EFFICIENT PROTOCOL VERIFICATION USING RULE-BASED SYSTEMS

by

Ritwik Bhattacharya

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing

The University of Utah

March 2006

Copyright © Ritwik Bhattacharya 2006

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Ritwik Bhattacharya

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

______________________ ______________________________________
Chair: Dr. Ganesh Gopalakrishnan

______________________ ______________________________________
Dr. Steven German

______________________ ______________________________________
Dr. Gary Lindstrom

______________________ ______________________________________
Dr. Seungjoon Park

______________________ ______________________________________
Dr. Konrad Slind

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Ritwik Bhattacharya in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Dr. Ganesh Gopalakrishnan
Chair, Supervisory Committee

Approved for the Major Department

Dr. Martin Berzins
Chair/Dean

Approved for the Graduate Council

Dr. David S. Chapman
Dean of The Graduate School

ABSTRACT

The increasing complexity of industrial hardware systems has brought into prominence the role of formal verification in the debugging and validation of their designs. While simulation based testing, a long-time industry standard, is still of great use, the advantages, and complementarity, of formal verification is now beyond question. *Automated* verification techniques, in particular, have proven to be highly popular and effective in this regard. Verification techniques explore the *entire* state space of a design model, as opposed to the directed search that testing performs, and are consequently often able to find deep and elusive bugs that lurk in the corners of complex designs.

For the same reason, however, verification techniques also suffer from the *state explosion problem*. The large state spaces of complex designs continue to outgrow the ability of formal verification techniques to explore these spaces, posing a moving target to catch up with for verification research. A lot of progress has been made in recent years to enhance the capacities of automated verification tools and algorithms, including the advent of BDD and SAT based symbolic techniques. Unfortunately, most of these techniques are driven by, and therefore suited to, *hardware circuit* verification. Many other classes of designs, such as protocols of various kinds (cache coherence, communication, mutual exclusion, and security protocols), and, in recent times, software systems, are not easily amenable to either description, or verification, or both, using these symbolic techniques.

This thesis presents a set of techniques for increasing the power and reach of *explicit state-enumeration* verification tools for rule based systems, suitable for the above class of designs. The techniques have been implemented as extensions to the Murphi verifier, and have been tested on high-level protocol designs of varying sizes, with very promising results.

A major contribution of this thesis is that it brings, and extends, powerful existing theoretical strategies for state space reduction to the world of rule based systems, making these strategies more widely applicable than before, and thus increasing the availability of formal methods of verification. We have developed, for the first time, effective algorithms and techniques for making these strategies useful for rule-based systems. Based on simple insights into rule-based systems, and the fact that systems of this kind often exhibit parametricity, as well as symmetry, we have devised simple techniques to achieve upto 10 times state space reduction.

Another contribution is the design of a translator from the rule-based language of Murphi into the input language of a BDD and SAT based tool, NuSMV. Based on a simple symbolic simulation algorithm, this translator makes symbolic techniques, such as BDD-based model checking, and SAT-based bounded model checking, accessible to systems modeled in the high-level Murphi language. Designers are no longer forced to sacrifice the convenience of a full-featured language such as Murphi in order to use symbolic techniques of verification.

Implemented as the POeM and Mu2SMV tools, these techniques have been tested on applications such as cache coherence protocols, mutual exclusion algorithms, leader election protocols, and other distributed algorithms. As a result of these techniques, many complex systems can now be verified faster, and with much less memory.

To Mago and Babago

CONTENTS

ABSTRACT ii

LIST OF FIGURES viii

LIST OF TABLES ix

CHAPTERS

1. INTRODUCTION 1

1.1 Formal Verification 2
1.1.1 Theorem Proving 2
1.1.2 Model Checking 3
1.1.3 State Space Reduction Techniques 4
1.2 Formal Design Languages 5
1.2.1 Rule Based Languages 6
1.3 Contributions of this Dissertation 6
1.3.1 Partial Order Reduction 8
1.3.2 Transactional Systems 9
1.3.3 Parameterized Systems 9
1.4 Outline of the Dissertation 10

2. FORMAL DESIGN AND SPECIFICATION 11

2.1 Introduction 11
2.2 Modeling Reactive Systems 12
2.2.1 Formal Design Languages 13
2.2.2 Process-based descriptions 14
2.2.3 Rule-based descriptions 15
2.3 The Murphi Modeling Language 16
2.4 Property specifications using Linear Temporal Logic 18

3. PARTIAL ORDER REDUCTION FOR RULE BASED SYSTEMS 21

3.1 Introduction 21
3.2 Explicit State Enumeration Model Checking 21
3.3 State Explosion and Partial Order Reductions 23
3.3.1 Partial Order Reductions 24
3.3.2 Ample Set Based Partial Order Reduction 26
3.3.2.1 Independence and invisibility of actions 28
3.3.3 Sufficient conditions for ample set construction 31
3.4 Computing the independence relation 35

3.4.1 Syntactic computation 35
3.4.2 Computing independence symbolically 36
3.5 Symbolic independence computation for Murphi 37
3.5.1 Handling Rulesets 40
3.6 Computing Ample Sets 41
3.6.1 Approximating **C1** 41
3.6.2 On-the-fly cycle detection for **C3** 43

4. EXPLOITING SYMMETRY FOR PARTIAL ORDER REDUCTION 46

4.1 Introduction 46
4.2 Parameterized Systems and Symmetry in Murphi 47
4.3 Computing Independence for Parametric Systems 49
4.3.1 A First Order Representation of Parameterized Systems 51
4.3.2 The Carry Over Theorem 53

5. PARTIAL ORDER REDUCTION FOR TRANSACTIONAL SYSTEMS 58

5.1 Introduction 58
5.2 Transaction-based priorities for ample set construction 59
5.2.1 Assigning transition priorities 60
5.2.2 Determining transition positions within a transaction 61
5.3 Refining the independence relation 61
5.3.1 On-the-fly verification of strengthened guards 62

6. ENABLING SYMBOLIC EXPLORATION OF RULE BASED SYSTEMS 65

6.1 The NuSMV Verification System 66
6.1.1 The description language 66
6.2 Murphi's Rule Based Description Language 68
6.3 Translating Murphi 69
6.3.1 Variable declarations 70
6.3.2 Functions and Procedures 71
6.3.3 Rules 71
6.4 Conclusions and Future Directions 74

7. IMPLEMENTATION, EXPERIMENTS AND RESULTS 75

7.1 Introduction 75
7.2 **POeM**: Partial Order enabled Murphi 75
7.2.1 Symbolic simulation of Murphi rules 76
7.2.1.1 Assignment statements 77
7.2.1.2 Conditional statements 79
7.2.1.3 Loop statements 80
7.2.1.4 Function and procedure calls 80
7.2.2 Ample set construction 81
7.3 Experiments and Results 81
7.3.1 Guard Strengthenings and User-defined Priorities: A Case Study 82

7.4 Conclusions and Future Directions 87
7.4.1 Contributions 87
7.4.2 Future Directions 88

REFERENCES **91**

LIST OF FIGURES

1.1 Introduction and Detection of errors in the design cycle 2
2.1 Promela model of Peterson's algorithm 15
2.2 Murphi model of Peterson's algorithm 20
3.1 Basic Explicit State Enumeration Model Checking Algorithm 23
3.2 State explosion caused by interleaving concurrent actions 25
3.3 Ample Set Based Partial Order Reduction 27
3.4 Commutativity of independent actions 29
3.5 Stuttering equivalent paths 30
3.6 Violation of condition **C1** 32
3.7 The need for *invisible* ample set transitions 33
3.8 The *ignoring problem* 34
3.9 Programs representing independence conditions 38
3.10 Types of transitions at a given state 42
3.11 On-the-fly cycle detection during DFS 45
4.1 An example of a Murphi ruleset 48
4.2 A simple parameterized Murphi system outline 49
5.1 Interleaving of independent transactions 59
6.1 Example of a NuSMV system description 68
6.2 A Murphi rule and the translated NuSMV module 73
7.1 **POeM**'s architecture 76
7.2 **POeM**: Murphi rules 77
7.3 **POeM**: Lisp s-expressions 78
7.4 **POeM**: Enabledness and commutativity conditions 79

LIST OF TABLES

7.1 Performance of partial order reduction algorithm 82
7.2 Advantages of Guard Strengthening 85
7.3 User-defined priorities for the German protocol 86
7.4 User defined priorities vs. Variable reference based priorities 86

CHAPTER 1

INTRODUCTION

With modern advances in technology, the complexity of both hardware and software systems has increased dramatically, prompting a spurt in research on effective ways of ascertaining the conformance of system designs to their specifications. Intel's first processor, the 4004, consisted of 2300 transistors, compared to nearly 230 million transistors in today's Pentium Extreme Edition 840 [14]. Along with this increased complexity has come the rising cost of detecting and fixing errors in design. Further, errors becomes more expensive to repair, the later they are discovered in the design cycle. The fact that many of these errors introduced at the high level tend to involve *global properties* only exacerbates the problem. Figure 1.1[34] shows that, despite all of these factors, and the fact that most errors are actually *introduced* early on, current practice continues to result in most errors being found late in the design cycle.

Although the mainstays of error detection continue to remain testing and simulation, verification technology has matured sufficiently to provide a viable, and complementary, alternative to these traditional methods. Formal verification methods provide a means of detecting errors at *high-level design stages*, where simulation and testing are less practical. *Automated formal methods* are especially attractive in this regard, since they combine ease of use with powerful analysis techniques.

Formal verification techniques attempt to achieve the following task: given a formal description of a (usually finite-state) system, and a formal description of one or more desired properties, explore the reachable state space of the system, and verify that the properties hold. It is naturally desirable to perform the verification on the highest level of design possible, as state spaces are more manageable at higher

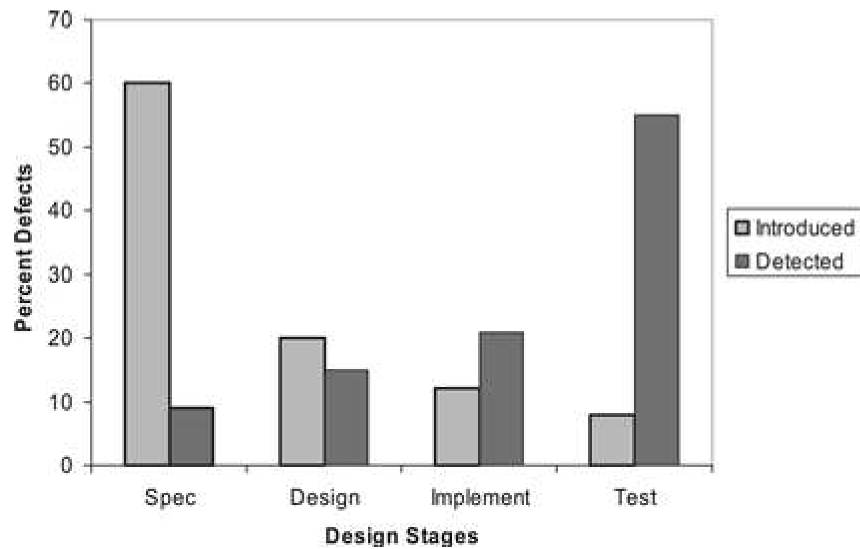

Figure 1.1. Introduction and Detection of errors in the design cycle

levels of abstraction, and also because, as mentioned earlier, most conceptual design errors are introduced at early stages of the design, where it is cheaper to detect and fix them.

1.1 Formal Verification

Under the general description of formal verification fall a number of different methods, which can be broadly classified as either theorem-proving techniques or model checking techniques.

1.1.1 Theorem Proving

Theorem proving techniques model systems in a higher order logic, and guide a proof assistant through proofs of the satisfaction of properties by the system. A very powerful technique, its limitation is primarily the degree of user interaction required, and the high level of user expertise necessary, which make it difficult to automate.

1.1.2 Model Checking

The second set of techniques are classified as model checking methods, which, in contrast to theorem proving, typically emphasize automation, at the potential cost of power. Model checking techniques are broadly based on the idea of building a representation of the reachable state space, typically as a graph, and checking properties on the graph. The graph building notion is both the greatest advantage of model checking methods, as it lends itself easily to automation and an almost push-button verification, and also its greatest drawback, since state spaces of realistic systems tend to be very large, and representing them fully is often impossible, even with the large memories of computer systems today. This problem is referred to as the *state explosion* problem.

Model checking techniques themselves can be further divided into two main classes - those that use an explicit representation of the state vectors of a system, and those that build a symbolic representation of states.

Symbolic model checking techniques, as the name suggests, are based on an indirect, symbolic representation of states, usually by characterizing a set of states using a Boolean formula, and encoding the formula with a variant of *binary decision diagrams* (BDDs) [7]. BDDs provide a compact, canonical representation of Boolean formulas, which makes operations such as comparison of formulas very inexpensive, and also reduces the space required to store states. However, BDDs are sensitive to *variable ordering*, as well as to the degree of global interdependencies among variables. For certain kinds of circuits, such as multiplication, BDDs are known to have exponential representations, irrespective of variable ordering. The interested reader is referred to Hu [23] for a good evaluation of BDD-based symbolic model checking.

The explicit-state approach is well suited to modeling systems with a lot of global interdependencies among state variables, such as communication protocols of message passing systems, or cache coherence protocols in multiprocessor systems. In the explicit state approach, a system is modeled by a set of *state variables*, each of which ranges over a finite domain. A state of the system is then an evaluation

of each of the state variables over their respective domains. An explicit-state enumeration model checker systematically explores all of the reachable states of such a system by walking the paths of the *state graph*, and storing each state encountered, typically in a *hash table*. Modern model checkers all perform the verification of properties *on-the-fly*, as the states are being explored. The biggest shortcoming of explicit-state methods, of course, is state explosion, made more acute because each state is represented in full. Many strategies for combating this problem have been devised, and such strategies are also the focus of this dissertation.

1.1.3 State Space Reduction Techniques

State space reduction techniques form a large and important part of verification research, and two of the most popular techniques that have been studied for explicit state model checking are symmetry and partial order reductions. Symmetry reduction techniques attempt to reduce the state space explored by defining an equivalence relation over the states, and then restricting the state exploration to only a single representative from each equivalence class. The equivalence classes are constructed based on the observation that for many systems (especially those with replicated components), the satisfaction of many properties of interest is often immune to certain permutations of the state vectors. As an example, in a typical MSI cache coherence protocol, properties may not distinguish processor identifiers, which means that a state s_1 where the cache of processor 1 is in the invalid state and processor 2 is in the shared state is equivalent to a state s_2 where processor 1 is in the shared state, and processor 2 is in the invalid state. A model checker with symmetry reductions enabled might visit state s_1 first, and thereafter, upon encountering state s_2, would not store it in the hash table. In a system with n identical components, symmetry reductions can result in a state space reduction of up to $n!$.

Partial order reductions, in contrast, reduce the explored state space by avoiding redundant interleavings of transitions that *commute*, (i.e., performing them in either order leads to the same final state). Explicit state model checkers typically preserve

concurrency semantics by interleaving simultaneously enabled transitions at a state in all possible ways. However, in cases where a pair of transitions commute, it is not always necessary to explore *both* interleavings. In a partial order reduction algorithm, the satisfaction of a set of sufficient conditions is used as a test for selecting a subset of all of the enabled transitions at any state, and only those parts of the sub-graph rooted at this state are explored that are reachable through the subset of enabled transitions selected. At the global level, this strategy avoids exploring multiple paths in the graph that are *stuttering equivalent*. The result is that in the smaller graph that is explored, there is guaranteed to be at least one path from each of the equivalence classes induced by the stuttering relation. Partial order reduction can often lead to exponential savings in state space.

One of the important contributions of this dissertation is a new partial order reduction algorithm for rule based systems, presented in Chapter 3. The primary difference between this algorithm and existing partial order reduction algorithms is that it replaces the syntactic check of the *commutes* relation by a much more precise, symbolic simulation based check. The algorithm has been implemented as an extension to the explicit state model checker Murphi [16]

1.2 Formal Design Languages

A formal description of a system is the starting point for all verification efforts, and a number of different paradigms have evolved for formal descriptions, mostly owing to the vast variety of systems that are sought to be verified. For finite state systems, which are the focus of this dissertation, the underlying mathematical formalism is the *Kripke structure*, which defines a system as a set of states, a *transition relation*, and a labeling function that labels states with propositions that hold at each state. However, it is not very practical to describe complex systems directly as Kripke structures, and most verification frameworks allow descriptions in some high level language. Theorem provers typically allow systems to be defined in first order, or higher order logics, which have great descriptive power, but add to the complexity of the verification itself. Model checkers, however accept descriptions in

simpler, restricted languages. For hardware circuits, *hardware description languages* (HDLs) such as VHDL and Verilog are the most commonly used. Another popular approach is to describe the transition relation on a per-variable basis. That is, how the state changes is described by defining a logic function representing the next state value of each individual variable of the system. An example of this style is the NuSMV [11] verification system, based on Ken McMillan's original SMV [32] symbolic model checker.

1.2.1 Rule Based Languages

For control-intensive systems like protocols, however, the paradigm of per-variable updates is quite inconvenient, as there are often global interdependencies among the variables, and it is not easy to extricate individual update functions for the state variables. Rule based languages such as Murphi [16], TLA+ [26] and BlueSpec [3] have become increasingly popular for specifying protocols and other communication based systems. In these languages, systems are described as a collection of global *rules*, which are guarded actions in the style of Dijkstra's guarded command language [15]. Most such languages provide rich data types such as arrays and records, and also allow programming-language style imperative specifications, including the use of functions and procedures. These languages are also well suited to describing *parameterized* systems, which consist of a number of replicated components with identical (or very nearly identical) behaviors.

1.3 Contributions of this Dissertation

As we mentioned earlier, explicit state model checking suffers from the well-known state explosion problem. In this dissertation, we investigate techniques for ameliorating this problem, especially in the context of rule-based systems. Rule based systems provide new opportunities and demands for designing state reduction algorithms, since many of the traditional reduction strategies used in explicit state model checking are not directly applicable. Foremost among these is the strategy of partial order reduction, which allows properties to be checked on a reduced graph with fewer states, under certain conditions, and guarantees equisatisfaction

of these properties over both the original and reduced graph. The reduced graph is constructed by exploiting the commutativity and independence of simultaneously enabled rules.

- One of the primary contributions of this dissertation is a new partial order reduction algorithm for rule based systems. The algorithm is based on the simple, but critical, insight that, for finite state systems, it is possible to symbolically simulate transitions, and thus obtain an accurate characterization of the independence of transitions. We also develop a number of new heuristics for effectively applying the algorithm to rule-based systems.

- Another feature of many rule based systems, especially parameterized, protocol-like systems, is their transactional nature. That is, many of these systems are based on the modalities of a request by one agent, followed by a sequence of actions eventually leading to either the grant or the denial of the request. In this dissertation, we develop a method to take advantage of this transactional nature of protocols, and further enhance the efficacy of partial order reductions for rule-based, transactional systems.

- Parameterized rule based systems also pose another interesting challenge, in that a verification of such a system is complete only when properties have been shown to hold for *all* parameter values. In general, proving such properties is known to be undecidable [2]. However, by imposing restrictions on the system description language, and/or the class of properties to be verified, it is possible to obtain decidability results that are still useful in practice. In this dissertation, we explore one such restricted class of systems, and develop a theory for parametrically carrying over the results of analyzing a smaller system to systems of higher size.

