fly algorithms—with respect to the problem they are applied to?
2. What are the bottlenecks of the compilation and refinement checking techniques presented in this chapter, and can they be improved upon in any way?

This section contains an analysis of these topics.

3.4.1 OBDD-based versus explicit refinement checking

Of all the tools that were used in the experimental study, FDR2 is the most relevant and interesting one. Both FDR2 and ARC/PP accept a very similar syntax of the same language (CSP), but differ vastly in terms of algorithms and implementation details. This makes them ideal candidates for a comparison targeted at the first question posed above.

It is important to note that the comparison we are interested in is not one of picking the better of the two tools. ARC is merely an academic, experimental prototype tool that emerged as a result of this research work while FDR2 is a robust commercial tool with a powerful input language, proven capabilities and frequent updates; this obviously makes FDR2 the better *tool product*. Rather, we are interested in a performance comparison of their back-end verification engines—the OBDD-based one in ARC/PP and the on-the-fly explicit state enumeration engine found in FDR2.

The approach we take in the algorithm comparison is to study the features of the examples for which one of the tools demonstrates a clear performance advantage over the other. In doing this, we temporarily ignore the effect that state compression (which can be seen as a smart finite state model compilation method rather than a reachability analysis technique) has on the verification times. Then, we identify a set of common properties of these examples which, provided our knowledge of the internal workings of FDR2 and ARC/PP, are most likely contributors to the performance discrepancy being observed.

The examples for which ARC/PP's back-end shows better performance are Synthetic, Milner's Schedulers (variant 2), Alternating Bit Protocol, and Solitaire. The Synthetic example is one with a very high number of reachable states and is composed of many identical processes with no interaction between them. It contains a high degree of nondeterminism since each sequential process is prepared to engage in the event a at any time. This example also exhibits a large ratio of LTS states to CSP process states ($2^n/n$, where n is the number of processes). These three factors result in small OBDDs for the transition relation and reached state space, fast and efficient computation of relational product for LTS traversal, and a small number of iterations of the main verification loop in the pair-by-pair algorithm presented in Section 3.3.2.

A common feature of the second variant of Milner's Schedulers and Solitaire is the extensive use of the hiding operator. Hiding has a two-fold effect on the OBDD-based algorithms. Firstly, it reduces the "irregularities" in the transition relation by making more transitions have the same label (τ) which is reflected in the OBDD representation of the relation in "smoothing" and size reduction. This has a positive effect on the speed of state space traversal. Secondly, hiding results in an increase in the LTS to CSP process states ratio which in turn reduces the number of iterations of the refinement checking algorithm.

Although the sequential components in Alternating Bit Protocol have no particularly regular structure, the example contains a lot of nondeterminism. The experimental results show that the performance benefit of the OBDD-based refinement checking algorithm in ARC/PP increases with the amount of nondeterminism. Again, the explanation of this observation can be seen in the increased

LTS to CSP process states ratio observed with the examples discussed above.

FDR2's verification engine is superior for the examples Dining Philosophers, Milner's Schedulers (variant 1), and Monkey Puzzle. The first and third of these lack any of the features that made ARC/PP excel in the examples discussed above: they lack a regular structure, are completely deterministic, and contain no hiding at all. Monkey Puzzle is the worst of the three with respect to performance of the OBDD-based algorithms as it combines a relatively large number of processes with small LTSs but engaging in a complex communication pattern. All these factors combined result in the largest OBDD size for a transition relation (more than 23,000 nodes) for any of the examples we have tried while the LTS has less than 24,000 reachable states.

The first variant of Milner's Schedulers does contain hiding, but that barely has any effect on the ratio of LTS to CSP process states. The number of iterations required for the pair-by-pair refinement checking algorithm from Section 3.3.2 grows exponentially with the number of schedulers instead of linearly as is the case with the second version of this example.

In general, we have found that state space traversal with OBDDs is less efficient than the explicit on-the-fly algorithms present in FDR2 when performed on a single state at a time. This is to be expected as the underlying OBDD operation—relational product—has a complexity proportional to the product of the sizes of the OBDDs representing the starting state and the labelled transition relation. The computation of next states in an optimised explicit model checking algorithm can be expected to have a complexity roughly linear to the number of sequential components in the example.

The picture changes considerably when the nature of the compiled CSP model is such that state space exploration that is otherwise progressing one CSP process state per iteration of the pair-by-pair OBDD-based refinement checking algorithm is performed on multiple LTS states at a time. Not only an explicit verification algorithm has to traverse the state space one LTS state at a time, but it also has to store and maintain them separately until the end of the check.

Arguably, state compression as used in FDR2 can achieve efficiency improvements comparable to those offered by OBDD-based verification algorithms for a certain class of examples—those with a lot of hiding and/or nondeterminism combined with some regularity of process compositions. However, there are examples (Solitaire and Alternating Bit Protocol from our problem suite) for which compression does not bring about any benefit and may even make matters worse due to very complex intermediate LTSs. ARC/PP is able to handle both examples in reasonable time. The opposite observation is also true—the author has come across an example describing message routing in a mesh grid for which state compression works while the OBDD-based approach has problems. Thus, we see

these two approaches complementary to each other; it is very difficult to predict which one will be able to verify a particular example in a purely analytical way.

Our findings in relation to the first question posed at the beginning of this section can be summarised as follows:

- The OBDD-based compilation and refinement checking algorithms presented in this chapter have an advantage over the explicit verification algorithm found in FDR2 for examples with a high ratio of LTS to CSP process states;
- Hiding and nondeterminism are the two factors which affect the ratio between LTS and CSP process states in a model the most—the more they are used the higher the ratio;
- Regularity and locality of inter-process communications is beneficial to OBDD-based algorithms as it results in a generally smaller OBDD encoding of the labelled transition relation of a process;
- Examples exhibiting a large number of processes constrained by a complex interaction pattern are more suitable to refinement checking using explicit state space exploration;
- State compression and OBDD-based verification engines can be seen as complementary approaches addressing the state space explosion problem.

3.4.2 Bottleneck analysis

The system complexity that can be handled by a verification algorithm can be limited either by space or time requirements. Thus, to identify the potential performance bottlenecks in the CSP compilation and refinement checking techniques presented in this chapter we should consider what determines the space and time complexity of our algorithms.

The compilation of CSP scripts into a corresponding LTS representation in OBDD form is not very time consuming as it performs a number of operations roughly proportional to the size of the syntax tree. Therefore, assuming that the input process does not contain excessively deep or infinite recursion, time is not a limiting factor for the compilation step. On the other hand, space is of concern, as the size of the OBDD required to encode the LTS representation of a process can become prohibitively large (i.e. exceeding the amount of the available physical memory). It is well-known that the performance of OBDD-based algorithms drops by orders of magnitude when using virtual memory due to the highly irregular pattern of memory access of these algorithms.

The refinement checking algorithm requires space to store all reached CSP states. When that space outgrows the available physical memory verification may not be able to complete. On the other hand, the time necessary to complete the pair-by-pair algorithm from Section 3.3.2 is proportional to the number of CSP states it has to explore; the latter is then a limiting factor to the overall verification performance.

To summarise our findings in relation to the second question posed at the beginning of this section, there are several potential bottlenecks in the algorithms presented in this chapter:

- Compilation of very large process descriptions into OBDDs may lead to OBDD blow-up. In this case verification—complete or even partial—is impossible;
- The state space exploration algorithm requires storing the reached CSP process states which is another potential source of OBDD blow-up. Only partial verification is possible in this case;
- The pair-by-pair nature of our refinement checking algorithm limits the full utilisation of the OBDD capabilities of handling a very large set of states at once. This may potentially result in a long run-time for the algorithm.

In the rest of this thesis, we present several improvements to the algorithms developed in this chapter addressing each of the issues listed above. Each enhancement is then implemented in the ARC framework and further experiments are carried out to assess the benefits and trade-offs of the proposed techniques.

Chapter 4

Hierarchical Partitioned Transition Relations

Overview

In this chapter we develop a refinement checking technique that does not require a monolithic OBDD representation of the full LTS of a process to be available. Instead, the complete model is kept in a hierarchical partitioned form combining the behaviour of separate components of the concurrent system called "synchronisation hierarchy". The structure of the synchronisation hierarchy reflects the static communication structure of the concurrent system's components. This technique is developed to address the danger of OBDD blow-up during the compilation of CSP processes with many sequential components and a very large state space. Performance results of an implementation of the proposed method in the ARC tool are presented.

4.1 Motivation

An implicit requirement of the pair-by-pair refinement/equivalence checking algorithm as presented in Section 3.3.2 is the ability to build a four-tuple of boolean functions $\phi(\mathcal{L})$ for the LTS of both the specification and implementation processes. The OBDD $\nu_{\mathcal{R}}(\mathcal{R})$ encoding the transition relation of an LTS is usually the largest in size of the four OBDDs in $\phi(\mathcal{L})$ and typically requires many times more OBDD nodes than the other OBDDs in $\phi(R)$. We call $\nu_{\mathcal{R}}(\mathcal{R})$ a *monolithic* OBDD representation of the transition relation $\mathcal{R}$ because of the fact that the whole transition relation is encoded as a single boolean function.

Unfortunately, $\nu_{\mathcal{R}}(\mathcal{R})$ can become very large. If its space requirement out-

grows the available physical memory, the performance of OBDD operations and thus the speed of the actual refinement check sharply declines.

From our experience with process compilation in the ARC tool, the following factors have the most impact on the size of an OBDD encoding of the LTS representation of a process:

- The number of states in the sequential components of the concurrent system. This is mainly due to the fact that the larger the state space of the LTS of a process, the more OBDD variables are required to encode that LTS;

- The number of interacting processes and the complexity of their interaction (synchronisation) pattern. Two processes executing asynchronously (e.g. combined using the interleave operator) generally have a smaller OBDD representation than those same two processes synchronizing over a set of events. A good illustration of this phenomenon is the Synthetic example, which has an extremely large number of states and a very compact OBDD representation;

- The distance between the OBDD variables V_{S_P} and V_{S_Q} used to encode the LTS representation of two interacting processes P and Q. Recall that ARC assigns OBDD variables in the order of traversal of the syntax tree. Therefore, one should keep tightly coupled processes close in the CSP process terms. A similar rule has been observed in another context by Hu [Hu95];

- The amount of hiding used. Although hiding does not affect the number of states in the LTS representation of process semantics, it effectively "smooths" the OBDD representation and reduces the number of nodes required by renaming events encoded through the variables from $V_{\mathcal{E}}$.

The above observations provide useful hints to the end-user of the ARC tool for constructing processes in a way which minimises the OBDD representation of their LTSs. This could speed up the refinement checking algorithm or even limit the OBDD size blow-up for the LTS representation in certain cases. Nevertheless, the general problem of OBDD blow-up during compilation still remains. An automated technique that does not rely on the experience of the user to write CSP scripts that have an efficient internal representation is much more desirable. We explore some existing techniques addressing this issue in the next section.

4.2 Partitioned transition relations

The problem of OBDD representations of transition relations growing too large is well-known in the hardware verification domain. One of the proposed solutions is to keep the entire transition relation R in a partitioned form[1] that reflects the structure of the digital circuit being represented [BCL91a, BCL91b]. Each partition of the transition relation R_i represents the behaviour of a small part (e.g., a bit of the state vector) of the whole circuit. There are two types of partitioning—*conjunctive* and *disjunctive*—that have been proposed in the literature [BCL91a, BCL91b].

Conjunctive partitioning is applicable to the OBDD representation of transition relations of systems with synchronous behaviour, e.g. synchronous digital circuits. The conjunction operator is used for composing the transition relation together because of the synchronous behaviour being represented—for a transition to occur, every component in the system has to agree to it. In a conjunctive partitioned transition relation, the transition relation partitions R_i are subsequently treated as implicitly conjoined ($R = \bigwedge_i R_i$). Thus, instead of storing one potentially very large OBDD for the transition relation, one stores and manipulates a number of smaller OBDDs that encode the same transition relation.

Disjunctive partitioning, on the other hand, is applicable to the OBDD representation of transition relations of asynchronous digital circuits. The asynchronous behaviour means that when two or more transitions within separate partitions are possible, they may occur in any sequence or even simultaneously. Therefore, the partitions R_i are stored as separate boolean functions and treated as implicitly disjunctive ($R = \bigvee_i R_i$) in the operations involving the OBDD encoding of the corresponding transition relation.

The advantage of partitioning compared to a monolithic OBDD for the transition relation is that the sum of the sizes of R_i is typically much smaller than the size of R. The latter OBDD may grow, in the worst case, as the product of the sizes of the OBDDs for the partitions. In theory, R may also turn out to be smaller in size than the sum of sizes of R_i, however, the general consensus in the literature is that this is very rare in practice. Furthermore, even if R is quite small, there is no guarantee that the intermediate OBDDs required for its computation will not exceed the memory limitations. The use of partitioned transition relations provides at most a linear increase in the memory required (which is very unlikely) while avoiding a monolithic OBDD representation of the transition relation with a size exponential to the size of the partitions. In prac-

[1] In this section only, we adopt the shorter notation used by Burch *et al.* [BCL91a, BCL91b], whereby R (R_i) denotes the OBDD encoding of a transition relation rather than the relation itself.

tice, a partitioned transition relation can often be successfully computed when the OBDD for the full transition relation grows too large.

To make partitioned transition relations useful for reachability analysis, relational product has to be computed on OBDDs without building the entire transition relation. For a disjunctive partitioned transition relation, we have:

$$Image(S, R) = (\exists_{V_S}(S \wedge \bigvee_i R_i)) \Big|_{[V_S \leftarrow V'_S]} \tag{4.1}$$

where R_i are the OBDD encodings of the transition relation partitions using the variable vectors V_S and V'_S for encoding the current and next states, respectively, and S is the OBDD encoding of a set of states in the full transition relation.

It is known that disjunction distributes over existential quantification [Bry92]. In other words, if f and g are boolean functions and V is a vector of boolean variables, we have:

$$\exists_V(f \vee g) = \exists_V f \vee \exists_V g \tag{4.2}$$

By applying the above, Equation (4.1) can be rewritten as:

$$Image(S, R) = (\exists_{V_S}(\bigvee_i S \wedge R_i)) \Big|_{[V_S \leftarrow V'_S]} = (\bigvee_i \exists_{V_S}(S \wedge R_i)) \Big|_{[V_S \leftarrow V'_S]}$$

Thus, the relational product over the entire transition relation can be conveniently obtained by computing relational products over each of the individual implicitly disjunctive partitions.

The relational product for conjunctive partitioned transition relation is:

$$Image(S, R) = (\exists_{V_S}(S \wedge \bigwedge_i R_i)) \Big|_{[V_S \leftarrow V'_S]} \tag{4.3}$$

Since conjunction does not distribute over existential quantification, the rewriting technique used for disjunctive partitioning is not applicable. However, under certain conditions it is possible to move a conjunct outside of the existential quantification operator. If f and g are boolean functions and f does not depend on the variables in the vector V, then:

$$\exists_V(f \wedge g) = f \wedge \exists_V g \tag{4.4}$$

Assuming that each R_i in Equation (4.3) depends on a subset of variables from V_S, it is possible to conjoin each R_i with S iteratively while existentially quantifying out any variables on which the remaining partitions do not depend. The order in which the partitions are chosen can significantly impact the efficiency of this method—it can be expensive and in the worst case there could be no benefit

from the partitioning. In practice, finding a good order does not appear to be difficult [BCL91b].

The application of the above partitioning techniques to the domain of concurrent systems appears simple at first. Indeed, the concurrently composed components in a concurrent system are natural candidates for partitioning the monolithic OBDD for the LTS representation of such a system. However, the concurrent components may exhibit a combination of synchronous and asynchronous behaviour—processes execute the events they synchronise on in lock-step while the execution of the rest is independent. As a result, the semantics of the CSP parallel composition operator presented in Section 3.2.5 is much more complicated than the semantics of composition of digital hardware blocks. Therefore, it is not possible to apply just disjunctive or conjunctive partitioning in the form presented in this section to the labelled transition relation of a CSP process.

There is an additional difference in how partitioning is applied to hardware and how it should be applied to a concurrent system. In digital circuits, there is a single level of concurrency and that is naturally reflected in the conjunctive and disjunctive partitioning. The structure of a concurrent system is more complex in that it may form a hierarchy of connections between the components. Thus, a representation of the partitioning hierarchy is required for concurrent systems.

In the rest of this chapter, we present a form of hierarchical partitioning of the labelled transition relation of a CSP process called *hierarchical partitioned transition relation* (HPTR). We develop a technique for computing OBDD relational products using a combination of the rules of disjunctive and conjunctive partitioning and recursive relational product computation on the HPTR.

4.3 Hierarchical partitioned transition relations (HPTR)

This section introduces a hierarchical form of a partitioned transition relation of a concurrent system that is suitable for subsequent state space exploration and refinement checking.

4.3.1 Definition

As a first step, we define the subset of CSP terms on which our technique operates. We consider the following syntactic structures:

$$P = P \underset{A}{\parallel} Q \;\Big|\; P \;|||\; Q \;\Big|\; P \setminus A \;\Big|\; f[P] \;\Big|\; LeafProcess \tag{4.5}$$

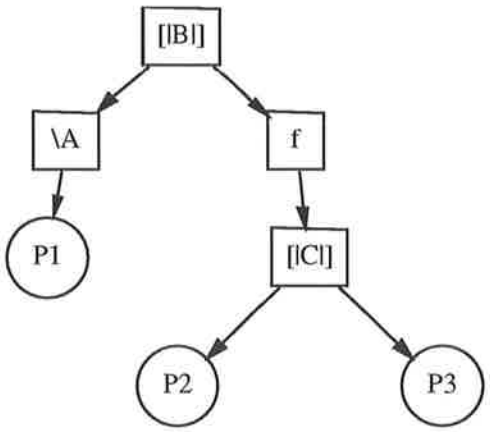


Figure 4.1: An example synchronisation hierarchy

where *LeafProcess* is a process with an OBDD representation of its LTS. The CSP operators from the above equation are termed *high-level* within the context of FDR2 [For97], while the remaining operators are referred to as *low-level*. This terminology is adopted here as well.

As is the case with partitioning for digital circuits, we are interested in partitioning the concurrently executing components, thus the set of operators in the Equation 4.5 includes parallel composition and interleaving. The other two operators included in that equation—hiding and renaming—have the property of not affecting the concurrent decomposition of the system.

Leaf processes can be seen as partitions in a concurrent system and are combined using the parallel, hiding and renaming operators to construct the behaviour of the system as a whole. The level of granularity of these processes depends on the use of low-level operators, because any process term with a top-level operator which is not high-level has to be, by definition, (a part of) a leaf process. The leaf processes are similar to the low-level components in the FDR tool [For93, For97] in the way they are used to construct the behaviour of the complete system using high-level operators, but differ from low-level processes in that they may contain high-level operators as well. For most practical purposes, however, high-level operators in leaf processes are unnecessary, and the leaf processes can be as fine-grained as the sequential components of the concurrent system [Ros97].

The syntactic structures defined by Equation (4.5) can be easily presented in a rooted tree form referred to as hierarchical partitioned transition relation (HPTR) which closely resembles the synchronisation hierarchy of a compound process. The leaves of the tree are the leaf processes, and the non-leaf nodes contain the

operators and related information—a synchronisation event set for the parallel operator, a hiding event set for the hiding operator, or an event renaming function f for the renaming operator. Figure 4.1 contains the synchronisation hierarchy for the process P defined as follows:

$$P \triangleq (P_1 \setminus A) \underset{B}{\parallel} f[P_2 \underset{C}{\parallel} P_3]$$

Synchronisation hierarchies are a natural way of representing a structural decomposition of a concurrent system. The full LTS model of this system is implicitly represented by the concrete OBDD encodings of the LTS representations of the leaf processes, the structure of the tree, and the operators applied within non-leaf nodes.

4.3.2 Structural optimisation

Synchronisation hierarchies closely resemble the syntax tree of a process term and can be derived from the latter relatively easily[2]. Leaf processes in the tree of a synchronisation hierarchy are identified by the first occurrence of a low-level operator. The LTS of a leaf process is derived in the usual monolithic OBDD form as described in Section 3.2.5. The initial HPTR obtained in this way can be subjected to certain structural optimisations presented in this section. Here by structural optimisation we mean a transformation of the tree of the synchronisation hierarchy aimed at reducing the size of the OBDDs for the labelled transition relations of leaf processes and minimising the number of non-leaf nodes in the HPTR.

Note that the example from Figure 4.1 contains a hiding node which is applied to the leaf process P_1. Since hiding reduces the size of an OBDD encoding of the LTS of a process (see Section 4.1), one can expect that moving the hiding node (i.e. "pushing" it down) as part of the leaf process $P1$ would provide a more space-efficient representation of the concurrent system and, as discussed in Section 3.4, likely result in a more time-efficient reachability analysis. This example can be generalised as it is always beneficial to "push" hiding down the synchronisation hierarchy towards and into the leaf nodes, subject to the CSP algebraic laws [Ros97]:

$$\begin{aligned}
(P \underset{A}{\parallel} Q) \setminus C &= ((P \setminus (C - A)) \underset{A}{\parallel} (Q \setminus (C - A))) \setminus (C \cap A) \\
f[P] \setminus A &= f[P \setminus f^{-1}(A)] \\
(P \setminus A) \setminus B &= P \setminus (A \cup B)
\end{aligned}$$

[2]A slight complication is the possibility that the syntax tree might span several process definitions. However, this can be easily overcome and is beyond the scope of this chapter.

where f^{-1} is the inverse of the injective event renaming function f. All of these algebraic laws are aimed at reducing the size of the monolithic OBDD representation of the transition relation of a leaf process. The number of non-leaf nodes may increase or decrease depending on which of the rules is used most—the first rule above increases that number by two, the second rule does not change it, while the last rule decreases the non-leaf nodes by one.

A similar transformation can be applied to the renaming nodes in the HPTR as well. In fact, assuming that the renaming function is injective, this operator distributes over all other CSP operators [Ros97]. Thus, it is possible to improve the compactness and efficiency of the HPTR representation by "pushing down" the renaming nodes all the way into the leaf nodes by using the following CSP laws for injective renaming functions f [Ros97]:

$$\begin{aligned} f[P \underset{A}{\|} Q] &= f[P] \underset{f(A)}{\|} f[Q] \\ f[P \setminus A] &= f[P] \setminus f(A) \end{aligned}$$

The successive application of these laws can effectively remove all renaming nodes from a HPTR by making them a part of the leaf processes. However, the size of the monolithic OBDD encoding of the labelled transition relations of leaf processes is not impacted significantly.

4.4 Recursive image computation on HPTRs

The computation of the refusal and divergence relations as well as the pair-by-pair refinement checking algorithm presented in Chapter 3 require the ability to compute both forward and backward images from a given set of LTS states and/or events. Therefore, to enable the operation of these algorithms on an HPTR instead of a monolithic OBDD, a method is required to compute relational product on an HPTR representing the LTS and an OBDD representing a set of LTS states.

Let the HPTR of a process P consist of non-leaf nodes N_i and leaf process nodes P_i. Each P_i is compiled into a monolithic LTS $\mathcal{L}_{P_i}$ represented in OBDD form $\phi(\mathcal{L}_{P_i})$ which uses variable vectors $V_{\mathcal{S}_i}$ and $V'_{\mathcal{S}_i}$ for the encoding of the state space of P_i. The vector V_{N_i} (V'_{N_i}) is defined to be the concatenation of the vectors $V_{\mathcal{S}_j}$ ($V'_{\mathcal{S}_j}$) of all leaf processes that are part of the process defined at N_i. For a non-leaf node N_i with two children nodes we annotate the latter with $left(N_i)$ and $right(N_i)$; we assume that for non-leaf nodes with a single descendant node in the tree $left(N_i) = right(N_i)$. Let N_0 be the root node of the HPTR of P.

Since both forward and backward state space traversal using OBDDs involve the same three basic operations—conjunction, existential quantification and vari-

able renaming—applied to different sets of variables, it suffices to present our approach on one concrete example. The derivation of any other form of relational product computation is very similar and is left out of our presentation.

We consider the relational product which computes the pairs of next states and events given a set of states Γ:

$$RelProd(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L}_P)) = (\exists_{V_{\mathcal{S}}}(\nu_{\mathcal{S}}(\Gamma) \wedge \nu_{\mathcal{R}}(\mathcal{R}))) \Big|_{[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]}$$

Because of the recursive structure of an HPTR, it is convenient to compute *RelProd* by traversing its nodes and deriving a partial relational product at each node. We denote this partial product as $RelProd_{HPTR}(F, N_i)$, where F is an OBDD that is passed to the node N_i from its parent. For the particular relational product under consideration F is used to encode Γ, however, in a different context F could be used to pass other information (e.g. an encoded set of events) specific to that context.

It is known that injective (one-to-one) OBDD variable renaming is a relatively inexpensive operation compared to disjunction and existential quantification, especially when the relative order of variables is maintained. Thus, it can be left out of the recursive image computation and applied at its end. The *RelProd* function can then be expressed via $RelProd_{HPTR}$ as follows:

$$RelProd(\nu_{\mathcal{S}}(\Gamma), \phi(\mathcal{L}_P)) = RelProd_{HPTR}(\nu_{\mathcal{S}}(\Gamma), N_0) \Big|_{[V_{\mathcal{S}} \leftarrow V'_{\mathcal{S}}]}$$

The other node types in a HPTR are hiding, parallel composition and interleaving, as well as the leaf nodes. In the following sections, we present the computation of $RelProd_{HPTR}$ for each of these node types.

4.4.1 Leaf nodes

Since the OBDD for the transition relation of a leaf process is readily available in a pre-computed form, $RelProd_{HPTR}$ of a leaf process P can be computed directly as:

$$RelProd_{HPTR}(F, P) = \exists_{V_{S_i}}(F \wedge \nu_{\mathcal{R}}(\mathcal{R}_P)) \tag{4.6}$$

One can easily observe from the above that the result of this computation is an OBDD which depends on the variables in $V_{\mathcal{E}}$, V'_{S_P} and the support variables of F except V_{S_P}. Indeed, it is an invariant of the recursive image computation for the chosen relational product that the intermediate result at any node N_i depends on the variables in $V_{\mathcal{E}}$, V'_{N_i} and those supporting F bar the vector V_{N_i}.

4.4.2 Hiding nodes

A hiding node passes F down to its child unchanged, but modifies the partial relational product derived by the child by replacing the events from the hiding set A with the internal τ event. Thus, for a hiding node N which represents the construct $left(N) \setminus A$ we have:

$$\begin{aligned} RelProd_{HPTR}(F, N) \;=\; & \exists_{V_\mathcal{E}}(RelProd_{HPTR}(F, left(N)) \wedge \nu_\mathcal{E}(A)) \wedge \nu_\mathcal{E}(\tau) \;\vee \\ & RelProd_{HPTR}(F, left(N)) \wedge \neg\, \nu_\mathcal{E}(A) \end{aligned} \tag{4.7}$$

4.4.3 Parallel and interleave nodes

The computation of the partial relational product for nodes representing a parallel composition of processes is non-trivial and is a key component of our technique of utilising HPTRs. This computation has to combine the two partial relational products from its children nodes and take into account both synchronous and asynchronous execution of events by the children processes. Since $P \;|||\; Q = P \underset{\emptyset}{\|} Q$, interleave nodes can easily be handled as a special case of the parallel ones.

We consider a node N that represents the construct $left(N) \underset{A}{\|} right(N)$. To aid the brevity of presentation, we introduce a number of abbreviations:

$$\begin{aligned} R_1 &= RelProd_{HPTR}(F, left(N)) \\ R_2 &= RelProd_{HPTR}(F, right(N)) \\ V_1 &= V_{left(N)} \\ V_1' &= V'_{left(N)} \\ V_2 &= V_{right(N)} \\ V_2' &= V'_{right(N)} \end{aligned}$$

and define the boolean function Id_i as:

$$Id_i = \bigwedge_{v_j \in V_i, v_j' \in V_i'} (v_j \wedge v_j' \vee \neg\, v_j \wedge \neg\, v_j')$$

According to the CSP compilation rule for parallel composition from Section 3.2.5 and the translation of set operators to OBDD operators in Sections 2.3.1 and 2.3.2, the OBDD encoding of the partial transition relation at a node N (denoted as $\mathcal{R}_N$) can be computed from the transition relations of the children of N as follows:

$$\begin{aligned} \nu_\mathcal{R}(\mathcal{R}_N) \;=\; & \neg\, \nu_\mathcal{E}(A) \wedge R_1 \wedge Id_2 \vee \\ & \neg\, \nu_\mathcal{E}(A) \wedge Id_1 \wedge R_2 \vee \\ & \nu_\mathcal{E}(A) \wedge R_1 \wedge R_2 \end{aligned}$$

The first line of the above equation corresponds to the left child of N making a transition that is not in the synchronisation set A, while the state of the right child remains unchanged. The second line represents the case when the right child performs an event different from those in A while the left one makes no transition. The last line of the equation corresponds to the case when both children of N make a simultaneous transition on an event from the synchronisation set A.

To derive the partial relational product at node N, we have to compute:

$$\begin{aligned} RelProd_{HPTR}(F, N) \;=\; & \exists_{V_1,V_2}(F \wedge \nu_{\mathcal{R}}(\mathcal{R}_N)) = \\ & \exists_{V_1,V_2}(F \wedge \neg\, \nu_{\mathcal{E}}(A) \wedge R_1 \wedge Id_2 \vee \\ & F \wedge \neg\, \nu_{\mathcal{E}}(A) \wedge Id_1 \wedge R_2 \vee \\ & F \wedge \nu_{\mathcal{E}}(A) \wedge R_1 \wedge R_2) = \\ & \exists_{V_1,V_2}(F \wedge \neg\, \nu_{\mathcal{E}}(A) \wedge R_1 \wedge Id_2) \vee \\ & \exists_{V_1,V_2}(F \wedge \neg\, \nu_{\mathcal{E}}(A) \wedge Id_1 \wedge R_2) \vee \\ & \exists_{V_1,V_2}(F \wedge \nu_{\mathcal{E}}(A) \wedge R_1 \wedge R_2) \end{aligned}$$

where the last transformation applies the disjunctive partitioning rule (4.2). Let us denote the three terms in the above disjunction with T_1, T_2, and T_3.

