A BDD-BASED ENVIRONMENT FOR FORMAL VERIFICATION OF HARDWARE SYSTEMS

by

Ramin Hojati

Memorandum No. UCB/ERL M96/44

12 July 1996

A BDD-BASED ENVIRONMENT FOR FORMAL VERIFICATION OF HARDWARE SYSTEMS

by

Ramin Hojati

Memorandum No. UCB/ERL M96/44

12 July 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

A BDD-Based Environment for Formal Verification of Hardware Systems

by

Ramin Hojati

Doctor of Philosophy in Computer Science
University of California, Berkeley
Professor Robert K. Brayton, Chair

Validation is the bottleneck in many designs. If the designers are to design more complicated circuits in less time with higher quality, better validation techniques are needed. Automatic formal verification is a very promising approach, which has received wide-spread interest, especially after the advent of Binary Decision Diagrams (BDDs) which have enabled the verification of some realistic examples. This thesis describes an environment for formal verification based on BDDs. Such an environment can be used as the basis of a prototype for an industrial-strength verification tool for hardware systems.

To create such a tool, several issues have to be addressed: concurrency model, fairness constraints, specification formats, efficient BDD-based algorithms for verifying an implementation versus its specification, early failure detection, generation of error traces, state minimization, and data abstraction. In this thesis, solutions for some aspects of the above issues is provided, trading expressiveness with efficiency in many cases. Many of the algorithms have been implemented in our prototype verification tool, HSIS, developed at UC Berkeley.

R. K. Brayton

Professor Robert K. Brayton
Dissertation Committee Chair

Contents

List of Lemmas and Theorems

List of Figures

List of Tables

Acknowledgment

This work would not have been possible at all without the constant help and encouragement of my advisor professor Robert K. Brayton. I am very much indebted to Dr. Robert Kurshan of AT&T Bell laboratories for introducing me to the subject of formal verification. My gratitude goes to the members of my dissertation committee professors Alberto Sangiovanni-Vincentelli and Leo Harrington.

Many other individuals contributed significantly to this work. First, I would like to thank my co-authors on several papers which made their way into this dissertation Adrian Isles, Sriram Krishnan, Paul Loewenstein, Dr. Robert Mueler-Thuns, Tom Shiple, Serdar Tasiran, and Vigyan Singhal. I would also like to thank the co-developers of the HSIS software, the co-presenters at the verification theory seminars, and the co-instructors for EE290H1993 Adnan Aziz, Felice Balarin, S. T. Cheng, Tim Kam, Sriram Krishnan, Rajeev Ranjan, Tom Shiple, Vigyan Singhal, Serdar Tasiran, Tiziana Villa, and Huey Wang. From outside Berkeley, I have benefitted very much from discussions with David Cyrluk, professor David Dill, professor Allen Emerson, Dr. Patrice Godefroid, Dr. David Long, Dr. Ken McMillan, Dr. John Rushby, professor Shmuel Safra, Dr. Hamid Savoj, Dr. Mandayam K. Srivas, Dr. Natarajan Shankar, Dr. Gary York, and professor Moshe Vardi.

I am thankful to professors Blum, Karp, Vazirani, Yelick in computer science, and professors Bade, Hald, Rieffel, Ribet, Solovay who all contributed to my education enormously. I am also indebted to Semi-conductor Research Corporation for funding this research. Last but not least, I am thankful to my friends and family who believed in me.

Chapter 1

Introduction

1.1 Motivation

In designing modern microprocessors and other complicated hardware systems, validation engineers begin their work as soon as the design process begins. The validation teams are often as large or larger than the design teams. Currently, validation is done using simulation, where two commonly used techniques are developing ad hoc behavioral simulators for specific parts of the design, and providing test vectors for conventional HDL simulators. This process is costly both in terms of man-power and CPU time. Worse, there is no guarantee that the whole design has been tested, and this problem is getting more severe with the increasing complexity of designs. For example, [Hsu93] reports that in the design of a commercial microprocessor, a more aggressive design had to be abandoned since the verification effort was expected to be too large.

Formal design verification (verification hereafter) is the process of proving properties of designs. Since verification checks all possible behaviors, it provides a guarantee that the verified property holds for the design. Although, verification is still only as good as the properties checked, generally it is much easier (and less time-consuming) to come up with a good set of properties, than to develop a good set of test vectors. Verification has the promise of reducing the simulation time, as well as increasing the level of confidence in the design.