- Rule based descriptions have by tradition been tied to explicit state model checkers (e.g., Murphi and NuSMV), whereas symbolic checkers typically have a per-variable update based system description language (NuSMV). Since

protocol-like systems are more amenable to rule based descriptions, it has thus far been difficult to apply symbolic methods to protocol verification. We develop an automatic translation scheme from Murphi into NuSMV that addresses this problem.

1.3.1 Partial Order Reduction

As mentioned earlier, partial order reduction techniques reduce the state space that is visited by a model checker by exploring only a subset of the enabled transitions at each state. The key idea is that model checkers enforce concurrency semantics by interleaving simultaneously enabled transitions in all possible ways. However, when a pair of transitions are *independent*, which means that they commute, and do not disable each other, it is often sufficient to explore only one interleaving of the two transitions. By not exploring some interleavings, partial order reduction algorithms avoid visiting the intermediate states generated by those interleavings. Thus, only a subset of the full state graph is actually explored, leading to a saving of space, and often time, required by the model checker. These reductions also preserve the truth of a large class of temporal properties, so very little expressive power for properties is sacrificed in adopting these techniques. One of the critical computations on which the efficacy of any partial order reduction algorithm depends is the computation of the independence relation. Previous algorithms perform a syntactic check for updates of common variables, to determine independence among transitions. However, for rule based languages that allow data types like arrays and records, syntactic checks are too conservative, and lead to many missed reductions. In the case of many protocols described in the rule based formalism of Murphi, we show that partial order reduction based on a syntactic computation of independence are completely ineffectual. In Chapter 3, we develop a new partial order reduction algorithm that uses a symbolic analysis to compute the independence relation. We show that this algorithm outperforms the previous algorithms in nearly every case.

1.3.2 Transactional Systems

As mentioned earlier, many protocol-like systems have a transactional nature, and their computations are a series of requests, followed by a sequence of actions culminating in either the grant or the denial of the request. In this dissertation, we label each such sequence starting from a request, and ending in either a grant or denial, a *transaction*. Since partial order reduction algorithms pick a subset of enabled transitions at each state, it is natural to assume that an algorithm that is aware of the transactional nature of the protocol being verified will be able to pick these subsets more efficiently, since it can preferentially pick transitions belonging to the current transaction, effectively postponing the exploration of other transactions until after the present transaction is complete. In Chapter 5, we show how our partial order reduction algorithm can be modified to exploit the transactional nature of protocols, and demonstrate that it leads to as much as threefold savings *over* those achieved by the partial order reduction algorithm.

1.3.3 Parameterized Systems

We mentioned that the rule based formalism is also suitable for describing parameterized systems, which are composed of a number of replicated components with identical, or nearly identical, behaviors. The number of components of the system is termed the *parameter* of the system in this dissertation. The verification of parameterized systems is complete in theory only when a property has been shown to hold of the system, *independently* of the parameter value. In general, this problem has been shown to be undecidable. Therefore, parameterized verification efforts often restrict either the class of systems, or the kinds of properties that can be verified. However, in practice, it is often the case that parameterized systems are verified for a few small instances of the parameter size, to gain some measure of confidence in the correctness of the system. Partial order reduction can then be applied to each of these individual verification runs. In Chapter 4, we show that for many parameterized systems, there is a (small) bound on the parameter size for which the independence relation needs to be computed, and it can be extrapolated

to get the independence relation for any higher size.

1.4 Outline of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we give an overview of formal design languages, rule based descriptions, and the Murphi modeling language. Chapter 3 gives an overview of explicit state enumeration based model checking and partial order reduction, discusses related work, and outlines the problems that rule based systems such as Murphi present in applying partial order reduction strategies. It also describes our solution to these problems, and introduces the core partial order reduction algorithm we have developed. Chapter 4 describes how our algorithm can be applied more efficiently to a certain restricted class of parameterized systems, and also discusses other properties which can be verified of such parameterized systems. Chapter 5 describes our technique for exploiting the transactional nature of many rule based systems, and also discusses a technique for iteratively refining the rules of a system to improve the performance of our partial order reduction algorithm. In Chapter 6, we discuss a technique that is an off-shoot of our main research, allowing us to translate rule based system descriptions in Murphi into the input language for the symbolic model checker NuSMV [11]. This enables the symbolic model checking of rule based systems. Finally, in Chapter 7, we describe our implementations of the algorithms, discuss experimental results, and examine directions for future research based on this work.

CHAPTER 2

FORMAL DESIGN AND SPECIFICATION

2.1 Introduction

Model checking, as the name suggests, is a procedure for checking whether a given formally defined system is a model of a given formally specified property or not. A system is said to be a model of a property if the property is true at every state the system can ever reach. In this chapter, we look at how systems can be formally defined, and in particular, at the rule based formalism for describing systems. We then give an overview of *linear temporal logic* (LTL), one of a number of logics that are commonly used to express temporal properties of interest.

In the model checking context, a *system* is a finite set of *states*, and a *transition relation* that describes how the system moves from one state to another. Typically, a system specification will also include a designation of certain states as the *initial states* of the system. Most systems considered in this dissertation belong to a class known as *reactive systems* [31]. As with ordinary finite state machines, reactive systems perform computations that evolve over time, and that are modeled by their transition from state to state. However, typical reactive systems don't have terminating states. That is, their computations are *infinite* sequences of states. Common examples of reactive systems include operating systems, sequential hardware circuits, cache coherence protocols, message passing systems, and so on. Often, we are interested in reasoning about the *temporal* properties of such systems.

Digital circuits, and programs/protocols form the two main classes of reactive systems. Circuits can be either *synchronous* or *asynchronous*. Synchronous circuits operate as a sequence of steps, where in each step, the inputs to the circuit change, the circuit stabilizes, and then a clock pulse is applied to change the values of

the outputs (which, along with the inputs, form the state variables of the circuit). Asynchronous circuits, on the other hand, are conceived of as having a number of components, with exactly one output each. Each component changes its output instantaneously upon the application of inputs, and the change is assumed to be rapid enough that it is impossible for two components to change value at the same time. Therefore, asynchronous circuits are typically modeled using *interleaving concurrency semantics*, in which exactly one component changes at a time. Programs and protocols, the main focus of this dissertation, are also modeled as asynchronous systems. As with asynchronous circuits, protocols are modeled as *concurrent systems* consisting of a number of communicating components. The most common modes of communication adopted are *message passing*, where the components communicate by exchanging messages over a network, and *shared variables*, where the components have shared access to a set of global variables, and communicate by appropriately updating the values of these variables.

2.2 Modeling Reactive Systems

All of the above types of systems share the common characteristic of being state transition systems. A commonly used mathematical formalism for defining state transition systems is the *Kripke structure*. A Kripke structure defines a set of states, a transition relation, and a labeling function from states to sets of *atomic propositions* which determines the set of propositions that are true at a given state. Formally, let AP be a set of atomic propositions. A Kripke structure M over AP is a four tuple $M = (S, S_0, R, L)$ where

- S is the set of states (finite)
- S_0 is the set of initial states
- R is the transition relation, which must be total (i.e., for every state s, there is a state s', such that $R(s, s')$)
- $L : S \to 2^{AP}$ is the labeling function

Paths in a Kripke structure are sequences of states such that every pair of successive states is related through the transition relation. Paths represent the computations of the system. Although conceptually simple, Kripke structures are sufficiently expressive to capture most temporal behaviors of reactive systems.

2.2.1 Formal Design Languages

While Kripke structures provide a formal mathematical basis for modeling and reasoning about systems, it is impractical to expect designers to define transition relations of systems directly as a mathematical object. Firstly, for systems of realistic size, it is often impossible to define a monolithic transition relation, owing to the complicated relationships between state variables, and how their values change with time. Secondly, such an attempt would also be highly prone to errors. Therefore, higher level formal design languages play a key role in enabling the use of formal verification techniques.

Formal design languages can be broadly classified into those that support the modeling of hardware circuits, and those that support programs or protocol models. Since this dissertation is mainly concerned with the verification of protocol like systems, we defer a discussion of languages for hardware design to Chapter 6, where we describe a translation scheme for applying symbolic model checking techniques to specifications in what we call the *rule based* paradigm, which we discuss below.

Two dominant paradigms have emerged for the specification of programs and protocols, both loosely based on Dijkstra's *guarded command language* [15]. In the literature, these have been referred to as the *process-based* and *rule-based* paradigms. The salient feature that both of these modeling techniques borrow from Dijkstra's language is the notion of guarded commands or statements. As in programming languages, the fundamental statement type is the assignment or update type. The semantics of updates in descriptions of temporal systems is that the state variable being updated assumes the updated value in the next time step of the system's computation. A *guarded statement* additionally imposes a predicate over the current values of the state variables, whose satisfaction is a precondition for the update statement to be executed. A guarded statement is said to be *enabled* at a point in

time if its guard predicate is satisfied at that time step.

2.2.2 Process-based descriptions

As mentioned earlier, programs and protocols are frequently modeled as concurrent systems of communicating components. The process based paradigm, as the name suggests, aligns closely with this model, and allows the designer to specify the system as a collection of *processes*, each of which is a sequential set of statements. Models can include both variables that are local to processes, and global variables accessible to all processes. Each process is also often associated with a *program counter*, which points to the current statement of the process scheduled for execution. The semantics of process-based descriptions are usually given in terms of the interleaving model of concurrency, whereby at each time step, exactly one process is chosen, and the statement pointed to by the program counter for that process is executed. In the case that multiple processes each have an enabled statement at a point in time, one of the processes is chosen nondeterministically.

The process based paradigm is particularly well suited to modeling communication protocols, and other asynchronous protocols where there is a natural separation of the state variables into local and global components. A popular design language based on the process paradigm is Promela, the modeling language used in the Spin model checker [21]. Promela models are a collection of processes, *channels*, and state variables. Channels are specialized message passing devices that processes can use to communicate, so Promela allows both the shared variable and message passing approaches to communication between processes. Channels allow both asynchronous (buffered) and synchronous communication between processes. Basic variable types include Booleans, bytes, integer subranges, and enumerated types. Arrays of basic types are also permitted, and array variables can be indexed by any valid expression that evaluates to a value type-consistent with the array's index type. Figure 2.1 shows a Promela model of Peterson's mutual exclusion algorithm [37].

In the figure, `_pid` is a special Promela variable whose value is the id of the running process.

```
 1 #define true       1
 2 #define false      0
 3 bool flag[2];
 4 bool turn;
 5 active [2] proctype user()
 6 {
 7     flag[_pid] = true;
 8     turn = _pid;
 9     (flag[1-_pid] == false || turn == 1-_pid);
10     crit: skip; /* critical section */
11     flag[_pid] = false
12 }
```

Figure 2.1. Promela model of Peterson's algorithm

2.2.3 Rule-based descriptions

Rule based specifications form another important, and widely used, paradigm for describing many types of systems, such as cache coherence protocols, security protocols, and mutual exclusion algorithms. Rule based formalisms represent state transition systems by a collection of unordered *rules*. Each rule is a guarded *action*, where an action can consist of one or more update statements. As before, a rule is said to be *enabled* if it's guard predicate is true at the current state. State holding variables in a rule based system are usually all global, and can be updated by any rule. Rule based systems are also given interleaving concurrency semantics, with rules replacing processes as the units of concurrency. That is, at each time step, exactly one enabled rule is picked, and its action executed, updating state variables appropriately. A rule is typically executed *atomically*, which means that once a rule r_1 is picked for execution at a given time step, no other rule r_2 can execute until r_1 has completed execution. In case multiple rules are simultaneously enabled, one of them is chosen nondeterministically.

Certain kinds of protocols, such as cache coherence protocols for distributed shared memory systems, are more naturally modeled as a collection of unordered

rules than as a collection of processes. As mentioned earlier, this has to do with whether there is a natural partitioning of state variables into local and global components or not. In the case of cache coherence protocols, for example, it is not always evident whether each hardware node ought to be a process, or each kind of message. Each of the choices presents difficulties in identifying local and global variables. In Chapter 3, we present a more detailed comparison of the two modeling techniques, and analyze them from the point of view of state space reduction strategies.

Popular rule based specification languages include Unity [9], Murphi [16], TLA+ [29] and BlueSpec [3]. We now describe the Murphi language in some detail, since it is used throughout the rest of this dissertation.

2.3 The Murphi Modeling Language

Murphi models consist of constant and type declarations, variable declarations, function and procedure declarations, rule declarations, and invariant declarations. States of a Murphi model are assignments of values to the variables, and the rules specify how the system transitions from state to state. Figure 2.2 shows a Murphi model of Peterson's mutual exclusion algorithm. In the figure, `Startstate` describes the initial states of the model.

Constant and type declarations:

- Integer constants can be declared at the beginning of a description.
- Murphi allows the same basic types as Promela, of Boolean, integer subranges, and enumerated types. It also allows the definition of array and record types.

Variable declarations:

- All the state variables of a Murphi model are global, and can be modified by any rule of the system.

Function and procedure declarations:

- Function and procedure declarations are similar to those in high level programming languages such as C and Pascal. Parameters to functions and procedures can be passed either by value or by reference. Recursion is also allowed, although seldom used in practice.

Rule declarations:

- Rules describe the transition relation of a Murphi model. Each rule, as described before, is a *guarded command* consisting of a predicate over the state variables, called the *guard*, and a sequence of updates to the state variables, called the *action*.

- The initial states are described using rules called `startstate`.

- Families of similar rules are declared using a `ruleset`, which is a parameterized collection of rules, corresponding to an actual rule for each combination of values of the parameters.

The guards of rules are Boolean expressions over the global variables, and can involve the standard Boolean operators such as conjunction, disjunction and negation. Quantification over finite ranges is also allowed in these expressions. Actions have sequential semantics, and each assignment to a variable is executed in the environment modified by previous assignments in the same rule.

The Murphi modeling language, as we have seen, provides a number of high-level features, such as arrays, records, functions and procedures, commonly found in programming languages such as C or Java. These features enable the description of systems at suitably high levels of abstraction, freeing the designer to focus on algorithmic correctness rather than implementation details. As we noted in Chapter 1, many of the most expensive bugs in designs are introduced at the highest levels of design, and it is one of the goals of this dissertation to facilitate verification of high level designs. However, the presence of these high-level features in Murphi provides interesting new challenges in applying standard state space reduction techniques, and developing new ones. In the next few chapters, we will describe some of these problems, and our solutions.

2.4 Property specifications using Linear Temporal Logic

In this section, we give a brief overview of *linear temporal logic*, which is commonly used to specify properties of reactive systems. Although this dissertation's focus is on *invariant safety* properties, this section is included for the sake of completion, and to provide the context for our discussion of the partial order reduction strategy in Chapter 3.

Temporal logics allow the description of sequences of transitions of reactive systems. Assertions such as that *eventually*, a certain event occurs, or that *always*, a certain property holds, can be expressed in these logics, using *temporal operators*. These operators are combined with ordinary Boolean connectives to form propositions of the logic. Different temporal logics provide different sets of temporal operators, and consequently, have differing powers of expressibility.

Linear temporal logic (LTL) formulas express properties of *computation trees* of a reactive system. A computation tree is simply the state graph formed by unwinding a state transition system starting from the designated initial states of the system. A computation tree captures all of the possible executions of the system. LTL formulas are *state formulas*, which describe what properties must hold at all computation paths starting at that state. The temporal operators allowed in LTL are:

- **X**: Asserts that a property must hold at the second state of the path.
- **F**: Asserts that a property must hold at some state on the path.
- **G**: Asserts that a property must hold at every state along the path.
- **U**: A binary operator that asserts that the second property must hold at some state on the path, and, at every preceding state, the first property must hold.
- **R**: The dual of **U**, this operator asserts that the second property must hold along every state of the path up to the first state where the first property holds. It does not, however, require that the first property must eventually hold.

Given a set of atomic propositions AP, the following are valid LTL formulas:

- For every $p \in AP$, p is an LTL formula.
- If p and q are LTL formulas, $\neg p$, $p \vee q$, $p \wedge q$, $\mathbf{X}p$, $\mathbf{F}p$, $\mathbf{G}p$, $p\mathbf{U}q$ and $p\mathbf{R}q$ are LTL formulas.

```
Type
  pid: 0..1;
Var
  turn: pid;
  flag: array [pid] of boolean;
  pc: array [pid] of 1..3;

Ruleset i: pid do
  Rule ``Make request''
    (pc[i] = 1) ==>
  Begin
    flag[i] := true;
    turn := 1 - i;
    pc[i] := 2;
  End;

  Rule ``Enter critical section''
    (pc[i] = 2 & ((! flag[1 - i]) | (turn = i))) ==>
  Begin
    pc[i] := 3;
  End;

  Rule ``Exit critical section''
    (pc[i] = 3) ==>
  Begin
    flag[i] := false;
    pc[i] := 1;
  End;
Endruleset;

Startstate
Begin
  For i: pid do
    flag[i] := false;
    turn := 0;
    pc[i] := 1;
  Endfor;
End;
```

Figure 2.2. Murphi model of Peterson's algorithm

CHAPTER 3

PARTIAL ORDER REDUCTION FOR RULE BASED SYSTEMS

3.1 Introduction

In the last chapter, we outlined how reactive, concurrent systems can be formally defined so that they are amenable to model checking techniques, with an emphasis on rule based systems. We also described the underlying mathematical formalism (the Kripke structure) that is commonly used to reason about these systems. In this chapter, we first provide an overview of *explicit state enumeration* model checking techniques. We then introduce the partial order reduction strategy for combating state explosion, and discuss issues and challenges with its application to rule based specifications. Finally, we present our solution to these problems, based on the simple observation about finite state systems that every predicate over state variables can be converted into a propositional formula with the aid of symbolic simulation.

3.2 Explicit State Enumeration Model Checking

As described earlier, *model checking* [18] is the name given to a set of techniques that have the goal of exploring the reachable states of a finite state system, and verifying that the states satisfy properties of interest. Explicit state enumeration methods, as the name suggests, achieve this by using an explicit representation of the states of the Kripke structure, usually referred to as *state vectors*. Each state vector is an assignment of values to the state variables. Before we describe the verification algorithm of explicit state enumeration model checkers, we need to introduce some notation and definitions.

Definition 3.1. A *state graph* is a four tuple $G = \langle Q, Q_0, \Delta, \emptyset \rangle$, where Q is a set of states, Q_0 is a set of initial states, Δ is a *transition relation* over states such that there is an edge between states q_1 and q_2 exactly when $\langle q_1, q_2 \rangle \in \Delta$. $\emptyset$ is a unique *error* state such that whenever $\langle \emptyset, q \rangle \in \Delta$, $q = \emptyset$.

Every Kripke structure induces a corresponding state graph.

Definition 3.2. For every pair of states $\langle q, q' \rangle \in \Delta$, q' is called the *successor* of q, and q the *predecessor* of q'.

Definition 3.3. A *path* is a finite sequence of states $q_0, q_1, \ldots, q_k$, such that $q_0 \in Q_0$, and $\langle q_i, q_{i+1} \rangle \in \Delta, 0 \leq i < k$.