In T_1, we can move $\neg\, \nu_{\mathcal{E}}(A)$ outside of the existential quantification operator as it only depends on variables in $V_{\mathcal{E}}$:

$$T_1 = \neg\, \nu_{\mathcal{E}}(A) \wedge \exists_{V_1,V_2}(F \wedge R_1 \wedge Id_2)$$

Next, we can rewrite operator $\exists_{V_1,V_2}$ as two consecutive operators $\exists_{V_2}\exists_{V_1}$. We can further apply the conjunctive partitioning rule (4.4) to move R_1 and Id_2 which do not depend on variables in V_1 outside of the second existential quantification operator to obtain:

$$T_1 = \neg\, \nu_{\mathcal{E}}(A) \wedge \exists_{V_2}(R_1 \wedge Id_2 \wedge \exists_{V_1}(F))$$

Due to the way in which R_1 is derived, we know that $R_1 \wedge \exists_{V_1}(F) = R_1$, from which we can further reduce T_1 to:

$$T_1 = \neg\, \nu_{\mathcal{E}}(A) \wedge \exists_{V_2}(R_1 \wedge Id_2)$$

Since $\exists_{V_2}(R_1 \wedge Id_2) = R_1\big|_{[V_2' \leftarrow V_2]}$, we conclude that:

$$T_1 = \neg\, \nu_{\mathcal{E}}(A) \wedge (R_1\big|_{[V_2' \leftarrow V_2]}) \tag{4.8}$$

As T_2 is symmetrical to T_1, we have:

$$T_2 = \neg\, \nu_{\mathcal{E}}(A) \wedge (R_2\big|_{[V_1' \leftarrow V_1]}) \tag{4.9}$$

A similar line of reasoning can be applied to T_3 as well. We move $\nu_{\mathcal{E}}(A)$ outside of the existential quantification and remove the dependence on F:

$$T_3 = \nu_{\mathcal{E}}(A) \wedge \exists_{V_1}(R_2 \wedge \exists_{V_2} R_1) \tag{4.10}$$

The order of application of the existential quantification operators in the above equation is chosen to keep the OBDD sizes of the intermediate products in the computation as small as possible, which positively affects the time and space requirements of the implementation. Equations (4.8), (4.9), and (4.10) can be combined to derive the final equation for the partial relational product at a parallel composition node of a HPTR:

$$\begin{aligned} RelProd_{HPTR}(F, N) \;=\; & \neg\, \nu_{\mathcal{E}}(A) \wedge (R_1 \Big|_{[V_2' \leftarrow V_2]}) \vee \\ & \neg\, \nu_{\mathcal{E}}(A) \wedge (R_2 \Big|_{[V_1' \leftarrow V_1]}) \vee \\ & \nu_{\mathcal{E}}(A) \wedge \exists_{V_1}(R_2 \wedge \exists_{V_2} R_1) \end{aligned} \tag{4.11}$$

4.5 Experimental results

We have developed an implementation of an HPTR-based refinement checking algorithm utilising Equations (4.6), (4.7), and (4.11) to implement forward and backward relational product in the pair-by-pair refinement checking algorithm from Section 3.3.2 and incorporated it in the ARC tool. We call this new implementation ARC/PP+HPTR. To assess its performance compared to the monolithic transition relation used in ARC/PP, we use two examples from Appendix A:

- Monkey Puzzle, which is the largest example from Appendix A in terms of size of the OBDD encoding of the labelled transition relation of the implementation process;
- Dining Philosophers, which allows us to study the performance of HPTR when the size of the example is varied.

Of primary interest are the amount of savings in terms of OBDD size for the transition relation of the implementation process and the change in the total run-time for the examples. The experimental results obtained are summarized in Table 4.1. In the table, $time_1$ and $size_1$ refer to the total execution time (in seconds) for a traces check and the size of the transition relation OBDD for ARC/PP, and $time_2$ and $size_2$ refer to the same data for ARC/PP+HPTR.

The number of OBDD nodes in an HPTR (i.e. $size_2$) is computed as the sum of the sizes of the OBDDs for the transition relations of the leaf processes and does not reflect the possibility of node sharing between these OBDDs. Thus,

Example	ARC/PP		ARC/PP+HPTR		Ratios	
	$time_1$	$size_1$	$time_2$	$size_2$	$\frac{time_2}{time_1}$	$\frac{size_1}{size_2}$
3 dining philosophers	0.4	577	0.9	225	2.25	2.56
4 dining philosophers	2.0	1021	7.0	306	3.5	3.34
5 dining philosophers	17.5	1589	71.0	385	4.06	4.13
Monkey puzzle (with re-encoding)	295	21449	5769	3451	19.6	6.2
Monkey puzzle (w/o re-encoding)	3079	35147	16975	4714	5.51	7.45

Table 4.1: Performance with and without the use of HPTR

$size_2$ is actually an upper bound for the number of OBDD nodes required by the corresponding HPTRs. The last two columns in the table contain the space saving offered by ARC/PP+HPTR ($size_1/size_2$) and the corresponding slowdown of the refinement check ($time_2/time_1$) compared to ARC/PP.

Prior to analysing the data in Table 4.1, it should be noted that none of the examples run actually required more memory than physically available. This means that the potential speed benefit of refinement checking using HPTR is not revealed by our experiments. Such a benefit is possible when a check not using HPTR hits the physical space limit while the same check using HPTR does not. The slow, non-sequential access to virtual memory that a refinement check not using HPTR has to perform in this scenario is expected to more than offset the generally slower recursive image computation using HPTR. Unfortunately, we do not have an example large enough to properly test such a scenario; the largest LTS we could obtain was that of Monkey Puzzle with state re-encoding disabled. Nevertheless, we hope that the results presented in this section demonstrate the potential of HPTRs.

Looking at the results for Dining Philosophers in Table 4.1, we notice that the space advantages of using HPTR instead of monolithic transition relation are offset by a slow-down of an almost equal factor across the tests with three, four, and five philosophers. This represents a trade-off between space and time complexity and confirms similar findings by other researchers using partitioned transition relations [BCL91b, BCL+94]. As expected, $size_2$ grows linearly with the number of processes in the example.

Monkey Puzzle presents a slightly different case—the application of HPTR slows the refinement check by a factor of nearly 20 while achieving a reduction in the size of transition relation of slightly over six-fold. But is this really unexpected? This example consists of 20 fairly simple processes, ranging from 1 to 7 states each, and the bulk of the complexity is in the complex synchronisation

pattern between these processes. As a result, the leaf processes in the HPTR for Monkey Puzzle are of very fine grain which impacts the performance of the recursive image computation negatively. We have run the same example with state re-encoding disabled, which results in larger OBDDs for the transition relations of leaf processes in the HPTR as well as a larger OBDD for the monolithic LTS. This has positive impact on HPTR-based refinement checking compared to the previous case—a greater than seven-fold reduction in number of OBDD nodes for the transition relation at the expense of less than six-fold slow down.

4.6 A further enhancement

The literature on partitioned transition relations recommends combining several small partitions into a larger one (assuming, of course, that the compound partition does not require more memory than is available), claiming that both space and time advantages are possible [BCL91b, BCL+94]. In the case of HPTR, the size of the partitions is determined by the syntactic representation of the compiled process description. Of course, it is quite feasible to modify the derivation of an HPTR described in Section 4.3 to allow for non-leaf nodes in the HPTR to be converted into leaf nodes with a monolithic OBDD-based transition relation.

A good strategy for obtaining an HPTR with larger partitions is to attempt building a monolithic OBDD-based transition relation for each non-leaf node during the construction of the HPTR. Then, if this operation does not result in an OBDD blow-up and the size of the monolithic transition relation does not exceed a predefined threshold, the non-leaf node is substituted by a leaf node containing that transition relation; otherwise the non-leaf node and the sub-tree under it is kept as is. The application of this strategy will result in constructing an HPTR with potentially fewer leaf and non-leaf nodes.

A simpler variation of the above strategy is to impose a limit on the depth of the synchronisation hierarchy, so that any sub-tree below a particular level under the root node is "cut-off" and substituted by a leaf node containing a monolithic OBDD-based transition relation. This technique has been implemented in ARC/PP+HPTR.

Table 4.2 presents our experimental findings for Monkey Puzzle with varying cut-off depth. It shows that the user of the tool can fine-tune the desired space versus time trade-off by varying the depth of the cut-off. The best cut-off point is one where the available physical memory is utilised to the highest possible degree without actually exhausting it.

Cut-off depth	ARC/PP		ARC/PP+HPTR w/cut-off		Ratios	
	$time_1$	$size_1$	$time_2$	$size_2$	$\frac{time_2}{time_1}$	$\frac{size_1}{size_2}$
3	295	21449	1047	16993	3.55	1.26
6	295	21449	1960	12474	6.64	1.72
9	295	21449	2982	8733	10.1	2.46
12	295	21449	4061	5871	13.8	3.65

Table 4.2: Experiments with ARC/PP+HPTR and cut-off depths on Monkey Puzzle example

4.7 Related work

To the best of the knowledge of the author, HPTRs are the first partitioning technique for the OBDD encoding of a labelled transition relation derived from an algebraic description of a concurrent system. However, the idea of utilising a structured (hierarchical) representation of a transition relation is not new.

Hu [Hu95] defines a high-level language for describing interacting systems and distributed protocols. Not surprisingly, this work also addresses the potential size blow-up of monolithic OBDD encodings of transition relations. Hu suggests that three of the high-level language statements—sequence, if-then-else, and nondeterministic choice—can be used to compute successor and predecessor states with a simple OBDD recursive algorithm rather than simple relational product operation on a pre-computed transition relation.

Alur *et al.* [AHR98] present an OBDD-based algorithm for the exploration of a multi-level hierarchy of transition systems. The state space hierarchy is implicitly defined by the use of a special temporal abstraction operator within the input description of a circuit design. The authors develop a method of computing the successors of a given state set based on conjunctive partitioning for parallel (synchronous) composition of circuit modules and an algorithmic technique for the temporal abstraction operator.

The technique presented in this chapter differs from the existing methods in several aspects. Firstly, our approach combines disjunctive and conjunctive partitioning in a novel way to enable the addition of parallel composition to the HPTR. Secondly, we include a set of structural optimisation rules that may significantly improve the performance of the recursive image computation. Finally, our technique is based on a very different set of operators than Hu's [Hu95] and, unlike the approach by Alur *et al.* [AHR98], does not rely on the user to explicitly or implicitly define the structure of the HPTR.

Chapter 5

Pseudo-Root States

Overview

This chapter presents a novel tool-independent reachability analysis technique that is applicable to both OBDD-based and on-the-fly model checking. This technique identifies and selectively discards certain states, referred to as pseudo-root states, which cannot be reached from outside of the reached state space. An experimental implementation of this approach in the ARC tool has shown a two to sixteen fold improvement in peak state storage requirements over conventional reachability analysis. This chapter is based on material presented by the author at TACAS'97 [PY97].

5.1 Existing reachability analysis methods

Exhaustive exploration of a finite state model is a key part of most automated verification algorithms, which rely on checking all reachable states of the model against a specification of desired properties. Conventional algorithms for traversing the model starting from its initial states require storing the set of all visited states. This is done in order to guarantee that each state is visited exactly once and thus avoid redundant work when a reachable state may be reached more than once through a different sequence of transitions.

The need to store all reached states is a major performance bottleneck for automated verification algorithms. The amount of memory required for storing these states is, in general, proportional to their number. Although some tools such as FDR feature state space storage mechanisms targeting minimal time overhead when non-direct access memory is used [Ros94], there is a general consensus in the literature that the use of virtual memory considerably slows down state

matching [Hol88, GHP92]. Thus, it is highly desirable that the verification is performed within the bounds of the available physical memory. Since reachability analysis involves a search through the stored states for each newly reached state, there is a growing time overhead as well. Several approaches have been proposed to alleviate this problem.

A popular approach for verifying complex systems is symbolic model checking [BCM+92, McM92a] which uses OBDDs for encoding both the finite model and the set of reached states. This brings about several potential advantages. Firstly, the finite model is not constructed explicitly and, given a well chosen variable ordering, an OBDD consisting of a few thousand nodes may represent a model with a vast state space—10^{30} states and more. Secondly, if the set of visited states exhibits some form of regularity, the OBDD representing this set may require many orders of magnitude less space than, for example, a hash table containing the corresponding state vectors. Thirdly, symbolic model checking allows sets of states, not only a single state, to be checked at each iteration of the reachability analysis procedure. Unfortunately, the size of the OBDDs representing the visited states is unpredictable, and in the worst case may be much larger than the sum of sizes of the corresponding state vectors. A good example of this in the context of the ARC tool is Monkey Puzzle (described in Appendix A), for which the tool has to store 66 OBDD nodes plus the associated overhead instead of the (minimum of) 66 bits required by methods based on storing state vectors.

Another reachability analysis method proposed in the literature is *scatter searching* [Hol87]. The idea is to explore, within the limits imposed by the available memory or time, a subset of the full finite model and then declare the system "error-free" with a certain degree of probability. It is possible to further increase this probability by performing multiple runs with randomly chosen hashing functions for state storage, effectively exploring random subsets of the global state space. An error found in any of these subsets is an error in the complete model. However, not finding any errors does not imply the complete correctness of the model. The *bit-state hashing* technique [Hol88] enhances this method and allows scatter searching to be carried out considerably faster and in less space. However, this is essentially a probabilistic method that may not be acceptable in cases where a definitive result is required.

State space caching approach [Hol87, JJ91] stores in memory as many reached states as possible. After the available memory is filled up, newly generated states replace states from memory (chosen randomly or otherwise), the latter serving as a cache of the set of visited states. As states discarded from memory may be reached again at a later stage of the reachability analysis, there is a considerable risk of the redundancy of repeated work. In practice, using a cache of size less than a third of all reachable states brings unacceptable time penalties [Hol87, GHP92].

An enhancement of the state space caching technique [GHP92] uses *partial order* methods [God94, Val93, Pel94] to deal with interleavings of independent transitions and reduce the number of times a state is visited during reachability analysis. However, a state may be reachable more than once for reasons other than pure interleaving—one example being cyclic behaviour. Thus, the risk of revisiting discarded states, although reduced, is still present.

More recently, an approach combining state space reduction, partial order driven reachability analysis, and a modification of the bit-state hashing technique has been proposed [MK96]. A pre-computation stage involving exhaustive state space exploration of the finite model is used to gather approximate information about the number of times each state of the model is to be revisited during finite model exploration. This information is then utilised in the course of the actual model checking for a more selective removal of states from memory when space limits are reached compared to that used earlier [GHP92]. This is a sensible approach due to the number of states in the finite model often being much lower than the number of the product states explored during model checking.

With the exception of the refined state space caching technique [GHP92] and its further improvement [MK96], a common feature of the existing reachability analysis methods is that states are treated as separate entities outside of the finite model they belong to. In other words, the potentially advantageous information about states and transitions in the model is not utilised; states are stored obliviously.

In this chapter, a different view of the state space storage problem resulting in a novel reachability analysis algorithm is presented. Whereas the refined state space caching technique extracts the necessary information by applying higher-level syntactic rules on the system description, our method is lower-level in that it works by utilising information at the level of the model itself. More specifically, our approach keeps track of the visited predecessors for each state and as soon as all predecessors of a state are explored it is discarded from memory. States that are no longer reachable from the unexplored state space are referred to as *pseudo-root states* (PRS). We present an efficient algorithm for the identification and deletion of a visited state that becomes a PRS. Complexity analysis shows that this algorithm incurs at most a two-fold increase of the run-time compared to the conventional reachability analysis algorithm. Our experimental results indicate that this technique allows exhaustive state space exploration visiting each state *exactly once* while storing, at any one time, between 6–45% of the reachable state space.

The rest of this chapter is organised as follows. Section 5.2 considers a generic version of the conventional reachability analysis algorithm, and presents two examples which illustrate the intuitions behind our approach. Section 5.3 defines

the notion of pseudo-root states and presents the modified state space exploration algorithm. Then the time and space complexity of our algorithm is analysed. Section 5.4 comments on the performance results obtained with the ARC tool [PY96a]. In Section 5.5 we discuss issues concerning the practical application of the pseudo-root state method.

5.2 Background and motivation

5.2.1 Conventional reachability analysis

A slightly modified version of the pure reachability analysis algorithm (Algorithm 1 from Section 2.2.1) which applies the verification function *Check* on each reached state is provided by Algorithm 9. Two sets of states are maintained, *Checked* for the states that have been visited, and *Pending* for the states that have been reached but not checked yet. We use interchangeably the terms visited, explored or checked for states from the set *Checked*, and refer to states from the set *Pending* as pending. We call the states in $Checked \cup Pending$ reached, and the rest of the states in $\mathcal{S}$ unreached.

Algorithm 9 Conventional reachable states checking

Require: An LTS $\mathcal{L}$
Ensure: *Check* is called on each reachable state in $\mathcal{L}$ exactly once

```
1:  Checked ← ∅
2:  Pending ← I
3:  while Pending ≠ ∅ do
4:    select s from Pending
5:    Pending ← Pending − {s}
6:    Check(s)
7:    Checked ← Checked ∪ {s}
8:    for r ∈ Image({s}, L) do
9:      if r ∉ Checked ∪ Pending then
10:       Pending ← Pending ∪ {r}
11:     end if
12:   end for
13: end while
```

Initially, *Checked* is empty, and *Pending* contains the set of initial states of $\mathcal{L}$. Algorithm 9 is iterative and terminates when *Pending* becomes empty. Each iteration explores one state s selected from *Pending*, which is then removed from that set and added to *Checked*. Next, a call to the checking function with the

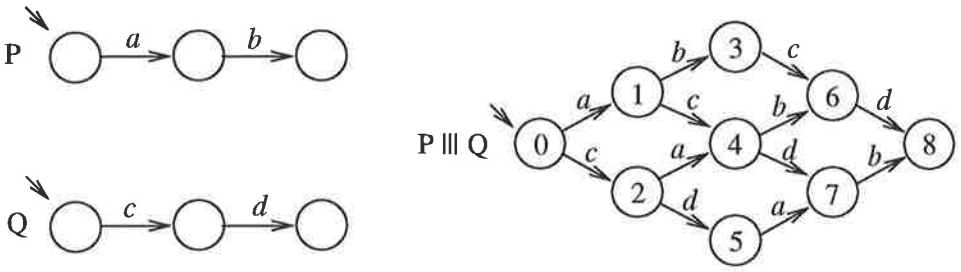


Figure 5.1: LTS of the interleaved processes example

current state s is performed. At the end of each iteration the successor states of s are examined and those not reached yet are added to *Pending*. Termination is assured by the monotonic growth of *Checked* and the finiteness of $\mathcal{S}$—eventually all pending states will have to be checked.

Two strategies for selecting a state from *Pending* have become most popular. Breadth-first search (BFS) tracks the pending states in a FIFO queue (a list), while depth-first search (DFS) stores these states in a LIFO queue (a stack)[1].

It may be observed that the set *Checked* is used solely for determining if the successor states of s have been reached already, ensuring that each state is visited exactly once. However, some states in *Checked* may be no longer reachable from the outside of the explored state space. Keeping such states in *Checked* is unnecessary as well as undesirable because that brings about performance penalties:

- These states take up valuable space;
- The computation of the condition $r \notin \mathit{Checked} \cup \mathit{Pending}$ is slower due to the larger size of *Checked*.

In the rest of this section we present two simple examples which demonstrate that some checked states need not be kept throughout the search. These examples served as motivation and encouragement for the development of the PRS approach to reachability analysis.

5.2.2 Interleaved processes example

Consider two very simple models P and Q combined by interleaving (Figure 5.1) and assume that the product LTS is explored by Algorithm 9 performing BFS.

[1]Strictly speaking, the DFS strategy actually requires a slight modification of Algorithm 9 along the lines of Algorithm 2.

The search starts from the initial state 0, which is then put in *Checked* while its successor states 1 and 2 are put into *Pending*. However, state 0 is a root state (a state with no incoming transitions) in the LTS, and therefore it is not reachable from any other state in that LTS. It may be safely discarded from *Checked* without affecting the future behaviour of Algorithm 9.

At the next iteration state 1 is explored and moved into *Checked*. States 3 and 4 are put into *Pending*. State 1 may only be reached from state 0, which itself has been already identified as unreachable. Therefore, state 1 is no longer reachable, too, and its removal from *Checked* will not affect the rest of the LTS exploration. A similar conclusion may be drawn for state 2 as well.

Following this line of reasoning at every step until the end of the reachability analysis, it becomes evident that the LTS can be exhaustively explored, visiting each of the nine states exactly once, by storing no more than three states (all in *Pending* and none in *Checked*) at any one time. All that is required is extracting and utilizing some information about the states from the LTS.

5.2.3 Counting example

This example models the behaviour of a finite counter. Figure 5.2 presents the LTS of a counter between 0 and 3. Each of the states is labelled with the value of the counter, which may be incremented by an *inc* transition and decremented by a *dec* transition. Again, let us assume that the LTS of this example is explored by Algorithm 9 using BFS.

Reachability analysis starts from the initial state 0. State 0 is moved to *Checked* and its only successor state 1 is put into *Pending*. At the next iteration, state 1 is explored and put into *Checked*, while its successor state 2 is stored in *Pending*. At this point, *Checked* consists of states 0 and 1. However, any path in the LTS starting from an unreached state and ending at state 0 must contain state 1, as state 1 is the only predecessor of state 0. Therefore, state 0 may be discarded from *Checked* without any risk of revisiting it again.

When state 2 is explored, it is moved into *Checked* and state 3 is put into *Pending*. With states 1 and 2 in *Checked*, state 1 may only be reached through state 2 similarly to the previous iteration. State 1 need not be kept in *Checked* and may be discarded. State 3 is the last to be explored, as state 2 is its only successor state, *Pending* becomes empty and reachability analysis terminates.

Although we have used a counter from 0 to 3 as a particular example, it is not hard to see that any finite counter can be exhaustively explored while storing only two of its states simultaneously. As the exploration progresses, we are able to minimise the number of states in *Checked* by leaving in only a subset of the reached states which, in a way, still uniquely represents (identifies) the full set of

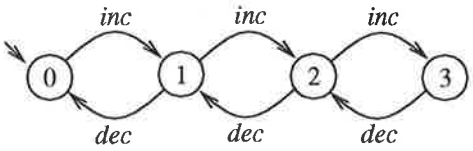


Figure 5.2: LTS of the counting example

reached states.

5.3 PRS-based reachability analysis

5.3.1 Pseudo-root states

The informal reasoning used in Sections 5.2.2 and 5.2.3 is based on two propositions for safely discarding an explored state. Suitable candidates for discarding from *Checked* are either root states (which can obviously be only members of $\mathcal{I}$) or states reachable only through other visited states. We call a state of the latter kind a *pseudo-root state*.

Definition 11 A state σ in an LTS $\mathcal{L}$ is a pseudo-root state with respect to a set of states $\Gamma \subseteq \mathcal{S}$ if and only if Γ contains all predecessors of σ:

$$BackImage(\{\sigma\}, \mathcal{L}) \subseteq \Gamma$$

The interesting case is when Γ in the above definition is the set *Checked* from Algorithm 9, and in what follows pseudo-root state actually means pseudo-root state w.r.t. *Checked*.

Theorem 2 Once a state becomes PRS during state space search, it remains PRS until the end of the search.

Proof: The state set *Checked* representing the set of states visited during state space exploration grows monotonically while the reachability analysis progresses. Therefore, once all the predecessors of a state are included in *Checked*, they remain there until the end of the state space traversal. ■

The above theorem presents a very useful property. However, to achieve a space saving compared to conventional reachability analysis such as Algorithm 9, a mechanism is needed to remove states from *Checked* without affecting the state space search in any way. The following theorem proves a basic property to support such a state removal mechanism:

Theorem 3 Let $\sigma \in Checked$ be a pseudo-root state. Then, σ can be removed from *Checked* without affecting Algorithm 9.

Proof: Let us assume that the removal of σ from *Checked* does affect the subsequent reachability analysis performed by Algorithm 9. An inspection of the algorithm reveals that the only computation that depends on the state set *Checked* occurs in the condition of an **if** statement on line 9 of the algorithm. The removal of σ from *Checked* can affect this condition in one of two ways:

1. Changing the condition from *true* to *false*. However,

$$r \notin Checked \cup Pending \Rightarrow r \notin Checked \cup Pending - \{\sigma\}$$

   which is a contradiction;

2. Changing the condition from *false* to *true*. This can only be the case if r and σ are the same state. Also, if r and σ are the same state, then they should have the same set of predecessor states. Since σ is PRS, all of its predecessors have been in *Checked* prior to visiting state s. However, state r has at least one predecessor, s itself, which was not in *Checked* prior to exploring state s. This is a contradiction.

Both cases above lead to contradiction, which means that the initial assumption that removing σ from *Checked* affects the subsequent reachability analysis is contradictory. This proves the theorem. ■

To implement the removal of pseudo-root states from *Checked*, a technique for identifying pseudo-root states is required. This is discussed in the next section.

5.3.2 PRS-enhanced reachability analysis algorithm

A state becomes pseudo-root as soon as all of its predecessors have been visited. A straightforward implementation of this rule would be to keep, for every reached state, the set of its visited predecessors and upon each update of that set compare the latter with the set of all predecessors of the state. Although feasible, such a solution would be rather inefficient, because the space and time overheads for keeping sets of states for each reached state could be substantial.

Fortunately, there is a simpler way of identifying pseudo-root states. Assuming that memory limits are not hit, Algorithm 9 has the property of exploring each state in the LTS exactly once, although a state may be reached more than once. Therefore, the set of its predecessor states need not be kept in memory or re-computed at each iteration of the search—a simple counter of the number

Algorithm 10 PRS-enhanced reachability analysis algorithm

Require: An LTS $\mathcal{L}$
Ensure: *Check* is called on each reachable state in $\mathcal{L}$ exactly once

1: $Checked \leftarrow \emptyset$
2: $Pending \leftarrow \mathcal{I}$
3: **for** $s \in \mathcal{I}$ **do**
4: {Set the reference counters of initial states}
5: $Counter(s) \leftarrow \mid BackImage(\{s\}, \mathcal{L}) \mid$
6: **end for**
7: **while** $Pending \neq \emptyset$ **do**
8: select s from $Pending$
9: $Pending \leftarrow Pending - \{s\}$
10: $Check(s)$
11: **if** $Counter(s) > 0$ **then**
12: {s is not PRS, include it in $Checked$}
13: $Checked \leftarrow Checked \cup \{s\}$
14: **end if**
15: **for** $r \in Image(\{s\}, \mathcal{L})$ **do**
16: **if** $r \in Checked$ **then**
17: **if** $Counter(r) = 1$ **then**
18: {r becomes PRS, discard it}
19: $Checked \leftarrow Checked - \{r\}$
20: **else**
21: $Counter(r) \leftarrow Counter(r) - 1$
22: **end if**
23: **else if** $r \in Pending$ **then**
24: $Counter(r) \leftarrow Counter(r) - 1$
25: **else**
26: {r is an unreached state}
27: $Pending \leftarrow Pending \cup \{r\}$
28: $Counter(r) \leftarrow \mid BackImage(\{r\}, \mathcal{L}) \mid -1$
29: **end if**
30: **end for**
31: **end while**

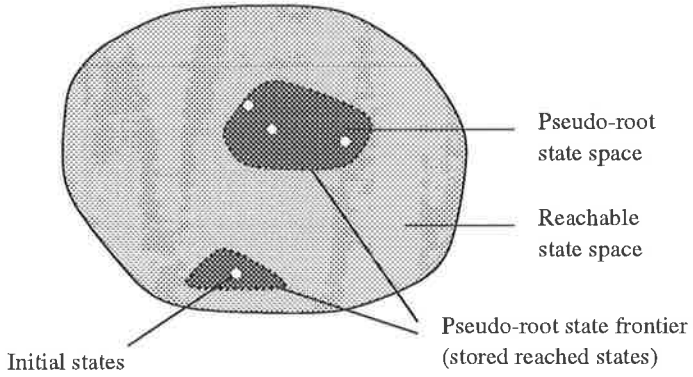


Figure 5.3: Visualisation of the PRS-based reachability analysis

of predecessors yet to be visited would suffice. This idea is similar to the *reference counting* approach used for garbage collection of dynamically allocated memory [Coh81, Har90].

An implementation of the above pseudo-root state detection method is given in Algorithm 10. Our modified reachability analysis algorithm is based on Algorithm 9. Every state stored in *Checked* or *Pending* is paired with an integer number *Counter* equal to the number of times that state is to be reached again. At the beginning, *Checked* is empty and the initial states of $\mathcal{L}$ are put into *Pending* (lines 1–6 of the algorithm). Upon each iteration of the main loop, a state s is selected for exploration and removed from *Pending*. If the reference counter of s is not zero then s may be reached again through a predecessor which has not been visited itself yet, and thus s is put into *Checked*. If the reference count of s is zero then s has become pseudo-root and, therefore, can be discarded without ever been put in *Checked*.

Next, all successor states r of s are considered (lines 15–30). If r is a member of *Checked*, then its reference counter is at least one. If the reference counter is equal to one, then s is the last of all predecessors of r which identifies r as pseudo-root and r is discarded from *Checked*. If the reference counter of r is greater than one, it is decremented by one to reflect the fact that another of its predecessors has been visited. In case r is a member of *Pending*, its reference counter is also decremented by one but the state is not discarded even if the result is zero, because a pending state has yet to be visited. If r has not been reached yet, it is included in *Pending* paired with its number of predecessors decreased by one (as s, one of r's predecessors, is already a checked state).

The operation of Algorithm 10 in progress is visualised in Figure 5.3. The lightly shaded area represents the reachable state space of an LTS, the white dots

are the initial states of the LTS, and the dark shaded area denotes the explored state space. A conventional LTS traversal algorithm has to keep all states in the dark shaded area, whereas the algorithm presented in this section only stores the states on the border between the light and dark shaded areas (the dark dots), because the inside of the reached state space is pseudo-root. As the algorithm advances, the area encompassed by black dots will grow; the two separate dark shaded areas will be joined and eventually all reachable states will be covered and the black dots will disappear. This process is remarkably similar in nature to burning dry grass on an island—the fire line quickly expands in all directions from the fire's ignition points until it is extinguished at the shores of the island.

5.3.3 Reaching physical memory limits

The PRS-based reachability analysis algorithm as presented in the previous section does not consider the possibility of exhausting the available memory. An interesting question is what can be done when, despite all efforts, the size of *Checked* ∪ *Pending* exceeds the available space. If the algorithm is to complete successfully, then states which are *certain* to be reached again have to be discarded.

One option is to discard a pending state with a reference counter greater than zero. Such a state is certain to be reached again and therefore will not be missed if deleted from *Pending*. The disadvantage of discarding a pending state with a non-zero reference counter is that when that state becomes pending again its reference counter will not reflect the fact that it has already been reached at least once. As a result, the state will not become pseudo-root and will remain in *Checked* until the end of the state space search. This approach, of course, does not completely solve the space shortage problem, but rather provides means for a graceful performance degradation of Algorithm 10.

At some stage, a checked state which is not pseudo-root may have to be discarded. When this happens, state space exploration may "leak" back into the reached state space and visit a state more than once. Under these conditions, a DFS strategy is required in order to guarantee termination of the algorithm. Essentially, the PRS-based reachability analysis algorithm starts to approximate the state space caching algorithm with DFS presented elsewhere [Hol87, JJ91]. It should be noted, however, that Algorithm 10 is expected to reach the memory limits much later in the search than the other algorithms because of the slower growth of *Checked*.