The approach originally proposed for verifying software programs was to use a theorem-prover for first or higher-order logic to prove the correctness of the program. This approach failed mainly due to large human interaction necessary to prove correctness of programs. Two developments in the late 1970's and early 1980's made verification of hardware systems feasible. First, it was suggested in [Pnu77] that temporal logic can be used for correctness specification. Since temporal logic is more restrictive than first and higher order logics, the problem of checking an implementation versus its specification becomes easier. Second, it was suggested by [EC81] and [QS81] that finite-state systems can be used to capture implementations, which makes the problem of verifying an implementation versus its specification decidable.

Most automatic verification techniques enumerate the set of reachable states, and then check the property on the set of reachable states. Using explicit enumeration techniques, only up to a few million states could be checked using this methodology. Bryant in [Bry86] suggested a data structure, called Ordered Binary Decision Diagrams (BDDs), which could be used to capture Boolean functions efficiently. Another major breakthrough was suggested in [CBM89], where it was shown that the set of reachable states of finite state systems can be efficiently produced and manipulated in many cases using BDDs. This allowed systems with many more orders of magnitude to be verified. Since then intense research has gone into driving formal verification towards an everyday practice. This thesis studies several central issues which have to be considered in making a BDD-based formal verification tool for hardware systems.

1.2 Overview

In order for verification to be used widely, it has to fit in the current design methodologies. The central issue here is to use the same description in verification as in synthesis and simulation. To allow for multiple specification languages, and to create a formal entry language, in chapter 2, we introduce a concurrecny model called combinational/sequential (C/S), and an intermediate format called BLIF-MV supporting this concurrecny model. To remove some unwanted behavior introduced by non-determinism, expressive edge-Streett fairness constraints are added.

The C/S descriptions often involve many small relations. The problem of building BDDs from the C/S descriptions is formulated as the *early quantification* problem in chapter 3. It is shown that a formulation of this problem is NP-complete in the number of variables and relations involved, and efficient heuristic algorithms are given. These algorithms are also used to give a solution to the *partition transition relations* problem, which reduces the memory requirement for the BDD of the transition relation of the product machine.

A set of algorithms for verification using language containment and model checking is presented in chapter 4. At the heart of these, is a set of algorithms computing various approximations to the set of fair states of an ω-automaton, which is used for checking language emptiness. The *debugging problem* for language containment and fair CTL model checking is studied, and proved NP-complete in chapter 5. Then heuristic algorithms are given. The material described so far forms the basis of the formal verification tool, HSIS, developed at UC Berkeley, and released to general public through anonymous FTP.

A set of more advanced topics is studied in the next few chapters. For language containment, the concept of *fairness graphs* is introduced in chapter 6. These are used to speed up language containment when many fairness constraints are involved. *State minimization* is another way to reduce the number of reachable states. However, the complexity of this task is PSPACE-complete. Two approximations to language containment based on simulation relations extended to fairness constraints are presented in chapter 7, and their complexities are given. Since these problems are still hard, heuristics are presented.

In many cases, often due to the size of the datapath elements, there is an explosion of the state space. In chapter 8, a concurrency model, called Integer C/S (ICS), is introduced, which is an extension of C/S, and allows uninterpreted and interpreted integer functions and predicates. A set of results on *datapath abstraction* of subsets of ICS models is proved, which allows the verification to proceed on small instantiations. Since these subsets can be automatically recognized, the abstractions can be produced without any user intervention. For cases where finite instantiations do not work, a symbolic execution algorithm is presented. Since the state space of ICS models can be infinite, the algorithm may not always terminate. However, it can be used to check that no bad behavior in n steps, where n is a user-defined parameter.

The thesis is concluded in chapter 9 by looking at the correctness problems for *memory systems* which execute their instructions out of order. As an example, the SUN's SPARC-8 ([SUN90]) memory architecture is verified using the language containment paradigm.

1.3 Related Research

In this section, we look at some important related research topics not covered in this thesis. In each case, we give pointers to some introductory references.

1.3.1 BDD Research

There are four important optimization problems when dealing with BDDs: fast BDD package, early quantification and partitioned transition relations, BDD minimization in presence of don't cares, and variable ordering.

Fast BDD Package. The first BDD package was introduced in [Bry86]. In [BRB90], garbage collection was introduced as a means of speeding up BDD computations. Recently, BDD packages are being rewritten so that the number of cache misses are minimized ([KR96]). This research is related to the breadth-first search approaches which try to minimize the effect of page faults when the BDDs no longer fit in the main memory ([OYY93], [AC94]).

BDD Minimization in Presence of don't cares. Most BDD-based reachability routines explore the state space in a breadth-first search manner. Using this technique, the previous set of reachable states can be used as a don't care set to minimize the frontier set. [SHBS94] introduces and systematically studies a set of techniques for minimizing BDDs in presence of don't cares.