Definition 3.4. A *run* is an infinite sequence of states $q_0, q_1, \ldots, q_k, \ldots$, such that $q_0 \in Q_0$, and $\langle q_i, q_{i+1} \rangle \in \Delta, 0 \leq i < k$.

Note that every finite prefix of a *run* is a *path*.

Definition 3.5. A state$q$ is *reachable* if there exists a path $q_0, \ldots, q$.

Definition 3.6. A state q is a *deadlock* state if its only successor is itself, that is, $\forall q' \in Q . \langle q, q' \rangle \in Delta \Rightarrow q' = q$.

We will often represent $\langle q_1, q_2 \rangle \in \Delta$ by $q_1 \rightarrow q_2$.

The typical explicit state enumeration model checking algorithm, as shown in Figure 3.1, checks whether the error state $\emptyset$ is reachable. This is accomplished *on-the-fly*, as the states are being generated. The algorithm starts an exploration of the state graph at each initial state, and generates new states by applying the transition relation to the current state being examined. Visited states are stored in memory, typically in a hash table, so that it can be efficiently determined whether newly generated states have already been examined or not. The state graph is traversed either in *breadth-first* or *depth-first* order.

```
1 Hash Visited;
2 Queue Unexplored;
3 Boolean Deadlocked;

4 Model_Check()
5 begin
6   Visited = StartStates;
7   Unexplored = StartStates;
8   while !(Unexplored = Empty) {
9     CurrentState = pickAState(Unexplored);
10     Deadlocked = True;
11     NextStates = applyTransRel(CurrentState);
12     foreach NextState in NextStates
13     do
14       if (NextState = ErrorState)
15         reportError(NextState);
16       endif;
17       if !(NextState = CurrentState)
18         Deadlocked = False;
19       endif;
20       if !member(NextState, Visited)
21         add(NextState, Visited);
22         add(NextState,Unexplored);
23       endif;
24     endfor;
25     if (Deadlocked)
26       reportError(NextState,Deadlocked);
27     endif;
28   endwhile;
29 end;
```

Figure 3.1. Basic Explicit State Enumeration Model Checking Algorithm

3.3 State Explosion and Partial Order Reductions

The algorithm described above has been implemented, with minor modifications, in most state-of-the-art explicit state model checkers, such as SPIN [21], Murphi [16] and TLC [29]. Software, and protocol-like systems, however, pose some

challenges to this approach. Highly concurrent, asynchronous systems typically consist of a large number of independently executing actions, which makes the state explosion problem especially acute for these systems.

Concurrent systems are commonly represented using the *interleaving model*, where all of the actions of an execution are arranged in a linear order called an *interleaving sequence*. Concurrently scheduled actions are arbitrarily ordered with respect to each other in any given execution. Since most logics for specifying properties can distinguish between executions in which actions appear in different orders, model checkers must explore *all* possible interleavings of concurrent actions. This can lead to an exponential increase in the state space of the system.

3.3.1 Partial Order Reductions

A powerful set of techniques that addresses this problem are collectively known as *partial order reduction* [41, 20]. These techniques exploit the *independence* of simultaneously enabled actions. Actions are independent if they result in the same global state when executed in either order. Partial order techniques reduce the number of interleavings of independent actions that need to be explored, and consequently reduce the number of states that are visited by the model checker. If a specification cannot distinguish between two different executions of a system that only differ in the order in which independent actions are executed, it is sufficient to explore only one of those executions. Partial order reductions are based on the *partial order model of program execution* [19, 25], which only partially orders executions, by leaving concurrently enabled actions unordered. Thus, each execution can correspond to a number of different interleaving sequences. Whenever it is not possible to differentiate these sequences, it is enough to pick one representative sequence.

A number of different techniques based on the idea of avoiding redundant interleavings of actions have been researched over the years. One of the earliest techniques is due to Overman [35], who considered a restricted concurrency model without looping and nondeterminism. Model checking algorithms based on the partial order reduction technique include the ideas of *stubborn sets* [41], *persistent*

sets [20], and *ample sets* [36]. The algorithms of this dissertation are based on the latter technique of ample sets.

A partial order reduction algorithm attacks the state explosion problem by attempting to build a smaller, reduced state graph, instead of the full state graph. Therefore, the set of executions of the system that are explored is a subset of the full set of possible executions of the system. To be sound, it must be possible to show that the executions that are not explored do not yield more information about the system than the smaller subset, with respect to the properties being verified of the system. Partial order reduction techniques guarantee this by ensuring that for every execution that is *not* explored in the reduced graph, an *equivalent* execution *is* included in the reduced graph.

A slightly different, but related technique, is *transaction based reduction* [30] that works by detecting *commit points* within transactions, and controlling the scheduling of threads based on these commit points.

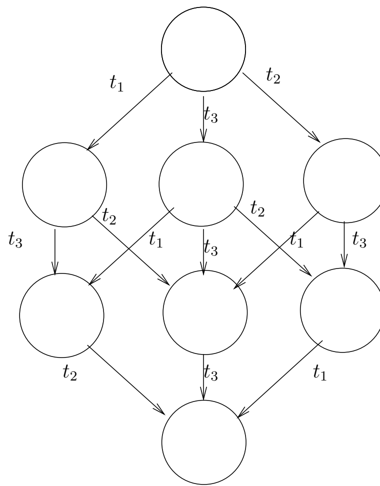

Figure 3.2. State explosion caused by interleaving concurrent actions

To see how exploring all possible interleavings of concurrently enabled actions can lead to state explosion, consider the state graph of Figure 3.2. In the figure, t_1, t_2, and t_3 are three transitions that are simultaneously enabled, and independent of each other. Exploring the six different possible orderings leads to eight states

being visited. If, however, the property being checked does not distinguish between these orderings, it would be much more efficient to check only one of the orderings, and consequently explore only four states. In general, given n concurrently enabled transitions, there are $n!$ different orderings possible, and 2^n different states. Thus, partial order reductions can lead to potentially exponential savings in state space, since exploring only one of the orderings visits $n+1$ states.

3.3.2 Ample Set Based Partial Order Reduction

The ample set based partial order reduction algorithm is most straightforwardly implemented as a modification of the standard *depth-first search*(DFS) algorithm for state exploration, as shown in Figure 3.3. This algorithm is very similar to the one presented earlier for explicit state enumeration model checkers, the crucial difference being that in line `11`, the call to `pickAState` is replaced by a call to `computeAmpleSet`. Thus, only the successors generated by transitions in the ample set are explored from each state. Also, whenever a new state is pushed on to the DFS stack, a bit is set by the call to `markOnStack` on line `22`. This is useful for computing the ample set, as we will see later.

To formally describe the ample set based partial order reduction technique, we modify slightly the Kripke structure introduced earlier in Chapter 2. Since individual transitions and the independence between them plays an important role in partial order reductions, we modify the transition relation of a Kripke structure to be a collection of individual *transitions*.

Definition 3.7. A labeled finite state transition system $\mathcal{F}$ is a 5-tuple $\langle S, T, I, AP, L\rangle$ where

- S is a finite set of *states*,
- T is a finite set of deterministic *transitions*, such that every $t \in T$ is a partial function $t : S \mapsto S$,
- $I \subseteq S$ is the set of initial states,
- AP is a set of atomic propositions, and

```
1 Hash Visited;
2 Stack Unexplored;
3 Boolean Deadlocked;

4 Model_Check()
5 begin
6   Visited = StartStates;
7   Unexplored = StartStates;
8   while !(Unexplored = Empty) {
9     CurrentState = pop(Unexplored);
10     Deadlocked = True;
11     NextStates = computeAmpleSet(CurrentState);
12     foreach NextState in NextStates
13     do
14       if (NextState = ErrorState)
15         reportError(NextState);
16       endif;
17       if !(NextState = CurrentState)
18         Deadlocked = False;
19       endif;
20       if !member(NextState, Visited)
21         add(NextState, Visited);
22         markOnStack(NextState);
23         push(NextState,Unexplored);
24       endif;
25     endfor;
26     if (Deadlocked)
27       reportError(NextState,Deadlocked);
28     endif;
29   endwhile;
30 end;
```

Figure 3.3. Ample Set Based Partial Order Reduction

- $L : S \mapsto 2^{AP}$ labels each state with a set of propositions that are true in the state.

Definition 3.8. A *labeled path* of a finite state system is a finite or infinite sequence starting with a state and then alternating transitions and states,

$$s_0, t_0, s_1, t_1, s_2, t_2, \ldots,$$

where $\forall i \geq 0 : t_i(s_i) = s_{i+1}$.

A labeled path is called a labeled *run* if it starts with a state in I. Let the set of all labeled paths of a finite state system be $\mathcal{P}$.

Definition 3.9. A transition t is said to be *enabled* at a state s if $\exists s' \in S : t(s) = s'$.

We define the predicate $\mathbf{en}(s,t)$ to be true exactly when t is enabled at s. We also define the predicate $\mathbf{enabled}(s) = \{t \in T \mid \mathbf{en}(s,t)\}$.

Definition 3.10. For any labeled path p of a system, the predicate $\mathbf{before}(p, t_1, t_2)$ is true when t_1 occurs before the earliest occurrence of t_2 in p, or t_2 does not occur in p.

Definition 3.11. The *restriction* of $\mathcal{P}$ with respect to a state s, written $\mathcal{P}_{|s}$, is the set of all labeled paths in $\mathcal{P}$ starting from the state s.

3.3.2.1 Independence and invisibility of actions

In concurrent systems, it is very often the case that multiple transitions are *enabled* at the same state. It is exactly in these situations that partial order techniques can reduce the number of states explored, by exploiting the independence of these transitions.

Definition 3.12. Two transitions t_1 and t_2 are *independent* iff the following conditions hold:

- *Enabledness*: $\forall s \in S : \mathbf{en}(s,t_1) \wedge \mathbf{en}(s,t_2) \Rightarrow \mathbf{en}(t_1(s),t_2) \wedge \mathbf{en}(t_2(s),t_1)$
- *Commutativity*: $\forall s \in S : \mathbf{en}(s,t_1) \wedge \mathbf{en}(s,t_2) \Rightarrow t_1(t_2(s)) = t_2(t_1(s))$

Figure 3.4 shows two independent transitions, and the effect of executing them in either order from a given state. It is easy to see that independent transitions form a good candidate for partial order reduction, since, as in the figure, simply executing the transitions in any *one* of the two orders leads us to the same final

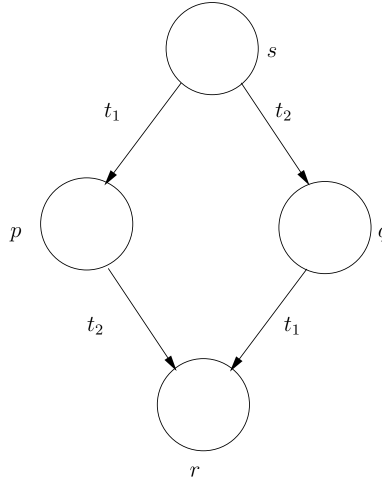

Figure 3.4. Commutativity of independent actions

global state. However, independence alone is not sufficient, since we must take care to ensure that by not visiting one of the intermediate states, say s_2 in the figure, we are not missing an error state (either because s_2 itself violates the property of interest, or because one of the successors of s_2 violates the property). *Invisibility* of transitions refers to how transitions affect the truth of the property of interest, and is an important concept in formulating conditions for partial order reduction:

Definition 3.13. For any property π, we define $\mathbf{props}(\pi) \in 2^{AP}$ as the set of propositions occurring in π.

Definition 3.14. A transition t is *invisible* with respect to a property π, written as $\mathbf{inv}_\pi(t)$, iff:

$$\forall s_1, s_2 \in S : t(s_1) = s_2 \Rightarrow L(s_1) \cap \mathbf{props}(\pi) = L(s_2) \cap \mathbf{props}(\pi)$$

Invisibility of transitions allows us to define the notion of *stuttering*.

Definition 3.15. Two infinite sequences of states $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ and $\rho = \langle r_0, r_1, r_2, \ldots \rangle$ are *stuttering equivalent* [28], written as $\sigma \sim_{st} \rho$ if there exist two infinite sequences of integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$, such that, for every $k \geq 0$:

$$L(s_{i_k}) = L(s_{i_k+1}) = L(s_{i_k+2}) = \cdots = L(s_{i_{k+1}-1}) =$$
$$L(r_{j_k}) = L(r_{j_k+1}) = L(r_{j_k+2}) = \cdots = L(r_{j_{k+1}-1}).$$

Intuitively, this says that two paths are stuttering equivalent if each path can be partitioned into contiguous sequences of states (*regions*) that are identically labeled, and the kth regions from the two paths have the same labeling. This is illustrated in Figure 3.3.2.1. Stuttering equivalence can be extended to structures in a straightforward manner [13, Chapter 10], such that two transition systems M and M' are stuttering equivalent if and only if they have the same set of initial states, and:

- For every path σ in M starting at an initial state, there is a path σ' in M' starting at the same initial state, such that $\sigma \sim_{st} \sigma'$.
- For every path σ' in M' starting at an initial state, there is a path σ in M starting at the same initial state, such that $\sigma' \sim_{st} \sigma$.

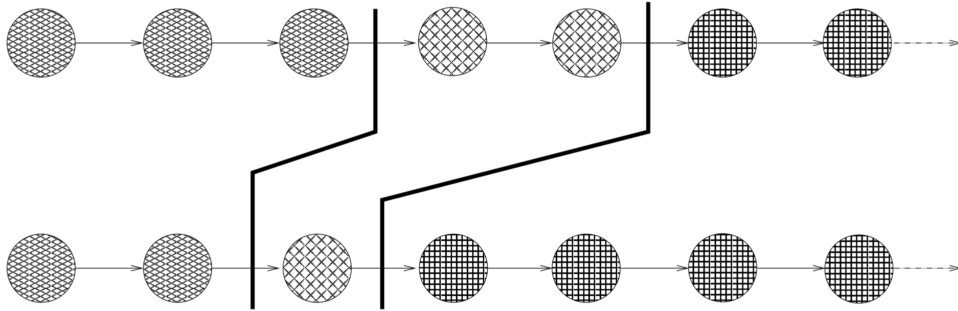

Figure 3.5. Stuttering equivalent paths

Stuttering equivalent paths have interesting and desirable properties from the point of view of verification. In particular, if we let LTL_X stand for the subset of LTL without the next time operator X, it has been shown that all properties expressible in LTL_X are *invariant under stuttering* [13, Chapter 10]. That is, whenever two paths σ and ρ are stuttering equivalent, for every LTL_X formula $\mathbf{A}\, f$, $\sigma \models f$ if and only if $\rho \models f$. LTL_X formulas are often termed *stuttering closed* for this reason.

As with all other equivalences, it is possible to partition the set of all paths of a system into *equivalence classes* based on stuttering equivalence. Therefore, it would be sufficient to check at least one representative path from each equivalence class, to determine whether the system satisfies any given stuttering closed property. Partial order reduction algorithms construct reduced graphs that guarantee the existence of at least one such path for each class. The ample set based methods we study in this dissertation achieve this by constructing subsets of enabled transitions at each state. We now describe a set of sufficient conditions for constructing these ample sets that guarantee the desired inclusion of at least one representative path from each stuttering equivalence class.

3.3.3 Sufficient conditions for ample set construction

The goal of the ample set based partial order reduction technique is to employ a locally optimal heuristic to construct a globally reduced state graph. By picking a small subset of the enabled transitions at each state as an ample set, the algorithm aims to construct a commensurately small state graph. As mentioned in the previous section, this must be done while ensuring that stuttering equivalence between the two structures is maintained. The following conditions have generally been used in the literature [36] as sufficient conditions for constructing reduced graphs based on the independence and commutativity of transitions:

C0: $\forall s \in S : \mathbf{ample}(s) = \emptyset \Leftrightarrow \mathbf{enabled}(s) = \emptyset.$

This condition ensures that an ample set is empty if and only if there are no enabled transitions. In other words, whenever there is at least one enabled transition in the original graph, there will also be a transition in the reduced graph.

C1:

$$\begin{aligned}
&\forall s \in S : \forall t_1, t_2 \in T : \\
&\qquad t_1 \in \mathbf{ample}(s) \wedge t_2 \notin \mathbf{ample}(s) \wedge \mathbf{dep}(t_1, t_2) \Rightarrow \\
&\qquad \forall p \in \mathcal{P}_{|s} : \exists t_3 \in \mathbf{ample}(s) : \mathbf{before}(p, t_3, t_2)
\end{aligned}$$

The most complex of the conditions, **C1** says that along every path in the *full*

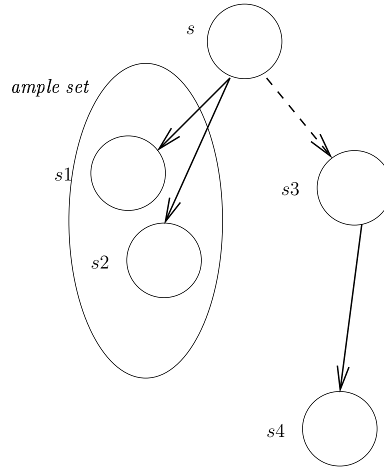

Figure 3.6. Violation of condition **C1**

state graph that starts at a state s, the following must hold - if there is an enabled transition that depends on a transition in the ample set, it is not taken before some transition from the ample set is taken. The satisfaction of this condition guarantees that all ample set transitions are independent of all other enabled transitions at that state. Otherwise, as Figure 3.6 illustrates, there will be a path in the full state graph that violates **C1** ($\langle s, s_3, s_4, \dots \rangle$ in the figure).

While picking transitions for the ample set, we have to ensure that no paths in the full state graph are left out that will not have stuttering equivalent paths in the reduced graph. Condition **C1** serves to restrict the shape of paths that could possibly be missed, to one of the following:

- A path with the prefix $\langle \beta_1, \beta_2, \dots, \beta_k, \alpha, \dots \rangle$, where α is an ample set transition, and all the β_is are transitions independent of α

- A path of the form $\langle \beta_1, \beta_2, \dots,$ where all the β_is are transitions independent of α.

In the first case, assume that a path of the form $\langle \beta_1, \beta_2, \dots \alpha \rangle$ reaches a state r. We can show that there will exist a path in the reduced graph that also reaches r. Since α is independent of all of the β_is, a path $\langle \beta_1, \beta_2, \dots \alpha \rangle$ can be constructed by applying the commutativity conditions that will also reach the same state r.