5.3.4 Complexity analysis

As with any other approach addressing the state space explosion problem, the performance of the PRS-based reachability analysis algorithm greatly depends on the actual example it is applied to, and more specifically, on the properties of the LTS graph of the model. An in-depth complexity analysis should therefore consist of not only worst-case, but also best-case and, if possible, average-case scenario performance.

It is also useful to anticipate the trade-offs (in this case, space versus time) of the new algorithm in order to evaluate its applicability to a particular verification problem. Certainly, there is no replacement for experimental data, which is provided later in Section 5.4. To estimate the space and time complexity of the proposed algorithm we present a relative comparison with Algorithm 9.

Space complexity

As Algorithm 9 stores all reached states in $\mathcal{L}$, its space requirements are of the order of $\mid \mathcal{S} \mid \times K$, where K is the size of a single state in bits. Note that $K \geq \lceil \log_2 \mid \mathcal{S} \mid \rceil$. In general, Algorithm 10 is expected to store far fewer reached states, but for each of these states requires additional space for storing a reference counter. The size of the reference counter must be at least $\lceil \log_2(d_{\max}) \rceil$ bits, where $d_{\max}$ is the maximum number of predecessors for a state in the LTS.

The worst-case space requirements of Algorithm 10 can be expected when $d_{\max} = \mid \mathcal{S} \mid$, that is, when there is a state in S which has incoming transitions from all states in $\mathcal{S}$. Assuming that the pseudo-root state method cannot discard any states from *Checked* until the end of the search, and that $K = \lceil \log_2 \mid \mathcal{S} \mid \rceil$, we obtain that the worst-case space complexity of PRS-based state space exploration is twice as much as that of Algorithm 9.

Algorithm 10 would perform best on a model in which every state has at most one predecessor—much like the counting example from Section 5.2.3, but without the *dec* transitions. Then one bit would suffice for the reference count, and Algorithm 10 would store at most one state at any one time. Such an LTS can be explored using only $K + 1$ bits of storage.

For most practical examples we have come across, each state has at most several incoming transitions and thus a reference counter of only a few bits length is sufficient. On the other hand, the state vector of such examples may require hundreds of bits [Hol88, God94]. Therefore, the average increase in space requirements for a single state is negligible. Also, our experimental results presented in Section 5.4 show that PRS-based reachability analysis needs to store, at any one time, between 2 to 16 times fewer states than Algorithm 9. This far outweighs the additional storage required by the reference counters.

Time complexity

In general, both algorithms execute the same number of iterations, exploring each state exactly once and following each transition exactly once. However, the PRS-based state space exploration performs two extra computations compared to Algorithm 9. Firstly, it sets the reference counter of every reached state by computing the set of its predecessor states, and secondly, it decrements this counter for every reached predecessor. The total number of decrements is less than or equal to the number of transitions in the LTS, because a state is connected to each of its predecessors by one or more unique transitions. As decrement and comparison operations are much faster (and take constant time) compared to state space search, we may ignore them in the analysis. The important factor in this analysis remains only the time required for the computation of predecessor states.

In a given LTS $\mathcal{L}$, the total number of successor states is equal to the total number of predecessor states. If we assume that computing predecessor states is of the same computational cost as deriving successor states then, in the worst case, an iteration of Algorithm 10 would require approximately twice as much time as an iteration of Algorithm 9 does. For both algorithms the number of iterations is equal to the number of reachable states in $\mathcal{S}$. Therefore, we may conclude that the worst case running time of PRS-based reachability analysis is approximately double that of Algorithm 9.

This informal worst-case analysis ignores the time required to search through reached states. To estimate average and best-case time performance of Algorithm 10 one would require some knowledge of the time complexity of state matching relative to the complexity of computing a successor state, which is implementation dependent (e.g. the LTS may be in OBDD form or computed on-the-fly, states may be stored in a hash table or in a balanced tree, etc.). In ARC, for example, the search through the reached states takes only a small fraction of the total run-time and any improvements in state matching brought about by the exclusion of pseudo-root states from the search set have only a marginal effect on the run-time. In tools like SPIN, where state matching takes up a substantial portion of the total run-time [Hol88], the average and best case time performance of the PRS-based reachability analysis algorithm may be considerably better than the worst case.

5.4 Experimental results

In this section we present experimental results for the PRS-based state space exploration algorithm as implemented in the ARC tool and applied on some of the

Number of philosophers	Number of states stored by ARC/PP	Peak number of states stored by ARC/PP+PRS	Percentage of total states
3	154	69	44.8%
4	832	370	44.5%
5	4474	1940	43.4%
6	24040	10171	42.3%

Table 5.1: State savings for Dining Philosophers example

examples from Appendix A. We refer to our implementation of the PRS-enhanced refinement checking algorithm as ARC/PP+PRS. Since ARC/PP uses breadth-first search LTS traversal strategy, ARC/PP+PRS implements Algorithm 10 with BFS as a state selection strategy in line 8 of the algorithm. Note that choosing a different state selection strategy may result in obtaining different numbers (more on this in Section 5.5.2).

For most examples we have run, the execution time of the refinement check with PRS-based reachability analysis averages to approximately twice that of the standard pair-by-pair refinement check. For larger examples, the overhead grows beyond this figure. However, we consider this to be a specific feature of our prototype OBDD-based implementation rather than a general deficiency of the algorithm itself.

Table 5.1 summarises the experimental data obtained for the Dining Philosophers example. The space reductions offered by Algorithm 10 for this example are approximately two-fold and are in fact the lowest of all examples we have run. We attribute this to the cyclic behaviour of both philosophers and forks and the form of the LTS of this example, which has a bisectional width of the reachability graph that is several times larger than the depth of that graph.

ARC/PP+PRS achieves a similar saving in the number of peak states required during reachability analysis for the Alternating Bit Protocol example. In particular, when DATASIZE is set to 5, up to 29 of the 66 reachable states (or 43.9% of the reachable state space) need to be stored at any one time.

Another example on which we have run ARC/PP+PRS is Milner's Schedulers. Again, we compare the peak number of states stored by Algorithm 10 to the total number of reachable states (Table 5.2). Perhaps the most interesting observation that can be made from Table 5.2 is that space improvements grow with the number of schedulers. This can be explained by the slower growth of the bisectional width of the LTS of this example as opposed to the growth in the number of states. A similar although much less pronounced trend can be observed in Table 5.1 as well.

Number of processes	Reachable states	Peak number of states stored by Algorithm 2	Percentage of total states
3	33	11	33.3%
5	193	50	25.9%
7	1025	206	20.0%
9	5121	816	15.9%
11	24577	3262	13.3%

Table 5.2: State savings for Milner's Schedulers

The next example on which ARC/PP+PRS has been used is Monkey Puzzle. We have modified Algorithm 10 to terminate immediately after the first solution has been found. In our test run 23,679 states of the puzzle have been explored before finding a solution to it while only a maximum of 1,425 states have been stored at any one time. In other words, with this example Algorithm 10 results in more than 16 fold reduction in memory requirements.

Finally, ARC/PP+PRS achieves comparable state savings on the Solitaire example. Since a valid move in this puzzle irreversibly changes the number of pegs on the board, a state reachable in n moves in Solitaire becomes PRS as soon as all states reachable in $n-1$ moved are explored. As a result, a peak of slightly over 27 million states reachable after the eighteenth move is required to explore each one of the 187,636,299 reachable states in this example. This represents close to a seven-fold reduction in the peak number of stored states.

As previously noted in Section 3.3.3, due to the use of hiding in Solitaire the entire reachable space of this example is computed in a single call to $TauExpand_{OBDD}$ that iterates through all possible moves in the puzzle. The application of PRS-enhanced reachability analysis within the $TauExpand_{OBDD}$ function results in storing only the set of all peg configurations reachable after a certain number of moves rather than all reachable states. This significantly reduces the size of the OBDD encoding of the set of states maintained by $TauExpand_{OBDD}$, as shown in Figure 5.4. This reduction has a rather positive effect on the run-time of ARC, which drops from 24016 to 4389 seconds—a nearly six-fold reduction in run-time.

5.5 Discussion

5.5.1 Computing predecessor states

As it was stated in Section 5.3.4, the time complexity of the proposed PRS-based state space exploration algorithm is dependent on the ability to effectively

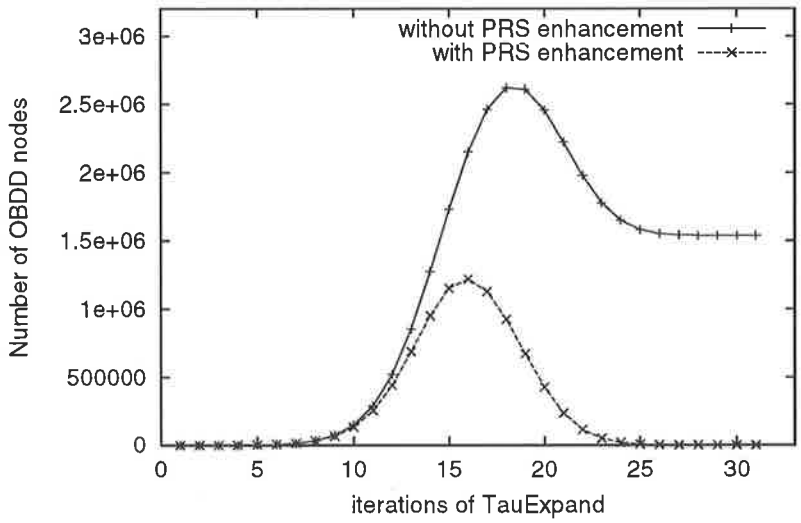


Figure 5.4: OBDD size for the reachable states in Solitaire with and without the use of PRS

compute the number of predecessor states.

When the full LTS is available in OBDD form (as is the case with the ARC tool) computing predecessors poses no significant problems. In this case, having the complete transition relation of the finite model allows the computation of the predecessor states to be done using the same relational product operation proposed for the derivation of successor states [BCM+92, BCL+94].

If on-the-fly verification technique is used, computing predecessor states might be slightly more complicated, but not necessarily less efficient. For example, the FDR2 tool [For97] stores the global LTS implicitly by storing the LTSs of the low-level processes and applying a set of synchronisation rules to infer the complete operational behaviour including successor states. Computing predecessors in this context appears to be not too different from the problem of computing successors and likely requires a comparable amount of time.

5.5.2 BFS, DFS and alternatives

Although our prototype implementation of Algorithm 10 uses BFS as the underlying search strategy, BFS may not be optimal for every LTS. One can expect that for reachability graphs whose bisection width is greater than their depth, DFS would tend to offer better space savings. In general, neither BFS nor DFS can be expected to provide the best space savings for all cases. An open question

for future exploration is whether there exists a heuristic criterion for dynamically selecting a state from *Pending* that can be used to maximise the space benefits of discarding pseudo-roots from the set of stored states.

5.5.3 Comparison to previous work

The main difference between our approach and Godefroid *et al.*'s refined state space caching technique [GHP92] is in the type of information about states that is used by the algorithm and the level at which it is obtained. Algorithm 10 works by utilizing information about the predecessors of a state extracted from the LTS, whereas the sleep set method presented by Godefroid *et al.* [GHP92] makes use of information about transition dependencies extracted at the higher syntactic level of system description (the source code or script). Our method is independent of the input language and its semantics and, therefore, is applicable to a wide range of tools. Also, it allows space reduction for tightly coupled concurrent systems such as Monkey Puzzle, for which a partial order technique would have had little or no effect.

The approach discussed by Miller and Katz [MK96] relies on *approximate* information on each state predecessor gathered in the pre-processing of the finite model. A hash function is used to map states of the model onto a (smaller) number of statically allocated predecessor counters. The total number of predecessors for all states that map to a particular counter is calculated and stored in that counter during the pre-processing stage. During actual model checking, every time a state is reached, its corresponding counter is decreased by one and a state can be safely discarded only when its corresponding counter becomes zero.

This technique can be seen as a very coarse approximation of the PRS approach, which associates a predecessor counter with each state and thus provides *exact* information on state predecessors. With our approach states are discarded as soon as they become pseudo-root, whereas the technique described by Miller and Katz [MK96] requires that a set of states, all mapped to the same counter, become pseudo-root in order to safely discard them. An additional advantage of the PRS-based state space exploration is that predecessor counters are allocated and discarded dynamically with their corresponding states, instead of being statically allocated.

Chapter 6

Exploiting Partial Orders

Overview

In this chapter, the pseudo root states approach presented in Chapter 5 is further developed into a yet more space and time efficient reachability analysis algorithm. This is achieved in two steps. Firstly, we show how the sleep set partial order technique can be applied to CSP refinement checking. Secondly, we uncover properties of the sleep set enhanced reachability analysis algorithm that allow pseudo root state identification to be performed concurrently with state space exploration. The lower space and time requirements of the resulting reachability algorithm are explained by its ability to identify certain states as being pseudo-root earlier in the search compared to the original pseudo-root state approach. Finally, we present experimental results that empirically confirm our performance expectations.

6.1 Motivation

Encouraged by the good performance of the pseudo-root state method described in Chapter 5, we set ourselves to find out if any further improvements to this technique are possible. As the main benefit of the PRS-enhanced reachability analysis is the saving in the number of states that need to be stored during state space exploration, of most interest to our research are methods that hold the promise of further reducing the space requirements at the expense of an acceptable time overhead. This would make our technique even more beneficial due to an improved space/time trade-off.

What are the avenues open to achieve such an improvement? One possibility (already discussed in Section 5.5.2) is that the method for selecting a state from

Pending can have an effect on the space saving offered by the PRS-enhanced state space exploration. However, after further analysis it has been decided not to explore this option, for the following reasons. Firstly, any such method would constitute a heuristic rather than a systematic approach, because the global properties of an LTS, e.g. whether the bisection width of its reachability graph is greater than its depth, cannot be known before the LTS is explored. Therefore, the performance of such a heuristic would very likely be sensitive to the LTS it is applied to. Secondly, such a heuristic could only deliver a marginal improvement in space at the cost of some time overhead required to make the choice of state to be selected from *Pending*.

Another option for further improvement of the PRS-enhanced reachability analysis can be identified by analysing the way it operates. The PRS technique depends on counting the number of predecessors of a given state that are yet to be visited in the state space search. Therefore, a reduction in the number of predecessors a state σ has in the reachability graph of an LTS would likely result in σ becoming pseudo-root earlier in the search. This, in turn, can be expected to reduce the peak number of states to be stored during reachability analysis. Further, if the number of predecessors of σ can be reduced to only one, σ would become pseudo-root immediately after it is visited by the algorithm.

As discussed in Section 5.1, partial order methods [God94, Val93, Pel94] have been developed to reduce the amount of redundant work done during reachability analysis. In particular, these methods can reduce the number of times a state is visited in state space exploration which has already been utilised by Godefroid *et al.*'s enhanced state space caching technique [GHP92].

Therefore, it appears very promising to seek the combination of the benefits of the PRS and partial order methods, thus deriving a technique resulting in a better space reduction than would be possible by either one alone. Indeed, with such a combined approach it is to be expected that both fewer transitions will be traversed and also fewer states need to be stored as visited. However, there is little literature on the application of partial order methods in the context of process algebras, and virtually none in the context of CSP. In the next two sections we study these methods to establish their applicability to CSP and refinement checking.

6.2 Partial order methods

CSP, similarly to many other process algebras, uses interleaving as a model of concurrency. While this choice results in elegant and concise operational and denotational semantics, interleaving can be seen as a major contributor to the state

space explosion phenomenon limiting the size of systems that can be successfully verified. Indeed, a parallel composition of a set of processes with little interaction results in a model that has, in general, a larger number of states and transitions than a composition with tighter interaction of the same set of processes.

In addition, interleaving increases the redundancy in state space exploration, since a particular state can be reached through a larger number of paths (sequences of transitions or events) in the model. Intuitively, this redundancy could be avoided under certain conditions. For example, if a property of interest is insensitive to the way a state is reached during state space exploration, it may be unnecessary to explore all valid paths in the reachability graph in order to verify that property.

Partial order methods [God94, Val93, Pel94] are a collection of techniques aimed at reducing the burden of interleaving on state space exploration and verification. The main idea behind these methods is to treat concurrent executions as partial orders imposed on the transitions involved in these executions. Expanding the partial orders into a corresponding set of interleavings can then be avoided when the order of execution is irrelevant to the property of interest—it may suffice to examine just a small subset (sometimes even one) of the many interleavings in order to check that property. Compared to reachability analysis techniques that exhaustively explore every state and every transition in a finite model, the use of partial order methods may bring savings in terms of both states and transitions that need to be visited while still guaranteeing the correctness of the outcome of verification.

Partial order methods rely on the notion of *transition dependence* to identify the interleavings that can be represented by partial orders. Informally, two transitions are independent when neither can influence (that is, enable or disable) the occurrence of the other, and dependent otherwise. More formally, event dependence in the context of LTS can be captured by a *dependence relation* as follows [God94]:

Definition 12 Let $\mathcal{L} = (\mathcal{S}, \mathcal{E}, \mathcal{R}, \mathcal{I})$ be an LTS and $\mathcal{D} \subseteq \mathcal{E} \times \mathcal{E}$ be a binary, reflexive, and symmetric relation. $\mathcal{D}$ is a valid *dependency relation* for $\mathcal{L}$ if and only if for all independent pairs of events $a, b \in \mathcal{E} : (a, b) \notin \mathcal{D}$ the following two properties are implied ($\lambda, \lambda', \lambda'', \gamma \in \mathcal{S}$):

$$\lambda \xrightarrow{a} \lambda' \in \mathcal{R} \Rightarrow (\exists \lambda'' \in \mathcal{S} : \lambda \xrightarrow{b} \lambda'' \in \mathcal{R} \Leftrightarrow \exists \gamma \in \mathcal{S} : \lambda' \xrightarrow{b} \gamma \in \mathcal{R}) \tag{6.1}$$

$$\lambda \xrightarrow{a} \lambda' \in \mathcal{R} \wedge \lambda \xrightarrow{b} \lambda'' \in \mathcal{R} \Rightarrow \exists \gamma : \lambda \xrightarrow{ab} \gamma \in \mathcal{R} \wedge \lambda \xrightarrow{ba} \gamma \in \mathcal{R} \tag{6.2}$$

In the above definition, Equation (6.1) expresses the previously mentioned property that independent transitions can neither enable nor disable each other.

In addition to that, independent transitions are commutative as reflected in Equation (6.2)—in other words, when a and b are both enabled in a particular state, there exists a unique state that is reachable by executing these events in either order.

A valid dependence relation $\mathcal{D}$ can be used to define an equivalence relation on sequences of transitions as proposed by Mazurkiewicz [Maz86, God94]: two sequences of transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions. Naturally, this equivalence relation allows sequences of transitions to be grouped into equivalence classes called Mazurkiewicz's traces. Mazurkiewcz's trace semantics, which characterise the behaviour of a concurrent system by its set of Mazurkiewicz's traces, is often referred to as a *partial order semantics* because it is possible to define a correspondence between Mazurkiewicz's traces and partial orders of occurrences of transitions [Maz86, God94].

Partial order reachability analysis methods owe their name to the underpinning role of Mazurkiewicz's work [God94]. Of greatest importance to the foundations of partial order techniques is the following theorem (formulated and proven by Godefroid [God94]):

Theorem 4 If two equivalent sequences of transitions in the reachability graph of an LTS originate from the same state, then they end in the same state.

Essentially, the above theorem formally states two properties of the reachability graph of a concurrent system informally discussed previously in this section:

- A state may be reachable through more than one sequence of transitions due to interleaving; and
- All interleavings of independent transitions (that is, sequences of transitions from the same equivalence class) starting from a particular state (e.g. an initial state) end in the same state.

At first, Definition 12 appears to bring little practical value, because a brute-force computation of $\mathcal{D}$ would be at least as expensive as the traversal of the corresponding LTS. Fortunately, there are purely syntactic techniques that can be shown to be sufficient to guarantee transition (and event) independence [God94, Pel94]. For example, Godefroid [God94] has defined a set of sufficient syntactic conditions for transition independence in a LFCS.

Since the dependence relation is central to all partial order methods, to enable their use in the context of CSP one has to develop a similar syntax-based approach for computing $\mathcal{D}$ for CSP process terms. We present an algorithm for this in Section 6.5.

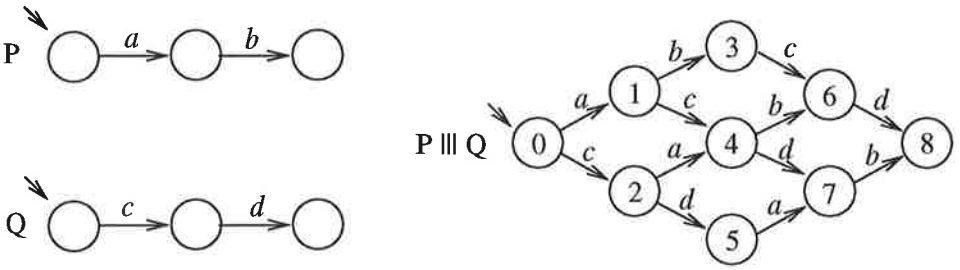


Figure 6.1: Interleaved processes example revisited

6.3 Sleep sets method

The sleep set (SS) reachability analysis technique is a partial order method developed by Godefroid [God90, GHP92] to avoid the wasteful exploration of all possible interleavings of independent transitions while providing a guarantee that each reachable state would be visited at least once in the process of state space exploration. Therefore, sleep sets provide means to reduce the number of transitions that are followed. This brings a reduction in time complexity of reachability analysis.

Despite not providing any savings in terms of visited states, the SS-enhanced state space exploration has a considerable advantage over those methods which do provide such a saving (e.g. Godefroid's persistent set method [God94])—it is independent of the property being checked and thus can be applied to any state-based verification algorithm including refinement checking for CSP. In fact, the application of the SS-enhanced technique requires only the availability of $\mathcal{D}$ and some bookkeeping during the reachability analysis.

The intuition behind the notion of sleep sets is best illustrated by the interleaved process example previously used in Section 5.2.2. For convenience, Figure 5.1 is reproduced in Figure 6.1. Since the two processes P and Q are combined by interleaving, the pairs of events $(a, c), (a, d), (b, c)$, and (b, d) are the independent transitions in this example; all other pairs are dependent transitions.

Let us assume that the exploration of the compound LTS from Figure 6.1 is performed in a depth-first manner, i.e. by Algorithm 2 presented in Section 2.2.1. Reachability analysis starts from state 0. Suppose that the transition labelled with event a leading to state 1 is selected first. Then, eventually, all states reachable from state 1 are explored, after which the DFS algorithm backtracks to state 0 and follows the transition labelled with c to reach state 2. This state has an outgoing transition labelled with a ending in state 4.

At this point, classic DFS state space exploration (Algorithm 2) would follow

that transition, identify state 4 as being already visited, and backtrack to state 2. On the other hand, a and c are independent events and both are enabled in state 0. Therefore, by Definition 12, it is known that traces $\langle a, c\rangle$ and $\langle c, a\rangle$ lead to the same state. This means that, in our scenario, a state space exploration utilising this knowledge may avoid following the transition from state 2 to state 4. The same reasoning can be applied to the transition from state 5 to state 7, because event d leading to state 5 cannot disable the occurrence of the independent event a.

The essence of Godefroid's approach of avoiding unnecessary transition traversal is to associate a set of transition labels (events) called a *sleep set* with every reachable state [God94]. The events in the sleep set of a state are enabled in that state but an outgoing transition labelled with an event from the sleep set is not traversed during reachability analysis. The sleep sets of the initial states are empty. The sleep sets of all other reachable states are computed the first time they are reached by the SS-based reachability analysis algorithm on the basis of the following two rules:

1. If a state $\gamma \in \mathcal{S}$ has the sleep set $ss(\gamma)$ and a transition $\gamma \xrightarrow{a} \lambda$ is followed, then $ss(\lambda)$ contains the events in $ss(\gamma)$ that are independent from a since a can neither enable nor disable them in λ;

2. If a state $\gamma \in \mathcal{S}$ has the sleep set $ss(\gamma)$ and a transition $\gamma \xrightarrow{a} \lambda$ is followed after a few other transitions out of γ labelled with events from a set $ft(\gamma)$ were followed, then $ss(\lambda)$ contains the events in $ft(\gamma)$ that are independent from a. The rationale for this requires some elaboration. Let $b \in ft(\gamma)$ and $(a, b) \notin \mathcal{D}$. Since b is explored before a, all states reachable through the trace $\langle b\rangle\hat{}s\hat{}\langle a\rangle$ have been already reached by the time λ and its successors are traversed. Since a and b are independent, $\langle a\rangle\hat{}s\hat{}\langle b\rangle$ will, by definition, lead to those same reached states, therefore, $b \in ss(\lambda)$.

The above two rules can be easily integrated into a DFS state space exploration procedure such as Algorithm 2. The resulting SS-enhanced reachability analysis algorithm is presented as Algorithm 11. The stack of the original DFS algorithm is modified to contain pairs of states and sleep sets. The recursive procedure DFS-SS maintains an event set *FollowedTrans* that accumulates the explored transition labels at each state in the stack. *FollowedTrans* is then utilised in the application of Rule 2 for the computation of the sleep set of the newly reached states.

Theorem 5 Algorithm 11 explores the same set of reachable states as Algorithm 2.

Algorithm 11 SS-enhanced reachability analysis algorithm

Require: An LTS $\mathcal{L}$
Ensure: *Reached* contains the set of all reachable states in $\mathcal{L}$

```
procedure DFS-SS
Pop (γ, CurrentSleep) from Stack
Reached ← Reached ∪ {γ}
FollowedTrans ← ∅
for all a ∈ {b | γ -b-> λ ∈ R ∧ b ∉ CurrentSleep} do
  FollowedTrans ← FollowedTrans ∪ {a}
  for all λ ∈ {θ | γ -a-> θ ∈ R} do
    if λ ∉ Stack ∪ Reached then
      {Compute the sleep set for λ—first apply Rule 1, then Rule 2}
      NewSleep ← {b | b ∈ CurrentSleep ∧ (a, b) ∉ D}
      NewSleep ← NewSleep ∪ {b | b ∈ FollowedTrans ∧ (a, b) ∉ D}
      Push (λ, NewSleep) onto Stack
      call DFS-SS
    end if
  end for
end for
end procedure
{Start of the Algorithm}
Push {(γ, ∅) | γ ∈ I} onto Stack in any sequence
Reached ← ∅
call DFS-SS
```

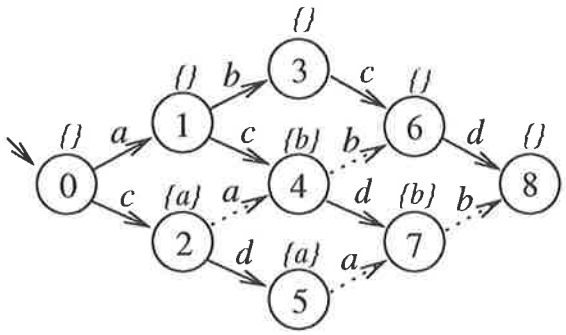


Figure 6.2: SS-enhanced exploration of the interleaved processes example

Proof: Can be found elsewhere [GHP92]. ■

The operation of Algorithm 11 on the example from Figure 6.1 is shown in Figure 6.2. Each state is annotated with its sleep set. It is assumed that the algorithm first explores the sequence of states along the trace $\langle abcd \rangle$ and then backtracks to state 1. At that stage, the transition labelled with c into state 4 is followed and, since the set *FollowedTrans* of state 1 contains the event b which is independent of c, the sleep set of state 4 is $\{b\}$. This prevents the transition ending in state 6 from being explored, and, therefore, only the transition from state 4 to state 7 labelled with d is followed. Again, b and d are independent events, thus the sleep set of state 7 is $\{b\}$. As there are no non-sleep transitions out of state 7, Algorithm 11 backtracks to state 1. Applying a similar line of reasoning as for state 4, the sleep sets of states 2 and 5 are computed to be $\{a\}$.

For this simple example Algorithm 11 traverses only 8 of the 12 transitions in the LTS, whereas classic DFS or BFS has to explore all transitions. The cost of achieving this reduction is the necessity to maintain the sleep sets and followed transitions for all states in *Stack*. Therefore, the time trade-off offered by the SS-enhanced reachability analysis algorithm depends on the relative overhead of computing the outgoing transitions of a state and the on-going computation and storage of sleep sets. Our experience with SS-enhanced reachability analysis in the context of the ARC tool is discussed in Section 6.6.1.

6.4 Combining PRS and sleep sets

An inherent property of the SS-enhanced reachability analysis is that it reduces the number of actual state predecessors because of the fact that some transitions in the LTS are not explored. This is the key property that enables sleep sets to perform very well in the enhanced state space caching method presented by

Godefroid *et al.* [GHP92]. Indeed, the reduction in the number of state predecessors directly lowers the probability of a state to be reached more than once during reachability analysis, which in turn greatly enhances the effectiveness of state caching. However, this probability is still non-zero and, therefore, Godefroid *et al.*'s refined state space caching technique [GHP92] still has to explore some parts of the state space more than once.

In Chapter 5 we have presented a novel state space exploration method utilising so-called pseudo-root states (PRS) that achieves a space reduction for the storage of reached states while guaranteeing that all reachable states are explored exactly once. An examination of the LTS in Figure 6.2 reveals that every state in that LTS has at most one predecessor when only the transitions followed by Algorithm 11 are considered, but most states in the LTS have two predecessors if all transitions are considered. Such a reduction in the number of predecessors would have resulted in every state becoming PRS as soon as it is reached.

Unfortunately, a naive combination of the original PRS-enhanced reachability analysis algorithm (Algorithm 10) with an SS-enhanced state space exploration algorithm such as Algorithm 11 would not work well because of the mismatch between the number of predecessors that Algorithm 10 would compute and the actual number of predecessors from which each state would be reached because of the use of sleep sets. This mismatch would result in many reachable states never becoming PRS and, hence, the benefits of the PRS-enhanced reachability analysis may be considerably diminished.

The key to enabling the sleep set and pseudo-root state techniques to work together is to modify the approach to the computation of the number of predecessor states (line 28 of Algorithm 10). This modification has to account for the *sleep* transitions—those that are present in the LTS but not explored by the SS-enhanced state space traversal technique (the dashed line transitions on Figure 6.2)—and adjust the number of predecessors paired with each state accordingly. In the rest of this section we present a slightly refined version of a technique first described by Yantchev and Parashkevov [YP97] that identifies incoming sleep transitions by computing and maintaining so-called *insleep* sets in parallel with the sleep sets. When the predecessors of a state are to be computed, the insleep sets are taken into account to exclude the incoming sleep transitions.