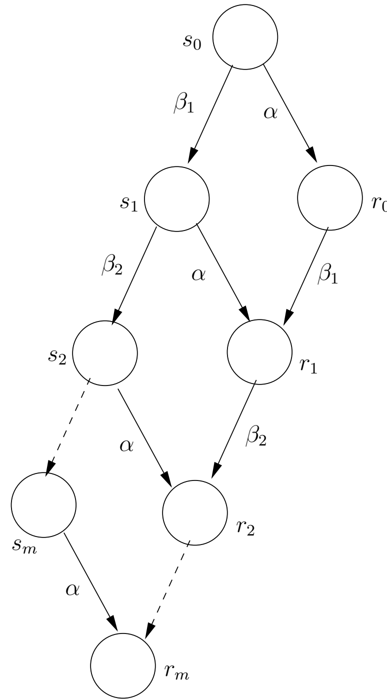

Figure 3.7. The need for *invisible* ample set transitions

However, to ensure that this path is *stuttering equivalent* to the original path, α must be an *invisible* transition, as Figure 3.7 makes apparent. This leads us to the next condition:

C2:

$\forall s \in S :$

$$\mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \forall t \in \mathbf{ample}(s) : \mathbf{inv}_{\pi}(t)$$

This condition says that if a state is not fully expanded, then every transition in the ample set is invisible with respect to the property being verified. Selecting only invisible transitions for inclusion in the ample set guarantees that stuttering equivalent paths for each path of the two types above will be included in the reduced graph.

The final condition avoids what is known in the literature as the *ignoring problem*. In brief, this condition ensures that an enabled transition is always eventually expanded. Figure 3.8 shows how, in the absence of this condition, it

is possible to miss exploring an enabled transition, and therefore miss potential error states. In the figure, assume that transition β is independent of all of the α_i transitions, and that each α_i is invisible. It would be perfectly legal to pick, as the ample set, transition α_i at state $s_i, 1 \leq i \leq 3$. This would, however, result in state s_4 never being explored, and if it was an error state, then, since it would not be included in the reduced graph, the reduction would be unsound. Therefore, we impose the following condition on ample sets:

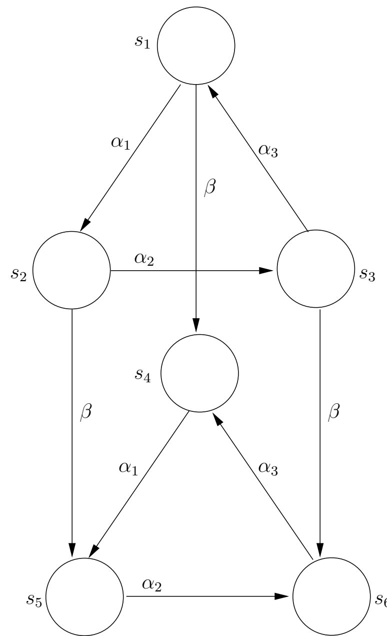

Figure 3.8. The *ignoring problem*

C3[1]**:**

$\forall s \in S:$

$\qquad \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow$

$\qquad \exists t \in \mathbf{ample}(s) : t(s) \notin \mathbf{onstack}(s)$

[1]For a proof of the sufficiency of this form of the condition see [22]

There is no transition t that is enabled in a state that is part of a cycle, and is not in the ample set of any state in that cycle.

3.4 Computing the independence relation

3.4.1 Syntactic computation

Traditional partial order reduction algorithms rely on syntactic methods for computing the dependence relation [13, Chapter 10]. Syntactic methods use occurrences of variables to determine dependence instead of solving for the possible values of variables. In order to be sound, syntax-based dependence computation algorithms must conservatively under approximate the independence relation, which can result in missed reduction opportunities. Unfortunately, this means that traditional partial order reduction algorithms do not work well with rule-based specifications such as used in Murphi [17] and TLC [29]. To illustrate the difficulties with a syntactic approach, consider a simple Murphi system with the variables $i : 0..2, a : 0..3, b :$ array[0..2] of boolean, and the following rules:

- rule "1": (True `==>` $a := (a + 1)\%4$)
- rule "2": (True `==>` $a := (a + 2)\%4$)
- rule "3": (($i >= 0$) `==>` $b[i + 1]$:=True)
- rule "4": (($i <= 0$) `==>` $b[i + 2]$:=False)

A syntax-based approach (such as used in SPIN) would classify rules 1 and 2 as dependent on each other, since they both update the same variable `a`. The binary operator `n % 4` gives the value of its first argument mod 4. However, a symbolic evaluation will discover that the values of `a` after applying both rules 1 and 2 in either order are identical. Similarly, rules 3 and 4, which both access the same array variable, would be classified as dependent by a syntax-based approach. However, it is evident that at all states where 3 and 4 are simultaneously enabled ($i = 0$), different elements of the array are updated, and therefore, the rules are actually independent.

As the examples above show, the problem with attempting to compute the independence relation syntactically is that the unavailability of exact information about variable values often means that these methods must mark as dependent, transitions that are actually independent. This problem was identified in [22], in the context of SPIN's partial order reduction algorithm. However, since SPIN uses a process-based modeling language, and has dedicated data structures for communication buffers (channels), SPIN's syntax based algorithm works quite well in the Promela context. For rule based languages such as Murphi, which are not process-based, and make heavy use of high-level data structures, the syntactic approach severely reduces the efficacy of partial order reduction, as we show in our discussion of experimental results in Chapter 7.

3.4.2 Computing independence symbolically

For rule based finite state systems, the subject of this dissertation, it is however possible to compute a more precise independence relation. To see how this improvement can be achieved, let us revisit the definition of independence from Section 3.3.2.1. Recall that it is composed of two different sub conditions, *enabledness* and *commutativity*:

Enabledness: $\forall s \in S : \mathbf{en}(s, t_1) \wedge \mathbf{en}(s, t_2) \Rightarrow \mathbf{en}(t_1(s), t_2) \wedge \mathbf{en}(t_2(s), t_1)$

Since each transition of a rule based system is a *guard/action* pair, we can rewrite the above as:

Enabledness: $\forall s \in S : g_1(s) \wedge g_2(s) \Rightarrow g_1(a_2(s)) \wedge g_2(a_1(s))$

where $t_1 = \langle g_1, a_1 \rangle$ and $t_2 = \langle g_2, a_2 \rangle$, and, for any state s, and transition $t = \langle g, a \rangle$, $g(s)$ represents the evaluation of the predicate g at the state s, and $a(s)$ represents the execution of the action a at the state s.

Remark 3.16. For any transition $t = \langle g, a \rangle$ of a finite state system, and a completely general symbolic state s, the predicate $g(s)$ is a propositional formula over the symbolic state variables.

Remark 3.17. For any transition $t = \langle g, a \rangle$ of a finite state system and a completely general symbolic state s, $a(s)$ can be symbolically simulated to produce another

symbolic state.

We will later see that in practice, there are certain restrictions we must impose on the *action* to be able to perform the symbolic simulation.

These two observations form the basis of an improved algorithm for computing independence. By symbolically simulating the action, and evaluating the guard predicates over the resulting symbolic state in the consequent of the implication above, the entire enabledness condition can be converted into a propositional formula over a general symbolic state. Since the enabledness condition must hold at every state of the system (it is universally quantified over the set of states), we can check whether the condition holds for a given pair of transitions by checking the validity of the resulting propositional formula. This, in turn, is equivalent to checking the satisfiability of the negated formula. The commutativity condition can also be similarly transformed into a propositional formula. In the next section, we describe how these ideas are implemented in the Murphi system.

3.5 Symbolic independence computation for Murphi

A Murphi system description, as we have discussed in Chapter 2, uses *rules* to encode the transition relation. Each rule is a guard/action pair, where the guard is a predicate over the current values of the state variables, and the action is a set of assignments to the next-state values of the state variables. Rules can also be defined parametrically, within *rulesets*. Rulesets have one or more parameters, and all of the rules within a ruleset are instantiated for every combination of values for all of the parameters of the ruleset.

Computing independence requires checking the *enabledness* and *commutativity* conditions for each pair of rules. Figure 3.9 shows the programs we use to represent the *enabledness* and *commutativity* conditions, for a pair of rules $r_1 \equiv (guard1, action1)$ and $r_2 \equiv (guard2, action2)$.

The enabledness check is carried out by symbolically simulating the action of one of the rules when both the rules are enabled (their guards are true), and then checking whether the other rule remains enabled (its guard is still true). In the

Enabledness

```
(if (and guard1 guard2)
    (begin
       action1
       (:= CHECK_ENABLED ,guard2)
    )
  (:= CHECK_ENABLED true))
(invariant CHECK_ENABLED)
```

Commutativity

```
(if (and guard1 guard2)
    (begin
       action1
       action2
       action2_r
       action1_r
       (:= CHECK_S1 (= s1 s1_r))
       (:= CHECK_S2 (= s2 s2_r))
             :
             :
       (:= CHECK_Sn (= sn sn_r))
    )
   (begin
      (:= CHECK_S1 true)
      (:= CHECK_S2 true)
            :
            :
      (:= CHECK_Sn true)
   ))
   (invariant CHECK_S1 & CHECK_S2 & ... CHECK_Sn)
```

Figure 3.9. Programs representing independence conditions

program above, this is achieved by checking whether the variable `CHECK_ENABLED` is always true.

The commutativity check is carried out by creating two sets of symbolic state

variables. The actions of the rules are then symbolically simulated, first in one order, and then in the reverse order, as the figure shows. The latter simulation updates the second set of state variables (hence `action1_r` and `action2_r`). The values of the two sets of variables are then compared for equality, to determine whether the rules commute or not. In the program, `s1`...`sn` and `s1_r`...`sn_r` represent the two sets of `n` state variables. The commutativity condition is true if each of the `CHECK_S1`...`CHECK_Sn` variables is always true.

In order to compute the full independence relation, we must carry out the above checks for *each pair of rules* in a given Murphi model description. Since Murphi allows both rules and rulesets, there are four possible types of pairs of rules:

1. Two single rules
2. A single rule and a rule from a ruleset
3. Two rules from different rulesets
4. Two rules from the same ruleset

A direct approach to the independence analysis would begin by expanding out each rule of a ruleset into its individual instances, which would result in a list of single rules, which could be analyzed pairwise for independence. However, the combinatorics in case of rulesets with a large parameter value makes this approach expensive and infeasible. For example, a pair of rulesets with 3 rules each, indexed over a parameter of size 4, would involve 66 pairwise checks for independence. Since it is one of our goals to design an algorithm that does not involve an overhead that outweighs the benefits of state space reduction, we have developed a number of different approaches to handling the combinatorial complexity.

Checking the above conditions for two single rules is straightforward, and is carried out exactly as outlined above. All of the other cases involve instantiating rules from a ruleset, and the various strategies used are described in the next section.

3.5.1 Handling Rulesets

To be complete, the independence checks for rules from a ruleset must be carried out for each instance of the rule. However, this can become quite expensive, since each evaluation of one of the expressions above involves a call to a SAT solver. Therefore, we use conservative approximations to reduce the number of SAT calls. For the case of a single rule and a rule from a ruleset, the single rule is checked for independence against the parameterized rule of the ruleset. Thus, if the single rule is found to be independent of the parameterized rule, it is guaranteed to be independent of each instance of the parameterized rule. However, if the single rule is not independent of *any one* of the instances of the parameterized rule, it will conservatively be marked dependent on *every* instance.

Two rules from different rulesets are checked by instantiating them to arbitrary symbolic values of the parameters. If these symbolic instances are determined to be independent, then all of the actual instances can be marked independent. It is a conservative check, because if the symbolic instances are found dependent, then it is not necessarily the case that all of the actual instances are dependent. However, marking them all dependent is conservative, and therefore sound.

For two rules from the same ruleset, we can sometimes do a little better than the above. In particular, if the ruleset has exactly one parameter, we check two different instances of the rules. In one instance, we check independence of the rules with the same symbolic value for the parameter, and in the other instance, we check independence of the rules with a different symbolic value for the parameter. If the first instance turns out to be independent, all of the actual instances with the same parameter value are marked independent. Similarly, if the second instance is independent, all of the actual instances with different parameter values can be marked independent. However, if the ruleset has more than one parameter, then the combinatoric explosion of cases to be checked make it prohibitively expensive, and we fall back to the standard algorithm for checking independence of rules from different rulesets.

3.6 Computing Ample Sets

Once the independence relation has been computed, the verification run can begin. The independence relation is stored as an *independence matrix*, so that looking up the independence of any pair of transitions is an $O(1)$ operation. At each state s encountered during a depth-first search of the state space, we must now compute an ample set of transitions to be explored from that state. In algorithms for process-based paradigms, a common heuristic is to pick all of the enabled transitions of one process as the ample set. Our algorithm, however, relies directly on the definition of an ample set. It starts out by picking a *seed transition* t_s, a candidate transition around which the ample set is to be constructed. Our initial heuristic for picking a seed transition was to pick the transition with the least number of references to state variables, as this is likely to minimize the number of other transitions that are dependent on the seed transition. In Chapter 5, we look at how to take advantage of the *transactional* nature of many protocols to improve the ample set selection algorithm.

Once a seed transition has been chosen as the initial member of the ample set, a least fixed point of this set is computed with respect to the dependence relation. That is, the *candidate ample set* CA is computed, where:

$$t_s \in CA$$
$$\forall t \in T. \forall t' \in CA. \mathbf{dep}(t, t') \wedge \mathbf{en}(s, t) \Rightarrow t \in CA$$

After thus computing the candidate ample set, we check the ample set for the satisfaction of the sufficient conditions **C0** – **C3**. **C0** is true by construction, of course, since we only pick enabled transitions. **C2** is checked simply by looking up the value of the precomputed invisibility predicate for each transition in CA.

3.6.1 Approximating C1

As has been previously noted, an exact check for **C1** is as hard as reachability itself, and is therefore obviously not feasible. Instead, heuristics that conservatively approximate **C1** are employed. Recall that the **C1** condition requires that there is no path in the full state graph starting at a state s where a transition dependent on

the ample set is taken before any ample set transition is taken. Figure 3.10 shows three of the four disjoint sets into which the transitions can be partitioned at any state s: the ample set transitions ($\mathcal{A}(s)$), the set of enabled transitions independent of those in the ample set ($\mathcal{I}(s)$), and the set of disabled transitions ($\mathcal{D}(s)$). The fourth set, of disabled transitions that are independent of those in the ample set, is irrelevant to our discussion, and is therefore omitted.

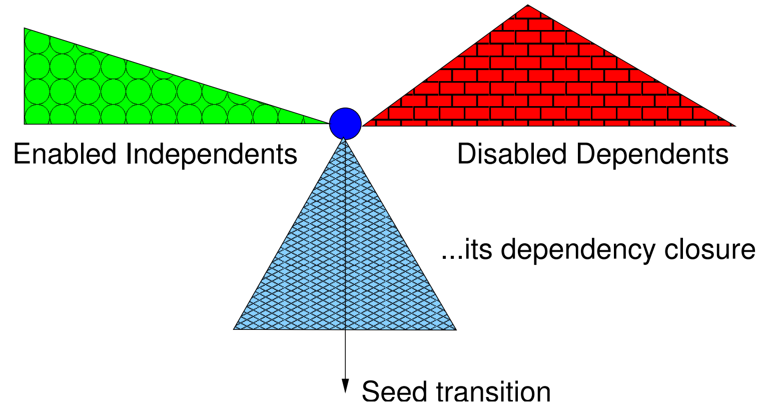

Figure 3.10. Types of transitions at a given state

We have experimented with two different algorithms for approximating the **C1** condition:

- In the first algorithm, we simply check whether $\mathcal{D}(s)$ is empty. This is a simple, but rather extreme, check. Given the set $\mathcal{D}(s)$, the check itself is just an $O(1)$ operation. However, it is only satisfied if *every* disabled transition is independent of *every* transition in the ample set.

- In our second algorithm, we check whether it is the case that every transition in $\mathcal{D}(s)$ can only be enabled by an ample set transition. To do this, we precompute the following *disabledness* condition for each pair of transitions, at the same time that the independence relation is computed:

$$\forall s \in S : \mathbf{en}(s, t_1) \wedge \neg\mathbf{en}(s, t_2) \Rightarrow \neg\mathbf{en}(t_1(s), t_2)$$

> At runtime, we then check whether the disabledness condition is true for every $t_1, t_1 \notin \mathcal{A}(s)$, and every $t_2 \in \mathcal{D}(s)$. This check is more expensive, since we need to perform a pairwise check for disabledness between every non-ample set transition and every transition in $\mathcal{D}(s)$, which is $O(n^2)$ in the worst-case. However, it is a closer approximation of the **C1** condition than our first algorithm.

We examine the effectiveness of the two algorithms in Chapter 7.

If the candidate ample set satisfies the **C1** check, we then check **C2**, which simply ensures that all of the transitions in the ample set are invisible. Finally, we check **C3**, which requires that at least one of the transitions in the ample set leads to a state that does not close a cycle in the state graph. Here, the naive implementation was to perform this check as part of the ample set computation, by executing each ample set transition and checking whether the state reached was on the DFS stack or not. However, this is highly inefficient, as each ample set transition ends up being executed twice, once as part of the ample set computation, and then again as part of the graph traversal. In the next section, we describe a more efficient, on-the-fly algorithm for cycle detection.

3.6.2 On-the-fly cycle detection for C3

Since our partial order reduction algorithm performs a depth first search (DFS) of the state graph, determining whether a newly generated state completes a cycle or not is equivalent to checking whether it is on the DFS stack or not. In this section, we describe how this check can be performed by keeping two bits of extra information in the *hash table*, which explicit-state enumeration model checkers use to store states that have been visited.

Figure 3.11 shows our modified DFS algorithm. The first bit (**onstack**) records whether the state is on the current DFS stack. The second bit (**goodC3**) records whether the **C3** check at that state was successful or not. These bits are then set and used in the following manner. Every time a new state is encountered during DFS, it is added to the hash table, and **onstack** is set (line 28). If the state has

already been visited, it's **onstack** bit is turned on (line 28). The ample set is then computed at that state, and, assuming that **C0 - C2** are satisfied, the ample set transitions are explored. As each transition is explored, the state it leads to is inserted into the hash table. During this insertion, if the state is found to be already in the hash table, with its **onstack** bit set, it is on the current stack. However, if a state is found that is either not in the hash table, or is in the hash table and does not have the **onstack** bit set, we have found a transition that leads to a state which does not complete a cycle, and therefore, the **C3** check at the previous state was successful. This is noted by following the parent pointer to the current state, and setting the parent state's **goodC3** bit (lines 21 and 26). When a state is about to be popped from the stack, we check if its **goodC3** bit is set, and if so, clear its **onstack** bit and pop the state (lines 34 and 35). If not, the **C3** check at that state was unsuccessful, and we explore all of the other enabled transitions from that state (lines 37 and 38).

Our symbolic SAT based partial order reduction algorithm has been implemented as an extension to Murphi. Details of the implementation, and an analysis of experimental results are presented in Chapter 7.

```
1 Hash Visited;
2 Stack Unexplored;
3 Boolean Deadlocked;

4 Model_Check()
5 begin
6   Visited = StartStates;
7   Unexplored = StartStates;
8   while !(Unexplored = Empty) {
9     CurrentState = top(Unexplored);
10     Deadlocked = True;
11     NextState = fireNextAmpleSetTransition(CurrentState);
12     if (NextState != NULL)
12       if (NextState = ErrorState)
15         reportError(NextState);
16       endif;
17       if !(NextState = CurrentState)
18         Deadlocked = False;
19       endif;
20       if (member(NextState, Visited) & !onStack(NextState, Visited))
21         setGoodC3(CurrentState, Visited);
22       endif;
23       if !member(NextState, Visited)
24         add(NextState, Visited);
25         push(NextState,Unexplored);
26         setGoodC3(CurrentState, Visited);
27       endif;
28       setOnStack(NextState, Visited);
29     else
30       if (Deadlocked)
31         reportError(NextState,Deadlocked);
32       endif;
33       if goodC3(CurrentState, Visited)
34         clearOnStack(CurrentState, Visited);
35         pop(CurrentState, Visited);
36       else
37         addEnabledtoAmple(CurrentState);
38         setgoodC3(CurrentState, Visited);
39     endif;
40   endwhile;
41 end;
```

Figure 3.11. On-the-fly cycle detection during DFS

CHAPTER 4

EXPLOITING SYMMETRY FOR PARTIAL ORDER REDUCTION

4.1 Introduction

In the last chapter, we introduced our symbolic, SAT based partial order reduction algorithm for rule based systems. Symbolic simulation allowed us to compute a more precise independence relation among transitions than that possible through the use of syntactic checks for variable references. The resulting partial order reduction algorithm is effective in reducing the state space of the finite state concurrent systems that are typically specified in rule based languages. Another characteristic shared by many rule based system descriptions is *parametricity.* That is, many of these systems are comprised of a number of identical, or nearly identical, components that interact with each other. The number of these components is referred to as the *parameter* of the system. Thus, a system description where the parameter value is left unspecified represents an entire family of concrete systems, one for each value of the parameter.

Verifying properties of parameterized systems is known to be undecidable in general [2]. Therefore, efforts at verifying parameterized systems focus on either restricted classes of systems, or on proving restricted types of properties. However, in practice, it is often the case that even with these techniques, push-button model checking is difficult to perform on parameterized systems. Attention is therefore often restricted to particular instances, such as, for example, verifying a cache coherence protocol for sizes of up to four or eight nodes.

For verifying particular instances of parameterized systems, partial order re-

duction techniques can naturally play a substantial role in making the verification tractable. However, if each of these systems is treated independently of each other, no advantage is taken of the similarity between the various instances. Consequently, a lot of the analysis must be repeated for each instance. For our algorithm, this means that the symbolic SAT based analysis for computing independence must be repeated for each pair of transitions from each instance of the system.

In this chapter, we present a technique for exploiting the structural similarity of a large class of parameterized systems to avoid duplicating the independence analysis. We show that it is possible to calculate a bound on the parameter size for which the independence relation needs to be computed. For systems of all higher sizes, the independence relation computed for the system of this bounded size can be extended in a straightforward way.

4.2 Parameterized Systems and Symmetry in Murphi

Parameterized systems, as mentioned earlier, consist of a number of similar, replicated components. In this dissertation, we restrict our attention to systems with exactly one parameter. Examples of such systems include cache coherence protocols, communication protocols, mutual exclusion algorithms, and so on. Murphi's description language naturally accommodates the specification of parameterized systems, with the *ruleset* construct. A ruleset specifies a family of similar rules that are indexed by the ruleset parameter. Typically, this will mean that they access different elements of parameterized data structures such as arrays, and perform identical actions on the respective elements. Figure 4.1 shows a simple ruleset consisting of a single parameterized rule which resets two array elements. It describes a set of rules, each updating the program counter (variable `pc`) and ticket number for one of a number of components participating in the Bakery mutual exclusion algorithm [27].

Many parameterized systems are also *symmetric* [24] with respect to the parameter. For the purposes of this dissertation, a system is symmetric with respect to a variable if states and properties are impervious to permutations of the elements of

```
ruleset id:pid
do
  rule "reset pc and proc"
  pc[id] = 3
    ==>
  begin
    ticket[id] := 0;
    pc[id] := 1;
  end;
endruleset;
```

Figure 4.1. An example of a Murphi ruleset

the type of the variable. A typical example would be the `pid` type in Figure 4.1, the set of identifiers for the components involved in the algorithm. Although it is common to use integer subranges to represent such sets, systems rarely rely on the actual integer properties of these variables. For example, assigning integer identifiers to the components imposes an implicit ordering on the components, although most systems will not distinguish behaviors of components based on their place in the ordering. Instead, such identifiers serve primarily to distinguish one component from another, so it would usually be sufficient to be able to tell when two components have the same, or different, identifiers.

Murphi supports the specification of symmetry using the *scalarset* construct [24]. A scalarset type is declared by specifying the size of the set. Variables of scalarset type cannot be assigned particular elements from the domain, and can only be compared for equality and inequality with other variables of the same type. Also, when used in `for` loops, the effect of the loop must be independent of the order in which the iterations are performed. Except the latter condition, all of the restrictions on scalarset variables are enforced by the Murphi compiler.

Given a Murphi description of a parameterized, symmetric system, we now describe how to calculate a bound on the parameter size, such that it is sufficient

to compute the independence relation for the system of that size, to be able to deduce the independence relation for systems of any higher size.

4.3 Computing Independence for Parametric Systems

```
CONST
  num_clients : 3;
TYPE
  message : enum{empty, req_shared, req_exclusive,
                invalidate, invalidate_ack,
                grant_shared, grant_exclusive};
  cache_state : enum{invalid, shared, exclusive};
  client: scalarset(num_clients);
VAR
  channel1: array[client] of message;
  cache: array[client] of cache_state;
RULESET cl: client do
  RULE "client requests shared access"
    cache[cl] = invalid & channel1[cl] = empty ==>
    BEGIN channel1[cl] := req_shared END;
  RULE "client requests exclusive access"
    (cache[cl] = invalid | cache[cl] = shared )
     & channel1[cl] = empty ==>
    BEGIN channel1[cl] := req_exclusive END;
END;
```

Figure 4.2. A simple parameterized Murphi system outline

As a simple example, consider the Murphi system outline of Figure 4.2. This is an extract from the parameterized German protocol, and shows the two transitions responsible for making new requests for access to a cache line, in either the shared or exclusive mode. The parameter of this system is the number of clients, represented by `num_clients`. The first thing to note is that there are actually multiple instances of each transition (rule) in the system, for any value of `num_clients`. In this case, there are 3 instances of each rule, corresponding to the range of the variable `cl`.

Theoretically, therefore, we need to check 9 pairs of rules for dependence (each of the 3 instances of the first rule against each of the 3 instances of the second rule). However, as we show in [6], it suffices to check a pair where the indices have the same value, and a pair where they have different values, and conservatively extrapolate the results to all pairs. So in the given system, we might choose to check rule 1[`cl` $\leftarrow$ cl_1] (the instance of the first rule with `cl` set to cl_1) against rule 2[`cl` $\leftarrow$ cl_1] (a pair with the same index value), and rule 1[`cl` $\leftarrow$ cl_1] against rule 2[`cl` $\leftarrow$ cl_2] (a pair with different index values). Here, cl_1 and cl_2 are symbolic values that are left unconstrained in the propositional expressions that are passed to the SAT solver. In the first case, if the SAT solver is unable to find a satisfying assignment, this implies that none of the rule instances with the same parameter value are dependent, since we left the parameter value unconstrained. In the second case, our algorithm conjoins to the propositional expressions representing enabledness and commutativity, a clause that constrains cl_1 and cl_2 to be different from each other. Thus, if the SAT solver is unable to find a satisfying assignment, it is evident that none of the rule instances with different parameter values are dependent on each other.

What does the truth of the above checks for one value of `num_clients` tell us about the truth of the corresponding checks for a different value of `num_clients`? Consider first the case of the two rules with the same value for the ruleset parameter `cl`. Obviously, these two rules are dependent, for they pass neither the enabledness check nor the commutativity check. This is because they are essentially requests from the same node, and therefore each request disables the other. In this case, it would not make any difference what the range of the index variable `cl` was, since the rules do not count the range in any manner, and only refer to state variables that are directly indexed over `cl`, which we have already instantiated to a particular value. Therefore, for these particular rules, it is sufficient to check independence for a particular instance, to be able to conclude that for every instance of the parameterized system, instantiations of these rules with the same index value will never be independent. Now consider the case of the two rules with different values

for the parameter `cl`. In this case, the rules involve entirely disjoint sets of state variables, and hence the rules are independent. However, this is true as long as the values of `cl` are different for the two rules, irrespective of what *particular* values they are. Therefore, here too, it is sufficient to check independence for one instance, and infer independence for all instances.

4.3.1 A First Order Representation of Parameterized Systems

Recall that a *scalarset* variable in Murphi is a variable such that the system description is completely symmetric with respect to permutations of the elements of the domain of the scalarset variable. A parameterized Murphi specification, with a single scalarset parameter N, can be described in terms of a first order language over the set of variables of the specification. Following Pnueli *et al*'s notion of *bounded data systems* [38], we partition the set of variables into three broad classes, as follows:

- $\mathcal{V}_1 = \{x_1, x_2, \dots x_a\}$ where x_i is interpreted over $\mathbb{B}$, the boolean domain, and $a \in \mathbb{N}$, the set of natural numbers.
- $\mathcal{V}_2 = \{y_1, y_2, \dots y_b\}$ where each y_i is a *scalarset* variable interpreted over the integer subrange $[1 \dots N]$, and $b \in \mathbb{N}$.
- $\mathcal{V}_3 = \{ar_1, ar_2, \dots ar_c\}$ where each ar_i is an array with index type $[1 \dots N]$, each array's cell type is interpreted over $\mathbb{B}$, and $c \in \mathbb{N}$.

The terms of the language are the boolean constants **True** and **False**, variables of type $\mathcal{V}_1$ or $\mathcal{V}_2$, and array references of the form $ar_i[y_j]$, where $y_j \in \mathcal{V}_2$ and $ar_i \in \mathcal{V}_3$. The valid atomic formulas of our language are partitioned into the set of *ordinary* atomic formulas $\mathcal{O}$ and *quantified* atomic formulas $\mathcal{Q}$, where: $\mathcal{O} = \{x_i \mid x_i \in \mathcal{V}_1\} \cup \{ar_i[y_j] \mid ar_i \in \mathcal{V}_3, y_j \in \mathcal{V}_2\} \cup \{y_i = y_j \mid y_i, y_j \in \mathcal{V}_2\}$, and $\mathcal{Q} = \{\forall x \in 1 \dots N. ar_i[x] \mid ar_i \in \mathcal{V}_3\} \cup \{\forall x \in 1 \dots N. \neg ar_i[x] \mid ar_i \in \mathcal{V}_3\} \cup \{\exists x \in 1 \dots N. ar_i[x] \mid ar_i \in \mathcal{V}_3\} \cup \{\exists x \in 1 \dots N. \neg ar_i[x] \mid ar_i \in \mathcal{V}_3\}$. The set of formulas is then the standard extension of the atomic formulas using the boolean

connectives $\wedge$, $\vee$ and $\neg$. We say that the set of all formulas over a set of variables $\mathcal{V}$, $\mathcal{L}(\mathcal{V})$ is the *language* of our logic.

A Murphi system description, which consists of a set of variable declarations, and a set of transitions (rules) defined as *guard/action* pairs, can be mapped into our first order language by mapping the variable definitions to the variables of the language, mapping guards to formulas of the language, and mapping actions as sets of substitutions of variables by terms or formulas of the language. Since the variables in $\mathcal{V}_1$ and $\mathcal{V}_3$ are of boolean type, they can be assigned any valid formula of the language, because Murphi allows arbitrary boolean expressions as rvalues in assignments. For a complete description of the allowed substitutions, and the corresponding Murphi constructs, see [5]. A state of a Murphi system can thus be seen as an interpretation of the variables of the logic. We denote the set of all states of a system as S, and, in particular, the set of all states of a *parameterized* system with parameter N as $S(N)$.

We restrict our attention to systems with *scalarset symmetry.* In such systems, both states (interpretations) and the satisfaction of formulas, are symmetric with respect to any permutation of the indices, as we describe below. Our notion of symmetry follows [38] closely.

Let $\Pi : [1..N] \rightarrow [1..N]$ be a permutation on the indices $[1..N]$. The state $\tilde{s}$ is a Π-variant of s, denoted as $\tilde{s} = s[\Pi]$ if the following conditions hold:

- $\tilde{x}_r = x_r$, for every $r \in [1..a]$
- $\tilde{y}_r = \Pi^{-1}(y_r)$, for every $r \in [1..b]$
- $\tilde{z}_r[h] = z_r[\Pi(h)]$, for every $r \in [1..c]$, $h \in [1..N]$

where $\tilde{v}$ denotes the value of $v \in \mathcal{V}$ in $\tilde{s}$, and v denotes the value in s.

A Murphi system S is symmetric with respect to a permutation Π if:

- $s \in S \iff s[\Pi] \in S$
- $\forall s \in S, f \in \mathcal{L}(\mathcal{V}).\ s \models f \iff s[\Pi] \models f$

4.3.2 The Carry Over Theorem

We now show that in the above setting, we can compute the dependence relation between transitions for all parameter sizes $N > 1$, by computing the relation for a small size, calculated as described below.

Enabledness: Given a pair of rules $\langle g_1, a_1 \rangle$ and $\langle g_2, a_2 \rangle$, they satisfy the enabledness condition when:

$$g_1(s) \wedge g_2(s) \Rightarrow g_1(a_2(s)) \wedge g_2(a_1(s)) \tag{4.1}$$

is valid over S, the set of all states (interpretations) s, $g_1(s)$ denotes the evaluation of the formula g_1, given the interpretation s of the variables, $a_1(s)$ (with a slight abuse of notation) denotes the application of the substitutions represented by a_1 to the variables, followed by an evaluation of the resulting terms over the interpretation s. Similarly for g_2 and a_2.

Note that the set S is actually parameterized over the *system size* N. We will henceforth denote by $S(N)$ the set of all interpretations to the variables for system size N.

We would like to find a bound, $\widehat{N}$, such that 4.1 is valid over $S(N)$ for all N, $N > 1$ iff it is valid over $S(N)$ for all N, $1 < N \leq \widehat{N}$. To arrive at such a bound, we proceed as follows: first, we convert guards g_1 and g_2 into CNF, and also push negations inside atomic formulas of type $\mathcal{Q}$ (ie, a formula $\neg \forall i.ar_j[i]$ is converted into the equivalent formula $\exists i.\neg ar_j[i]$, and so on for every atomic formula of type $\mathcal{Q}$). Let the cardinality of the set $\mathcal{V}_2$ be k, the number of *existentially quantified* atomic formulas of type $\mathcal{Q}$ in a guard g_i be e_i, and the number of *universally quantified* atomic formulas of type $\mathcal{Q}$ in g_i be u_i.

Claim: 4.1 is valid over $S(N)$ for all N, $N > 1$ iff it is valid over $S(N)$ for all N, $1 < N \leq \widehat{N}$, where $\widehat{N} = b + 2(\max_{1<i<2} e_i + \max_{1<i<2} u_i)$.

Proof: To show 4.1, it is sufficient to show that its negation:

$$g_1(s) \wedge g_2(s) \wedge (\neg g_1(a_2(s)) \vee \neg g_2(a_1(s))) \tag{4.2}$$

is satisfiable for $N > \widehat{N}$ iff it is satisfiable for some N, $1 < N \leq \widehat{N}$. To show this, moreover, it is sufficient to show that if 4.2 is satisfiable over $S(N)$, for $N > \widehat{N}$, it is satisfiable over $S(\widehat{N})$.

Since $g_1(s)$, $g_2(s)$, $g_1(a_2(s))$, and $g_2(a_1(s))$ are all formulas of the language, they can be converted into CNF. Doing so, we can rewrite 4.2 as:

$$\underbrace{\bigwedge_{1<i<k} c_i(s)}_{g_1(s)} \wedge \underbrace{\bigwedge_{1<j<l} d_j(s)}_{g_2(s)} \wedge \Big(\big(\underbrace{\neg \bigwedge_{1<i<k} c_i(a_2(s))}_{\neg g_1(a_2(s))} \big) \vee \big(\underbrace{\neg \bigwedge_{1<j<l} d_j(a_1(s))}_{\neg g_2(a_1(s))} \big) \Big) \tag{4.3}$$

where the c_i are the clauses in the CNF representation of g_1, and d_j are the clauses in the CNF representation of g_2. Given an interpretation $s \in S(N)$ that satisfies 4.3, we now show how to construct an interpretation $\hat{s} \in S(\widehat{N})$ that also satisfies it. Clearly, to satisfy 4.3, we need to satisfy each of the following:

$$\bigwedge_{1<i<k} c_i(s) \tag{4.4}$$

$$\bigwedge_{1<j<l} d_j(s) \tag{4.5}$$

$$\Big(\big(\neg \bigwedge_{1<i<k} c_i(a_2(s)) \big) \vee \big(\neg \bigwedge_{1<j<l} d_j(a_1(s)) \big) \Big) \tag{4.6}$$

Let us consider each of the above formulas in turn. Each clause c_i (d_j) in 4.4 (4.5) is a set of disjuncts, where each disjunct is an atomic formula. Recall that atomic formulas are either well-typed relational expressions over terms, or formulas of type $\mathcal{Q}$. Without loss of generality, we can introduce new variables to the set $\mathcal{V}_2$ to replace every quantifier variable in existentially quantified atomic formulas in c_i (d_j), one for each such formula. Doing so yields a total of $e_1 + e_2$ new variables $ev_1, ev_2, \ldots ev_{e_1+e_2}$ of type $\mathcal{V}_2$.

The formula in 4.6 can be rewritten as:

$$\Big(\bigvee_{1<i<k} \neg c_i(a_2(s)) \;\vee\; \bigvee_{1<j<l} \neg d_j(a_1(s))\Big)$$

Each clause c_i (d_j) is a set of disjuncts $c_{i1}, c_{i2}, \ldots c_{im_i}$ ($d_{j1}, d_{j2}, \ldots d_{in_j}$). Pushing the negation inside, therefore, we get:

$$\Big(\Big(\bigvee_{1<i_1<k} \bigwedge_{1<i_2<m_{i_1}} \neg c_{i_1 i_2}(a_2(s))\Big) \;\vee\; \Big(\bigvee_{1<j_1<l} \bigwedge_{1<j_2<n_{j_1}} \neg d_{j_1 j_2}(a_1(s))\Big)\Big)$$

which can be written in CNF by multiplying out the conjuncts:

$$\begin{aligned}
&(\neg c_{i_1 1}(a_2(s)) \vee \neg c_{i_2 1}(a_2(s)) \vee \ldots \neg c_{i_k 1}(a_2(s)) \\
&\quad \vee \neg d_{j_1 1}(a_1(s)) \vee \neg d_{j_2 1}(a_1(s)) \vee \ldots \neg d_{j_l 1}(a_1(s))) \\
\wedge&(\neg c_{i_1 1}(a_2(s)) \vee \neg c_{i_2 2}(a_2(s)) \vee \ldots \neg c_{i_k 1}(a_2(s)) \\
&\quad \vee \neg d_{j_1 1}(a_1(s)) \vee \neg d_{j_2 1}(a_1(s)) \vee \ldots \neg d_{j_l 1}(a_1(s))) \\
\wedge&\ldots \\
\wedge&(\neg c_{i_1 m_{i_1}}(a_2(s)) \vee \neg c_{i_2 m_{1_2}}(a_2(s)) \vee \ldots \neg c_{i_k m_{i_k}}(a_2(s)) \\
&\quad \vee \neg d_{j_1 n_{j_1}}(a_1(s)) \vee \neg d_{j_2 n_{j_2}}(a_1(s)) \vee \ldots \neg d_{j_l n_{j_l}}(a_1(s)))
\end{aligned}$$

As before, each disjunct above is an atomic formula. In this case, since each atomic formula has an outer negation, universally quantified atomic formulas become existentially quantified atomic formulas, and *vice versa*, upon pushing the negation inside. Therefore, we can replace the quantifier variable in each new existentially quantified atomic formula by a new variable of type $\mathcal{V}_2$. Doing so introduces $u_1 + u_2$ new variables $uv_1, uv_2, \ldots uv_{u_1+u_2}$. Added to the earlier $e_1 + e_2$ variables, we have totally introduced $e_1 + e_2 + u_1 + u_2$ new variables of type $\mathcal{V}_2$. Note that:

$$e_1 + e_2 + u_1 + u_2 \leq 2(\max_{1<i<2} e_i + \max_{1<i<2} u_i) \tag{4.7}$$

To summarize, we now have the following sets of variables that are assigned values by the interpretation $s \in S(N)$ that satisfies 4.3:

- $V_1 = \{x_1, x_2, \ldots x_a\}$ of boolean type

- $V_2 = \{y_1, y_2, \dots y_b, ev_1, ev_2, \dots ev_{e_1+e_2}, uv_1, uv_2, \dots uv_{u_1+u_2}\}$ of type $[1..N]$
- $V_3 = \{ar_1, ar_2, \dots ar_c\}$, arrays with index $[1..N]$ of boolean type.