Our proposed technique for computing and maintaining insleep sets as an extension of Algorithm 11 is based on two key properties of that algorithm. The first property is that a sleep transition only occurs when there is a reachable state with a pair of independent outgoing transitions. It is captured formally in the following theorem:

Theorem 6 Let Algorithm 11 traverse two consecutive independent transitions

$\lambda \xrightarrow{a} \lambda', \lambda' \xrightarrow{b} \gamma \in \mathcal{R}$ labelled, respectively, with $a, b \in \mathcal{E} : (a, b) \notin \mathcal{D}$. Then, there is an incoming sleep transition into γ labelled with a.

Proof: Since independent transitions can neither enable nor disable each other (Definition 12, Equation 6.1) and b is enabled in λ', b is enabled in λ. Therefore, there exists $\lambda'' \in \mathcal{S}$ such that $\lambda \xrightarrow{b} \lambda'' \in \mathcal{R}$. Then, according to Equation 6.2, there is also a transition $\lambda'' \xrightarrow{a} \gamma \in \mathcal{R}$. Since Algorithm 11 has traversed the transition $\lambda' \xrightarrow{b} \gamma$, then b is not in the sleep set of state λ' which, in turn, means that the transition $\lambda \xrightarrow{b} \lambda''$ will be followed by the algorithm after transition $\lambda \xrightarrow{a} \lambda'$. Then, when $\lambda \xrightarrow{b} \lambda''$ is traversed, a will be in the sleep set of λ'' which makes $\lambda'' \xrightarrow{a} \gamma$ a sleep transition. This proves the theorem. ■

The above theorem provides insight on how events are to be included in the insleep sets during state space traversal as implemented by Algorithm 11. An additional property of the algorithm, captured in the following theorem, addresses the issue of propagating events in insleep sets between consecutive states in state space traversal.

Theorem 7 Let state $\gamma \in \mathcal{S}$ be a state reached by Algorithm 11, and the insleep set of that state be $A \subseteq \mathcal{E}$. If the algorithm follows a transition $\gamma \xrightarrow{a} \lambda \in \mathcal{R}$, then the insleep set of λ shall contain the set $\{b \mid b \in A \wedge (a, b) \notin \mathcal{D}\}$.

Proof: Let $b \in A$ and $(a, b) \notin \mathcal{D}$. Since there is an incoming sleep transition labelled with b in state γ, there must be a reachable state θ such that $\theta \xrightarrow{b} \gamma$. Since independent transitions can neither enable nor disable each other (Definition 12, Equation 6.1) and a is enabled in γ, there must be a state $\theta' \in \mathcal{S}$ such that $\theta \xrightarrow{a} \theta' \in \mathcal{R}$. Further, from Equation 6.2 we derive that there exists a transition $\theta' \xrightarrow{b} \lambda \in \mathcal{R}$. Since $\theta \xrightarrow{b} \gamma$ is a sleep transition, the sleep set of θ includes b. Then, since $(a, b) \notin \mathcal{D}$, the sleep set of θ' will also include b, which makes $\theta' \xrightarrow{b} \lambda$ a sleep transition, too. This proves that b is in the insleep set of λ. ■

Theorems 6 and 7 underpin a new reachability analysis algorithm combining sleep sets and pseudo-root states. We refer to it as Algorithm 12. It can be seen as an extension of Algorithm 11. Each reachable state is annotated with two sets of events—sleep and insleep sets, the event through which it has been initially reached, and the number of predecessors through which it has to be reached to become pseudo-root. Every time a transition $\gamma \xrightarrow{a} \lambda$ is explored and λ is a new reachable state, its insleep set is computed as:

$$Insleep(\lambda) = \{b \mid b \in Insleep(\gamma) \cup \{c\} \wedge (a, b) \notin \mathcal{D}\}$$

Algorithm 12 PRS- and SS-enhanced reachability analysis algorithm

Require: An LTS $\mathcal{L}$

Ensure: *Check* is called on each reachable state in $\mathcal{L}$ exactly once

procedure DFS-SS-PRS
Pop $(\gamma, SleepSet(\gamma), InsleepSet(\gamma), Event(\gamma), Counter(\gamma))$ from *Stack*
$Check(\gamma)$
if $Counter(\gamma) > 0$ **then**
{γ is not pseudo-root, include it in *Checked*}
$Reached \leftarrow Reached \cup \{\gamma\}$
end if
$FollowedTrans \leftarrow \emptyset$
for all $a \in \{b \mid \gamma \xrightarrow{b} \lambda \in \mathcal{R} \wedge b \notin SleepSet\}$ **do**
$FollowedTrans \leftarrow FollowedTrans \cup \{a\}$
for all $\lambda \in \{\theta \mid \gamma \xrightarrow{a} \theta \in \mathcal{R}\}$ **do**
if $\lambda \in Reached$ **then**
if $Counter(\lambda) = 1$ **then**
{λ becomes PRS, discard it}
$Reached \leftarrow Reached - \{\lambda\}$
else
$Counter(\lambda) \leftarrow Counter(\lambda) - 1$
end if
else if $\lambda \in Stack$ **then**
$Counter(\lambda) \leftarrow Counter(\lambda) - 1$
else
{λ is an unreached state}
{Compute the sleep set for λ and store it in *NewSleep*}
{Compute the insleep set for λ and store it in *NewInSleep*}
{Compute the PRS counter for λ and store it in *NewCounter*}
Push $(\lambda, NewSleep, NewInsleep, \{a\}, NewCounter)$ onto *Stack*
call DFS-SS
end if
end for
end for
end procedure
{Start of the Algorithm}
Push $\{(\gamma, \emptyset, \emptyset, \emptyset, \mid BackImage(\{\gamma\}, \mathcal{L}) \mid) \mid \gamma \in \mathcal{I}\}$ onto *Stack*
$Reached \leftarrow \emptyset$
call DFS-SS

where c is the event through which γ was first reached. The insleep sets are used to account for incoming sleep transitions each time the number of predecessors for a newly reached state is computed, which enables the operation of the pseudo-root state deletion technique.

6.5 Computing the dependence relation

As mentioned before, a necessary condition for the application of any partial order technique is the ability to compute the dependence relation $\mathcal{D}$. We have not been able to track any background art discussing this computation in the context of a process algebra like CSP. Fortunately, Godefroid's thesis [God94] provides a generic rule for transition (and event) independence in the context of concurrent systems: a sufficient syntactic condition for independence between two transitions a and b is that the set of processes that can perform a is disjoint from the set of processes that can perform b.

How can one apply Godefroid's generic rule to the CSP algebra? Our choice is to operate at the level of leaf processes $\{P_i\}_{i=1}^{n}$ in the synchronisation hierarchy of a process P (as introduced in Section 4.3). According to the rule, any two events that can be performed by a leaf process are dependent. Therefore, the computation of $\mathcal{D}(P)$ iterates over all P_i, determines $\alpha(P_i)$, and updates $\mathcal{D}$ to include each $(a, b) : a, b \in \alpha(P_i)$. This is captured by Algorithm 13.

Algorithm 13 Computing the dependence relation of a CSP process

Require: A CSP process P
Ensure: $\mathcal{D}$ is a valid dependence relation for P over $\mathcal{E}_P \times \mathcal{E}_P$
  $\mathcal{D} \leftarrow \emptyset$
  {Determine the leaf processes $\{P_i\}_{i=1}^{n}$ in the synchronisation hierarchy of P}
  for $Q \in \{P_i\}_{i=1}^{n}$ **do**
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(a, b) \mid a, b \in \alpha(Q)\}$
  end for

The main motivation for using this approach is its relative simplicity, especially in terms of implementation in the ARC tool. However, in some cases, the proposed technique for computing $\mathcal{D}$ can be overly pessimistic, that is, it may rule two events dependent when they are actually independent. Consider, for example, $P = a \rightarrow (Q \underset{C}{\|} R)$, where Q and R are purely sequential processes. Since P itself is a leaf process, any pair of events is computed as being dependent. In practice, two events from $\alpha(P)$ are dependent only if both are in $\alpha(Q)$ or both are in $\alpha(R)$.

Despite the potentially pessimistic dependence relation computed by Algorithm 13, it performs very well for virtually all practical examples we have come across, because most leaf processes are in practice purely sequential. This certainly holds for all examples in Appendix A.

6.6 Experimental results

Both reachability analysis algorithms presented in this chapter have been implemented in the ARC tool. We refer to our implementation of the SS-enhanced reachability analysis method (Algorithm 11) as ARC/PP+SS, and to the implementation of the combined sleep set and PRS-based state space exploration Algorithm 12) as ARC/PP+PRS+SS. In this section, we present experimental results obtained using these tools on examples from Appendix A.

6.6.1 Performance of ARC/PP+SS

The performance of ARC/PP+SS is compared to that of ARC/PP to establish whether the sleep set technique brings about time savings over the basic pair-by-pair refinement checking algorithm from Section 3.3.2. Table 6.1 presents the results for variant one of Milner's Schedulers example and Table 6.2 the data for Dining Philosophers. A number of observations can be made on the basis of the information in those tables:

- For transition relations of small size, there is no observable time penalty due to the overhead of using sleep sets in state space exploration;
- The savings offered by ARC/PP+SS over ARC/PP are both in terms of followed transitions and run-time, and grow with the size of reachable state space of the example;
- For the largest transition relations for the two examples, a time saving of a factor of three and a reduction in the number of traversed transitions of five to six times is achieved by using the SS-enhanced reachability analysis.

In contrast with the above results, the use of the sleep set technique does not improve performance when applied to the Alternating Bit Protocol example, since the same number of transitions is followed by both ARC/PP and ARC/PP+SS. This can be seen as a worst-case scenario for the use of sleep sets because none of the reachable states contain two independent outgoing transitions. Again, ARC/PP+SS does not appear to introduce measurable time overhead over ARC/PP.

N	Reachable states	ARC/PP		ARC/PP+SS	
		transitions	run-time, sec	transitions	run-time, sec
3	25	49	0.3	27	0.3
5	161	481	0.9	164	0.7
7	897	3585	3.9	906	2.6
9	4609	23041	27.4	4635	13.9
11	22529	135169	213.6	22548	77.0

Table 6.1: Experimental results of ARC/PP and ARC/PP+SS on variant one of Milner's Schedulers example

N	Reachable states	ARC/PP		ARC/PP+SS	
		transitions	run-time, sec	transitions	run-time, sec
3	154	411	0.5	209	0.5
4	832	2964	2.2	1122	1.6
5	4474	19925	16.7	6454	9.1
6	24040	128478	146.3	27901	47.0

Table 6.2: Experimental results of ARC/PP and ARC/PP+SS on Dining Philosophers example

6.6.2 Performance of ARC/PP+PRS+SS

In order to experimentally check our expectation that Algorithm 12 requires a lower peak number of states to be stored during the search than the original PRS-enhanced reachability analysis technique (Algorithm 10), the performance of ARC/PP+PRS+SS is benchmarked against ARC/PP+PRS. Table 6.3 presents the results for variant one of Milner's Schedulers. Quite unexpectedly, the peak number of stored states for ARC/PP+PRS+SS grows linearly with the numbers of schedulers. This is the best improvement achieved on any example run with ARC/PP+PRS+SS by the author.

Algorithm 12 provides less of an improvement for the Dining Philosophers example (Table 6.4) by reducing the peak number of states required by three to four fold compared to ARC/PP+PRS.

N	Reachable states	Peak states	
		ARC/PP+PRS	ARC/PP+PRS+SS
3	154	12	14
5	832	51	25
7	4474	206	42
9	24040	814	59
11	22529	3271	71

Table 6.3: Peak stored states for variant one of Milner's Schedulers

N	Reachable states	Peak states	
		ARC/PP+PRS	ARC/PP+PRS+SS
3	154	69	46
4	832	380	168
5	4474	1963	592
6	24040	10171	3378

Table 6.4: Peak stored states for the Dining Philosophers example

Chapter 7

The ARC Tool

Overview

This chapter contains a brief overview of the ARC tool—its design and architecture details, user interface and command-line options. A concise user guide to applying ARC in practice is provided along with some examples.

7.1 Introduction

The practical usefulness of an algorithm or collection of algorithms cannot always be reliably predicted by theoretical complexity analysis alone. This is undoubtedly the case for the majority of state-based formal verification algorithms that often have a worst case computational complexity that is at least exponential to the size of the modelled system. However, practical implementations of these algorithms rarely follow the theoretical predictions when applied to real-world problems. Also, practical implementations of two algorithms that have the same worst case complexity in theory may exhibit vastly different run-times when applied to the same problem. Therefore, the development of new state-based formal verification algorithms should always be augmented with an experimental implementation to gather empirical data on the performance of those algorithms and enable their analysis and further optimisation.

The ARC prototype tool, containing experimental implementations of most of the algorithms discussed in this thesis, has been developed in the course of this work. In the author's experience, the practical experimentation with ARC proved to be an invaluable learning tool on its own. It has demonstrated that an algorithm that appears simple and elegant on paper may not be as efficient in practice as another that is much more complex. Experimentation has also made

us appreciate the unpredictable behaviour of OBDDs used for representing sets and relations.

This chapter provides an overview and a short user guide for the ARC tool. Throughout the thesis, ARC has been used in conjunction with a set of selected benchmark examples described in Appendix A. These examples are useful both for testing ARC's algorithm implementation and for performance comparison with other similar tools. The actual performance data on ARC with these examples is provided in the chapters discussing the particular algorithms.

7.2 Architecture

The high-level internal architecture of the ARC tool is shown in Figure 7.1. ARC employs a modular software architecture obtained by structured analysis and design of the verification problem it is solving. There are two major separate steps in the data flow inside the tool: CSP compilation and encoding followed by state space exploration and model checking.

Luckily, existing software artifacts could be reused in ARC—two of its five modules shown in Figure 7.1 use publicly available software that is utilised with little or no modification. This allowed us to concentrate on the more interesting and relevant to our research tasks (compilation and verification). Consequently, the first working prototype of ARC with rudimentary CSP and basic pair-by-pair refinement checking support has been built within a couple of months of initial design and implementation work.

The CSP parser module reuses Scattergood's public domain parser [Sca92]. It has been slightly customised by adding a second pass in the parsing process that rewrites the original parse tree in a more convenient format while populating the symbol table. The second pass is also responsible for the parsing of `pragma` statements that are not processed by the first pass.

The symbol table module is responsible for storage and access to all symbol related information. Each input symbol (or, as it is often called, atom) falls into one of the following categories:

- Process definition identifier. For these symbols, a reference to the syntax tree node of the definition is stored. The compilation of process references relies on this information;

- Constant and function identifiers. Again, a reference to the syntax tree node of the constant or function definition is stored and later utilised in expression evaluation during compilation;

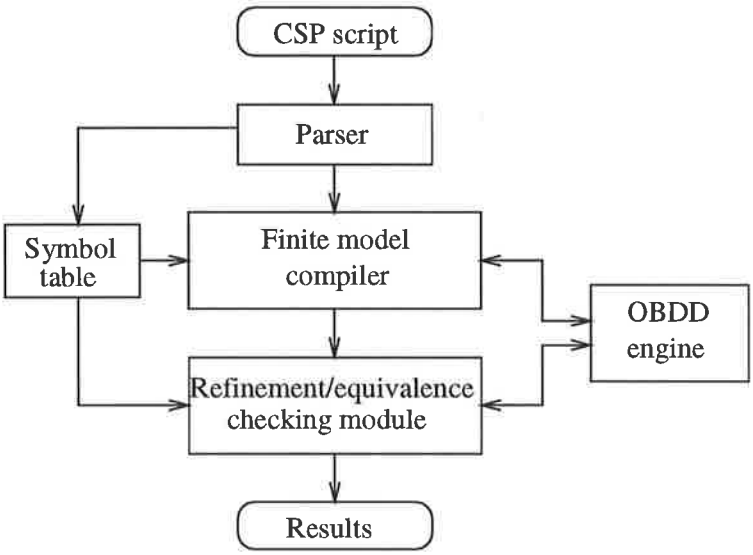


Figure 7.1: The architecture of the ARC tool

- Channel name identifier. For these symbols, the type of the channel is stored. This information is later referred to during the compilation of channel input/output operators;
- Predefined identifiers. These include the internal symbols for the CSP operators, sets and set operations, arithmetic and boolean operators, etc.;
- Other symbols, e.g. identifiers. No specific information is stored in the symbol table.

The finite model compiler module is responsible for the semantic check of the syntax tree and the generation of the OBDD encoding of the LTS representation of the specification and implementation processes in preparation for the subsequent refinement check. The syntax tree of the process definitions is explored via recursive descent to build the final LTS in OBDD form in accordance with the CSP compilation rules presented in Section 3.2.5.

David Long's public domain OBDD library [Lon93] is reused "as is" in the OBDD engine module. It is one of the first OBDD libraries to become publicly available; much more powerful and complete implementations exist nowadays (for example, the CUDD library [Som98][1]). An additional software layer is added on

[1]A comparison of OBDD packages (including Long's library and CUDD) in the context of model checking can be found elsewhere [YBO+98].

top of the OBDD library to provide functions for encoding sets and relations and exploring nondeterministic finite state automata encoded as OBDDs.

The verification module performs one of the refinement and equivalence checking algorithms developed in this thesis as selected by the user using ARC's command-line options. Since verification may take a long time, the user is continuously informed of the progress of the search (the number of unique CSP state pairs explored). When a check fails, debugging information is provided to the user. This information includes the trace that leads to the error and a description of the error—for example, a trace error would provide a list of next events for the specification and implementation processes.

The ARC tool is implemented in approximately 6,800 lines of C code (including comments but excluding the parser and OBDD engine modules). Its execution is highly customisable through the use of command-line options discussed in Section 7.4. Section 9.2 covers various existing and future extensions of the ARC tool.

7.3 Input language

As the main emphasis of this research is on OBDD encoding and verification algorithms implemented in the ARC tool, the finite model compiler in ARC has been implemented to handle only the core CSP language features[2]. This includes the subset defined in (2.1) with the necessary extensions to handle expressions and channel declarations. Thus, even though the input syntax is compatible with that of the FDR1 tool because of our choice to build on Scattergood's public domain CSP parser, there are several CSP script features not supported by ARC. These include functional extensions, sequences, replicated operators, and data types.

It should be noted that, although ARC does not support the same rich input language as the commercial FDR1 and FDR2 tools do, the CSP subset that ARC supports is sufficient to define any process that the richer language can, although it may require a greater effort on the user's part. This is demonstrated in practice by the wide range of examples using the ARC subset of CSP presented in Appendix A.

Despite our intent to keep ARC's syntax as close to that of FDR1 as possible, several changes have been introduced:

- The specification and implementation processes for the refinement check need to be declared in separate `pragma` constructs specific to ARC:

[2]However, there are extensions to the ARC tool as described here, which handle much larger subset of the language. See Section 9.2 for more details.

```
pragma spec COPY
pragma impl SYSTEM
```

- Channel type definitions have to be separated into a single definition, and each sub-range (element of the dot construct) has to be a contiguous sequence of integers specified using the '..' operator. An example of legal channel declarations in ARC is as follows:

```
XMIN = 1
XMAX = 3
YMIN = -1
YMAX = 1
Type = {XMIN..XMAX}.{YMIN..YMAX}
pragma channel left,right: Type
pragma channel done
```

  The new style of channel declarations introduced in FDR2 that makes the keyword `pragma` obsolete is not supported in ARC.

- Identifiers in process definitions have to be capitalised:

```
count(X) = if (X < 0) then STOP else a -> count(X-1)
```

- To simplify the building of complex concurrent systems such as those modeling the behaviour and the state space of various puzzles, a new parallel operator notation is introduced. This operator forces its left and right operand processes to synchronise on all common events, similar to Hoare's parallel operator working with implicit process alphabets [Hoa85]. The implementation of this operator computes the alphabets of its operands as the set of events in which they can participate. In order to avoid changes to the CSP parser an existing non-process operator (.) has been reused for the new notation, as illustrated in the following example:

```
impl = (full(1,1,BWBWBW).full(1,2,BBBWWW).full(1,3,BWBWWB).
       full(2,1,WWBWBB).empty(2,2).full(2,3,WWBWBB).
       full(3,1,WBBWBW).full(3,2,BBBWWW)) \ hideset
```

7.4 Command-line options

ARC supports a number of command-line options which can be seen by running the tool with no options supplied—Figure 7.2 contains the help screen output by ARC. These options can be split into several categories:

```
ARC - a CSP refinement/equivalence checker v1.2
Copyright 1995-99 AP, UofA, All Rights Reserved
-----------------------------------------------
USAGE:  arc [options] [input-file]
OPTIONS:
        -bdd<n> Bdd node limit (default: 1000000 nodes)
        -cf     Check failures
        -cin    Read from standard input instead of disk file
        -cpu<n> Change the ratio speed to memory usage (default: 2; range: 0-3)
        -ct     Check traces (default: failures-divergencies)
        -def    Define symbols (example: -def N 10)
        -dfs    Use DFS strategy (default: use BFS strategy)
        -eq     Equivalence check (default: refinement check)
        -fa     Use full abstraction check (default: don't use it)
        -hptr   Use HPTR for LTS (default: use a monolithic LTS)
        -id     Use strict state identity function (default: relaxed function)
        -intf   Check process interfaces before the verification phase
        -ord    Use the BDD vector order E<S (default: S<E)
        -prs    Apply PRS reachability analysis algorithm
        -qfe    Quit after the first encountered error
        -sb     Use sequential component re-encoding
        -ss     Use the sleep set algorithm (forces DFS strategy)
        -v<n>   Set verbosity level to <n> (default: 2; range: 0-4)
```

Figure 7.2: ARC's help screen

- Options controlling the type of check;
- Options relating to the finite model compiler module;
- Options controlling the OBDD engine module;
- Options relating to the refinement/equivalence checking module;
- General options

In the following sections, we present a brief description of each run-time option by category.

7.4.1 Verification mode options

`-ct` Makes the check in the traces model ($\mathcal{T}$). By default, a check in the failures-divergences model ($\mathcal{N}$) is performed.

`-cf` Makes the check in the failures model ($\mathcal{F}$). By default, a check in the failures-divergences model ($\mathcal{N}$) is performed.

`-eq` Checks the specification and implementation processes for equivalence in the chosen semantic model. By default, the tool checks if the refinement relation between the specification and implementation processes holds in the chosen semantic model.

7.4.2 Compilation options

`-def` Provides a definition of integer constants at the command-line. This makes possible running ARC in batch mode on verification problems of varying sizes. For example, the option "`-def N 10`" shall set `N` to be the integer constant 10 within the CSP script being read in the run.

`-id` Enforces a more strict definition of the identity relation on the state space of a sequential process. The end result is a larger OBDD for the LTS representation of the process, but also fewer unreachable states in the final LTS. This is useful when the PRS reachability analysis is used, but slows down the check otherwise. Thus, a relaxed definition of the identity relation is used by default.

`-sb` Enables the re-encoding of sequential components as per Section 3.2.6. This option is often useful for processes composed of many sequential components with complex descriptions. Re-encoding is disabled by default.

`-hptr` Directs the tool to use an HPTR instead of a monolithic transition relation as described in Chapter 4. This option should be used only if the monolithic LTSs obtained are very large or cannot be computed. HPTR compilation mode is disabled by default.

7.4.3 OBDD related options

`-cpu<n>` Controls the trade-off between memory consumption and speed. Lower values from the allowed range $0 \leqslant n \leqslant 4$ favour smaller memory footprint at the expense of a slower execution, whereas larger values trade faster execution for increased memory requirements. The default value is $n = 3$.

`-bdd<n>` Set the OBDD node limit to n nodes. The run will be aborted if that limit is exceeded and internal garbage collection in the OBDD engine cannot free any nodes. One useful application of this option is to avoid the use of virtual memory by ARC, which is known to slow it down considerably. As a rule of thumb, one should specify an OBDD node limit of $m/25$, where m is the number of bytes of physical memory available to the application. The default value is $n = 1000000$.

`-ord` Allows control over the relative order of the OBDD variables used in encoding the state space of a transition relation ($V_{\mathcal{S}}$ and $V_{\mathcal{S}'}$) and events ($V_{\mathcal{E}}$). By default, the state variables are before the event ones; if this option is specified the order is reversed. On most examples we have run, the default ordering results in better performance.

7.4.4 Verification algorithms options

`-dfs` Directs the state space exploration algorithm of the pair-by-pair algorithm to use depth-first search strategy. By default, breadth-first search is used.

`-fa` Performs the refinement checking using the full abstraction technique described in Chapter 8. This option works only for checks done in the traces model ($\mathcal{T}$).

`-ss` Directs the state space exploration algorithm of the pair-by-pair algorithm to utilise sleep sets. When specified, this option enforces a depth-first search strategy. By default, using sleep sets is disabled.

`-prs` Directs the state space exploration algorithm of the pair-by-pair algorithm to use the pseudo-root state technique from Chapter 5. This option can be combined with `-ss` for even better results. By default, the PRS technique is disabled.

`-intf` Enables a check of interfaces between the specification and implementation processes prior to verification; the run is then aborted if the implementation can potentially perform an event that the specification cannot engage in. This option is there for purely historic reasons. No interface check is done by default.

`-qfe` Directs the tool to quit after the first error has been found. By default, ARC continues the check after the first error to uncover as many other problems as possible in a single run.

7.4.5 General options

`-v<n>` Sets the verbosity level during the run of the tool to n, which is a number between 0 and 4. A higher number implies more detail is printed. Verbosity level 0 should be used when only the truthfulness of the relation being checked (e.g. a yes/no answer) is required; level 1 adds some run-time statistics. The default verbosity level (2) further includes progress updates during the check. Level 3 adds a print-out of every trace that is being checked. Finally, the highest verbosity level (4) outputs the compiled LTS of the specification and implementation processes.

`-cin` Directs ARC to read a CSP script from the standard input stream. By default, ARC expects an input file name to be specified at the command-line.

7.5 Example run sessions

This section aims to familiarise the user with running ARC and the information printed by the tool during its execution. Due to the large number of options supported by the tool, it is very difficult to cover all of them; instead, a set of few representative ARC sessions is used.

In this section, we use the Dining Philosophers example taken from Appendix A. Our first attempt is to check the example for only two philosophers in the traces model:

```
rama::/home/ata/ARC
% ./arc -ct -def n 2 -v0 examples/dinphil-var1.csp

ARC - a CSP refinement/equivalence checker v1.2
Copyright 1995-99 AP, UofA, All Rights Reserved
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

spec [T= impl is TRUE.
```

Since we are only interested in the outcome of this check, we specified the `-v0` option. Because of this, ARC prints only its banner and a statement containing the relation being checked (including the model) and its truthfulness.

Next, we want to perform the exact same check in the failures-divergences model, but require more detail on the progress of the tool (thus the use of -v1 instead of -v0):

```
rama::/home/ata/ARC
% ./arc -def n 2 -v1 examples/dinphil-var1.csp
ARC - a CSP refinement/equivalence checker v1.2
Copyright 1995-99 AP, UofA, All Rights Reserved
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Specification 'spec': 1.200e+01 states,   4 BDD vars,     77 BDD nodes
Implementation 'impl': 9.000e+02 states,  10 BDD vars,    246 BDD nodes
BDD encoding took 0.3 secs CPU time
Traces check...
...done (so far 0.4 secs CPU time)
Computing acceptances...done (so far 0.5 secs CPU time)
Computing divergences...done (so far 0.5 secs CPU time)
Failures-divergencies check...

*** FAILURES ERROR at step 11 after trace
<sitsDown.1,sitsDown.2,picksUp.2.2,picksUp.1.1>

spec refuses {{picksUp.2.1,sitsDown.1,sitsDown.2,putsDown.1.1,
putsDown.2.1,getsUp.1,getsUp.2,putsDown.1.2,putsDown.2.2,
picksUp.1.2,picksUp.2.2},{picksUp.1.1,sitsDown.1,sitsDown.2,
putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,putsDown.1.2,
putsDown.2.2,picksUp.1.2,picksUp.2.2},{picksUp.1.1,picksUp.2.1,
sitsDown.2,putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,putsDown.1.2,
putsDown.2.2,picksUp.1.2,picksUp.2.2},{picksUp.1.1,picksUp.2.1,
sitsDown.1,putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,
putsDown.1.2,putsDown.2.2,picksUp.1.2,picksUp.2.2},{picksUp.1.1,
picksUp.2.1,sitsDown.1,sitsDown.2,putsDown.2.1,getsUp.1,getsUp.2,
putsDown.1.2,putsDown.2.2,picksUp.1.2,picksUp.2.2},{picksUp.1.1,
picksUp.2.1,sitsDown.1,sitsDown.2,putsDown.1.1,getsUp.1,getsUp.2,
putsDown.1.2,putsDown.2.2,picksUp.1.2,picksUp.2.2},{picksUp.1.1,
picksUp.2.1,sitsDown.1,sitsDown.2,putsDown.1.1,putsDown.2.1,
getsUp.2,putsDown.1.2,putsDown.2.2,picksUp.1.2,picksUp.2.2},
{picksUp.1.1,picksUp.2.1,sitsDown.1,sitsDown.2,putsDown.1.1,
putsDown.2.1,getsUp.1,putsDown.1.2,putsDown.2.2,picksUp.1.2,
picksUp.2.2},{picksUp.1.1,picksUp.2.1,sitsDown.1,sitsDown.2,
putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,putsDown.2.2,
picksUp.1.2,picksUp.2.2},{picksUp.1.1,picksUp.2.1,sitsDown.1,
sitsDown.2,putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,
putsDown.1.2,picksUp.1.2,picksUp.2.2},{picksUp.1.1,picksUp.2.1,
sitsDown.1,sitsDown.2,putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,
putsDown.1.2,putsDown.2.2,picksUp.2.2},{picksUp.1.1,picksUp.2.1,
sitsDown.1,sitsDown.2,putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,
putsDown.1.2,putsDown.2.2,picksUp.1.2}}
impl refuses {{picksUp.1.1,picksUp.2.1,sitsDown.1,sitsDown.2,
```

```
putsDown.1.1,putsDown.2.1,getsUp.1,getsUp.2,putsDown.1.2,
putsDown.2.2,picksUp.1.2,picksUp.2.2}}

(so far 0.7 secs CPU time)
...done

spec [FD= impl is FALSE.
Found 0 trace error(s), 1 failures error(s) and 0 divergence error(s).
28 states and 50 transitions were explored.

*** Total 0.7 secs CPU time
```

The above contains considerably more detail than the first run. Following the ARC banner, the tool provides information regarding the compilation and encoding of the specification and implementation processes: the number of states[3], the number of OBDD variables required for encoding the state space of the process, and the number of OBDD nodes in the transition relation of the process. The time taken by parsing and compilation of the CSP script is printed prior to commencing the refinement check. The latter proceeds by performing state space exploration and traces check first, followed by computing the refusal and divergence relations and failures-divergences check. The traces check is separated from the failures-divergence one for performance reasons only—due to OBDD caching effects this leads to 10-20% speed improvement.

Complete error information—type of property violation, trace after which it is observed, and the discrepancy between the behaviour of the specification and implementation processes—is printed by the tool for the well-known deadlock state in the implementation process. At the end of the ARC run, the type and number of errors is reported in addition to the verification result and the total run-time for this run.

Next, we would like to check this example for $n = 5$. Since we know this would take more than a few seconds, we use the default verbosity level (2) and enable the sleep set method. The following is a printout from the ARC run excluding the rather long error information:

```
ARC - a CSP refinement/equivalence checker v1.2
Copyright 1995-99 AP, UofA, All Rights Reserved
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Specification 'spec': 3.000e+01 states,   5 BDD vars,    203 BDD nodes
Implementation 'impl': 2.430e+07 states,  25 BDD vars,   1336 BDD nodes
BDD encoding took 0.7 secs CPU time
```

[3]This number can be seen as the upper limit on the number of reachable states, as it is the product of the number of states of each of the sequential components in the corresponding process.

```
Traces check...
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850 900
950 1000 1050 1100 1150 1200 1250 1300 1350 1400 1450 1500 1550 1600
1650 1700 1750 1800 1850 1900 1950 2000 2050 2100 2150 2200 2250 2300
2350 2400 2450 2500 2550 2600 2650 2700 2750 2800 2850 2900 2950 3000
3050 3100 3150 3200 3250 3300 3350 3400 3450 3500 3550 3600 3650 3700
3750 3800 3850 3900 3950 4000 4050 4100 4150 4200 4250 4300 4350 4400
4450 ...done (so far 19.8 secs CPU time)
Computing acceptances...done (so far 20.5 secs CPU time)
Computing divergences...done (so far 20.5 secs CPU time)
Failures-divergencies check...
50 100 150 200 250 300 350 400 450
<error found, debugging information printed here>
(so far 24.2 secs CPU time)
500 550 600 650 700 750 800 850 900 950 1000 1050 1100 1150 1200 1250
1300 1350 1400 1450 1500 1550 1600 1650 1700 1750 1800 1850 1900 1950
2000 2050 2100 2150 2200 2250 2300 2350 2400 2450 2500 2550 2600 2650
2700 2750 2800 2850 2900 2950 3000 3050 3100 3150 3200 3250 3300 3350
3400 3450 3500 3550 3600 3650 3700 3750 3800 3850 3900 3950 4000 4050
4100 4150 4200 4250 4300 4350 4400 4450 ...done

spec [FD= impl is FALSE.
Found 0 trace error(s), 1 failures error(s) and 0 divergence error(s).
4474 states and 6454 transitions were explored.
Maximum DFS depth was 110

*** Total 52.5 secs CPU time
Memory manager bytes allocated: 11567112
Approximate bytes used: 1673272
Number of nodes: 31810
Node limit: 1000000
Overflow: no
Approximate bytes per node: 52.60
Cache entries: 13609
Cache size: 131042
Cache load factor: 0.10
Cache look ups: 2421913
Cache hits: 404164
Cache hit rate: 0.17
Cache insertions: 2017749
Cache collisions: 1863152
Number of variables: 1544
Number of variable associations: 52
Number of garbage collections: 1
Number of nodes garbage collected: 407431
Number of find operations: 1534919
```

There are few differences with the second run. Firstly, a progress indicator

(the number of pair of states explored) is printed during the refinement check. Secondly, the sleep set method enforces the use of depth-first state space exploration and the maximum depth of the reachability graph is reported after the number of states and transitions explored. Finally, a number of OBDD engine statistics are printed at the end of the session.

Chapter 8

Extensions and Future Work

Overview

This chapter discusses a number of topics that extend the work presented so far in this thesis, and highlights avenues for further investigation. We present an application of the full abstraction property of CSP semantics that allows the reduction of a refinement checking problem to a simpler reachability problem that can be efficiently computed using breadth-first OBDD-based reachability analysis on an unlabelled transition relation. This technique also enables the use of promising SAT-based formal verification methods in the context of CSP. We explore the notion of semi-formal verification and its applicability to our work.

8.1 Introduction

Due to its practical significance and fundamental computational complexity, research in automated formal verification can be seen as an open ended process, and we expect it to remain so for a long time to come. To ensure the continuity of this process, research activities are not only concerned with solving previously identified and known problems, but also searching for and formulating new problems and challenges that are yet to be addressed. The present work is no exception to this.

In the author's view, the results presented so far in this text can be seen as a coherent and, in a way, complete work. However, we have also been successful in identifying a number of research areas where further investigation appears to be quite promising. This chapter summarises these open areas as opportunities for further advancement of our knowledge.

8.2 Reducing refinement checking to a reachability problem

8.2.1 Pair-by-pair refinement checking

An inherent property of all refinement checking algorithms presented so far is that their execution explores the product of the reachable state spaces of the specification and implementation processes one pair of CSP states at a time. While this works fairly well when the ratio between LTS and CSP process states is high (see Section 3.4), it becomes a disadvantage when the ratio becomes close to 1 : 1, for example when the processes being checked for refinement are deterministic. This worst case scenario is demonstrated by the Monkey Puzzle example, for which the OBDD-based refinement checking is about eight times slower than explicit state space exploration techniques (see Section 3.3.3). This phenomenon is due to the relatively high cost of performing relational product on OBDDs—an operation used in the computation of successor states. This cost is amortised over the LTS states reachable from another set of LTS states through a single visible event.

The need to perform the refinement checking on a pair-by-pair basis is necessitated by the relatively complex refusal and divergence semantics that have to be computed at each iteration of the algorithm. On the other hand, it is known that OBDDs are usually more efficient when applied on sets of many states at a time. A typical reachability analysis procedure using OBDDs in the literature obtains the states reachable through exactly n transitions from an initial state on the n-th iteration of the reachability analysis algorithm (see, for example, Burch *et al.*'s original symbolic model checking algorithm [BCM+92]).

There is, however, a well-known class of safety properties that allows the extensive use of hiding on the implementation process for the purposes of refinement checking in the traces model. These are properties that check the ability of the implementation to perform a specific event. This event may be named *fail* or *error* if it represents a deviation from expected behaviour, or *done* in the context of finding solutions to combinatorial problems such as puzzles. A refinement check of this nature can be performed by reachability analysis of the LTS of the process $IMPL \setminus \Sigma - \{fail\}$ followed by a check if any of the reachable states have outgoing transitions labelled with *fail*. If there is no such state, the refinement:

$$Stop \sqsubseteq_{\mathcal{T}} IMPL \setminus \Sigma - \{fail\}$$

holds, and vice versa. One of the examples from Appendix A on which this approach could be successfully applied is the Solitaire puzzle. We estimate that

the use of hiding in this manner has reduced the run-time by at least two orders of magnitude[1].

While the approach described above appears to have a restricted application—checking a specific property in the traces model only—it made us aware of the feasibility of a more general refinement checking algorithm that would operate on all reachable LTS states at once instead of performing the check one pair of CSP states at a time. Such an algorithm holds the promise of providing a better fit for the strengths of OBDD-based LTS representations.

In the rest of this section, we show how such an algorithm can be built by exploiting the full abstraction property of the CSP semantics. Essentially, the check for refinement between two processes is translated into a reachability search on an LTS of a so-called *composite* process suitably derived from the specification and implementation processes of the refinement relation.

8.2.2 Full abstraction in CSP

The notion of full abstraction is rather theoretical and involved [Ros97, Section 9.3]. Generally speaking, it measures how "good" the chosen semantics of a language is. In particular, appropriately chosen semantics should provide for the existence of a *context* $C[\cdot]$ for distinguishing between processes based on some simple *test* $\mathbf{T}$.

A context can be seen as a procedure for building a process $C[P]$ (which we call a *composite* process) from another one denoted as P. As an example, a context may put the original process in parallel with a certain environment process ENV and possibly hide some of the events of the compound process:

$$C[P] = ENV \underset{A}{\|} P \setminus B$$

A test $\mathbf{T}$ is a simple well-defined aspect of the behaviour of a process, which can be either true or false. Examples of such tests are the propositions "the process deadlocks after trace $\langle\rangle$", "the process' traces contain $\langle fail \rangle$", etc. A test may contain several propositions combined using boolean operators.

In a semantic model that is fully abstract with respect to $\mathbf{T}$, any two processes P and Q with unequal semantic values should be clearly distinguished by a context $C[\cdot]$ applying $\mathbf{T}$ to $C[P]$ and $C[Q]$, whereby exactly one of the composite processes passes the test and the other does not.

Roscoe [Ros97] has established the existence of suitable semantic contexts and tests in each of the three CSP models. He has formulated the following tests:

[1] In fact, this is a very conservative estimate since we have never been able to run a complete refinement check on Solitaire without the use of hiding due to space restrictions.

- $\mathbf{T}_{\mathcal{T}}$: A process fails this test if its traces include a specific one, e.g. $\langle fail \rangle$, and passes the test otherwise. The traces model $\mathcal{T}$ of CSP is fully abstract with respect to $\mathbf{T}_{\mathcal{T}}$;

- $\mathbf{T}_{\mathcal{F}}$: A process fails this test if either its traces contain $\langle fail \rangle$ or if it deadlocks immediately on trace $\langle\rangle$, and passes the test otherwise. The failures model $\mathcal{F}$ is fully abstract with respect to $\mathbf{T}_{\mathcal{F}}$;

- $\mathbf{T}_{\mathcal{N}}$: A process fails this test if it immediately deadlocks or diverges on trace $\langle\rangle$, and passes the test otherwise. The failures-divergences model $\mathcal{N}$ is fully abstract with respect to $\mathbf{T}_{\mathcal{N}}$.

8.2.3 Refinement checking with observers

The main idea behind the application of full abstraction to refinement checking can be summarised in the following three step technique for verifying the relation $P \sqsubseteq_M Q$:

1. From the description of the specification P derive an *observer* process $O_M(P)$;

2. Construct the LTS of the composite process:

$$(O_M(P) \underset{\Sigma'}{\|} Q) \setminus \Sigma'$$

   where $\Sigma' = \Sigma - \{fail\}$ ($fail$ is a special event that is not part of the alphabet of P nor Q);

3. Apply the test $\mathbf{T}_M$ to the composite process. The outcome of the test reflects the truthfulness of $P \sqsubseteq_M Q$.

Step 1 above requires the computation of an observer process (in LTS form) derived directly from the specification P. Essentially, the observer monitors the behaviour of the implementation process without affecting or restricting it in any way. More importantly, the observer process is capable of detecting behaviour that is not permitted by the specification process, and signaling the presence of that erroneous behaviour in a way that allows detection through the application of the test $\mathbf{T}_M$.

Since the semantics of refinement depends on the CSP model M, the algorithm for deriving the observer process is specific to each of the three CSP semantic models. The significance of the time and space complexity of step 1 above is limited by the fact that the specification process P, unlike the implementation

Q, usually has a small number of number of states. In practice, it is acceptable that the computation of the observer has a similar computational complexity to process normalisation—a procedure that is a pre-requisite to refinement checking in FDR2 [For97].

Step 2 above combines the observer and implementation processes to execute in a lock-step mode, meaning that the observer process is able to observe every visible event that the implementation can engage in, and thus monitor its behaviour. The construction of the LTS of composite process in OBDD form is straightforward using the compilation techniques outlined in Chapter 3.

Step 3 above performs the actual refinement checking. Recall that the pair-by-pair algorithm has to explore two LTSs—one for the specification and one for the implementation—and perform the relatively complicated and time consuming computation of CSP semantics on each pair of CSP states. In contrast to this, step 3 of the proposed approach traverses only one LTS (that of the composite process) and the tests that are to be checked can be applied to all reachable LTS states at once. Given the set of reachable states and the transition relation of the composite process, the propositions contained in the tests $\mathbf{T}_{\mathcal{T}}$, $\mathbf{T}_{\mathcal{F}}$ and $\mathbf{T}_{\mathcal{N}}$ can be checked as follows:

- The proposition "Composite process never performs a certain event" (i.e. the *fail* event) is equivalent to determining that no reachable state has an outgoing transition labelled with that event;

- The proposition "Composite process cannot immediately deadlock" can be reduced to determining that all reachable states have at least one outgoing transition;

- The proposition "Composite process cannot immediately diverge" can be tested by ensuring that there is no unbroken loop of internal events reachable from any of the initial states by a sequence of τ events.

The application of the above checks is relatively easy and thus the computationally intensive part of Step 3 is reachability analysis.

This novel refinement checking algorithm achieves the objectives set in Section 8.2.1 because the reachability analysis in Step 3 can be performed using OBDD-based breadth first search on a single LTS in contrast to the pair-by-pair refinement checking algorithm. The full abstraction results proven by Roscoe guarantee the soundness of this approach.

In the presented three step checking refinement technique, the construction of the observer process $O_M(P)$ from an arbitrary specification process P is the key step. It is discussed in the next section.

8.2.4 Transforming specifications into observers

Because of the semantic correlation between an observer process $O_M(SPEC)$ derived from a specification process $SPEC$, one would intuitively expect that the construction of the former from the latter process would be semantically driven. Therefore, it is proposed that the construction is performed in two stages:

- Firstly, the LTS representation of the specification process is normalised to obtain the LTS of the process $\texttt{normal}(SPEC)$;
- Secondly, a transformation is performed on the LTS representation of the process $\texttt{normal}(SPEC)$ to obtain $O_M(SPEC)$;

The procedure for obtaining a normal form representation from an LTS representation of a process obtained from its CSP description has been extensively discussed in the literature [Ros94, For97, Ros97]. The normal form of a process is closely related to its semantics and behaviour. In particular, two processes are semantically equivalent if and only if their normal forms are syntactically equivalent [Ros97]. For our purposes, the most important property of $\texttt{normal}(SPEC)$ is that for each trace in $traces(SPEC)$ there exists a unique state from $\texttt{normal}(SPEC)$ that corresponds to a sequence of transitions labelled with the events from that trace starting from the initial state of the normal form.

The normal form of a process can be represented using an LTS augmented with a couple of mapping functions[2]:

- $minaccs : \mathcal{S} \rightarrow \mathbb{P}(\mathcal{E})$ maps a set of minimal acceptances to each of the states in the LTS. Semantically, the elements of the set $minaccs(\gamma)$ for a state $\gamma \in \mathcal{S}$ represent sets of events $\{A_i\}_{i=1}^{n}$ that the normal form can stably accept in γ and the normal form cannot stably accept any subsets of A_i in γ. Also, $\bigcup_{i=1}^{n} A_i$ should be the same as the set of event labels on the outgoing transitions of γ;
- $div : \mathcal{S} \rightarrow \mathcal{B}$ maps a boolean divergence condition to each of the states in the LTS. Semantically, a state $\gamma \in \mathcal{S}$ is divergent if and only if $div(\gamma)$ is true.

The second stage of obtaining an observer process—computing $O_M(SPEC)$ from $\texttt{normal}(SPEC)$—is dependent on the model M in which the refinement check is to be performed. In the rest of this section, we show how this can be achieved for the traces model.

[2]This results in a representation that is essentially the same as Roscoe *et al.*'s Generalised LTS [RGG+95].

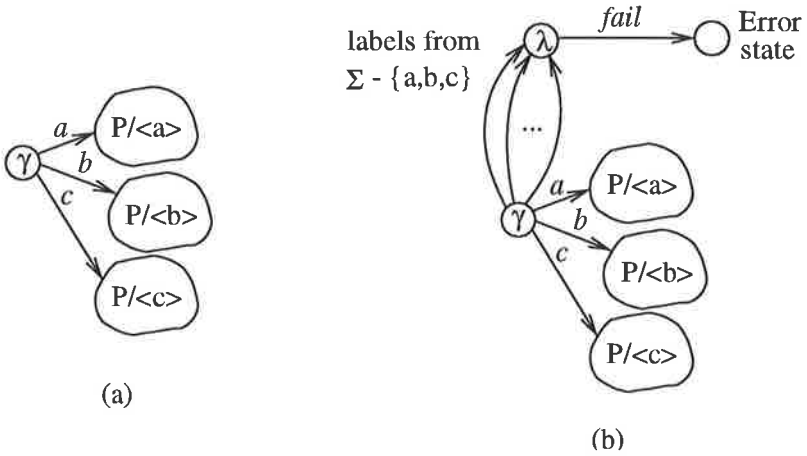


Figure 8.1: Transformation of a state in $\texttt{normal}(SPEC)$ to obtain a state in $O_{\mathcal{T}}(SPEC)$

To apply $\mathbf{T}_{\mathcal{T}}$ to the composite process, an observer process for checking $\sqsubseteq_{\mathcal{T}}$ has to perform the *fail* event whenever the observed (implementation) process can engage in an event that *SPEC* does not allow after a particular trace t. Since any trace $s \in traces(SPEC)$ can be mapped uniquely to a state of $\texttt{normal}(SPEC)$, if the occurrence of a disallowed event in any of the states in the normal form can be detected, then one can also detect a traces refinement violation after any trace of *SPEC*.

For a given state γ of $\texttt{normal}(SPEC)$, the enabled events $enabled(\gamma)$ can be computed simply as the union of the event labels of all outgoing transitions or, alternatively, as the union of all minimal acceptance sets of γ. To detect an occurrence of a disallowed event in γ, new outgoing transitions are added as follows:

$$\{\gamma \xrightarrow{a} \lambda \mid a \in \Sigma - enabled(\gamma)\}$$

where λ has a single outgoing transition labelled with the event *fail*.

This operation is depicted on an example in Figure 8.1. Figure 8.1a shows a state γ from $\texttt{normal}(SPEC)$ which has three outgoing transitions labelled with the events a, b and c connected with the rest of the LTS of $\texttt{normal}(SPEC)$. Figure 8.1b illustrates the transformation of γ to obtain the corresponding state in $O_{\mathcal{T}}(SPEC)$. This involves adding extra transitions from γ to λ labelled with each event that is disallowed by γ in $\texttt{normal}(SPEC)$, and a transition from λ to a special error state labelled with the event *fail*. Note that it is not necessary to add λ and error states to $O_{\mathcal{T}}(SPEC)$ for every state in the normalised specification—one pair of these states is sufficient for an observer in the traces model.

By performing this transformation on all states of $\texttt{normal}(SPEC)$, an observer process $O_{\mathcal{T}}(SPEC)$ is obtained. This observer can then be used to build a composite process to perform refinement checking of $SPEC \sqsubseteq_{\mathcal{T}} Q$ for any process Q as described in Section 8.2.3.

8.2.5 OBDD-based reachability analysis

Once the refinement checking problem is reduced to a reachability analysis problem on an LTS in OBDD form, a number of techniques known in the context of symbolic model checking become applicable. In this section, we briefly discuss the techniques that appear most relevant to our needs.

The LTS representation of the composite process obtained using the approach discussed in Section 8.2.3 has transitions that are labelled with at most two different events—τ and *fail*. At the implementation level, it is possible to substitute the use of the *fail* event in $\mathbf{T}_{\mathcal{T}}$ and $\mathbf{T}_{\mathcal{F}}$ with a special "error" state. An "error" state of the LTS of the composite process contains the observer process state which has a single outgoing transition labelled with the event *fail*. Avoiding the explicit use of *fail* in the composite process makes all the transitions in its LTS internal, which in turn enables the use of an unlabelled transition relation during reachability analysis.

During the reachability analysis, which proceeds in breadth-first manner using OBDD image computation, it is often acceptable to abort the state space exploration after a sequence of transitions from the initial states to an error state is found. This technique is often referred to as "short-circuiting" the symbolic model checking algorithm [McM92a], and can be implemented in a straightforward manner by checking whether an error state has been added to the set of reachable states after each iteration of the OBDD image computation.

While all of the reachability analysis techniques discussed so far perform forward traversal of the state space starting from the initial states, transforming the refinement checking problem into a reachability problem allows the use of backwards transition relation traversal. With this approach, the search starts from the "error" states in the LTS and proceeds backwards until either an initial state is reached or a fixed point is reached. In many cases, backwards traversal has been found to be more efficient than forward traversal.

Last but not least, full abstraction and the use of observer and composite processes makes it possible to perform refinement checking with SAT-based model checking techniques that are discussed later in Section 8.6.

8.2.6 Related work

The idea of transforming a specification process to obtain an observer process and constructing a composite process to check the validity of a refinement relation is related to the automata-theoretic approach to temporal logic model checking [WVS83, VW86, Var95]. The latter approach uses the theory of automata to capture computations that can be described both through programs (models) and specifications (temporal formulae). The translation from programs and specifications to the common automata representation allows questions about programs and specifications to be reduced to questions about automata.

More specifically, given a temporal logic formula ϕ, it is possible to construct a so-called Büchi automaton [Buc62] A_ϕ that accepts all computations satisfying ϕ [Var95]. This construction is generally of exponential complexity to the length of the formula ϕ, but this is rarely a problem in practice because the formulae to be checked are usually very short. To determine whether a model A satisfies a temporal logic formula ϕ, a product automaton A_P of the model and $A_{\neg\,\phi}$ is built. Then, by construction, A_P accepts all computations that satisfy $\neg\,\phi$, that is, violate the original property being checked. Thus, the model check will succeed if and only if A_P is empty. There are several algorithms for checking emptiness of a Büchi automaton [GH93] that work in linear time to the size of the automaton and can be performed while the product automaton A_P is being built. SPIN can be seen as a typical linear temporal logic (LTL) model checking tool utilising this approach [Hol97].

Despite the apparent similarities between the refinement checking technique presented in this section and the automata-theoretic approach to temporal logic model checking, there are also a number of differentiating factors. CSP refinement checking is inherently a single language approach, whereas temporal logic model checking requires separate languages for describing properties and models. Consequently, the observer process in our approach is obtained by LTS transformation rather than by compilation of a temporal logic property into automata. Another difference can be seen in the construction of the composite process. Our technique utilises parallel composition and hiding in a similar way to CSP term compilation, deriving an LTS in OBDD form, whereas the automata theoretic approach to temporal logic model checking proposes to build the product Büchi automaton on the fly.

Perhaps surprisingly, there is very little research on the links and relationships between temporal logic model checking and refinement-based verification techniques in general and in the context of CSP in particular. One notable exception is Leuschel *et al.*'s work [LMC00], which explores the problem of performing LTL model checking via CSP refinement checking using FDR2. It has been found

that converting an LTL formula into a specification process suitable for refinement checking against the model process is very difficult and, in fact, impossible in the general case. On the other hand, applying the observer/composite process method to this problem has been successful. Leuschel *et al.* [LMC00] have shown how a Büchi automaton $A_{\neg\,\phi}$ derived from the LTL formula ϕ can be converted into an observer process (called tester process in their paper). The latter is then appropriately composed with the model process being checked and two refinement checking tests are applied to establish the truthfulness of the original LTL model checking problem.

8.3 Optimised pair-by-pair refinement checking

By construction, the basic pair-by-pair refinement checking algorithm implemented in ARC/PP and presented in Section 3.3.2 explores all pairs of CSP states (Γ, Λ) of the specification and implementation processes being checked. In essense, this algorithm works on the normal forms of the specification and implementation, and Γ and Λ are actually states of the corresponding normal forms. Therefore, the number of pairs of states that are explored is proportional to the size of the normal form of the implementation.

In contrast to ARC/PP, the FDR2 tool explores the product of the normal forms of the specification process and the LTS states of the implementation process [Ros94]. Therefore, the refinement checking algorithm in FDR2 may have to explore considerably fewer pairs of states when the normal form of the implementation process is larger than its LTS, e.g. if the implementation process contains complex nondeterminism. The inefficiency of the refinement checking algorithm in ARC/PP arises from the fact that it may be proving that a given LTS state of the implementation refines a particular normal form state of the specification many times over instead of a one-off proof as in FDR2.

Fortunately, the issue discussed above can be resolved with a couple of small modifications to the original Algorithm 8 from Section 3.3.2, which results in the optimised pair-by-pair refinement checking procedure in Algorithm 14. Firstly, Algorithm 14 can no longer support equivalence checking as seamlessly as Algorithm 8 does, therefore, *Rel* is not an input parameter and all references to it in the body of the algorithm are removed. Secondly, the addition of new pairs to the set *Pending* described in lines 31-35 of Algorithm 8 is modified in the following way: for every new pair of states (Γ, Λ) visited, we remove from Λ all Θ such that the pair of states (Γ, Θ) is already visited or pending. This change is captured in lines 20-24 of Algorithm 14.

We suggest that, as a part of future research, Algorithm 14 is implemented

Algorithm 14 The optimised pair-by-pair refinement checking algorithm

Require: Process P and its LTS $\mathcal{L}_P$
Require: Process Q and its LTS $\mathcal{L}_Q$
Require: *Model* is a CSP model for the check—one of $\{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$
Ensure: All violations of the relation being checked are reported

1: {Start from the pair of initial states}
2: $Pending \leftarrow \{(TauExpand(\mathcal{I}_P, \mathcal{L}_P), TauExpand(\mathcal{I}_Q, \mathcal{L}_Q), \langle\rangle)\}$
3: {Initially, we have checked nothing}
4: $Checked \leftarrow \emptyset$
5: **while** $Pending \neq \emptyset$ **do**
6: {Get the first pending pair of states}
7: $(\Gamma_P, \Gamma_Q, Trace) \leftarrow \textbf{head}(Pending)$
8: $Pending \leftarrow \textbf{tail}(Pending)$
9: $divFlag \leftarrow Model = \mathcal{N} \wedge (divergent(\Gamma_Q, \mathcal{L}_Q) \vee divergent(\Gamma_P, \mathcal{L}_P))$
10: **if** $NextEvents(\Gamma_Q, \mathcal{L}_Q) \not\subseteq NextEvents(\Gamma_P, \mathcal{L}_P)$ **then**
11: {Traces error after *Trace*}
12: **else if** $Model = \mathcal{N} \wedge divergent(\Gamma_Q, \mathcal{L}_Q) \wedge \neg\, divergent(\Gamma_P, \mathcal{L}_P)$ **then**
13: {Divergence error after *Trace*}
14: **else if** $\neg\, divFlag \wedge Model \in \{\mathcal{F}, \mathcal{N}\} \wedge refusals(\Gamma_Q, \mathcal{L}_Q) \not\subseteq refusals(\Gamma_P, \mathcal{L}_P)$ **then**
15: {Failures error after *Trace*}
16: **end if**
17: $Checked \leftarrow Checked \cup (\Gamma_P, \Gamma_Q, Trace)$
18: **if** $\neg\, divFlag$ **then**
19: **for all** $a \in NextEvents(\Gamma_Q, \mathcal{L}_Q)$ **do**
20: $\Gamma'_P \leftarrow NextStates(\Gamma_P, \{a\}, \mathcal{L}_P)$
21: $\Gamma'_Q \leftarrow NextStates(\Gamma_Q, \{a\}, \mathcal{L}_Q) - \{\Lambda \mid (\Gamma'_P, \Lambda, X) \in Checked \cup Pending\}$
22: **if** $\Gamma'_Q \neq \emptyset$ **then**
23: $Pending \leftarrow \textbf{cons}(Pending, (\Gamma'_P, \Gamma'_Q, Trace^\frown\langle a\rangle))$
24: **end if**
25: **end for**
26: **end if**
27: **end while**

in the ARC tool and its empirical performance on a wide variety of examples is studied. While the optimised pair-by-pair refinement algorithm can clearly be expected to have better worst-case time requirements, it remains to be seen how its average-case performance compares to that of Algorithm 8. Another open research question is how Algorithm 14 compares to the refinement checking technique from Section 8.2, which bypasses the pair-by-pair approach completely in favour of the simpler reachability analysis.

8.4 Other partial order methods

In Chapter 6, we have looked at the application of one particular partial order method—sleep sets—in the context of refinement checking CSP. It has been established empirically that the use of sleep sets independently or in conjunction with the PRS-enhanced reachability analysis brings about considerable time and space savings. Specifically, the utilisation of sleep sets has provided a significant reduction in the number of transitions that need to be explored for a particular model. In the light of these findings, one can anticipate that other partial order reduction methods are quite likely to provide further improvements to our refinement checking algorithms.

One suggestion that may be fruitful is to study the application of Godefroid's persistent set technique [God94]. This technique enables the so called *persistent-set selective search*, which performs a restricted form of reachability analysis (i.e., not all reachable states are visited) that has useful properties—for example, it is guaranteed that all deadlock states are reached [God94]. Note that the sleep set technique does not offer a saving in terms of the states visited during reachability analysis, but only a reduction in the number of transitions followed. However, it is not clear how persistent sets can be applied to refinement checking of an arbitrary pair of processes and this is where the main challenge of this task is.

8.5 OBDD-based state compression

State compression techniques [RGG+95, For97] as applied in the FDR2 tool aim at reducing the reachable state space of an LTS of a process by computing a simpler LTS that is semantically equivalent to the original. In fact, one such technique—normalisation—has to be applied to the specification process in a refinement check for all versions of the FDR tool. As discussed in Sections 3.3.3 and 3.4.1, state compression is very useful for a number of practical problems, particularly when a concurrent system is built iteratively from components exhibiting locality of communication and repetitive structure, such as the Dining Philosophers

example. Although we have not looked at implementing OBDD-based versions of the state compression algorithms, it is certainly feasible to do so. However, what is to be gained from that apart from a straightforward application to examples for which we already know explicit state compression algorithms work well?

A closer look at the state compression algorithms as implemented in FDR2 reveals an implicit requirement to explicate the LTS of the process to be compressed [Ros97]. This is a potential bottleneck as explicating an LTS can be a space and time consuming task. In our experience, explication essentially prohibits compression algorithms for processes with more than approximately twenty to thirty thousand states in their explicated LTSs. This has been the case for the Alternating Bit Protocol example used in our empirical performance study in Section 3.3.3.

Therefore, the main advantage in pursuing OBDD-based implementations of state compression techniques can be seen in circumventing the need to perform an initial explication of the LTS. As demonstrated extensively in the literature and in this thesis, OBDDs can handle sets of states and relations quite efficiently. In fact, OBDD-based versions of other LTS minimisation problems have already been proposed. For example, an OBDD-based strong bisimulation minimisation algorithm has been developed by Bouali and de Simone [BdS92].

8.6 SAT-based verification

The general boolean satisfiability (SAT) problem has a very simple formulation: given a boolean function $f : \mathcal{B}^n \to \mathcal{B}$, determine if there is an assignment to the boolean variables of f that makes f evaluate to 1. Despite this apparent simplicity, SAT is a computationally hard problem—in fact, it is the first problem shown to be NP-complete [Coo71], and, therefore, currently has no known algorithmic solution of lower than exponential complexity[3]. Since many practical problems in a variety of application areas—scheduling, artificial intelligence, digital circuit design and verification, etc.—are in the same complexity class, it is possible to apply a SAT solver to any of these problems after a suitable translation.