The key idea behind finding an interpretation of size $\widehat{N}$ is to project the interpretation of variables in V_1 and V_3 from N to $\widehat{N}$, and to find a suitable permutation of the interpretation of size N from which to project the interpretations of variables in V_2, as follows. The interpretation s can assign at most α different values to the variables in V_2, $\alpha \leq \widehat{N}$ (from 4.7). Without loss of generality, assume that these values are $v_1 < v_2 < \dots < v_\alpha$. Since the system is symmetric, there is a permutation Π over the indices $[1..N]$, such that $\Pi^{-1}(v_k) = k$, for every $k \in 1..\alpha$. Let $\tilde{s}$ be the Π-variant of s, applying the permutation-induced transformation to the set of variables above. Clearly, $\tilde{s}$ is also an interpretation that satisfies 4.3. To construct the interpretation $\hat{s} \in \widehat{N}$ that satisfies 4.3, we let $\tilde{s}$ and $\hat{s}$ agree on the interpretation of the variables in V_1 and V_2. For the remaining variables $ar_1, ar_2, \dots ar_c$, we let $\tilde{s}$ and $\hat{s}$ agree on the values of all $ar_i[k]$, for $k \leq \alpha$. After our transformations to CNF, and replacing existentials by new variables, the formula 4.3 is a formula over the variables in V_1 and V_2, and universally (over the parameter size) quantified expressions over V_3. Since $\tilde{s}$ and $\hat{s}$ agree on the interpretation of all of the above, $\hat{s}$ satisfies 4.3 over $S(\widehat{N})$.

This proves that the *enabledness* relation for a particular pair of rules can be computed for a limited size, and deduced for all higher sizes. By lifting the analysis to all pairs of rules, it is clear that we can arrive at a globally maximum $\widehat{N}$ that suffices as a bound for checking the enabledness relation for *every* pair of rules.

Commutativity, given a pair of rules $\langle g_1, a_1 \rangle$ and $\langle g_2, a_2 \rangle$, is stated as follows:

$$(\forall s \in S.\ g_1(s) \wedge g_2(s) \Rightarrow a_1(a_2(s)) = a_2(a_1(s))) \tag{4.8}$$

As before, we would like to show that, to deduce the validity of 4.8 for all $N > 1$, it is sufficient to its validity for a small size.

Claim: 4.8 is valid over $S(N)$ for all N, $N > 1$ iff it is valid over $S(N)$ for all N, $1 < N \leq \widehat{N} = b + 2(\max_{0<i<R} e_i + \max_{0<i<R} u_i)$.

Proof: Reasoning as before, it is sufficient to show that, if:

$$g_1(s) \wedge g_2(s) \wedge \neg(a_1(a_2(s)) = a_2(a_1(s))) \tag{4.9}$$

is satisfiable over $S(N)$, for $N > \widehat{N}$, it is satisfiable over $S(\widehat{N})$.

Once again, we can convert the formulas representing the guards into CNF, to obtain:

$$\underbrace{\bigwedge_{1<i<k} c_i(s)}_{g_1(s)} \wedge \underbrace{\bigwedge_{1<j<l} d_j(s)}_{g_2(s)} \wedge (\neg(a_1(a_2(s)) = a_2(a_1(s)))) \tag{4.10}$$

Therefore, any interpretation that satisfies 4.10 must satisfy each of the following:

$$\bigwedge_{1<i<k} c_i(s) \tag{4.11}$$

$$\bigwedge_{1<j<l} d_j(s) \tag{4.12}$$

$$\neg(a_1(a_2(s)) = a_2(a_1(s))) \tag{4.13}$$

As before, we can replace the quantifier variables of existentially quantified expressions in c_i and d_j with $e_1 + e_2$ new variables $ev_1, ev_2, \ldots ev_{e_1+e_2}$ of type $\mathcal{V}_2$. Since actions are independent of the parameter size, if formula 4.13 is satisfied by an interpretation of any size, it is satisfied by some interpretation of every size. Therefore, we do not consider it any further.

Given an interpretation $s \in S(N)$ that satisfies 4.11 and 4.12, we can construct an interpretation $\hat{s}$, as before, that also satisfies these equations.

Thus, we have shown that both commutativity and enabledness, and therefore, the independence relation, can be computed for limited sizes, and then deduced for any higher size. Chapter 7 discusses experimental results using the carryover theorem for parameterized systems.

CHAPTER 5

PARTIAL ORDER REDUCTION FOR TRANSACTIONAL SYSTEMS

5.1 Introduction

Systems of interacting components are often designed to perform a certain task jointly, by following a *protocol* of some kind. Examples of such systems include cache coherence protocols, mutual exclusion algorithms, security protocols, and various network protocols, such as leader election protocols. The protocols enforce an orchestrated mechanism of interaction between the communicating components, which we call *transactions*. An example of a transaction is the sequence of steps carried out by a set of processors executing a directory based cache coherence protocol, starting from a request for a particular cache line by one processor, and terminating with either the grant or the denial of the request by the directory controller. transaction. A transaction can involve actions of multiple components (the requesting node and the directory controller in the example above). This is a slightly different notion than the notion of a transaction as a sequence of actions within a single thread (or component), as presented in [39], for example.

Commonly, transactional systems are designed to allow only a single ongoing transaction at any one time, or, even if multiple transactions are permitted, they are independent of each other. In such cases, it is unnecessary to interleave the individual steps of these transactions with each other. In this chapter, we discuss a method for taking advantage of this transactional nature to enhance the performance of partial order reduction algorithms.

Secondly, our symbolic, SAT based computation of the independence relation

starts from a completely general symbolic state, and is therefore conservative, since it actually only matters whether a given pair of transitions is independent at all *reachable* states. In this chapter, we also examine how the guards of transitions can be strengthened to refine the independence analysis, and how these strengthenings can be proved safe for the property of interest.

5.2 Transaction-based priorities for ample set construction

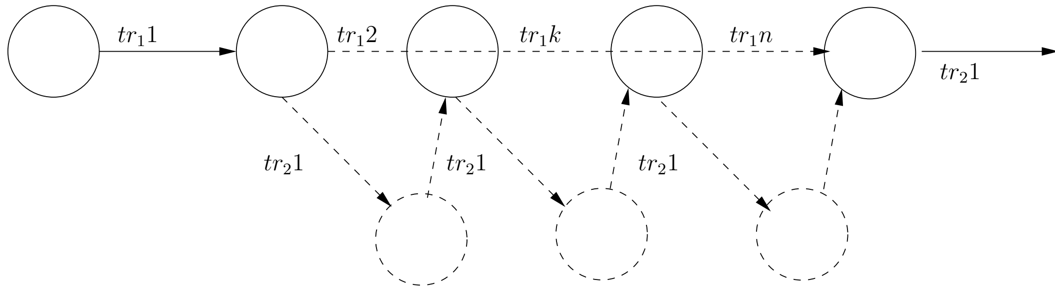

Figure 5.1. Interleaving of independent transactions

To understand how the transactional nature of systems can be exploited during partial order reduction, consider a typical scenario of directory based cache coherence protocols, as depicted in Figure 5.1. In this scenario, one cache controller makes a request for a line ($tr_1 1$), and, while this request is being processed, a second controller makes a request for a different line ($tr_2 1$). Since cache controllers typically process only one request at a time, the second request is recorded, but cannot progress further until the first request has been processed. However, the intermediate transitions of the ongoing transaction for the first request (the $tr_1 k$s), as well as the first step of the second request, *both* satisfy the ample set conditions. This means that the partial order reduction algorithm could choose either as an ample set. It is evident, however, that picking the transitions that belong to the ongoing transaction will result in fewer states being explored. In particular, none of the states in dotted lines in Figure 5.1 will be explored if we always choose the transitions from the ongoing transaction for the ample set.

To enable the partial order reduction algorithm to make that choice, we have developed the notion of *transaction guided priorities* for transitions. These priorities are then used to guide the process of computing an ample set at runtime.

5.2.1 Assigning transition priorities

Recall that our algorithm begins the process of constructing an ample set by picking a *seed transition.* Since the overall goal of a partial order reduction algorithm is to minimize the number of states visited globally, and to minimize the size of the ample set locally, our default criterion for picking a seed transition is to try and minimize the number of other transitions it is dependent on. This is achieved by computing, statically, the number of variable references in each transition. This information is used at runtime to pick the enabled transition with the smallest variable count as the seed transition.

In the case of transactional systems, we would like to change this strategy to pick a transition that is part of the ongoing transaction as our seed transition. To do this, we note that transitions can be classified into three broad groups, based on their position in a transaction. The first group is the set of transitions that *initiate* a transaction, such as a request for a cache line in a cache coherence protocol. The second is the set of transitions that *complete* a transaction. In this category would fall transitions such as the grant of the cache line, sent by the directory controller to the node that made the request. Finally, all of the other transitions can be classified as *intermediate* transitions.

Now, since we would like to preferentially pick transitions that are part of an ongoing transaction, we assign priorities to the transitions in *reverse* order of their position within transactions. That is, we assign the highest priorities to transitions that complete transactions, medium priorities to intermediate transitions, and the lowest priorities to transitions that initiate transactions. The effect of this scheme can be seen by revisiting Figure 5.1. Since all of the $t_1 i$s, being intermediate transitions, will be assigned a higher priority than $t_2 1$, an initiating transition, $t_2 1$ will be deferred until the transaction t_1 has completed. Effectively, this results in "scheduling" ongoing transactions with greater priority, and postponing the start of

new transactions as long as possible. Note that this is completely sound, because we will only be able to postpone the start of a new transaction as long as the transition that starts it is independent of transitions that belong to the ongoing transaction.

5.2.2 Determining transition positions within a transaction

In order to make effective use of this prioritization of transitions, it is necessary to identify the location of transitions within a transaction. Currently, our algorithm does not attempt to automatically deduce this information, relying instead on the user/designer to provide this input. Most designers clearly know the situation of rules within a transaction. Another idea we have tested [10] is to try to obtain the situation of a rule through concrete execution on small instances of the protocol. For example, if we are given, or can identify, the transitions that start various transactions of a protocol, we can run a simulation on a modified protocol where only the transition that begins a transaction is enabled at the initial state. Recording the sequence in which subsequent transitions become enabled gives us an idea of the positions of transitions in that transaction.

An important feature of this method is that, since it only *improves* the efficacy of the reduction, there is no penalty to pay for incorrectly identifying the position of transitions. It is possible, of course, in the case that a wrongly prioritized transition has a high variable reference count, for the optimization to actually perform worse than the default heuristic for picking a seed transition, but there is no risk of the reduction becoming unsound because of incorrect transition priorities. The results of applying this heuristic while computing ample sets are discussed in Chapter 7.

5.3 Refining the independence relation

Our algorithm for independence analysis, as presented so far, relies on performing a symbolic simulation of a transition $\langle g, a \rangle$ to characterize the state $a(s)$ arrived at by executing the transition from a given state s. Following the definitions of *enabledness* and *commutativity* (Chapter 2, we do not constrain the state s in any way. In particular, we do not attempt to restrict our attention to only the

reachable states of the system. However, it is evident that the independence of transitions at unreachable states is inconsequential to the partial order reduction algorithm. Thus, it would be sound to compute independence only with respect to the reachable states of a given system.

Restricting our attention to only the reachable states would require us to come up with a way of characterizing the reachable states, for example, using an invariant predicate. However, such predicates are difficult to arrive at without either a lot of manual/machine-assisted reasoning, or an actual model checking run. We can, nevertheless, attempt to refine the independence relation in a different way. Since our concern is with transitions which are independent at all reachable states, but dependent at unreachable states, we can attempt to *strengthen* the guards of these transitions, so that their independence is extended to *all* states of the system.

Of course, one can argue that for transitions which are actually independent at reachable states because they will never be enabled together, it does not matter whether the independence analysis classifies them as independent or not. While it does not matter for the actual selection of transitions to the ample set, recall that our implementations of condition **C1** check the set of *disabled* transitions dependent on transitions in the ample set. Here, a more accurate independence relation will increase the chances of satisfying the **C1** condition, and thus yield higher reductions. Coming up with predicates to strengthen guards is a hard problem, and often requires sharp insights into the nature of the protocol itself. In Chapter 7, we discuss strategies we have used to discover these strengthening predicates.

5.3.1 On-the-fly verification of strengthened guards

Once we have strengthened the guards of transitions, it is necessary to show that these strengthenings are sound, and do indeed preserve the semantics of the original transitions. We now show that it is sufficient to model check the strengthened system with a modified property, to be able to prove the soundness of the strengthenings.

Since Murphi transitions are guard action pairs (g_i, a_i), strengthening the guards corresponds to adding predicates p_i to the guards of transitions t_i. Define the

strengthening operator Θ over transitions such that:

$$\Theta(\langle g_i, a_i \rangle) = \begin{cases} \langle g_i \wedge p_i, a_i \rangle & \text{if } t_i \text{ is strengthened} \\ \langle g_i, a_i \rangle & \text{otherwise} \end{cases}$$

We extend Θ to apply to runs $\sigma = \langle s_1, t_1, s_2, \ldots, s_k, \ldots \rangle$ so that $\Theta(\sigma)$ results in the sequence (not necessarily a run) $\langle s_1, \Theta(t_1), s_2, \ldots, s_k, \ldots \rangle$. Let the original system be $\mathcal{F}$, and the modified system $\mathcal{F}'$. Note that both systems have the same set of states, and the same initial state predicate I. Assume that the property to be verified of the original system was P. We model check the new system with the property $P \wedge Str$, where:

$$Str = (g_{i_1} \rightarrow p_{i_1}) \wedge (g_{i_2} \rightarrow p_{i_2}) \wedge \ldots \wedge (g_{i_k} \rightarrow p_{i_k})$$

$g_{i_1} \ldots g_{i_k}$ are the k guards of the original system that have been strengthened with the predicates $p_{i_1} \ldots p_{i_k}$.

Definition. A run $\sigma = \langle s_1, s_2, \ldots, \rangle$ of a system satisfies an invariant property P, written as $\sigma \models P$, iff the property is true at every state in the run.

Definition. A system $\mathcal{M}$ satisfies an invariant property P, written as $\mathcal{M} \models P$, iff every run of the system satisfies the property.

Theorem 5.1.

$$\mathcal{F}' \models P \wedge Str \Rightarrow \mathcal{F} \models P$$

Proof: By contradiction. Assume that the antecedent of the theorem is true, and assume that there is a run $r = \langle s_1, t_1, s_2, t_2, \ldots, s_m, \ldots \rangle$ of $\mathcal{F}$ that does not satisfy the property P. Without loss of generality, we assume that $s_1, s_2, \ldots s_{m-1} \models P$, and $s_m \not\models P$. If $\Theta(r)$ is a run of $\mathcal{F}'$, $s_m \models P \wedge Str$, which implies that $s_m \models P$, contradicting our assumption. Therefore, assume that $\Theta(r)$ is not a run of $\mathcal{F}'$. Then, there is a t_k, such that $\Theta(t_k)$ is not enabled at s_k, and $\Theta(\langle s_1, t_1, s_2, \ldots s_k \rangle)$ is a valid prefix of a run of $\mathcal{F}'$, such that $s_1, s_2, \ldots s_k \models P \wedge Str$. Since r is a run of $\mathcal{F}$, t_k is enabled at s_k.

Case 1: $\Theta(t_k) = \langle g_k, a_k \rangle$. In this case, since $\Theta(t_k)$ is not enabled at s_k, this implies that $s_k \not\models g_k$. But we know that t_k is enabled at s_k. That is, $s_k \models g_k$, leading to a contradiction.

Case 2: $\Theta(t_k) = \langle g_k \wedge p_k, a_k \rangle$. In this case, we have $s_k \not\models g_k \wedge p_k$. However, we know that $s_k \models g_k$. Also, $s_k \models P \wedge Str$. That is, $s_k \models g_k \rightarrow p_k$. Therefore, $s_k \models g_k \wedge p_k$, leading to a contradiction.

Thus, in every case, we arrive at a contradiction, and hence, the theorem is true, and, by model checking the strengthened system for the property $P \wedge Str$, we can prove that the strengthenings of the guards are sound. If the model check fails, on the other hand, we have to manually examine the error trail to determine whether the property failed, or whether one of the strengthenings does not hold, and rerun the model check after making the necessary changes, in the latter case.

CHAPTER 6

ENABLING SYMBOLIC EXPLORATION OF RULE BASED SYSTEMS

Rule based languages have traditionally been used to specify systems that are verified using explicit-state model checking methods, such as in Murphi. In many cases, this approach is the most efficient, since rule based specifications often have many global variables, and global dependencies among the variables, which is known to cause BDD sizes to blow up [23]. However, there are cases where a BDD based analysis might prove to be equally, or more efficient, and the lack of tools with support for symbolic state exploration of rule-based systems has so far made such experiments difficult to perform. Symbolic model checkers have traditionally been aimed at the verification of hardware circuits, and employ a per-variable update modeling paradigm that makes it difficult to specify protocol like systems with significant global interdependencies among variables.

In this chapter, we describe a translation scheme from the rule-based description language of Murphi into the input language of the symbolic verification tool NuSMV. This translation makes the current state-of-the-art symbolic techniques available to rule-based system descriptions, and no longer forces designers to choose between the convenience of a feature-rich, high-level language, and the compact representation of symbolic methods.

As with any translation scheme, it is important, while translating Murphi specifications, to preserve the semantics of the specifications. This poses an interesting problem, since specifications in Murphi consist of *guard*/*action* pairs called rules, which are interleaved in all possible ways to simulate concurrency. Moreover, an *action* is a sequence of updates to variables that are executed serially, and atomically,

whenever a rule is fired. The primary challenge in achieving the translation was to convert these serial, atomic-update semantics of Murphi rules into the parallel, per-variable update style of NuSMV. Here, we show a simple, symbolic simulation based solution to this problem.