Although the first practical algorithm for SAT solving—the Davis-Putnam procedure [DP60]—has been around for more that forty years, major improvements to it in the past few years [MSS99, MMZ+01] have made SAT solving powerful enough to be able to work on functions with hundreds of thousands

[3]What the author finds even more exciting is that finding efficient SAT solving algorithms is directly related to the fundamental mathematical question of whether P=NP. It is hard to overestimate the significance of this seemingly theoretical debate on our perception of the world and may be even day-to-day life.

variables. These algorithmic advances have, in turn, generated a lot of interest in using SAT techniques as an alternative to OBDDs for model checking.

8.6.1 Bounded model checking (BMC)

The idea of using SAT to perform a restricted form of model checking called *bounded model checking* (BMC) has been introduced by Biere *et al.* [BCCZ99]. BMC solves the problem of finding a path from the initial states to an error state of a finite state system, whereby the path may consist of up to k transitions. Since bounded model checking is concerned with finding error traces of length limited by the bound k, it is targeted at finding short counter-examples and is not exhaustive since the presence of error traces of length greater than k is not checked for. Despite this apparent limitation of BMC, its industrial application in the context of digital circuit verification has demonstrated its practical effectiveness [CFF+01]. Therefore, it appears that there is empirical evidence to suggest that a significant amount of errors in real-world models can be exposed via a relatively short sequence of transitions.

Similarly to the OBDD-based model checking approach, BMC requires that the transition relation of the system model is encoded as a Boolean formula using, for example, characteristic functions as described in Section 2.3.1. Let *Init* be the boolean encoding of the initial states of a finite model, *Rel* the encoding of its transition relation, and *Err* the encoding of the error states. The propositional formula describing a path of length k starting from one of the initial states is built by unrolling the transition relation as follows:

$$Path_k = Init \wedge \bigwedge_{i=1}^{k} Rel_i$$

where Rel_i encodes the set of possible transitions after $i-1$ transitions from the initial states and is obtained from *Rel* by variable renaming. Then, the formula:

$$F_k = Path_k \wedge (\bigvee_{i=1}^{k} Err_i)$$

where Err_i encodes the error states occurring after i transitions from the initial states of the model, is satisfiable if and only if an error state is reachable from the initial states after m ($m \leq k$) transitions [BCCZ99]. Any variable assignment that makes F_k evaluate to 1 is a boolean encoding of a sequence of transitions leading to an error state, which is useful in diagnosing and fixing the error.

A refinement of the basic BMC technique that is currently being researched aims to exploit the structural similarities between the boolean encodings of the

BMC problem when the bound is incremented by one [WKS01]. Since $Path_{k+1}$ is obtained from $Path_k$ by conjoining the latter with Rel_{k+1}, information obtained when SAT solving F_k, such as the learned implicants of F_k, can be useful when checking F_{k+1}. This is expected to lead to performance and useability improvements in using bounded model checking.

How would BMC apply to the work presented in this thesis? At first, it might appear that a change from an OBDD- to a SAT-based verification engine would require a significant rework of the CSP compilation techniques developed in this thesis. However, we believe that there is a significant scope for reuse since OBDDs are already encodings of boolean formulae. In particular, the CSP compilation rules from Chapter 3 should remain largely intact; one aspect that has to change is the way sequential components are composed together and unfolded to produce *Rel*.

A positive sign with regards to adding SAT-based refinement checking to the ARC tool is also the fact that the Bounded Model Checker tool (BMC) [BCCZ99] is built on top of the popular OBDD-based model checking tool SMV [McM92b]. Since ARC can also be viewed as an OBDD model checking tools targeted at the CSP framework, the amount of work needed to accommodate a SAT-based verifier should be comparable to those made by Biere *et al.* [BCCZ99].

8.6.2 Applying induction and hybrid methods

Since the amount of verification done by bounded model checking can be directly controlled by increasing the bound k, it is, in principle, possible to perform exhaustive verification by setting k to be equal or larger than the size of the longest non-cyclic sequence of transitions from the initial states of the finite state model (that is, the diameter of the corresponding reachability graph) being verified [BCCZ99]. However, there is no known general and computationally inexpensive algorithm to compute the diameter of the system and, therefore, this technique can be applied only in certain special cases. For example, in a concurrent system composed of a number of sequential components executing concurrently, it is possible to achieve complete verification by computing the worst-case diameter of the reachability graph of the system as the product of the diameters of its sequential components. Still, the worst-case diameter obtained in this way may well be much larger than the actual one, and the bounded model checking algorithm could be potentially performing a lot of unnecessary computations.

To overcome this limitation of BMC, a number of researchers have explored other techniques for SAT-based reachability analysis and model checking. One proposal is to augment the path formula $Path_k$ with additional predicates guaranteeing that only non-cyclic paths are considered [SSS00]. Then, it is attempted

to prove (via a suitable encoding as a SAT problem) that if no error trace exists when a bound of k is applied, then no error state can be reached with a bound of $k+1$. If this proof succeeds, then, by induction, no error state can be reached after any number of transitions. Note that the induction step will succeed when k is equal to the diameter of the reachability graph of the model, although it may as well succeed for a much lower k.

A hybrid approach that utilises a non-canonical representation of boolean formulae called Boolean Expression Diagrams (BEDs) and a symbolic reachability analysis algorithm that uses both OBDDs and SAT algorithms for checking satisfiability is presented by Williams *et al.* [WBCG00]. Since a standard reachability analysis algorithm is used, this approach, unlike BMC, provides for exhaustive verification.

The main requirement for the implementation of both the induction-based and hybrid methods discussed above is the same as for BMC—the availability of the transition relation as a boolean formula. Therefore, either of these techniques can be implemented in ARC.

8.7 When everything else fails: semi-formal verification

Despite the promising technological advances in the area of automated formal verification, there will always be systems that are sufficiently complex to be well beyond the capacity of formal verification techniques due to the high computational complexity of this verification methodology. This is quite unfortunate, because the more complex a system is, the higher the amount of verification it will require to identify and eliminate design errors. In other words, formal verification may fail to deliver on system designs that are most critical to verify and get right.

Of course, one can always apply the traditional informal testing and simulation methodology discussed in Section 1.1, because it only requires some form of executable model to work with and, therefore, scales up in terms of space requirements for the model very well[4]. However, test-based verification is hardly automated and requires a significant amount of manual effort to reach a given level of design coverage. Thus, one can see formal and informal (test-based) verification occupying the opposite ends of the spectrum in terms of capacity versus automation and coverage trade-off, leaving a large gap in between the two

[4]Indeed, if a system being designed can be built in practice, then a model of that system, which by definition is a mathematical abstraction of that real system, can also be built.

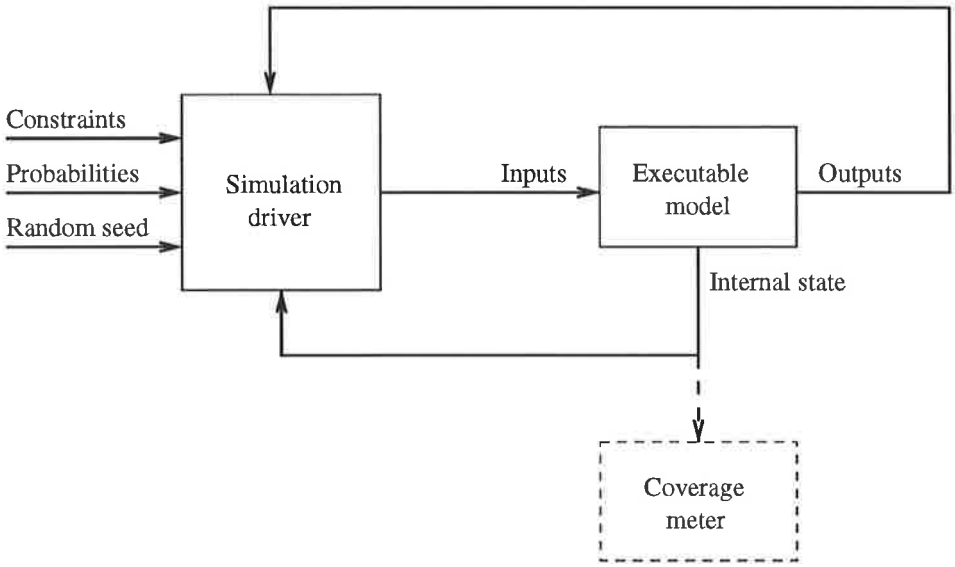


Figure 8.2: The semi-formal verification setup

methodologies. There is a growing need to fill this gap by inventing verification techniques that combine the advantages of the formal and informal verification methods.

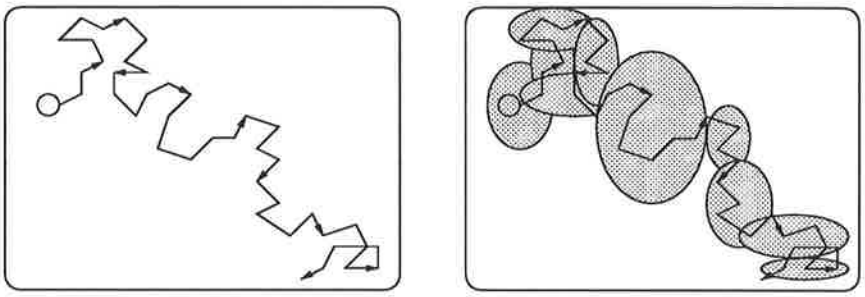
One approach to addressing the verification gap that the author is familiar with through his current professional experience is presented by Yuan *et al.* [YSP+99]. This technique is commonly referred to as user-directed random testing and simulation or, perhaps less accurately, semi-formal verification. Figure 8.2 presents a diagram of the verification setup used by this method. The design that has to be verified is represented as an executable model with inputs, outputs and an internal state that can be sampled at any time during the verification procedure. This model receives input stimuli from its environment, performs certain internal computation and, in turn, provides input to the environment. The environment is modelled by a simulation driver that generates sequences of stimuli for the executable model being verified. The sequence being generated is determined by a set of constraints and probabilities and a random seed provided by the verification engineer. The constraints are boolean expressions over the set of model inputs, outputs and internal state, which need to be satisfied for any valid sequence of input stimuli. The probabilities determine how often would a particular input value be offered to the model. The provision of an initial random seed allows verification runs to be deterministically repeated at a later stage, for example as a part of a regression test suite.

A third optional component in the semi-formal verification setup—a coverage meter—keeps a record of all visited states in selected components of the executable model. This provides valuable feedback about the coverage achieved with a particular set of constraints and probabilities and allows for fine-tuning these verification parameters.

Unlike formal verification techniques, semi-formal methods do not suffer from model capacity limitations and can be applied to any design provided an executable model is available. However, semi-formal verification does not guarantee complete coverage of all possible scenarios. In contrast to informal verification methods, it does not require laborious test case generation, instead offering a formal framework (constraints and probabilities) for describing the environment of the system under test. User-provided constraints can also be seen as a formal design documentation of the interfaces in a larger system. They also facilitate hierarchical design verification practices whereby a large system is constructed and verified in a bottom-up fashion. In this case, environment constraints at one level of the design hierarchy will act as specifications (proof obligations) at the next (higher) hierarchy level.

How relevant is semi-formal verification to CSP refinement checking? Recall that an on-the-fly model checker essentially creates an executable finite state model that is exhaustively explored by recording the set of states that are visited during reachability analysis. That fits very well with semi-formal verification, which, however, does not need visited state storage. Since FDR2 uses on-the-fly model checking, it could be easily adapted to semi-formal verification by adding the simulation driver and the optional coverage meter components.

On the other hand, the ARC tool builds an OBDD-based LTS (either monolithic or partitioned) of the processes that are to be checked, which is not particularly suitable as an executable model for two reasons. Firstly, for large processes it may not be possible to build the OBDDs for the corresponding transition relations, therefore the scalability of the implementation would suffer. Secondly, computing next possible transitions and states would involve computationally intensive OBDD relational product operations, which would certainly take longer time than on-the-fly techniques. However, neither of these two factors is completely preventing the application of semi-formal techniques in ARC. In the next section, we propose a novel semi-formal technique that combines Yuan *et al.*'s approach [YSP+99] with local symbolic state space exploration using either SAT or OBDD-based algorithms.



(a) semi-formal reachability analysis

(b) symbolic semi-formal reachability analysis

Figure 8.3: State space exploration for semi-formal verification

8.7.1 A symbolic semi-formal verification method

In a number of examples of practical interest, it may be possible to build a symbolic (OBDD- or SAT-based) transition relation but impossible to perform complete reachability analysis and verification because of space or time constraints; it may only be possible to do state space exploration up to a limited bound (depth). For such examples, one can attempt to exploit the advantages of both formal and semi-formal techniques. This has been first recognised by Ho *et al.* [HSH+00] who have developed a tool called Ketchum unifying formal and simulation-based verification. The goal of Ketchum's algorithms is to explore the state space of a model while covering as many reachable states in selected model components as possible. Ketchum's reachability analysis algorithm uses heuristics to switch from semi-formal model exploration to local symbolic reachability techniques when the distance between the current and the target states is small [HSH+00]. The primary application of this method is sequential test pattern generation.

Figure 8.3a illustrates the operation of the state space exploration of a semi-formal verification algorithm by Yuan *et al.* [YSP+99]. The rounded box represents the entire reachable state space of the executable model, while the little circle is the set of initial states of the model. The simulation driver then generates sequences of inputs that change the state of the executable model driving it through an execution trace denoted by the seesaw line with arrows.

Assuming that a symbolic representation of the transition relation of the executable model can be built, it is possible to perform bounded symbolic reachability analysis from any given state of the model. This would increase the coverage density in the sense that not only a single trace but rather a set of traces guided by the original semi-formal verification trace are explored. This is illustrated by the shaded ellipses overlapping the semi-formal verification trace on Figure 8.3b.

The size of each ellipse and, correspondingly, the depth that each run of symboling state space exploration reaches can be either explicitly set by the user or implicitly determined by limiting the space and time available for each run.

We are now ready to construct a verification algorithm that utilises both a semi-formal and formal (bounded) reachability analysis (shown as Algorithm 15). Our proposal allows for a number of concrete implementations characterised by a set of functions required by Algorithm 15:

Algorithm 15 Symbolic semi-formal verification algorithm

Require: An LTS $\mathcal{L}$ and its symbolic representation $\phi(\mathcal{L})$
Require: Error states *ErrorStates*
Require: Bounded symbolic reachability analysis procedure *BSRA*
Require: Semi-formal simulation driver *SFSD*
Require: Simulation exit condition *ExitCondition*
Ensure: $ReachedErrorStates \subseteq ErrorStates$ contains the set of reached error states in $\mathcal{L}$
  $States \leftarrow \phi(\mathcal{I})$
  while $\neg$ *ExitCondition* **do**
    call $BSRA(\phi(\mathcal{L}), States)$
    $Depth \leftarrow$ number of iterations of *BSRA*
    $LocalStates \leftarrow$ states explored by *BSRA*
    $ReachedErrorStates \leftarrow ReachedErrorStates \cup (LocalStates \cap ErrorStates)$
    $States \leftarrow$ state reached by $SFSD(States, Depth + 1)$
  end while

- Function *BSRA* performs bounded symbolic reachability analysis. It may be utilising either SAT- or OBDD-based techniques. This function performs a breadth-first search on the reachability graph from a given set of states up to a certain number of iterations (depth). As discussed above, the depth of the search may be predefined or implicitly provided by limited space and time resources for the function;

- Function *SFSD* implements the semi-formal simulation driver algorithm. It is guiding the state space exploration similarly to the pure semi-formal method described by Yuan *et al.* [YSP+99];

- The predicate *ExitCondition* controls how long the symbolic semi-formal verification is run. Again, there are several options for defining this predicate based on the length of the trace explored, number of error states found, time limit, coverage threshold, etc.

The operation of Algorithm 15 is contained within a single **while** loop that is executed until the *ExitCondition* predicate becomes true. It starts with a bounded symbolic reachability analysis using *BSRA* from the set of initial states and up to some bound k. If the intersection of the states reached by *BSRA* and the error states is not empty, then the states from that intersection are added to *ReachedErrorStates*. Then, function *SFSD* is used to perform a user-guided random simulation from one of the initial states to a state that is reachable through exactly $k + 1$ transitions. This state is then used instead of the initial states and another iteration of the **while** loop is executed.

Although Algorithm 15 appears promising from the theoretical point of view as it improves the overall state coverage compared to basic semi-formal verification, its practical usefulness can only be measured empirically. A key open question is whether the simulation speed measured as the number of new reachable states explored per unit of time attainable by semi-formal verification alone can be maintained or improved upon when the more computationally expensive bounded symbolic reachability analysis is in Algorithm 15.

8.8 The need for multiple verification engines

The experience accumulated in the course of this research and author's ongoing experience in a broader formal verification context (e.g., hardware verification) leads us to believe that a very likely future trend in formal verification tools is to incorporate multiple back-end verification engines that can be chosen for execution by the user or automatically depending on the problem that is being solved. The rationale for this is really simple—there is no single dominant formal verification method. Consider, for example, the state-of-the-art in explicit and OBDD-based model checking. This work has demonstrated that, at least for concurrent system verification, there are examples that are handled very well using one of these methods but are very inefficient or even impossible to solve under comparable resource constraints using the other model checking technique, and vice versa.

We illustrate the current state of applicability of the known model checking methods in Figure 8.4. Each of the two basic techniques—explicit and OBDD-based state space exploration—can handle a certain subset of the domain of decidable problems called solvable problems. In the context of this discussion, solvable problems are problems that can be decided upon within economically viable space and time limitations, while decidable problems that are not solvable are called insolvable problems.

Although there is a significant overlap in terms of solvable problems for explicit

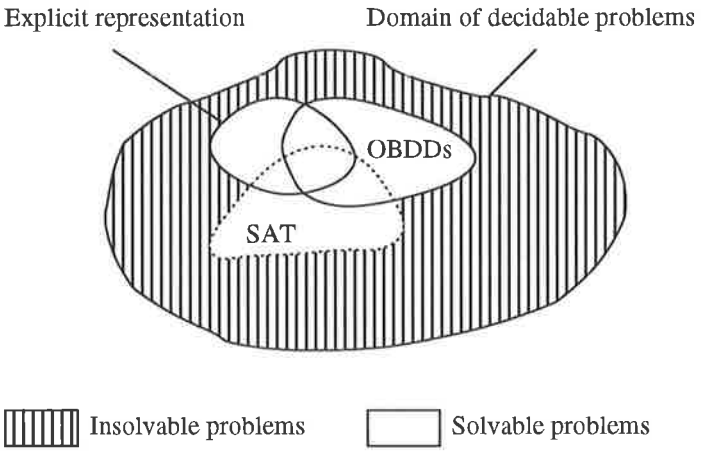


Figure 8.4: The domain of problems and verification techniques

and OBDD-based model checking, there are examples that are solvable using only one of the two—an observation already made in Section 3.4.1. The early research on SAT-based BMC has uncovered, yet again, that certain verification problems that amend themselves to BMC but not OBDD-based model checking.

A good integration of multiple back-end engines transparently to the end-user in a verification tool would result in a more capable and usable verification tool that provides an advantage for both the tool developer and the end-user. We are, therefore, proposing to study the integration issues concerned with implementing multiple back-end engines in a single verification tool. Of particular interest is the issue of making a choice of a verification engine based on the actual problem being verified.

Of course, verification problems that cannot be completely verified using any of the three techniques discussed above can be tackled using semi-formal techniques such as those described in Section 8.7. As we have shown, semi-formal verification methods can also benefit from bounded symbolic reachability analysis local to a particular state in the semi-formal verification trace. However, semi-formal techniques are inherently incomplete and, therefore, it is preferable to utilise them only after all formal verification engines are applied to a problem without success.

8.9 Acknowledgments

The ideas presented in Section 8.2 were born after an impromptu tutorial on full abstraction in CSP provided by Bill Roscoe. He and Jay Yantchev have

been independently exploring similar techniques in the context of aiding state compression in the FDR tool. The optimisation of the pair-by-pair refinement checking algorithm was suggested by Bill Roscoe after a review of a draft of this thesis.

Chapter 9

Conclusions

9.1 Summary

The general approach to formal verification has changed considerably since the early days of the pioneers Floyd [Flo66] and Hoare [Hoa69]. Most notable is the progress from hand-written mathematical notations and proofs to computer readable notations and automated verification programs that can reason about properties "at the click of a button". The initial concerns of the research community that automated formal verification is prohibitively expensive for all problems of practical significance because of its exponential worst-case complexity have been dispelled by empirical results obtained by applying software tools to a large variety of real-world examples. It has been shown that practical problems rarely tend to bring about the worst-case performance of the verification algorithms. The extreme complexity of the general verification case does not prevent the development of efficient solutions for certain classes of problems. This is illustrated, for example, by the successful use of specification process normalisation in both FDR1 and FDR2 [Ros94], and by the use of OBDDs for LTS encoding and refinement checking in the ARC tool.

Still, there is no silver bullet for the inherent computational complexity and corresponding resource requirements (most importantly, space and time) of automated formal verification algorithms. Instead, progress in this area is advancing in small but steadily paced incremental steps that continuously increase the number of problems of practical significance that can be successfully solved. This thesis contributes to this advancement with several new and improved methods and algorithms.

This work builds and expands on the existing body of research in automated formal verification and, in summary, achieves the following results:

- Provides an approach to compiling CSP process terms (scripts) into an LTS representation in a compact OBDD form using a syntax-driven technique. This translation approach is more sophisticated and efficient than existing methods and supports a number of advanced features such as parameterised recursion, identifiers, expressions, etc. The efficiency of encoding the CSP semantics is ensured by a number of specially developed methods including an algorithmic handling of the external choice operator, re-encoding of the OBDD representation of labelled transition relations for sequential components in a concurrent system, and storing the refusal relation in an implicitly conjunctive OBDD form;

- Presents a basic pair-by-pair refinement/equivalence checking algorithm operating on OBDDs working in any of the three CSP semantic models ($\mathcal{T}$, $\mathcal{F}$, and $\mathcal{N}$). The performance of this algorithm on a number of verification examples is experimentally measured. The strengths and potential areas for improvement of the basic pair-by-pair refinement/equivalence checking algorithm are identified based on an in-depth empirical performance analysis and comparison with existing formal verification tools for CSP on these examples;

- Develops solutions for the identified performance bottlenecks. The list of new solutions includes HPTR and PRS-based reachability analysis techniques, the application of partial order reduction methods separately and in conjunction with the PRS technique;

- Proposes a novel non-monolithic form of OBDD-based LTS representation called Hierarchical Partitioned Transition Relation (HPTR) that reflect the synchronisation patterns in a concurrent system, and an algorithm for image computation on HPTRs. This enables refinement checking of processes whose LTS is encoded as an HPTR. It is experimentally established that an HPTR can be up to an order of magnitude smaller in terms of OBDD nodes than an equivalent monolithic OBDD encoding of an LTS;

- Develops a novel reachability analysis algorithm that alleviates the need to store all states visited during state space exploration. This technique is based on an observation that some states, referred to as Pseudo-Root States (PRS) in the text, may no longer be reached from outside of the reached state space. The proposed new reachability analysis algorithm identifies and discards PRS from the set of reached states as early as possible. A study of the theoretical computational complexity of the algorithm is presented along with empirical data that shows more than sixteen-fold reduction in state storage requirements;

- Explores the applicability of partial order methods to CSP refinement checking. An OBDD-based refinement checking algorithm that utilises the sleep set technique is developed and it is experimentally shown that a speed-up by a factor of two to three compared to the basic pair-by-pair refinement checking algorithm is achievable. A novel refinement checking algorithm that exploits both pseudo-root state and sleep set enhancements is proposed. This algorithm delivers much greater state storage savings than the PRS-enhanced reachability analysis alone—for one of the examples, the peak number of stored states grows linearly instead of exponentially with the size of the example (number of processes);

- Presents a technique for performing a refinement check by reducing it to a reachability analysis problem. This technique is key to exploiting the advantages of performing OBDD-based breadth-first reachability analysis on an unlabelled transition relation known from temporal logic model checking research. It also provides a link to the utilisation of SAT-based and semi-formal verification in the context of CSP;

- Proposes a new symbolic semi-formal verification algorithm that combines the advantages of semi-formal verification with bounded symbolic (SAT- or OBDD-based) reachability analysis to achieve better coverage of the state space of the model being verified and maximise the chances of finding hidden errors in the model;

- Implements the novel algorithms developed in a practical tool called ARC. Applies and measures empirically the performance of the proposed algorithms.

Overall, we believe that this work represents a substantial improvement over most, if not all, OBDD-based formal verification tools targeting concurrent process algebras. This improvement can be seen in terms of the richness of the machine-readable notation in use, the variety of novel verification algorithms specifically developed to address verification bottlenecks, the number of examples being empirically studied, as well as the depth of understanding the advantages and disadvantages of using OBDDs instead of explicit on-the-fly model checking within a process algebra setting. We have also demonstrated that, on a number of examples, the techniques described in this thesis can rival the performance and capacity of a commercially developed formal verification tool such as FDR.

Our results are applicable mainly to the verification of control-dominated concurrent systems. Another factor that can be a large contributor to the state space explosion problem—large data types and operations on variables of those

data types—is not directly addressed. This can be seen as a limitation of the applicability of our research.

Finally, there are cases where the practical improvements offered by the verification techniques developed in this thesis fall short of our expectations. For example, HPTR-based refinement checking offers merely a trade-off between space and time requirements rather than providing a clear-cut advantage over basic pair-by-pair refinement checking. Further work is required to better understand the reasons why HPTRs do not work as well as the simple disjunctive or conjunctive partitions of a transition relation known from the hardware verification domain.

9.2 Research arisen from our work

Our research has spawned or otherwise supported a number of other projects.

Cao [Cao97] has worked on the extension of the input language of ARC with replicated CSP operators and sequences. He has also studied the effect of varying OBDD variable ordering for sequential components on the size of the final LTS representation and the speed of the pair-by-pair refinement checking algorithm.

Lam [Lam97] has developed a Graphical User Interface (GUI) front-end for the ARC tool. The interface is a lightweight client process that communicates with the ARC core server code through TCP/IP. The idea is that tool can be running on a quite powerful machine connected to the Internet and serving requests from various users across the globe. An additional feature of the GUI is the ability to visualise and interactively explore labelled transition systems, which is a valuable debugging aid.

Gao and Esser [GE99] have developed model checking functionality for checking for deadlock, livelock, and nondeterminism working directly on the OBDD representation of process semantics and built these into ARC. They also present an improved tool architecture that utilises ARC's back-end verification engine but provides a completely new and fully featured CSP compiler front-end comparable to that of FDR2. The integration with ARC is achieved through a TCP/IP based message exchange API. The compiler includes several advanced features such as interactive parse tree exploration with syntax highlighting, type checking, as well as code for data independence analysis that will eventually allow to verify examples with very large or even infinite state space using a smaller abstraction model.

Appendix A

CSP Examples

Overview

As discussed in Chapter 1, the literature on formal verification and model checking often presents results on problems that are not described in sufficient detail as to enable others to apply their techniques and tools to the same instances and therefore gather valuable experimental information. Although the hardware verification community has its set of standard albeit somewhat outdated benchmark examples (ISCAS85 and ISCAS89), to the best of the knowledge of the author no such database of practical problems has been put together for the concurrent systems verification area.

In this appendix, a description and CSP scripts (in the subset of the language acceptable by the ARC tool) for all examples used throughout the thesis are provided. It is hoped that, despite the need to translate these examples for use with other tools, they would provide a good starting point for the evaluation of new verification approaches. These examples may also be of use to assess the advantages of new extensions to ARC itself.

A.1 Synthetic example

Synthetic is a very simple CSP process with an unusually high number of states due to interleaving. The code for this example is given in Figure A.1. This process has 2^{2^n} states as it consists of 2^n processes, and each of them can engage in event a independently of the others. Although such a process is unlikely to be a part of a real system (hence the name Synthetic), it may be used as an example to assess the ability of a tool to cope with complexity caused by state space explosion from interleaving and nondeterminism.

```
-- Definition of specification and implementation processes
pragma spec spec
pragma impl impl

pragma channel a

n = 5

p(I) = if (I == 0) then a -> STOP
else p(I-1) ||| p(I-1)

spec = p(n)
impl = p(n)
```

Figure A.1: The CSP script for Synthetic example

The presence of n as a parameter in this example is very useful as it allows varying the complexity of a particular instance of this example.

A.2 Milner's Schedulers example

The n-schedulers problem [Mil89] is a widely used benchmark example for verification tools. The system under consideration is a ring of n schedulers which control the execution of a certain task. Event $a.i$ denotes the start of the i-th task, and event $b.i$ denotes its end. The purpose of the schedulers is to ensure that:

- Only one copy of each task is active at any one time;
- Tasks are activated in sequence—task 1, then task 2, and so on. Task 1 can be activated only after task n is activated.

The above two conditions are implemented by the schedulers by passing a token around the ring [Mil89].

As the system of schedulers is described with the use of n as a parameter, it is possible to build a whole family of examples and study the performance of a given tool or set of tools with varying n. Two variants of this example are encoded to check the two properties above.

A.2.1 Variant one

The CSP script to check that both properties of the n schedulers hold is given in Figure A.2. Comments are inserted as appropriate to clarify the purpose of each part of the script. Note that the token passing events $s.i$ as well as the token insertion event st (which only happens in the beginning of system operation) are hidden by the implementation as they are irrelevant to the specification being checked.

A.2.2 Variant two

This variant of the Milner's Schedulers example checks only whether tasks are activated in sequence (see Figure A.3). Thus, more events are hidden in the implementation process—in addition to the events $s.i$ and st, task ending $b.i$ events are also irrelevant. The result of the additional hidden events is that the implementation process has only n CSP states—a significant reduction over the exponentially growing number of states in the implementation process of variant one of this example.

A.3 Dining Philosophers example

Dining Philosophers is a well-known verification example studied for CSP by Hoare [Hoa85]. The system consists of n philosophers who eat at a round table and need two forks to have their meal. A philosopher can become hungry at any time, and after sitting at the table he will take first the fork on his left then the one on his right. If any of the two forks is currently occupied, the philosopher would wait until the fork is put down by the neighbour on the correspondent side. The verification question posed is whether the philosophers can die of hunger, or, simply put, whether the system being described is deadlock free.

The CSP script for this example, given in Figure A.4, closely follows the one provided by Hoare [Hoa85].