6.1 The NuSMV Verification System

NuSMV traces its roots to the SMV [32] tool, the first model checker based on BDDs, developed at CMU. It is a verification tool for finite state systems, and the latest version provides both BDD based full symbolic model checking, as well as SAT based bounded model checking, which we have already introduced in chapter 3. Both NuSMV and its predecessor, SMV, whose input language NuSMV has largely retained, were designed with the goal of verifying *hardware circuits*, and this is reflected in their choice of input language. NuSMV compiles a system description into BDDs, which can then be used either for symbolic model checking, or converted to SAT formulas for bounded model checking.

6.1.1 The description language

A NuSMV system description consists of a number of modules, with one unique `main` module. Each module may consist of a declaration of state variables, which can only be of finite types; an initialization of the state variables, which defines the initial state of the system; and a set of assignments representing the transition relation, which define the values of state variables in the next state, given their values in the current state. NuSMV uniformly adopts a *parallel assignment* semantics, and performs compile-time checks for circular or multiple assignments to variables. Figure 6.1 shows a description of a mutual exclusion protocol in the NuSMV language.

Variable declarations:

- Variable declarations are preceded by the `VAR` keyword. Variables are either state variables of boolean, integer subrange, enumerated, or array types, or names for instances of modules.

Initialization:

- Initializations are performed either in an `INIT` block, which can be used to define a predicate over the state variables that holds at the initial state, or in an `ASSIGN` block, such as in the figure.

Transition Relation:

- The transition relation can also be specified in one of two ways - either in a `TRANS` block, through a predicate relating the values of the state variables in the current state with their values in the next state, or in an `ASSIGN` block, as in the figure.

There are two notable aspects of the description above, with regards to our translation scheme. First, notice that, in NuSMV's world-view, the transition relation is described on a per-variable basis. That is, the next state assignment of each variable defines, completely, how that state variable changes its value as the system evolves. The second aspect of note is that, as mentioned earlier, NuSMV adopts the *parallel assignment* semantics. This means that any sequentiality in assignments can only be non-circular. We describe later the significance of this fact to our translation.

NuSMV allows the modeling of asynchronous systems, through the definition of modules as a collection of parallel processes. In this mode, NuSMV nondeterministically picks a module to run at each step, modeling interleaving concurrency. However, the next state assignments of the module picked are still carried out in parallel. Each process module is associated with a `running` boolean variable, which is true exactly when that process is running. Since NuSMV allows fairness constraints, which restrict attention to *fair execution paths*, this can be used in conjunction with the `running` variable to ensure that each process is scheduled infinitely often.

It is this mode of NuSMV that we will use in our translation, since it allows us to closely match Murphi's rule semantics.

```
MODULE main
 VAR
   semaphore : boolean;
   proc1 : process user(semaphore);
   proc2 : process user(semaphore);
 ASSIGN
   init(semaphore) := 0;

MODULE user(semaphore)
 VAR
   state : idle, entering, critical, exiting ;
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
       state = idle                   : {idle, entering};
       state = entering & !semaphore  : critical;
       state = critical               : {critical, exiting};
       state = exiting                : idle;
      1                              : state;
     esac;
   next(semaphore) :=
     case
       state = entering : 1;
       state = exiting  : 0;
       1                : semaphore;
    esac;
 FAIRNESS
   running
```

Figure 6.1. Example of a NuSMV system description

6.2 Murphi's Rule Based Description Language

Recall that a Murphi description consists of constant and type declarations, followed by variable declarations, function and procedure declarations, a startstate description that defines the initial states of the finite state machine being described, and rules - guard/action pairs that define the transition relation of the machine.

Murphi's description language permits the same basic types as NuSMV, and additionally allows record types. Murphi's overall program structure is as follows:

$$\begin{array}{lll} \langle Program \rangle & ::= & \langle decl \rangle \\ & & \langle procdecl \rangle \\ & & \langle rules \rangle \end{array}$$

$\langle decl \rangle$ includes constant, type and variable declarations. $\langle procdecl \rangle$ includes any function and procedure declarations. Finally, $\langle rules \rangle$ represent the rules (transitions) of the system. In the following sections, we describe our algorithm for translating each of these parts of a Murphi description into NuSMV.

6.3 Translating Murphi

As mentioned briefly at the outset, the primary challenge in translating Murphi descriptions into NuSMV lay in converting the sequential and atomic updates of state variables by Murphi's rules into the globally parallel, per-variable update semantics of NuSMV. To understand why this issue poses a problem, let us examine a fragment of a typical Murphi rule's action:

```
a := MAX;
for i: pid
do
  if (a < b[i])
  then
    a := b[i];
  endif;
endfor;
```

In the above example, the variable `a` is potentially assigned multiple times during the execution of the action. Since NuSMV only allows one update statement per variable, a direct translation of the action above is impossible. In general, the sequential update paradigm of Murphi means that within a single rule, updates to state variables can be *cumulative*, which cannot be modeled directly using parallel-update semantics.

Our solution to the above problem is to *symbolically simulate* each Murphi rule. By symbolically simulating the rule, we accumulate updates to each variable over the course of a rule's action. The final result is an expression for each variable's next state value in terms of the current values of the state variables. These expressions are then emitted as NuSMV code to update each variable. The code corresponding to each rule is packaged as a NuSMV *module*.

Since Murphi's semantics for rules allow full interleaving of concurrently enabled rules, we instantiate each module as a *process*. The details are in section 6.3.3.

6.3.1 Variable declarations

Murphi allows the same basic types as NuSMV, with the addition of record types. We divide the variables $\mathcal{V}$ of a Murphi system description into two disjoint subsets $\mathcal{V}_R$ and $\mathcal{V}_S$, such that $\mathcal{V}_R$ contains all of the variables that have a record component, and $\mathcal{V}_S$ contains the simple, non-record variables. The variables in $\mathcal{V}_S$ are converted directly into variables of the same name in NuSMV. A variable `v` in $\mathcal{V}_R$ has one of the following type-schemas:

```
v : record
      f1 : t1;
      f2 : t2;
         :
         :
      fk : tk;
    endrecord;
```

or,

```
v : array [ind] of record
                     f1 : t1;
                     f2 : t2;
                        :
                        :
```

```
fk : tk;
endrecord;
```

where `t1-tk` are any valid Murphi types, including record types. In the first case, we recursively translate the `k` new variables `v_f1`, `v_f2`, ... `v_fk`, of types `t1`, `t2`, ... `tk` respectively. In the second case, we recursively translate the `k` new variables `v_f1`, `v_f2`, ... `v_fk`, of types `array [ind] of t1`, `array [ind] of t2`, ... `array [ind] of tk` respectively. Effectively, this results in splitting all record types into multiple variables for each of the fields of the record, and pushing all of the array components to the end.

6.3.2 Functions and Procedures

NuSMV does not support functions and procedures, while Murphi allows them. Therefore, we handle functions and procedures much as a compiler would. Procedures are (possibly parameterized) blocks of code, and are handled by in-lining them (with any formal parameters replaced by actuals), before handing the code over to the symbolic simulator. Functions have return values, and are handled by converting all return statements in the function body into assignments to a special symbolic return variable. A function call is then symbolically simulated, after similarly replacing the formal parameters with actuals. The special return variable holds the symbolic return value of the function. Although Murphi supports recursive functions, we have rarely encountered them in practice, and do not support the translation of recursive functions.

6.3.3 Rules

Murphi expresses the transition relation of the system being described as a set of *rules*. Each rule is a guard/action pair, where the guard is a predicate over the current values of the state variables, and the action is a sequence of assignments to state variables. The semantics of assignment are that in the next time step, the state variables will have the values assigned to them by the current rule's action. Murphi also allows parameterized *rulesets*, which are used to represent a set of

rules that share the same schema, and typically operate on variables parameterized by the ruleset parameters. To translate the Murphi transition relation, we expand rulesets out into the set of actual rules they represent. We now have an expanded set of rules, each of which is a pair of the form $\langle g, a \rangle$. We convert each rule into the corresponding `if` statement `if g then a endif;`, where `g` is the guard of the rule, and `a` the action. This `if` statement is then symbolically simulated, to produce the effect of executing the rule. The resulting NuSMV statements generated are all packaged into a module, one for each rule. Figure 6.2 shows an example of a Murphi rule, and its translation into NuSMV.

As can be seen from the figure, the rule has been translated into a parameterized module, with the parameters being the state variables of the system. Since the semantics of NuSMV's module instantiation is call-by-reference, passing all of the state variables as actuals to each module allows the modules to make updates to the state variables. In the figure, for example, the variable `BB`'s next state value is set at the end of the module `RULE_1`.

To describe how the symbolic simulation proceeds, we now examine the translation of the rule in Figure 6.2. The rule essentially consists of a guarded *if-then-else* statement. NuSMV allows one to label expressions with a name, using the `DEFINE` construct, and `VAR_7` is used to name the expression representing the guard of the rule. Similarly, `VAR_8` is used to name the condition of the *if* statement. `VAR_10` then represents the condition when the guard is true, but the *if* statement's condition is false. That is, it represents the condition under which the *else* branch is taken. `VAR_9` is declared as a variable of the type `0 .. 100`, since that is the type of the variable `BB`, whose value it represents. Specifically, `VAR_9` represents the value of `BB` when the else branch is taken. It is assigned the result of a case statement, which has the value `0` when the condition for the *else* branch is true, and the current value of `BB` otherwise. `VAR_12` is the name of the condition for the *if* branch, and `VAR_11` is assigned the result of another case statement, which takes the value `(((VAR_9) + (2)) mod (100))` when the condition for the *if* branch is satisfied, and `VAR_9` otherwise. Finally, in the next state, `BB` is assigned `VAR_11`.

```
rule "1"
 bb < 32 ==>
 if ((bb % 2) = 0)
 then
   bb := ((bb + 2) % 100);
 else
   bb := 0;
 endif;
end;
```

(a) Murphi rule

```
MODULE RULE_1( BB)

DEFINE VAR_7 := (BB) < (32);
DEFINE VAR_8 := ((BB) mod (2)) = (0);
DEFINE VAR_10 := !(VAR_8) & (VAR_7);
VAR VAR_9 : 0 .. 100;
ASSIGN VAR_9 := case
(VAR_10) : (0);
TRUE : (BB);
esac;
DEFINE VAR_12 := (VAR_8) & (VAR_7);
VAR VAR_11 : 0 .. 100;
ASSIGN VAR_11 := case
(VAR_12) : (((VAR_9) + (2)) mod (100));
TRUE : (VAR_9);
esac;
next(BB) := VAR_11;
```

(b) NuSMV translation of Murphi rule

Figure 6.2. A Murphi rule and the translated NuSMV module

Effectively, this means that if the condition for the *if* branch is satisfied (which means `VAR_9` has the original value of `BB`), the value of `BB` in the next state will be `((BB + 2) mod 100)`, and otherwise, if the condition for the *else* part is satisfied, the value of `BB` will be `0`, which is in accordance with the semantics of the Murphi

rule.

6.4 Conclusions and Future Directions

In this chapter, we have described an algorithm for translating Murphi's rule based specifications into NuSMV's input language, thus opening up the possibility of symbolically verifying rule based specifications. Our initial results are encouraging, as we have been able to translate and verify properties such as mutual exclusion of non-trivial examples like the Peterson mutual exclusion algorithm. We have not attempted so far to translate the specifications of the properties themselves from Murphi into NuSMV, although this is straightforward for Murphi's invariant assertions, which can be converted into the equivalent `LTLSPEC G` assertions in NuSMV.

We believe that having the capability to employ symbolic techniques on rule based specifications will open up further avenues for research in combining explicit state enumeration based methods with symbolic techniques. As an example, it may be possible to partition the transition relation of a rule based specification into disjoint subsets, some of which may be more efficient to explore symbolically, while others may be amenable to explicit state techniques. One can imagine that, in the case of transaction based protocols, for example, it might be efficient to collect all of the rules belonging to a single transaction, and translate those rules into their symbolic equivalents. The model checking run of the protocol could then proceed by using an explicit state representation for the parts of the graph that are not within a transaction, and using symbolic representations for the states along a transaction's path.

CHAPTER 7

IMPLEMENTATION, EXPERIMENTS AND RESULTS

7.1 Introduction

In this final chapter, we describe in some detail our implementation of the ideas presented so far, discuss experimental results, and outline future directions for extending the research of this dissertation.

7.2 POeM: Partial Order enabled Murphi

We have implemented our ideas in the **POeM** tool, an extension to the Murphi system. The Murphi model checking system compiles a model description in Murphi's input language into a specialized executable verifier, which can be used to check properties of the system. Murphi's input language has already been described in Chapter 2. As an explicit state model checker, Murphi implements both a breadth-first, and a depth-first search of the state space. This, and various other options, are selectable through command-line options to the executable verifier that is compiled. For details, the reader is referred to the Murphi user manual [33].

POeM's architecture is outlined in Figure 7.1. There are two main phases to a **POeM** verification run. First, the transition system is analyzed to generate the independence relation. The Murphi model is then compiled as usual into an executable verifier, incorporating the independence information. The verifier itself includes an option to turn on partial order reduction, in which case a depth-first search using ample sets is performed. In the next sections, we describe each of these two phases of **POeM**.

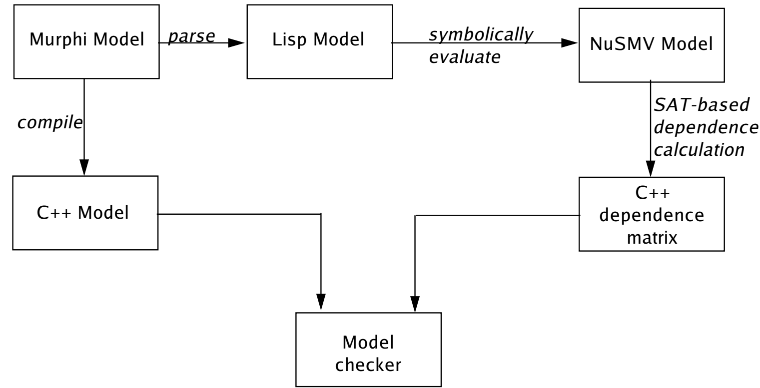

Figure 7.1. **POeM**'s architecture

7.2.1 Symbolic simulation of Murphi rules

POeM starts out by translating the given Murphi model into a set of Lisp s-expressions, for the purposes of symbolic simulation. Figure 7.2 shows a pair of rules from the Murphi model of the German protocol, and Figure 7.3 shows what they look like when converted into Lisp. Figure 7.4 then shows the enabledness and commutativity conditions expressed as Lisp s-expressions. It is these conditions that are actually symbolically simulated by **POeM**, to produce propositional expressions representing the enabledness and commutativity conditions.

The symbolic simulation proceeds by creating a hash table for each state variable. The hash table records the symbolic value of each state variable at every step of the simulation. The table entries for arrays and records are functions (lambda expressions) over the indices (for arrays), or field names (for records). For each value of the index or field name, the lambda expressions evaluate to a symbolic expression that represents the value of the referenced array cell or record field. Entries for scalar variables directly record the symbolic expression representing the variable's value. At the beginning of each symbolic simulation, the hash table is initialized to a set of fresh symbolic values for each variable. As the symbolic simulation progresses, NuSMV statements are emitted corresponding to the effect of simulating each statement. In the following, we describe how Murphi statements are simulated, and an outline of the corresponding NuSMV code that is generated.

```
rule "home sends reply to client -- shared"
  home_current_command = req_shared
  & !home_exclusive_granted
  & channel2_4[home_current_client] = empty ==>
begin
  home_sharer_list[home_current_client] := true;
  home_current_command := empty;
  channel2_4[home_current_client] := grant_shared;
end;

rule "home sends reply to client -- exclusive"
  home_current_command = req_exclusive
  & forall i: client do home_sharer_list[i] = false endforall
  & channel2_4[home_current_client] = empty ==>
begin
  home_sharer_list[home_current_client] := true;
  home_current_command := empty;
  home_exclusive_granted := true;
  channel2_4[home_current_client] := grant_exclusive;
end;
```

Figure 7.2. **POeM**: Murphi rules

7.2.1.1 Assignment statements

The assignment statement, in its basic form, updates the variable on the left hand side with the expression on the right hand side. For simplicity, let us assume that the variable on the left hand side is a scalar variable. Later we will outline how we extend the basic algorithm to array or record variables. Traditional symbolic simulation algorithms would store as the symbolic value of the variable on the left hand side, the expression on the right hand side, with variables occurring in the right hand side replaced by their symbolic values, as determined from the hash table. However, since we use NuSMV to generate the final SAT instance, we adopt a slightly different approach. We always store in the hash table the name of a NuSMV variable that represents the symbolic value of the Murphi state variable. As

```
(setq guard0
      '(AND (AND (= HOME_CURRENT_COMMAND REQ_SHARED)
                 (NOT HOME_EXCLUSIVE_GRANTED))
       (= (SUBSC CHANNEL2_4 HOME_CURRENT_CLIENT) EMPTY)))

(setq action0
      '(BEGIN (:= (SUBSC HOME_SHARER_LIST HOME_CURRENT_CLIENT) TRUE)
              (:= HOME_CURRENT_COMMAND EMPTY)
              (:= (SUBSC CHANNEL2_4 HOME_CURRENT_CLIENT) GRANT_SHARED)))
(setq guard1
      '(AND (AND (= HOME_CURRENT_COMMAND REQ_EXCLUSIVE)
                 (FORALL I 1 3 (= (SUBSC HOME_SHARER_LIST I) FALSE)))
            (= (SUBSC CHANNEL2_4 HOME_CURRENT_CLIENT) EMPTY)))

(setq action1
      '(BEGIN (:= (SUBSC HOME_SHARER_LIST HOME_CURRENT_CLIENT) TRUE)
              (:= HOME_CURRENT_COMMAND EMPTY)
              (:= HOME_EXCLUSIVE_GRANTED TRUE)
              (:= (SUBSC CHANNEL2_4 HOME_CURRENT_CLIENT) GRANT_EXCLUSIVE)))
```

Figure 7.3. **POeM**: Lisp s-expressions

mentioned earlier, the hash table is initialized with a set of fresh symbolic values for each variable. These are actually a set of NuSMV *input* variables, of the same type as the corresponding Murphi state variables. NuSMV input variables (in version 2.2 and older) represent unconstrained variables whose value is nondeterministically picked to be any one of the values from their respective domains, at each time step.

To evaluate the right hand side (rhs) of an assignment statement, we perform a lookup in the hash table for each state variable occurring in the rhs. Each of these lookups returns the name of a NuSMV variable, and we use these variables to construct, in NuSMV, an expression corresponding to the rhs. This expression is then assigned to a fresh NuSMV variable, and the hash table entry for the state variable on the left hand side is updated with this new variable name.

```
; Rule 0 independent of Rule 1
(setq r0r1 (list 'begin action0 action1))
(setq r1r0 (sublis var_list (list 'begin action1 action0)))
(setq commutes0_1
      (list 'if `(and ,guard0 ,guard1)
      (list 'begin r0r1 r1r0 check)
      (list 'begin
            '(:= CHECK_CHANNEL1 TRUE)
            '(:= CHECK_CHANNEL2_4 TRUE)
            '(:= CHECK_CHANNEL3 TRUE)
            '(:= CHECK_HOME_SHARER_LIST TRUE)
            '(:= CHECK_HOME_INVALIDATE_LIST TRUE)
            '(:= CHECK_HOME_EXCLUSIVE_GRANTED TRUE)
            '(:= CHECK_HOME_CURRENT_COMMAND TRUE)
            '(:= CHECK_HOME_CURRENT_CLIENT TRUE)
            '(:= CHECK_CACHE TRUE))))

(setq enabled0_1
      (list 'if `(and ,guard0 ,guard1)
      (list 'begin action0 `(:= CHECK_ENABLED ,guard1))
            '(:= CHECK_ENABLED true)))
(setq enabled1_0
      (list 'if `(and ,guard0 ,guard1)
      (list 'begin action1 `(:= CHECK_ENABLED ,guard0))
            '(:= CHECK_ENABLED true)))
```

Figure 7.4. **POeM**: Enabledness and commutativity conditions

7.2.1.2 Conditional statements

Conditional statements in Murphi are the `if-then-else` and the `case/switch` statements. The `if-then-else` is simulated by first evaluating the condition, and naming it using NuSMV's `DEFINE` construct [8]. Then, as each statement of the `else` block is simulated, we record the fact that we are in a statement block where the condition previously evaluated is false. Similarly, while evaluating the `if` block, we record that the condition is true. In case of nested conditionals, we thus

maintain a list of NuSMV names representing the various conditions, and their "polarities". Each assignment statement is then converted into a case statement in NuSMV, guarded by the conjunction of the various conditionals in the right polarities. Murphi's `case/switch` statement is similarly translated into a sequence of NuSMV case statement.

7.2.1.3 Loop statements

Although Murphi allows both `for` and `while` loops, we do not handle `while` loops, because the bound on such loops cannot always be statically computed. `For` loops are unrolled, and then treated as one large block of straight-line code, which is handled as above.

7.2.1.4 Function and procedure calls

Procedure calls do not return values, and are treated as macro expansions. Parameters can be passed either by value or by reference. The formal parameters and local variables of procedures are preprocessed to make their names unique, by prefixing the names with the procedure name. Entries for these names are then created in the hash table. In the case of parameters passed by value, the entries for the formal parameters are initialized to the values of the actual parameters. In the case of parameters passed by reference, references to the formal parameter in the procedure body are replaced by references to the actual parameter. The procedure body is then simulated as above.

Function calls are treated very similarly, except that they also have a return value. During preprocessing, all the `return` statements in the function body are replaced by assignments to a unique return variable. When a function call is encountered during symbolic simulation, the function body is simulated, and the function call replaced by the NuSMV variable representing the value of the return variable at the end of the simulation of the function body.

7.2.2 Ample set construction

The basic ample set construction algorithm has been outlined in Chapter 3. Here, we describe an interesting implementation detail that has led to improved performance in some cases. Recall that the ample set computation proceeds by picking a *seed transition*, and computing the transitive closure with respect to dependence. It is often the case that there are multiple candidate seed transitions. In such cases, our algorithm chooses the one with the highest priority (either by the variable reference heuristic, or the transaction based heuristic). However, it is possible that this choice leads to an improper ample set that does not satisfy one of the sufficient conditions. If the **C1** condition is violated, it is possible that a different ample set formed by choosing a different seed transition might still be feasible at that state[1]. In order to take advantage of this possibility, we allow the user to set a flag that instructs the algorithm to make two attempts to form an ample set. If the first attempt fails, the algorithm tries to pick an enabled transition not in the first ample set. While this increases the run time overhead, we have found that on a few examples, it can result in increased state reduction.

7.3 Experiments and Results

We have run **POeM** on a number of examples of different sizes, and Table 7.1 shows our overall results on some mutual exclusion algorithms and a cache coherence protocol. In the table, the columns under "Unreduced" represent the number of states explored, and the time taken for the verification to complete, without any partial order reduction. The columns under "Static PO" represent the same figures for the case where a static, syntax-based analysis was used to determine the independence relation, and the columns under "Symbolic PO" represent the figures for **POeM**. The final column, "Analysis Time", is the time taken by **POeM**'s symbolic evaluation based module to compute the independence relation. As can be seen, **POeM** is most effective on large examples, where the overhead of performing

[1]Note that **C2** is maintained by construction, since only invisible transitions are picked to be in the ample set, and **C3** is checked on-the-fly.

the symbolic analysis, and computing an ample set at each state, is outweighed by the savings that result from a far fewer number of states being explored.

Example	Static PO		Symbolic PO		Analysis
	States	Time	States	Time	Time
Bakery	157	0.1	119	0.1	8.9
Burns	82010	3.65	69815	11.67	52.9
Dekker	100	0.13	90	0.13	11.6
Dijkstra6	11664	0.88	4900	1.17	17.9
Dijkstra8	139968	8.81	33286	8.98	17.9
Dijkstra10	>1.5M	>1000.0	202248	82.6	17.9
DP6	1152	0.36	90	0.31	9.8
DP10	125952	7.86	823	0.48	9.8
DP14	>1.0M	>800.0	7395	2.6	9.8
Peterson2	26	0.15	24	0.15	4.6
Peterson4	22281	0.53	14721	0.58	4.6
German6	7378	1.36	2542	0.83	32.4
German8	42717	15.23	10827	4.6	32.4
German10	193790	131.83	36606	24.91	32.4

Table 7.1. Performance of partial order reduction algorithm

7.3.1 Guard Strengthenings and User-defined Priorities: A Case Study

We have tried our heuristics of user defined transition priorities and guard strengthenings on the German protocol, as well as the FLASH cache coherence protocol, and describe some of our experiences with these protocols here.

The German cache coherence protocol is a directory-based protocol for maintaining coherence among shared memory multiprocessors, proposed by Dr. Steven German as a verification challenge problem in 2000. The Murphi description of the protocol only models a single address/cache line, and a parameterized number of processors. We have included the descriptions of the protocol with and without guard strengthenings, as appendices to this report.

Our technique for generating predicates to strengthen guards is to first run **POeM** directly on the protocol, and analyze the resulting dependency matrix. For

pairs of rules that **POeM** marks dependent, we examine the test(s) that failed (enabledness, dependency, or both), and try to reason about predicates that, if added, would make the rules independent, without violating the properties we wish the protocol to hold. If we are able to come up with such predicates, we add them to the guard, and add the corresponding implication predicate to the invariant to be proved (see Chapter 5.

Run directly on the German protocol, **POeM** concludes that the rules "home receives invalidate acknowledgment" and "home sends invalidate message" are dependent on the rule "home sends reply to client - exclusive". In the first case, given that the German protocol only handles one outstanding transaction at a time, it is clear that these two rules can never even be enabled together, since, if the outstanding transaction is a request for exclusive access, the rule that grants access is only enabled when all of the invalidate acknowledgments have been collected by the home node. The guards for the two rules are:

```
rule "home receives invalidate acknowledgment"
    home_current_command != empty
    & channel3[cl] = invalidate_ack ==>

rule "home sends reply to client -- exclusive"
    home_current_command = req_exclusive
    & client_requests[home_current_client]
    & forall i: client do home_sharer_list[i] = false endforall
    & channel2_4[home_current_client] = empty ==>
```

In this case, it is reasonably obvious that the missing predicate to be added to the guard of the rule "home receives invalidate acknowledgment" is a predicate asserting that the client from which the invalidate acknowledgment is being received is actually on the sharer list. That is, the rule's guard is modified to:

```
rule "home receives invalidate acknowledgment"
    home_current_command != empty
```

```
    & channel3[cl] = invalidate_ack
    & home_sharer_list[cl] ==>
```

Adding this predicate helps **POeM** recognize that the two rules can never be simultaneously enabled, and since this is a precondition to the dependence tests, it will mark the rules independent.

In the second case, the guards are:

```
rule "home sends invalidate message"
    (home_current_command = req_shared & home_exclusive_granted
     | home_current_command = req_exclusive)
    & home_invalidate_list[cl]
    & channel2_4[cl] = empty ==>

rule "home sends reply to client -- exclusive"
    home_current_command = req_exclusive
    & client_requests[home_current_client]
    & forall i: client do home_sharer_list[i] = false endforall
    & channel2_4[home_current_client] = empty ==>
```

Here again, by similar reasoning, the two rules ought never to be enabled together, and therefore, marked independent. However, it is not apparent what predicate is to be added to enforce this. It is clear from the existing guards that, if the rules are to be simultaneously enabled, `home_current_command = req_exclusive` must be true. Looking at the rule "home picks new request", which sets this variable, leads to the realization that, in the case of a request for exclusive access, the home node copies the `home_sharer_list` to the `home_invalidate_list`. The protocol then clears an entry in the invalidate list once it sends out the invalidate message to that client, and clears the entry in the sharer list once the client has sent the acknowledgment to the invalidate. This means that, at the time the home node sends out an invalidate message to a client, that client must be on the sharer *and* invalidate lists. Therefore, as before, we can add the predicate

`home_sharer_list[cl]` to the guard for the rule "home sends invalidate message":

```
rule "home sends invalidate message"
    (home_current_command = req_shared & home_exclusive_granted
     | home_current_command = req_exclusive)
    & home_invalidate_list[cl]
    & channel2_4[cl] = empty
    & home_sharer_list[cl] ==>
```

Similar reasoning is used to strengthen the guards of other pairs of rules that we determine to have been falsely marked dependent by **POeM**, resulting in the final strengthened version in the appendix.

As is evident, the ability to effectively strengthen the guards of a given protocol depends on a good understanding of the workings of the protocol, and this is perhaps the reason that our efforts with the FLASH protocol have not been as productive. The description of the protocol that we worked with is quite detailed, with 32 rules, and many auxiliary variables, sometimes multiple variables representing the same combination of state variables, all of which make it hard to determine the appropriate predicates to add to increase independence.

Table 7.2 shows the results of running the protocols with and without strengthened guards.

Example	**Without Strengthening**				**With Strengthening**			
	Unreduced		**POeM**		**Unreduced**		**POeM**	
	States	**Time**	**States**	**Time**	**States**	**Time**	**States**	**Time**
German6	13270	2.68	4485	1.29	7378	1.42	2542	0.74
German8	81413	30.28	20104	8.87	42717	14.5	10827	4.56
German10	378236	260.96	69613	49.04	193790	126.6	36606	24.72
FLASH	6336	0.78	6336	1.46	2888	0.46	2146	0.64

Table 7.2. Advantages of Guard Strengthening

Rule	Priority
client requests shared access	7
client requests exclusive access	7
home picks new request	4
home sends invalidate message	4
home receives invalidate acknowledgment	4
sharer invalidates cache	4
client receives shared grant	0
client receives exclusive grant	0
home sends reply to client - shared	4
home sends reply to client - exclusive	4

Table 7.3. User-defined priorities for the German protocol

The next set of experiments run on the German protocol were aimed at testing the significance of user-defined priorities for rules, over the automatically computed priorities, which are based on the number of variable references. Table 7.3 shows the priorities we assigned to the rules of the German protocol, and Table 7.4 shows the comparison between the two methods of assigning priorities to rules. Lower weights translate into a higher priority for the rule to be picked as the seed transition.

Example	Without Strengthening				With Strengthening			
	Tr. wts		Varcount wts		Tr. wts		Varcount wts	
	States	Time	States	Time	States	Time	States	Time
German6	2521	0.66	4485	1.29	1166	0.36	2542	0.74
German8	8098	2.75	20104	8.87	2851	0.94	10827	4.56
German10	20968	10.52	69613	49.04	5890	2.57	36606	24.72

Table 7.4. User defined priorities vs. Variable reference based priorities

The user defined priorities were assigned in such a fashion as to give higher priority to rules that complete transactions, the transactions in this case being the requests for exclusive or shared access to a line. Rules that represent the

intermediate steps of a transaction were given medium priority, and rules that represent the start of a transaction were given the lowest priority.

In the case of the German protocol, user-defined priorities gave a distinct performance boost to **POeM**, resulting in up to an 80% reduction over the already reduced state space explored by **POeM** using the regular variable reference count based priorities.

7.4 Conclusions and Future Directions

7.4.1 Contributions

Formal verification techniques aim to provide an effective means of detecting errors in high-level hardware and software designs. However, one of the biggest impediments to a wider application of formal methods continues to be the state explosion problem. Thus, effective state space reduction techniques are critical to the success of formal verification.

In this dissertation, we have focused our attention on *rule based systems*, which form an important class of high-level design languages. Rule based languages are commonly the paradigm of choice for specifying high-level protocols such as cache coherence and security protocols, as well as communication based protocols such as mutual exclusion algorithms, leader election protocols, and so on. One of the most powerful class of state reduction techniques, *partial order reduction*, has so far difficult to apply to rule based systems. In this dissertation, we have addressed this shortcoming, and provided a number of approaches to effectively applying this reduction to rule based systems:

- We have developed a new partial order reduction algorithm based on symbolic simulation that is particularly applicable to rule based systems such as Murphi and TLA+. This algorithm overcomes the limitations of the traditional, syntax based check for independence among transitions, limitations that are particularly severe in the case of languages with high-level data structures such as arrays and records.

- We have developed a technique for taking advantage of symmetric, parameterized systems that lets us reuse the independence results from the analysis of a system with a small parameter value while verifying any instance of the system with a larger parameter size.

- Based on the observation that many protocol-like systems exhibit *transactional* behavior, we have developed a technique for soundly reordering transactions that enhances the efficacy of partial order reduction. By delaying the scheduling of an enabled transaction independent of the ongoing transaction, we are able to reduce the number of interleavings among transactions that the verifier explores.

- An initial independence analysis can often reveal possible predicates that, when conjoined with the guards of existing transitions, improve our algorithm's ability to deduce independence. We have developed an on-the-fly algorithm for proving the soundness of such *guard strengthening* predicates. This allows us to directly model check the system with strengthened guards, and verify the soundness of the strengthenings, as well as the correctness of the system property, in one model checking run.

- Finally, we have developed a symbolic simulation based translation algorithm from Murphi's rule based language into NuSMV's per-variable update style specification. This brings symbolic, BDD and SAT based model checking techniques to rule based designs, and opens up the possibilities of combining explicit state enumeration and symbolic methods for the verification of rule based systems.

7.4.2 Future Directions

This dissertation presents a number of techniques for combating the state space explosion problem in rule based systems. Further research is needed to examine the applicability of these techniques to other kinds of systems. There are also opportunities to improve the techniques presented here:

- We have presented a symbolic technique for computing independence that converts the independence conditions into propositional formulas which can be checked for validity using a SAT solver. Since Murphi's high level language includes features such as function calls, finitary arithmetic, and array lookups, it would be interesting to examine the comparative efficiencies of the current scheme versus directly harnessing a first-order decision procedure such as provided by tools like Stanford's CVC [40] and CVC-Lite [4].

- Our results for the carry over of independence information across instances of parameterized systems only currently apply to *bounded data* systems. We have explored the possibility of expanding the result to also encompass parametric systems that allow arrays whose element type is also parameterized, which would take us into the realm of unbounded data systems. While a direct extension of the proof is infeasible, we believe that a restricted type of unbounded data systems might be amenable to a similar carry over theorem. The primary difficulty lies in the multiple levels of indexing that arise from allowing arrays with parameterized data types, and a restriction on the nesting levels of array indices might allow us to extend the proved theorem.

- Our analysis of transactional systems relies so far on the designer/user providing the positions of transitions within a transaction. However, we have recently begun experimenting with the possibility of automatically deducing these positions from finite simulation runs of the system being verified [10]. Such an algorithm would further automate the model checking process, bringing us closer to the ultimate goal of a push-button model checking tool capable of powerful analysis.

- Strengthening the guards of transitions in order to improve the independence analysis presently requires the user to manually study the transitions and deduce the appropriate predicates. However, recent work on counter-example guided refinement techniques [12, 1] suggest that it may be possible to auto-

matically suggest candidate predicates by analyzing the satisfying assignment returned by the SAT solver in cases where a pair of transitions are found to be dependent.

REFERENCES

[1] Nina Amla and Kenneth L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *FMCAD*, pages 260–274, 2004.

[2] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[3] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification, Invited Talk. In *MEMOCODE*, pages 249–249, 2003.

[4] Clark W. Barrett and Sergey Berezin. Cvc lite: A new implementation of the cooperating validity checker category b. In *CAV*, pages 515–518, 2004.

[5] Ritwik Bhattacharya. http://www.cs.utah.edu/~ritwik/carryover.html.

[6] Ritwik Bhattacharya, Steven German, and Ganesh Gopalakrishnan. A symbolic partial order reduction algorithm for rule based transition systems. Technical Report UUCS-03-028, School of Computing, University of Utah, 2003.

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[8] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, and Marco Roveri. NuSMV 2.2 User Manual. http://nusmv.irst.itc.it/NuSMV/userman/v22/nusmv.pdf.

[9] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[10] Xiaofang Chen. personal communication.

[11] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[13] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[14] Intel Corporation. Intel microprocessor quick reference guide. http://www.intel.com/pressroom/kits/quickreffam.htm.

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[16] David Dill. The stanford murphi verifier. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, New Jersey, July 1996. Springer-Verlag. Tool demo.

[17] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[18] E. A. Emerson, E. M. Clarke, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. *ACM Trans. on Programming Languages and systems*, 8(2):244–263, 1986.

[19] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *CAV 93*, pages 438–449, 1993.

[20] Patrice Godefroid. Using partial orders to improve automatic verification methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 176–185, New Brunswick, NJ, USA, June 1990. Springer-Verlag.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[22] G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.

[23] Alan John Hu. Techniques for efficient formal verification using binary decision diagrams. Technical Report CS-TR-95-1561, Stanford University, 1995.

[24] C. Norris Ip and David L. Dill. Better verification through symmetry. In *Int'l Conference on Computer Hardware Description Language*, 1993.

[25] Shmuel Katz and Doron Peled. An efficient verification method for parallel and distributed programs. In *REX Workshop*, pages 489–507, 1988.

[26] L. Lamport. Specifying concurrent systems with TLA+. In *Calculational System Design*, 1999.

[27] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[28] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[29] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc., 2002.

[30] Vladimir Levin, Robert Palmer, Shaz Qadeer, and Sriram K. Rajamani. Sound transaction-based reduction without cycle detection. In *SPIN*, pages 106–122, 2005.

[31] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[32] Kenneth L. McMillan. SMV Model Checker. http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv/.

[33] Ralph Melton, David L. Dill, C. Norris Ip, and Ullrich Stern. Murphi Annotated Reference Manual. http://chicory.stanford.edu/dill/Murphi/Murphi3.1/doc/User.Manual.

[34] Arun Mulpur. Use Co-Simulation for the Functional verification of RTL Implementations. *Chip Design Magazine*, 2005.

[35] William T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, 1981.

[36] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design*, 8 (1):39–64, 1996. also in Computer Aided Verification, 1994.

[37] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), June 1981.

[38] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS '01*, pages 82–97, 2001.

[39] S. Qadeer, S. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 2004.

[40] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.

[41] Antti Valmari. A stubborn attack on state explosion. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 156–165, New Brunswick, NJ, USA, June 1990. Springer-Verlag.