A.4 Alternating Bit Protocol example

Alternating Bit Protocol is one of the simplest protocols that address the problem of providing reliable one-way communication over an unreliable communication media [Ros97]. The assumption made by this protocol is that messages sent over the media can be lost or duplicated, but not corrupted. Also, it is assumed that the order of messages is unchanged.

```
-- Definition of specification and implementation processes
pragma spec spec
pragma impl impl

-- 'n' must be an integer > 2
n = 5

-- Channel index range
range = {1..n}

pragma channel a, b, s: range
pragma channel st

-- The specification insists that the a.I events are executed
-- in sequence; a b.I event may be executed at any time after
-- a.I is.

-- 'control' specifies the in-order execution of all 'a'
-- events.
control = control1(1)
control1(I) = if (I == n+1) then
                control1(1)
              else
                a.I -> control1(I+1)

-- 'inseq' specifies that a.I is executed before b.I.
inseq = inseq1(1)
inseq1(I) = if (I == n-1) then
              inseq2(I) ||| inseq2(I+1)
            else
              inseq2(I) ||| inseq1(I+1)
inseq2(I) = a.I -> b.I -> inseq2(I)

-- 'spec' is just the parallel composition of 'control' and
-- 'inseq'.
spec = control.inseq
```

Figure A.2: The CSP script for Milner's Schedulers example, variant one

```
-- The 'start' process inserts a token into the ring of
-- schedulers.
start = st -> STOP

-- 'p_1' (the first scheduler) can be activated either by the
-- ring or by the 'start' process.
p_1 = (s.1 -> SKIP [] st -> SKIP) ;
      a.1 -> (s.2 -> b.1 -> p_1 []
              b.1 -> s.2 -> p_1)

-- The rest of the processes in the schedulers' ring behave as
-- 'p_i(I)'.
p_i(I) = s.I -> a.I -> ( s.((I%n)+1) -> b.I -> p_i(I) []
                         b.I -> s.((I%n)+1) -> p_i(I) )

-- A process to compose all schedulers in the system.
-- Events s.I and st are hidden as they are implementation-
-- specific and irrelevant to the specification.
sched(I) = if (I == 1) then
             (start [|{st}|] p_1) \ {st}
           else if (I == n) then
             (sched(I-1) [|{s.1,s.n}|] p_i(I)) \ {s.1, s.n}
           else
             (sched(I-1) [|{s.I}|] p_i(I)) \ {s.I}

-- Implementation process.
impl = sched(n)
```

Figure A.2: The CSP script for Milner's Schedulers example, variant one (continued)

```
-- Definition of specification and implementation processes
pragma spec spec
pragma impl impl

-- 'n' must be an integer > 2
n = 5

-- Channel index range
range = {1..n}

pragma channel a, b, s: range
pragma channel st

-- The specification insists that 'a.i' events are executed
-- in sequence
spec = spec1(1)
spec1(I) = a.I ->
             if (N == n) then
               spec1(1)
             else
               spec1(N+1)

-- The 'start' process inserts a token into the ring of
-- schedulers.
start = st -> STOP
```

Figure A.3: The CSP script for Milner's Schedulers example, variant two

```
-- 'p_1' (the first scheduler) can be activated either by the
-- ring or by the 'start' process. The additional process 'p1'
-- is required to hide event b.1 outside of the recursive
-- definition of 'p_1'.
p1  = p_1 \ {b.1}
p_1 = (s.1 -> SKIP [] st -> SKIP) ;
       a.1 -> (s.2 -> b.1 -> p_1 []
               b.1 -> s.2 -> p_1)

-- The rest of the processes in the schedulers' ring behave as
-- 'p_i(I)'. Again, 'pi(I)' is introduced to allow hiding of
-- event b.i outside of the recursion for 'pi(I)'.
pi(I)  = p_i(I) \ {b.I}
p_i(I) = s.I -> a.I -> ( s.((I%n)+1) -> b.I -> p_i(I) []
 b.I -> s.((I%n)+1) -> p_i(I) )

-- A process to compose all schedulers in the system.
-- Events s.i and st are hidden as they are implementation-
-- specific and irrelevant to the specification.
sched(I) = if (I == 1) then
             (start [|{st}|] p1) \ {st}
           else if (I == n) then
     (sched(I-1) [|{s.1,s.n}|] pi(I)) \ {s.1, s.n}
           else
             (sched(I-1) [|{s.I}|] pi(I)) \ {s.I}

-- Implementation process.
impl = sched(n)
```

Figure A.3: The CSP script for Milner's Schedulers example, variant two (continued)

```
-- Definition of specification and implementation processes
pragma spec spec
pragma impl impl

-- Number of philosophers
n = 6

range1 = {1..n}
range2 = {1..n}.{1..n}

-- Channel declarations
pragma channel sitsDown, getsUp : range1
pragma channel picksUp, putsDown: range2

-- Behaviour of the i-th philosopher.
phil(I) = sitsDown.I -> picksUp.I.I ->
          picksUp.I.(I-1+((n+1-I)/n)*n) ->
  putsDown.I.I ->
          putsDown.I.(I-1+((n+1-I)/n)*n) ->
          getsUp.I -> phil(I)

-- Behaviour of the i-th fork.
fork(I) = (picksUp.I.I -> putsDown.I.I -> fork(I)) []
  (picksUp.((I%n)+1).I ->
   putsDown.((I%n)+1).I -> fork(I))

-- Behaviour of i-th fork and philosopher together.
group(I) = (phil(I) [|{picksUp.I.I,putsDown.I.I}|] fork(I))

-- Synchronisation events for the fork between the first
-- and n-th philosophers.
final_events = {picksUp.n.n-1,picksUp.1.n,
                putsDown.n.n-1,putsDown.1.n}
```

Figure A.4: The CSP script for Dining Philosophers example

```
-- Composing group(I) together to form the system.
grouping(I) = if (I==1) then
                group(1)
              else if (I==n) then
                ( grouping(n-1) [|final_events|] group(n) )
              else
                ( grouping(I-1) [|{picksUp.I.(I-1),
                                   putsDown.I.(I-1)}|]
                  group(I) )

-- The implementation process.
impl = grouping(n)

-- The specification process simply expresses deadlock
-- freedom.
spec = events(n)

-- 'events' and 'events1' compose together a
-- nondeterministic choice over the events in which
-- each philosopher may engage.
events(I) = if (I==1) then
              events1(1)
            else
              events1(I) |~| events(I-1)
events1(I) = sitsDown.I -> spec |~|
             picksUp.I.I -> spec |~|
             picksUp.I.(I-1+((n+1-I)/n)*n) -> spec |~|
             putsDown.I.I -> spec |~|
             putsDown.I.(I-1+((n+1-I)/n)*n) -> spec |~|
             getsUp.I -> spec
```

Figure A.4: The CSP script for Dining Philosophers example (continued)

```
-- Declaration of the specification and implementation
-- processes.
pragma spec COPY
pragma impl SYSTEM

-- The data set communicated over the channels.
datasize = 5
range = {1..datasize}

-- The bit values.
bits = {0..1}

-- Compound bit and data values.
compound = {0..1}.{1..datasize}

-- Maximum number of transmission errors in the media
-- processes.
maxerrors = 50

-- Overall configuration (taken from FDR2 example)
--
--              a  PUT  b
--  left         /     \       right
--  ------> SEND        RECV ------>
--               \     /
--              d  GET  c
--

pragma channel left, right: range
pragma channel a, b: compound
pragma channel c, d: bits

-- 'PUT' models the media for communication from 'SEND' to
-- 'RECV' processes.
PUT = PUT1(maxerrors)
PUT1(Errors) = a?X?Y ->
               if (Errors == 0) then
                 b!X!Y -> PUT
               else
                 PUT1(Errors-1) |~|
```

Figure A.5: The CSP script for Alternating Bit Protocol example

```
                b!X!Y -> PUT1(Errors-1) |~|
                b!X!Y -> PUT

-- 'GET' models the media for communication from 'RECV' to
-- 'SEND' processes.
GET = GET1(maxerrors)
GET1(Errors) = c?X ->
               if (Errors == 0) then
                 d!X -> GET
               else
                 GET1(Errors-1) |~|
                 d!X -> GET1(Errors-1) |~|
                 d!X -> GET

-- 'SEND' models the sending process.
SEND = SEND1(0)
SEND1(Bit) = left?X -> SEND2(Bit, X)
SEND2(Bit, X) = a!Bit!X   -> SEND2(Bit,X) []
                d.Bit     -> SEND1(1-Bit) []
                d.(1-Bit) -> SEND2(Bit,X)

-- 'RECV' models the receiving process.
RECV = RECV1(0)
RECV1(Bit) = b.Bit?X -> right!X -> RECV1(1-Bit) []
             b.(1-Bit)?Y -> RECV1(Bit) []
             c!(1-Bit) -> RECV1(Bit)

-- The overall system put together
SYSTEM = (SEND.PUT.GET.RECV) \
         {c.0, c.1, d.0, d.1,
          a.0.1, a.0.2, a.0.3, a.0.4, a.0.5,
          a.1.1, a.1.2, a.1.3, a.1.4, a.1.5,
          b.0.1, b.0.2, b.0.3, b.0.4, b.0.5,
          b.1.1, b.1.2, b.1.3, b.1.4, b.1.5
         }

-- 'SYSTEM' should refine a single place buffer.
COPY = left?X -> right!X -> COPY
```

Figure A.5: The CSP script for Alternating Bit Protocol example (continued)

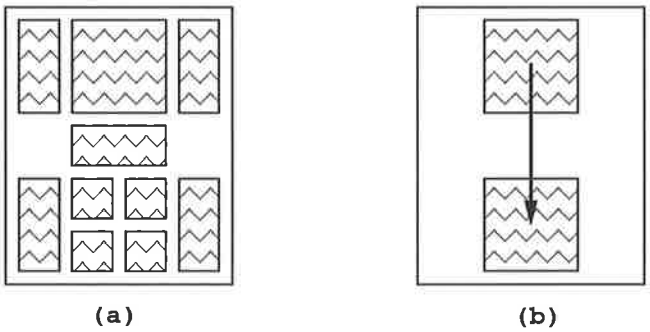


Figure A.6: The Monkey Puzzle board game

The protocol works by appending to each message a single bit that alternates between 0 and 1 with every message being sent and received successfully. The sender keeps sending a particular message with its assigned extra bit until it gets a message from the receiver acknowledging the receipt of a message with that extra bit. Similarly, upon receiving a message from the sender process the receiver continually sends acknowledge messages with the appropriate bit information back to the sender.

The CSP script for the Alternating Bit Protocol examples is in Figure A.5. Processes *PUT* and *GET* model the behaviour of a media that can lose or duplicate messages up to *maxerrors* number of times. Processes *SEND* and *RECV* model the behaviour of the sender and receiver, respectively. This example is parameterised by *datasize*, which affects the number of reachable states by changing the size of the data type passed along in a message, and *maxerrors* which controls the amount of nondeterminism present in the media processes.

A.5 Monkey Puzzle example

Monkey puzzle is a game by Holger Schemel played on a board of 4x5 square cells as shown in Figure A.6a. It is a variant of the popular 15 piece game played on a square 4x4 board. The monkey puzzle contains several pieces of unequal size, including a 2x2 piece called "monkey". The objective of this game is to put the "monkey" on its "feet" (Figure A.6b) starting from the initial placement of pieces as shown in Figure A.6a by sliding the pieces on the board sideways.

Solving the puzzle automatically involves exhaustive state space search and can be therefore carried out by a verification tool given a proper description of the puzzle. The CSP script used with ARC is adapted from a solution for FDR2 kindly provided by Bill Roscoe (see Figure A.7).

```
-- Contents of original COPYRIGHT file:
-- > This game is copyrighted (c) 1995 by Holger Schemel.
-- > Besides this, you can do what you want with it. :)

-- Declaration of the specification and implementation
-- processes.
pragma spec spec
pragma impl puzzle

Empty = 1  -- an empty square
OneSq = 2  -- a square occupied by
           -- a single square piece
HL    = 3  -- a square occupied by
           -- the left part of a horizontal bar
HR    = 4  -- a square occupied by
           -- the right part of a horizontal bar
VTop  = 5  -- a square occupied by
           -- the top part of a vertical bar
VBot  = 6  -- a square occupied by
           -- the bottom part of a vertical bar
MBL   = 7  -- a square occupied by
           -- the bottom left part of the monkey bar
MBR   = 8  -- a square occupied by
           -- the bottom right part of the monkey bar
MTL   = 9  -- a square occupied by
           -- the top left part of the monkey bar
MTR   = 10 -- a square occupied by
           -- the top right part of the monkey bar

-- Width and height
Wd = 4
Ht = 5

range = {0..Wd-1}.{0..Ht-1}

-- Channel declarations
pragma channel OneLeft,OneRight,OneUp:range
pragma channel OneDown,VLeft,VRight,VUp,VDown:range
pragma channel HLeft,HRight,HUp,HDown:range
pragma channel MLeft,MRight,MUp,MDown:range
```

Figure A.7: The CSP script for Monkey Puzzle example

```
-- Predicates
bas(I,J) = I>=0 and I<=Wd-1 and J<=Ht-1 and J>=0
-- renamed to 'or_' because 'or' is an internal symbol
or_(I,J) = I>=1 and bas(I,J)
ol(I,J) = I<Wd-1 and bas(I,J)
ou(I,J) = J>=1 and bas(I,J)
od(I,J) = (J<Ht-1) and bas(I,J)
hd(I,J) = (J<Ht-1) and (I<Wd-1) and bas(I,J)
hu(I,J) = J>=1 and (I<Wd-1) and bas(I,J)
hl(I,J) = (I<Wd-2) and bas(I,J)
hr(I,J) = (I<Wd-1) and I>=1 and bas(I,J)
vr(I,J) = (J<Ht-1) and I>=1 and bas(I,J)
vl(I,J) = (J<Ht-1) and (I<Wd-1) and bas(I,J)
vd(I,J) = (J<Ht-2) and bas(I,J)
vu(I,J) = J>=1 and (J<Ht-1) and bas(I,J)
mu(I,J) = J>=1 and (J<Ht-1) and (I<Wd-1) and bas(I,J)
md(I,J) = (J<Ht-2) and (I<Wd-1) and bas(I,J)
ml(I,J) = (I<Wd-2) and (J<Ht-1) and bas(I,J)
mr(I,J) = (I<Wd-1) and I>=1 and (J<Ht-1) and bas(I,J)

-- Specification allows all possible events except
-- the final move in a solution.
spec = run({OneDown.1.0,VUp.1.1,HRight.1.2,MLeft.1.3,
            VLeft.1.3,HDown.1.0,OneRight.1.2,MUp.1.1,
            VRight.1.2,HLeft.1.3,OneUp.1.1,HUp.1.1,
            VDown.1.0,OneLeft.1.3,MRight.1.2,MLeft.1.1,
            HRight.1.0,OneUp.1.3,MDown.1.2,OneRight.1.0,
            VLeft.1.1,VDown.1.2,HUp.1.3,OneDown.1.2,
            VRight.1.0,HRight.1.4,HDown.1.2,VUp.1.3,
            HLeft.1.1,OneLeft.1.1,OneRight.1.4,MUp.1.3,
            MLeft.1.2,OneUp.1.4,HRight.1.1,VLeft.1.2,
            HUp.1.4,OneRight.1.1,VRight.1.1,HDown.1.3,
            HLeft.1.2,OneDown.1.3,OneLeft.1.2,MRight.1.1,
            VLeft.1.0,VRight.1.3,HUp.1.2,VDown.1.1,
            HLeft.1.4,OneLeft.1.4,OneUp.1.2,MRight.1.3,
            MDown.1.1,HDown.1.1,HLeft.1.0,OneRight.1.3,
            MUp.1.2,OneLeft.1.0,OneDown.1.1,VUp.1.2,
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
            HRight.1.3,OneRight.3.1,OneUp.3.4,OneDown.3.3,
            VRight.3.1,OneUp.3.2,VRight.3.3,VDown.3.1,
            OneRight.3.3,OneDown.3.1,VUp.3.2,OneUp.3.3,
            OneRight.3.0,VDown.3.2,OneRight.3.4,VRight.3.0,
            OneDown.3.2,VUp.3.3,OneUp.3.1,VRight.3.2,VDown.3.0,
            OneRight.3.2,OneDown.3.0,VUp.3.1,VUp.0.1,MLeft.0.3,
            OneDown.0.0,VLeft.0.3,HDown.0.0,MUp.0.1,HLeft.0.3,
            OneUp.0.1,MDown.0.0,HUp.0.1,OneLeft.0.3,VDown.0.0,
            MLeft.0.1,OneUp.0.3,MDown.0.2,VLeft.0.1,HUp.0.3,
            VDown.0.2,HDown.0.2,HLeft.0.1,MUp.0.3,OneDown.0.2,
            OneLeft.0.1,VUp.0.3,MLeft.0.2,OneUp.0.4,VLeft.0.2,
            HUp.0.4,HLeft.0.2,HDown.0.3,OneLeft.0.2,
            OneDown.0.3,MLeft.0.0,HLeft.0.4,OneUp.0.2,
            MDown.0.1,OneLeft.0.4,VLeft.0.0,VDown.0.1,HUp.0.2,
            HDown.0.1,HLeft.0.0,MUp.0.2,OneDown.0.1,
            OneLeft.0.0,VUp.0.2,OneUp.2.4,HRight.2.1,VLeft.2.2,
            HUp.2.4,OneRight.2.1,VRight.2.1,HDown.2.3,
            OneDown.2.3,OneLeft.2.2,MRight.2.1,VRight.2.3,
            OneUp.2.2,MDown.2.1,VLeft.2.0,HUp.2.2,VDown.2.1,
            OneLeft.2.4,MRight.2.3,HDown.2.1,OneRight.2.3,
            MUp.2.2,OneLeft.2.0,OneDown.2.1,VUp.2.2,HRight.2.3,
            HRight.2.0,OneUp.2.3,MDown.2.2,OneRight.2.0,
            VLeft.2.1,VDown.2.2,HUp.2.3,VRight.2.0,HDown.2.2,
            OneRight.2.4,MUp.2.3,OneDown.2.2,HRight.2.4,VUp.2.3,
            OneLeft.2.1,MRight.2.0,VRight.2.2,OneUp.2.1,
            MDown.2.0,HUp.2.1,VDown.2.0,OneLeft.2.3,MRight.2.2,
            VLeft.2.3,HDown.2.0,OneRight.2.2,MUp.2.1,
            OneDown.2.0,HRight.2.2,VUp.2.1})

-- The composition of the puzzle.
puzzle =
 SquareVTop(0,4).SquareMTL(1,4).SquareMTR(2,4).SquareVTop(3,4).
 SquareVBot(0,3).SquareMBL(1,3).SquareMBR(2,3).SquareVBot(3,3).
 SquareEmpty(0,2).SquareHL(1,2).SquareHR(2,2).SquareEmpty(3,2).
 SquareVTop(0,1).SquareOneSq(1,1).SquareOneSq(2,1).SquareVTop(3,1).
 SquareVBot(0,0).SquareOneSq(1,0).SquareOneSq(2,0).SquareVBot(3,0)

-- Implementation process.
impl = puzzle
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
-- The behaviour of an empty square.
SquareEmpty(I,J) = if or_(I,J) then
                     OneRight.I.J ->
                     SquareOneSq(I,J) [] SquareEmpty_1(I,J)
                   else
                     SquareEmpty_1(I,J)
SquareEmpty_1(I,J) = if ol(I,J) then
                       OneLeft.I.J ->
                       SquareOneSq(I,J) [] SquareEmpty_2(I,J)
                     else
                       SquareEmpty_2(I,J)
SquareEmpty_2(I,J) = if vl(I,J) then
                       VLeft.I.J ->
                       SquareVBot(I,J) [] SquareEmpty_3(I,J)
                     else
                       SquareEmpty_3(I,J)
SquareEmpty_3(I,J) = if vr(I,J) then
                       VRight.I.J ->
                       SquareVBot(I,J) [] SquareEmpty_4(I,J)
                     else
                       SquareEmpty_4(I,J)
SquareEmpty_4(I,J) = if vl(I,J-1) then
                       VLeft.I.J-1 ->
                       SquareVTop(I,J) [] SquareEmpty_5(I,J)
                     else
                       SquareEmpty_5(I,J)
SquareEmpty_5(I,J) = if vr(I,J-1) then
                       VRight.I.J-1 ->
                       SquareVTop(I,J) [] SquareEmpty_6(I,J)
                     else
                       SquareEmpty_6(I,J)
SquareEmpty_6(I,J) = if hl(I,J) then
                       HLeft.I.J ->
                       SquareHL(I,J) [] SquareEmpty_7(I,J)
                     else
                       SquareEmpty_7(I,J)
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
SquareEmpty_7(I,J) = if hr(I-1,J) then
                       HRight.I-1.J ->
                       SquareHR(I,J) [] SquareEmpty_8(I,J)
                     else
                       SquareEmpty_8(I,J)
SquareEmpty_8(I,J) = if ml(I,J) then
                       MLeft.I.J ->
                       SquareMBL(I,J) [] SquareEmpty_9(I,J)
                     else
                       SquareEmpty_9(I,J)
SquareEmpty_9(I,J) = if mr(I-1,J) then
                       MRight.I-1.J ->
                       SquareMBR(I,J) [] SquareEmpty_10(I,J)
                     else
                       SquareEmpty_10(I,J)
SquareEmpty_10(I,J) = if ml(I,J-1) then
                        MLeft.I.J-1 ->
                        SquareMTL(I,J) [] SquareEmpty_11(I,J)
                      else
                        SquareEmpty_11(I,J)
SquareEmpty_11(I,J) = if mr(I-1,J-1) then
                        MRight.I-1.J-1 ->
                        SquareMTR(I,J) [] SquareEmpty_12(I,J)
                      else
                        SquareEmpty_12(I,J)
SquareEmpty_12(I,J) = if od(I,J) then
                        OneDown.I.J ->
                        SquareOneSq(I,J) [] SquareEmpty_13(I,J)
                      else
                        SquareEmpty_13(I,J)
SquareEmpty_13(I,J) = if ou(I,J) then
                        OneUp.I.J ->
                        SquareOneSq(I,J) [] SquareEmpty_14(I,J)
                      else
                        SquareEmpty_14(I,J)
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
SquareEmpty_14(I,J) = if hu(I,J) then
                        HUp.I.J ->
                        SquareHL(I,J) [] SquareEmpty_15(I,J)
                      else
                        SquareEmpty_15(I,J)
SquareEmpty_15(I,J) = if hd(I,J) then
                        HDown.I.J ->
                        SquareHL(I,J) [] SquareEmpty_16(I,J)
                      else
                        SquareEmpty_16(I,J)
SquareEmpty_16(I,J) = if hu(I-1,J) then
                        HUp.I-1.J ->
                        SquareHR(I,J) [] SquareEmpty_17(I,J)
                      else
                        SquareEmpty_17(I,J)
SquareEmpty_17(I,J) = if hd(I-1,J) then
                        HDown.I-1.J ->
                        SquareHR(I,J) [] SquareEmpty_18(I,J)
                      else
                        SquareEmpty_18(I,J)
SquareEmpty_18(I,J) = if vd(I,J) then
                        VDown.I.J ->
                        SquareVBot(I,J) [] SquareEmpty_19(I,J)
                      else
                        SquareEmpty_19(I,J)
SquareEmpty_19(I,J) = if vu(I,J-1) then
                        VUp.I.J-1 ->
                        SquareVTop(I,J) [] SquareEmpty_20(I,J)
                      else
                        SquareEmpty_20(I,J)
SquareEmpty_20(I,J) = if md(I,J) then
                        MDown.I.J ->
                        SquareMBL(I,J) [] SquareEmpty_21(I,J)
                      else
                        SquareEmpty_21(I,J)
SquareEmpty_21(I,J) = if md(I-1,J) then
                        MDown.I-1.J ->
                        SquareMBR(I,J) [] SquareEmpty_22(I,J)
                      else
                        SquareEmpty_22(I,J)
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
SquareEmpty_22(I,J) = if mu(I,J-1) then
                        MUp.I.J-1 ->
                        SquareMTL(I,J) [] SquareEmpty_23(I,J)
                      else
                        SquareEmpty_23(I,J)
SquareEmpty_23(I,J) = if mu(I-1,J-1) then
                        MUp.I-1.J-1 -> SquareMTR(I,J)
                      else
                        STOP

-- The behaviour of a square with a single square piece.
SquareOneSq(I,J) = if ou(I,J+1) then
                     OneUp.I.J+1 ->
                     SquareEmpty(I,J) [] SquareOneSq_1(I,J)
                   else
                     SquareOneSq_1(I,J)
SquareOneSq_1(I,J) = if od(I,J-1) then
                       OneDown.I.J-1 ->
                       SquareEmpty(I,J) [] SquareOneSq_2(I,J)
                     else
                       SquareOneSq_2(I,J)
SquareOneSq_2(I,J) = if or_(I+1,J) then
                       OneRight.I+1.J ->
                       SquareEmpty(I,J) [] SquareOneSq_3(I,J)
                     else
                       SquareOneSq_3(I,J)
SquareOneSq_3(I,J) = if ol(I-1,J) then
                       OneLeft.I-1.J -> SquareEmpty(I,J)
                     else
                       STOP

-- The behaviour of a square with the left part of a
-- horizontal bar.
SquareHL(I,J) = if hu(I,J+1) then
                  HUp.I.J+1 ->
                  SquareEmpty(I,J) [] SquareHL_1(I,J)
                else
                  SquareHL_1(I,J)
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
SquareHL_1(I,J) = if hd(I,J-1) then
                    HDown.I.J-1 ->
                    SquareEmpty(I,J) [] SquareHL_2(I,J)
                  else
                    SquareHL_2(I,J)
SquareHL_2(I,J) = if hr(I+1,J) then
                    HRight.I+1.J ->
                    SquareEmpty(I,J) [] SquareHL_3(I,J)
                  else
                    SquareHL_3(I,J)
SquareHL_3(I,J) = if hl(I-1,J) then
                    HLeft.I-1.J -> SquareHR(I,J)
                  else
                    STOP

-- The behaviour of a square with the right part of a
-- horizontal bar.
SquareHR(I,J) = if hu(I-1,J+1) then
                  HUp.I-1.J+1 ->
                  SquareEmpty(I,J) [] SquareHR_1(I,J)
                else
                  SquareHR_1(I,J)
SquareHR_1(I,J) = if hd(I-1,J-1) then
                    HDown.I-1.J-1 ->
                    SquareEmpty(I,J) [] SquareHR_2(I,J)
                  else
                    SquareHR_2(I,J)
SquareHR_2(I,J) = if hr(I,J) then
                    HRight.I.J ->
                    SquareHL(I,J) [] SquareHR_3(I,J)
                  else
                    SquareHR_3(I,J)
SquareHR_3(I,J) = if hl(I-2,J) then
                    HLeft.I-2.J -> SquareEmpty(I,J)
                  else
                    STOP
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
-- The behaviour of a square with the bottom part of a
-- vertical bar.
SquareVBot(I,J) = if vu(I,J+1) then
                    VUp.I.J+1 ->
                    SquareEmpty(I,J) [] SquareVBot_1(I,J)
                  else
                    SquareVBot_1(I,J)
SquareVBot_1(I,J) = if vd(I,J-1) then
                      VDown.I.J-1 ->
                      SquareVTop(I,J) [] SquareVBot_2(I,J)
                    else
                      SquareVBot_2(I,J)
SquareVBot_2(I,J) = if vr(I+1,J) then
                      VRight.I+1.J ->
                      SquareEmpty(I,J) [] SquareVBot_3(I,J)
                    else
                      SquareVBot_3(I,J)
SquareVBot_3(I,J) = if vl(I-1,J) then
                      VLeft.I-1.J -> SquareEmpty(I,J)
                    else
                      STOP

-- The behaviour of a square with the top part of a
-- vertical bar.
SquareVTop(I,J) = if vu(I,J) then
                    VUp.I.J ->
                    SquareVBot(I,J) [] SquareVTop_1(I,J)
                  else
                    SquareVTop_1(I,J)
SquareVTop_1(I,J) = if vd(I,J-2) then
                      VDown.I.J-2 ->
                      SquareEmpty(I,J) [] SquareVTop_2(I,J)
                    else
                      SquareVTop_2(I,J)
SquareVTop_2(I,J) = if vr(I+1,J-1) then
                      VRight.I+1.J-1 ->
                      SquareEmpty(I,J) [] SquareVTop_3(I,J)
                    else
                      SquareVTop_3(I,J)
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
SquareVTop_3(I,J) = if vl(I-1,J-1) then
                      VLeft.I-1.J-1 -> SquareEmpty(I,J)
                    else
                      STOP

-- The behaviour of monkey bottom left.
SquareMBL(I,J) = if mu(I,J+1) then
                   MUp.I.J+1 ->
                   SquareEmpty(I,J) [] SquareMBL_1(I,J)
                 else
                   SquareMBL_1(I,J)
SquareMBL_1(I,J) = if md(I,J-1) then
                     MDown.I.J-1 ->
                     SquareMTL(I,J) [] SquareMBL_2(I,J)
                   else
                     SquareMBL_2(I,J)
SquareMBL_2(I,J) = if mr(I+1,J) then
                     MRight.I+1.J ->
                     SquareEmpty(I,J) [] SquareMBL_3(I,J)
                   else
                     SquareMBL_3(I,J)
SquareMBL_3(I,J) = if ml(I-1,J) then
                     MLeft.I-1.J -> SquareMBR(I,J)
                   else
                     STOP

-- The behaviour of monkey bottom right.
SquareMBR(I,J) = if mu(I-1,J+1) then
                   MUp.I-1.J+1 ->
                   SquareEmpty(I,J) [] SquareMBR_1(I,J)
                 else
                   SquareMBR_1(I,J)
SquareMBR_1(I,J) = if md(I-1,J-1) then
                     MDown.I-1.J-1 ->
                     SquareMTR(I,J) [] SquareMBR_2(I,J)
                   else
                     SquareMBR_2(I,J)
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
SquareMBR_2(I,J) = if mr(I,J) then
                     MRight.I.J ->
                     SquareMBL(I,J) [] SquareMBR_3(I,J)
                   else
                     SquareMBR_3(I,J)

SquareMBR_3(I,J) = if ml(I-2,J) then
                     MLeft.I-2.J -> SquareEmpty(I,J)
                   else
                     STOP

-- The behaviour of monkey top left.
SquareMTL(I,J) = if mu(I,J) then
                   MUp.I.J ->
                   SquareMBL(I,J) [] SquareMTL_1(I,J)
                 else
                   SquareMTL_1(I,J)

SquareMTL_1(I,J) = if md(I,J-2) then
                     MDown.I.J-2 ->
                     SquareEmpty(I,J) [] SquareMTL_2(I,J)
                   else
                     SquareMTL_2(I,J)

SquareMTL_2(I,J) = if mr(I+1,J-1) then
                     MRight.I+1.J-1 ->
                     SquareEmpty(I,J) [] SquareMTL_3(I,J)
                   else
                     SquareMTL_3(I,J)

SquareMTL_3(I,J) = if ml(I-1,J-1) then
                     MLeft.I-1.J-1 -> SquareMTR(I,J)
                   else
                     STOP
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

```
-- The behaviour of monkey top right.
SquareMTR(I,J) = if mu(I-1,J) then
                   MUp.I-1.J ->
                   SquareMBR(I,J) [] SquareMTR_1(I,J)
                 else
                   SquareMTR_1(I,J)

SquareMTR_1(I,J) = if md(I-1,J-2) then
                     MDown.I-1.J-2 ->
                     SquareEmpty(I,J) [] SquareMTR_2(I,J)
                   else
                     SquareMTR_2(I,J)

SquareMTR_2(I,J) = if mr(I,J-1) then
                     MRight.I.J-1 ->
                     SquareMTL(I,J) [] SquareMTR_3(I,J)
                   else
                     SquareMTR_3(I,J)

SquareMTR_3(I,J) = if ml(I-2,J-1) then
                     MLeft.I-2.J-1 -> SquareEmpty(I,J)
                   else
                     STOP
```

Figure A.7: The CSP script for Monkey Puzzle example (continued)

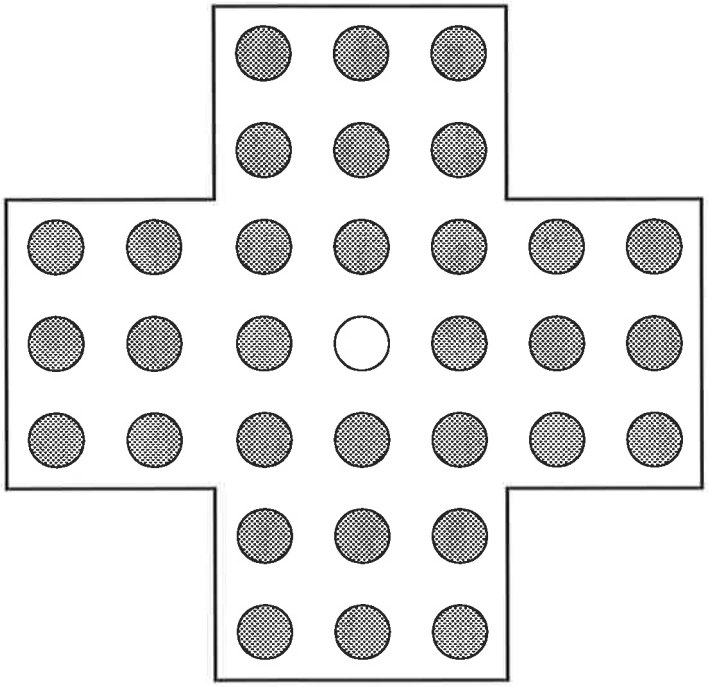


Figure A.8: The Solitaire board game

A.6 Solitaire example

The Solitaire example represents a well-known board puzzle the initial configuration of which is shown in Figure A.8. The game area consists of 33 holes in which 32 pegs (denoted by gray filled circles) are places so that the centre hole remains empty. Pegs can hop over other pegs adjacent to them (above, below, to the left or right) provided that the destination of the hop is not occupied by another peg; the peg over which the hop is performed is removed from the board. The goal is to find a sequence of moves that leaves a single peg on the board. Since there are 32 pegs on the board, a winning sequence would entail exactly 31 moves.

The CSP script given in Figure A.9 is based on a FDR2 script kindly provided by Bill Roscoe. The range of possible moves on the table is encoded as a set of events; each event contains information regarding the direction of the move and the coordinates of the hole where the hopping peg has "landed". Each hole is modelled through a separate process containing two states (empty and occupied) and as many transitions as necessary to encode all possible state transitions. A move involves the participation of exactly three holes: a "source" hole that is initially occupied and then emptied, a "hop" hole which contains a peg before

the move and is empty after the move is completed, and a "destination" hole that is empty at first but becomes full at the end of the move. The necessary synchronisation between the state changes of the holes involved in a move is achieved by putting in parallel of all hole processes.

Solitaire is an example with a very complex communication pattern and a very high number of reachable states—over 190 million despite the simplicity of each hole process. Thus, it represents a real challenge for any advanced reachability analysis techniques.

```
-- peg solitaire for FDR

-- Bill Roscoe, November 1992
-- adapted for ARC by Atanas Parashkevov, October 1997

-- An old and familiar puzzle:

--                XXX     6
--                XXX     5
--              XXXXXXX   4
--              XXX XXX   3
--              XXXXXXX   2
--                XXX     1
--                XXX     0
--                        Y
--              0123456 X

-- A peg can hop into an empty hole over one other, removing
-- the one it has hopped over. The numbers are the co-ordinate
-- system we use in the rest of this script. To get simple
-- defnitions, we extend the co-ordinates by 2 in each direction.

-- Definition of specification and implementation processes.
pragma spec SPEC
pragma impl IMPL

Range = {(0-2)..8}.{(0-2)..8}

-- The events represent hops in the stated direction to the
-- co-ordinates mentioned. Thus right.3.3 is the peg in
-- (1,3) (X is always the first co-ordinate) hopping to
-- (3,3) over (2,3).

pragma channel up,down,left,right:Range

-- The occurence of 'done' denotes success in solving the
-- puzzle.
pragma channel done
```

Figure A.9: The CSP script for Solitaire example

```
EMPTY = 1
FULL  = 2

SPEC = STOP

-- We will have a process for each hole, which is in state
-- Full or Empty, and allows the 12 events it can participate
-- in appropriately (The co-ordinates like -2 and 7 are there
-- to allow this definition to be used everywhere.)

-- Each hop will be synchronised on by three holes=processes:
-- from, over and to. Note that from and over start full and
-- become empty, to starts empty and becomes full.

Empty(I,J,Tg) = left.I.J  -> Full(I,J,Tg)
             [] down.I.J  -> Full(I,J,Tg)
             [] up.I.J    -> Full(I,J,Tg)
             [] right.I.J -> Full(I,J,Tg)
             [] (if (Tg == EMPTY) then
                   done -> Empty(I,J,Tg)
                 else
                   STOP)

Full(I,J,Tg) = left.(I-2).J -> Empty(I,J,Tg)
            [] down.I.(J-2)  -> Empty(I,J,Tg)
            [] up.I.(J+2)    -> Empty(I,J,Tg)
            [] right.(I+2).J -> Empty(I,J,Tg)
            [] left.(I-1).J  -> Empty(I,J,Tg)
            [] down.I.(J-1)  -> Empty(I,J,Tg)
            [] up.I.(J+1)    -> Empty(I,J,Tg)
            [] right.(I+1).J -> Empty(I,J,Tg)
            [] (if (Tg == FULL) then
                  done -> Full(I,J,Tg)
                else
                  STOP)
```

Figure A.9: The CSP script for Solitaire example (continued)

```
-- The various interface sets used to put the puzzle
-- together are as follows:

C02 = {up.0.4,down.0.2,done}
C03 = {up.0.4,down.0.2,done}
C12 = {up.1.4,down.1.2,done}
C13 = {up.1.4,down.1.2,done}
C52 = {up.5.4,down.5.2,done}
C53 = {up.5.4,down.5.2,done}
C62 = {up.6.4,down.6.2,done}
C63 = {up.6.4,down.6.2,done}
C22 = {up.2.3,up.2.4,down.2.2,down.2.1,done}
C23 = {up.2.4,up.2.5,down.2.3,down.2.2,done}
C21 = {up.2.2,up.2.3,down.2.1,down.2.0,done}
C24 = {up.2.6,up.2.5,down.2.3,down.2.4,done}
C20 = {up.2.2,down.2.0,done}
C25 = {up.2.6,down.2.4,done}
C32 = {up.3.3,up.3.4,down.3.2,down.3.1,done}
C33 = {up.3.4,up.3.5,down.3.3,down.3.2,done}
C31 = {up.3.2,up.3.3,down.3.1,down.3.0,done}
C34 = {up.3.6,up.3.5,down.3.3,down.3.4,done}
C30 = {up.3.2,down.3.0,done}
C35 = {up.3.6,down.3.4,done}
C42 = {up.4.3,up.4.4,down.4.2,down.4.1,done}
C43 = {up.4.4,up.4.5,down.4.3,down.4.2,done}
C41 = {up.4.2,up.4.3,down.4.1,down.4.0,done}
C44 = {up.4.6,up.4.5,down.4.3,down.4.4,done}
C40 = {up.4.2,down.4.0,done}
C45 = {up.4.6,down.4.4,done}

C2 = {up.2.2,up.2.3,down.2.1,down.2.0,
      up.2.4,up.2.5,down.2.3,down.2.2,
      up.2.6,down.2.4,done}

C3 = {up.3.2,up.3.3,down.3.1,down.3.0,
      up.3.4,up.3.5,down.3.3,down.3.2,
      up.3.6,down.3.4,done}
```

Figure A.9: The CSP script for Solitaire example (continued)

```
C4 = {up.4.2,up.4.3,down.4.1,down.4.0,
      up.4.4,up.4.5,down.4.3,down.4.2,
      up.4.6,down.4.4,done}

CE = {up.0.4,down.0.2,up.1.4,down.1.2,
      up.2.2,up.2.3,down.2.1,down.2.0,
      up.2.4,up.2.5,down.2.3,down.2.2,
      up.2.6,down.2.4,up.3.2,up.3.3,
      down.3.1,down.3.0,up.3.4,up.3.5,
      down.3.3,down.3.2,up.3.6,down.3.4,
      up.4.2,up.4.3,down.4.1,down.4.0,
      up.4.4,up.4.5,down.4.3,down.4.2,
      up.4.6,down.4.4,up.5.4,down.5.2,
      up.6.4,down.6.2,done}

R0 = {done,
      left.0.2,left.0.3,left.0.4,
      right.2.2,right.2.3,right.2.4}

R1 = {done,
      left.0.2,left.0.3,left.0.4,
      left.1.2,left.1.3,left.1.4,
      right.2.2,right.2.3,right.2.4,
      right.3.2,right.3.3,right.3.4}

R2 = {done,
      right.3.2,right.3.3,right.3.4,
      left.1.2,left.1.3,left.1.4,
      left.2.0,left.2.1,left.2.2,left.2.3,
      left.2.4,left.2.5,left.2.6,
      right.4.0,right.4.1,right.4.2,right.4.3,
      right.4.4,right.4.5,right.4.6}

R5 = {done,
      left.4.2,left.4.3,left.4.4,
      right.6.2,left.4.3,left.4.4}
```

Figure A.9: The CSP script for Solitaire example (continued)

```
R3 = {done,
      right.5.2,right.5.3,right.5.4,
      left.3.2,left.3.3,left.3.4,
      left.2.0,left.2.1,left.2.2,
      left.2.3,left.2.4,left.2.5,left.2.6,
      right.4.0,right.4.1,right.4.2,right.4.3,
      right.4.4,right.4.5,right.4.6}

R4 = {done,
      left.4.2,left.4.3,left.4.4,
      left.3.2,left.3.3,left.3.4,
      right.5.2,right.5.3,right.5.4,
      right.6.2,right.6.3,right.6.4}

RE = {done,
      left.0.2,left.0.3,left.0.4,
      left.1.2,left.1.3,left.1.4,
      right.2.2,right.2.3,right.2.4,
      right.3.2,right.3.3,right.3.4,
      right.5.2,right.5.3,right.5.4,
      left.3.2,left.3.3,left.3.4,
      left.2.0,left.2.1,left.2.2,
      left.2.3,left.2.4,left.2.5,left.2.6,
      right.4.0,right.4.1,right.4.2,
      right.4.3,right.4.4,right.4.5,right.4.6,
      left.4.2,left.4.3,left.4.4,
      right.6.2,left.4.3,left.4.4}

-- So CE and RE are the sets of all vertical and horizontal
-- actions ever possible in the puzzle, so that ...

-- OUTSIDE = diff(Sigma,union(RE,CE))

-- is the set of events that should never happen.
```

Figure A.9: The CSP script for Solitaire example (continued)

```
-- The puzzle is formed by combining 33 of these processes
-- together, first in columns ...

OUTSIDE = {done,
           up.0.2, up.0.3, up.0.5, up.0.6,
           up.1.2, up.1.3, up.1.5, up.1.6,
           up.2.0, up.2.1, up.2.7, up.2.8,
           up.3.0, up.3.1, up.3.7, up.3.8,
           up.4.0, up.4.1, up.4.7, up.4.8,
           up.5.2, up.5.3, up.5.5, up.5.6,
           up.6.2, up.6.3, up.6.5, up.6.6,

           down.0.0, down.0.1, down.0.3, down.0.4,
           down.1.0, down.1.1, down.1.3, down.1.4,
           down.2.(0-2), down.2.(0-1), down.2.5, down.2.6,
           down.3.(0-2), down.3.(0-1), down.3.5, down.3.6,
           down.4.(0-2), down.4.(0-1), down.4.5, down.4.6,
           down.5.0, down.5.1, down.5.3, down.5.4,
           down.6.0, down.6.1, down.6.3, down.6.4,

           right.0.2, right.0.3, right.0.4,
           right.1.2, right.1.3, right.1.4,
           right.2.0, right.2.1, right.2.5, right.2.6,
           right.3.0, right.3.1, right.3.5, right.3.6,
           right.5.0, right.5.1, right.5.5, right.5.6,
           right.6.0, right.6.1, right.6.5, right.6.6,
           right.7.2, right.7.3, right.7.4,
           right.8.2, right.8.3, right.8.4,

           left.(0-2).2, left.(0-2).3, left.(0-2).4,
           left.(0-1).2, left.(0-1).3, left.(0-1).4,
           left.0.0, left.0.1, left.0.5, left.0.6,
           left.1.0, left.1.1, left.1.5, left.1.6,
           left.3.0, left.3.1, left.3.5, left.3.6,
           left.4.0, left.4.1, left.4.5, left.4.6,
           left.5.2, left.5.3, left.5.4,
           left.6.2, left.6.3, left.6.4}
```

Figure A.9: The CSP script for Solitaire example (continued)

```
SIGMA = {up.0.3,left.3.4,left.4.1,right.0.2,down.2.2,
         right.5.3,right.4.6,up.5.4,right.2.4,up.3.2,
         left.6.3,up.2.5,right.3.1,left.1.2,down.4.4,
         down.3.3,right.6.4,down.4.0,down.2.6,right.1.3,
         up.2.1,left.5.2,up.1.4,left.4.5,right.2.0,
         down.6.2,left.3.0,left.2.3,up.3.6,right.4.2,
         down.0.4,right.3.5,up.4.3,down.5.3,left.1.4,
         down.4.6,left.2.1,down.0.2,right.3.3,up.4.1,
         right.2.6,up.3.4,right.4.0,right.0.4,up.1.2,
         left.4.3,left.3.6,down.3.1,right.6.2,up.6.3,
         down.2.4,down.1.3,right.4.4,up.5.2,up.4.5,
         down.2.0,left.3.2,down.6.4,left.2.5,down.4.2,
         left.0.3,down.3.5,right.2.2,up.3.0,up.2.3,
         left.5.4,down.6.3,left.3.1,left.2.4,down.1.2,
         right.4.3,right.3.6,up.4.4,right.1.4,up.2.2,
         left.5.3,left.4.6,right.2.1,down.4.1,left.0.2,
         down.3.4,down.2.3,right.5.4,up.6.2,down.3.0,
         left.4.2,right.0.3,up.0.4,left.3.5,down.5.2,
         left.1.3,down.4.5,left.2.0,up.2.6,right.3.2,
         up.4.0,right.2.5,up.3.3,left.6.4,down.4.3,
         left.0.4,down.3.6,right.2.3,up.3.1,left.6.2,
         up.2.4,right.3.0,up.0.2,left.3.3,left.4.0,
         left.2.6,up.4.6,down.2.1,right.5.2,down.1.4,
         right.4.5,up.5.3,down.0.3,right.3.4,up.4.2,
         up.3.5,right.4.1,left.2.2,down.5.4,down.3.2,
         right.6.3,up.6.4,down.2.5,right.1.2,up.2.0,
         up.1.3,left.4.4}

-- All events except for the 'done' event are hidden from
-- 'PUZZLE'.
IMPL = PUZZLE \ SIGMA

PUZZLE = done -> STOP [| OUTSIDE |]
         (COL0.COL1.COL2.COL3.COL4.COL5.COL6)
```

Figure A.9: The CSP script for Solitaire example (continued)

```
-- The following process is similar to PUZ but has the
-- first four moves leading to a solution "hinted".
RPUZ = left.3.3 -> up.4.3 -> right.4.2 -> down.4.1 ->
       left.4.2 -> run(SIGMA) [|SIGMA|] PUZ

COL0 = (Full(0,2,EMPTY)[|C02|]Full(0,3,EMPTY))[|C03|]
        Full(0,4,EMPTY)

COL1 = (Full(1,2,EMPTY)[|C12|]Full(1,3,EMPTY))[|C13|]
        Full(1,4,EMPTY)

COL2 = (Full(2,0,EMPTY)[|C20|]((Full(2,1,EMPTY)[|C21|]
       (Full(2,2,EMPTY)[|C22|]Full(2,3,EMPTY)))[|C23|]
        Full(2,4,EMPTY)))[|C24|]
       (Full(2,5,EMPTY)[|C25|]Full(2,6,EMPTY))

COL3 = (Full(3,0,EMPTY)[|C30|]((Full(3,1,EMPTY)[|C31|]
       (Full(3,2,EMPTY)[|C32|]Empty(3,3,FULL)))[|C33|]
        Full(3,4,EMPTY)))[|C34|]
       (Full(3,5,EMPTY)[|C35|]Full(3,6,EMPTY))

COL4 = (Full(4,0,EMPTY)[|C40|]((Full(4,1,EMPTY)[|C41|]
       (Full(4,2,EMPTY)[|C42|]Full(4,3,EMPTY)))[|C43|]
        Full(4,4,EMPTY)))[|C44|]
       (Full(4,5,EMPTY)[|C45|]Full(4,6,EMPTY))

COL5 = (Full(5,2,EMPTY)[|C52|]Full(5,3,EMPTY))[|C53|]
        Full(5,4,EMPTY)

COL6 = (Full(6,2,EMPTY)[|C62|]Full(6,3,EMPTY))[|C63|]
        Full(6,4,EMPTY)
```

Figure A.9: The CSP script for Solitaire example (continued)

Bibliography

[AHR98] R. Alur, T. A. Henzinger, and S. K. Rajamani. Symbolic exploration of transition hierarchies. In *TACAS '98*, pages 330–344. Lecture Notes in Computer Science, 1384, Springer Verlag, 1998.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. on Computers*, 27(6):509–516, June 1978.

[ATB94] A. Aziz, S. Tasiran, and R. K. Brayton. BDD variable ordering for interacting finite state machines. In *DAC'94*, pages 283–288, 1994.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS '99*, pages 193–207. Lecture Notes in Computer Science, 1579, Springer Verlag, 1999.

[BCL91a] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *DAC '91*, pages 403–407, 1991.

[BCL91b] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI '91*, pages 49–58, 1991.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *DAC '90*, pages 46–51, June 1990.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *CAV '92*, pages 96–108. Lecture Notes in Computer Science, 663, Springer Verlag, 1992.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[BK95] A. Biere and A. Kick. A case study on different modeling approaches based on model checking—verifying numerous versions of the alternating bit protocol with SMV. Technical Report IRATR-1995-5, University of Karlsruhe, 1995.

[BMTV94] G. Barrett, M. N. Marcigaglia, G. Tateo, and M. Vaccari. A tool for checking refinement between finite-state CSP processes. In *Transputer Applications and Systems '94, Proceedings of the 1994 World Transputer Congress*. IOS Press, 1994.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. Computers*, C-35(8):677–691, August 1986.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Bry95] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *ICCAD '95*, pages 236–243, November 1995.

[Bry96] R. E. Bryant. Bit-level analysis of an SRT divider circuit. In *DAC '96*, pages 661–665, 1996.

[Buc62] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congress on Logic, Method and Philos. Sci. 1960*, pages 1–12. Stanford University Press, 1962.

[Cao97] T. T. Cao. Extensions and enhancements of the Adelaide Refinement Checker. Master's thesis, Dept. of Computer Science, University of Adelaide, Australia, 1997.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, pages 365–373. Lecture Notes in Computer Science, 407, Springer Verlag, June 1989.

[CCFP95] F. Corno, M. Cusinato, M. Ferrero, and P. Prinetto. Proving testing preorders for process algebra descriptions. In *EDTC '95: IEEE European Design and Test Conference*. IEEE Press, March 1995.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchela, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV '2001*, pages 436–453. Lecture Notes in Computer Science, 2102, Springer Verlag, 2001.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *POPL'92*, pages 343–354. ACM Press, 1992.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolovsky, and S. Zhang. The concurrency factory—practical tools for specification, simulation, verification, and implementation of concurrent systems. In *DIMACS Workshop on the Specification of Parallel Algorithms, Princeton, NJ*, pages 75–90, 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CGZ96] E. Clarke, S. German, and X. Zhao. Verifying the SRT division algorithm using theorem-proving techniques. In *CAV '96*, pages 111–122. Lecture Notes in Computer Science, 1102, Springer Verlag, 1996.

[CJMW96] E. M. Clarke and et al. J. M. Wing. Formal methods: State of the art and future directions. *ACM Computer Surveys*, 28(4):626–643, December 1996.

[CK96] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, June 1996.

[CM77] E. Cerny and M. A. Marin. An approach to unified methodology of combinational switching circuits. *IEEE Trans. Computers*, C-26(8):745–756, August 1977.

[Coh81] J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13:341–367, 1981.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *Proc. of Third Annual ACM Symp. on Theory of Computing*, 1971.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1):36–72, January 1993.

[DB95] A. Dsouza and B. Bloom. Generating BDD models for process algebra terms. In *CAV '95*, pages 16–30. Lecture Notes in Computer Science, 939, Springer Verlag, 1995.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Dym95] K. Dymond. *A Guide to the CMM: Understanding the Capability Maturity Model for Software.* Process Inc. US, 1995.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In *CAV '91*, pages 203–213. Lecture Notes in Computer Science, 575, Springer Verlag, 1991.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL—Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

[Flo66] R. W. Floyd. Assigning meanings to programs. In 19^{th} *Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1966.

[FM91] J.-C. Fernandez and L. Mounier. A tool set for deciding behavioral equivalences. In *CONCUR '91*, pages 23–42. Lecture Notes in Computer Science, 527, Springer Verlag, 1991.

[For93] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 1.3*, August 1993.

[For97] Formal Systems (Europe) Ltd. *FDR2 User Manual*, October 1997.

[GE99] P. Gao and R. Esser. Adding dedicated model checking to the Adelaide Refinement Checker. Manuscript in preparation, Dept. of Computer Science, University of Adelaide, Australia, 1999.

[GH93] P. Godefroid and G. Holzmann. On the verification of temporal properties. In *Proc. 13th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 109–124. North-Holland, 1993.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *CAV '92*, pages 178–191. Lecture Notes in Computer Science, 663, Springer Verlag, 1992.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90*, pages 176–185. Lecture Notes in Computer Science, 531, Springer Verlag, 1990.

[God94] P. Godefroid. *Partial-order methods for the verification of concurrent systems.* PhD thesis, University of Liège, Belgium, 1994.

[Har90] P. H. Hartel. Comparison of three garbage collection algorithms. *Structured Programming*, 11(3):117–128, 1990.

[Hea94] G. E. Head. Six-sigma software using cleanroom software engineering techniques. *HP Journal*, pages 40–50, June 1994.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol87] G. J. Holzmann. Automated protocol validation in Argos: Assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13(6):683–696, 1987.

[Hol88] G. J. Holzmann. An improved protocol reachability analysis technique. *Software—Practice and Experience*, 18(2):137–161, 1988.

[Hol92] G. J. Holzmann. Protocol design: Redefining the state of the art. *IEEE Software*, pages 17–22, January 1992.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–295, 1997.

[HP94] G. J. Holzmann and D. Peled. An improvement in formal verification. In *FORTE '94*, pages 197–211, 1994.

[HSH+00] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal simulation engines. In *ICCAD '2000*, pages 120–126, 2000.

[HU69] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata.* Addison-Wesley, 1969.

[Hu95] A. J. Hu. *Techniques for Efficient Formal Verification using Binary Decision Diagrams.* PhD thesis, Department of Computer Science, Stanford University, December 1995.

[Hum97] W. S. Humphrey. *Introduction to the Personal Software Process.* Addison-Wesley, 1997.

[Inm88a] Inmos Ltd. *Compiler Writers Guide.* Prentice Hall, 1988.

[Inm88b] Inmos Ltd. *Occam2 Reference Manual.* Prentice Hall, 1988.

[Jan96] G. Janssen. *Private communication.* 1996.

[JH93] He Jifeng and C. A. R. Hoare. From algebra to operational semantics. *Information Processing Letters*, 45:75–80, 1993.

[JJ91] C. Jard and T. Jéron. Bounded-memory algorithms for verification on-the-fly. In *CAV '91*, pages 192–202. Lecture Notes in Computer Science, 575, Springer Verlag, 1991.

[JU91] M. B. Josephs and J. T. Utting. An algebra for delay-insensitive circuits. In *CAV'90*, pages 343–352. Lecture Notes in Computer Science, 407, Springer Verlag, 1991.

[Kel76] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[Kur97] R. P. Kurshan. Formal verification in a commercial setting. In *DAC '97*, pages 258–262, 1997.

[Lam97] C. K. Lam. Graphical user interface and LTS visualisation for the ARC tool. Honours' thesis, Dept. of Computer Science, University of Adelaide, Australia, 1997.

[Lee59] C. Y. Lee. Representation of digital circuits by binary-decision programs. *Bell Systems Technical Journal*, 38(4):985–999, 1959.

[LMC00] M. Leuschel, T. Massart, and A. Currie. How to make FDR spin. Technical Report DSSE-TR-2000-10, Department of Electronics and Computer Science, University of Southampton, UK, 2000.

[LO95] M. Leeser and J. O'Leary. Verification of a subtractive radix-2 square root algorithm and implementation. In *ICCD '95*, pages 526–531. IEEE Press, 1995.

[Lon93] D. E. Long. *BDDLIB man page.* Carnegie-Mellon University, 1993.

[Low96] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *TACAS '96*, pages 147–166. Lecture Notes in Computer Science, 1055, Springer Verlag, 1996.

[Maz86] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II*, pages 279–324. Lecture Notes in Computer Science, 225, Springer Verlag, 1986.

[McC98] S. McConnell. *Software Project Survival Guide.* Microsoft Press, 1998.

[McM92a] K. L. McMillan. *Symbolic Model Checking—an approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[McM92b] K. M. McMillan. *The SMV System.* School of Computer Science, CMU, draft manual edition, 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MK96] H. Miller and S. Katz. Saving space by fully exploiting invisible transitions. In *CAV '96*, pages 336–347. Lecture Notes in Computer Science, 1102, Springer Verlag, 1996.

[MM95] G. A. McCaskill and G. J. Milne. Sequential circuit analysis with a BDD based process algebra system. Technical Report CIS-95-010, School of Computer and Information Science, University of South Australia, January 1995.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC '2001*, pages 530–535, 2001.

[MNS95] P. D. Mosses, M. Nielsen, and M. I. Schwartzbach. Anatomy of the Pentium bug. In *TAPSOFT '95*, pages 97–107. Lecture Notes in Computer Science, 810, Springer Verlag, 1995.

[MSS99] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, May 1999.

[NIN89] R. De Nicola, P. Inverardi, and M. Nesi. Using the axiomatic presentation of behavioural equivalences for manipulating CCS specifications. In *Automatic Verification Methods for Finite State Systems*, pages 54–67. Lecture Notes in Computer Science, 407, Springer Verlag, June 1989.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Lecture Notes in Computer Science, 104*. Springer Verlag, 1981.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94*, pages 377–390. Lecture Notes in Computer Science, 818, Springer Verlag, 1994.

[PY95] A. N. Parashkevov and J. Yantchev. Efficient refinement checking for CSP using OBDDs. In *Australasian Conference on Parallel and Real-time Systems PART'95*, pages 243–250, 1995.

[PY96a] A. N. Parashkevov and J. Yantchev. ARC—a tool for efficient refinement and equivalence checking for CSP. In *IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing ICA3PP '96*, pages 68–75, 1996.

[PY96b] A. N. Parashkevov and J. Yantchev. ARC—a verification tool for concurrent systems. In *Australasian Conference on Parallel and Real-time Systems PART'96*, pages 69–76, 1996.

[PY97] A. N. Parashkevov and J. Yantchev. Space efficient reachability analysis through use of pseudo-root states. In *TACAS '97*, pages 50–64. Lecture Notes in Computer Science, 1217, Springer Verlag, 1997.

[RGG+95] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS'95 Workshop*, pages 133–152. Lecture Notes in Computer Science, 1019, Springer Verlag, 1995.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honor of CAR Hoare*. Prentice Hall, 1994.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[RSS96] H. Ruess, N. Shankar, and M. Srivas. Modular verification of SRT division. In *CAV '96*, pages 123–134. Lecture Notes in Computer Science, 1102, Springer Verlag, 1996.

[Sca92] B. Scattergood. *A Parser for CSP.* Oxford University, UK, December 1992.

[Sca98] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP.* PhD thesis, The Queen's College, Oxford University, UK, 1998.

[Sok96] O. Sokolsky. *Efficient Graph-Based Algorithms for Model Checking in the Modal Mu-Calculus.* PhD thesis, State University of New York at Stony Brook, May 1996.

[Som98] F. Somenzi. *CUDD—CU Decision Diagram Package, Release 2.3.0.* Department of ECE, University of Colorado at Boulder, 1998.

[SSS00] M. Sheeran, S. Singh, and G. Staalmarck. Checking safety properties using induction and a SAT solver. In *FMCAD '2000*, pages 108–125. Lecture Notes in Computer Science, 1954, Springer Verlag, 2000.

[TM96] T. Theobald and C. Meinel. State encodings and OBDD-sizes. Technical Report TR-04, FB IV - Informatik, Universität Trier, Germany, 1996.

[TSL+90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *ICCAD '90*, pages 130–133. IEEE Press, November 1990.

[Vac95] M. Vaccari. *Private communication.* 1995.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In *CAV '93*, pages 397–408. Lecture Notes in Computer Science, 697, Springer Verlag, 1993.

[Var95] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344. IEEE Computer Society Press, Washington, DC, 1986.

[WBCG00] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *CAV '2000*, pages 124–138. Lecture Notes in Computer Science, 1855, Springer Verlag, 2000.

[WG84] W. M. Waite and G. Goos. *Compiler Construction.* Springer Verlag, 1984.

[WKS01] J. Whittemore, J. Kim, and K. A. Sakallah. Satire: A new incremental satisfiability engine. In *DAC '2001*, pages 542–545, 2001.

[WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.

[YBO+98] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *FMCAD '98*, pages 255–289. Lecture Notes in Computer Science, 1522, Springer Verlag, 1998.

[YP97] J. Yantchev and A. N. Parashkevov. Memory and run-time efficient state space exploration algorithms. Unpublished manuscript, Dept. of Computer Science, University of Adelaide, Australia, 1997.

[YSP+99] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using BDDs. In *ICCAD '99*, pages 584–589, 1999.