Variable Ordering. Variable ordering has a great impact on the speed of verification using BDDs. [FOH93] introduces good heuristic algorithms for this problem. [Mc93] proves an interesting bound on the size of BDDs for the transition relation, based on the structure of the circuit. [Rud93] gives efficient dynamic re-ordering algorithms to automatically re-order variables during various BDD operations. [PSP94] enhances dynamic variable re-ordering by grouping symmetric variables during re-ordering, and moving them as a unit.

1.3.2 Theorem Proving

There has been considerable amount of research in hardware verification using theorem-proving. Most of these efforts have concentrated on checking the correctness of pipelined designs, by comparing the pipelined implementation to its non-pipelined specification. [Cyr93] describes one such methodology. The techniques presented in chapter 9 can also be used for this task.

1.3.3 Timing Verification

Timing verification involves verifying specifications and implementations which involve timing constraints. In verifying synchronous hardware systems, which is the majority of hardware systems, such constraints are not needed in practice. [Alu91] provides a good introduction to the subject.

1.3.4 Explicit State Exploration

Explicit state exploration techniques appear to be more suitable for higher-level protocols which can be specified in interleaving semantics. In this case, symmetry techniques ([ID93], [ES95]), state bit hashing ([Hol91]), and partial order techniques ([God94]) can be used to reduce the size of the state space.

1.4 Organization of the Thesis

Chapter 2 describes the C/S concurrency model, shows how one can translate other concurrecny models into C/S, and gives the syntax for BLIF-MV. The early quantification and partition transition relation problems are described in chapter 3. A set of BDD algorithms for language containment and model checking are presented in chapter 4. Debugging methods for fair CTL and language containment are presented in chapter 5. Chapter 6 describes a method for speeding up language containment in presence of fairness constraints. Fair simulation relations, which form the basis for state minimization of ω-automata and can be used as an approximation to language containment, are studied in chapter 7. ICS models, data abstraction techniques using finite instantiations, and reachability analysis of ICS models are described in chapter 8. Chapter 9 concludes the thesis by giving an application of verification to the correctness of memory systems with out-of-order execution.

Chapter 2

Combinational/Sequential Concurrency Model and BLIF-MV

2.1 Introduction

To make formal verification possible in real designs, several problems must be solved. In this chapter, we concentrate on how verification can be integrated with logic synthesis and simulation. We have addressed this problem by creating a formal concurrency model, combinational/sequential, and an associated intermediate format, BLIF-MV. BLIF-MV can be used for simulation, synthesis, and verification, where descriptions in high-level languages are compiled into it. This compilation process is described in [Che94] for the hardware description language, Verilog.

Our design methodology for synthesis and verification is as follows. The design is first written in abstract terms, using non-determinism, in the user's favorite high-level language. The design is then translated into BLIF-MV, and verification is performed on it. Simulation is used at this point to find easy bugs, whereas formal verification may be used to catch the harder ones. If the user is satisfied with the verified parts, they can be synthesized into hardware or possibly software.

In section 2.2, we introduce C/S and give its operational and denotational semantics. In designing a concurrency model, one has to worry about expressiveness and conciseness. In section 2.3, we show how other popular concurrency models can be translated into C/S efficiently. In section 2.4, BLIF-MV, as an implementation of C/S, is introduced. In BLIF-MV, we have taken care to introduce as few constructs as possible, while not sacrificing expressiveness and the ability to preserve the structure of the user's specifications. BLIF-MV is the logical extension of BLIF (Berkeley Logic Intermediate Format, [BLIF87]) to include non-determinism and multi-valued variables.

2.2 The Combinational/Sequential (C/S) Concurrency Model

2.2.1 Syntax

In this section, the combinational/sequential (C/S) concurrency model is described. There are three primitives: variables, tables (intuitively non-deterministic gates), and latches.

Definitions A ***variable*** takes values from some finite domain. A ***table*** is a relation defined over a set of variables. The variables of a table are divided into two sets: inputs and outputs. ***Latches*** are defined over two variables ranging over the same finite domain: input (or ***next state***) and output (or ***present state***) of the latch. Note that present and next state variables may overlap. This happens when an output of a latch is an input to another. With every latch, we associate a set of initial values, which is a subset of the set of values of its variables. Every variable is the output of exactly one table or latch. Hence, every input to a table or latch is the output of some other table or latch, i.e. our systems are closed[1]. A variable can be input to many tables or latches. A ***state*** is an assignment of values to the latches of a model, where a value assigned to a latch must be in its domain. An ***initial state*** is a state where every latch takes a value from its set of initial values.

The concurrency model of C/S is exactly the same as synchronous hardware also, modulo the fact that tables may be non-deterministic and signal may be non-deterministic instead of Boolean. If the tables are deterministic and the signals are Boolean, then we can think of them as general gates. In this case, the above model is syntactically and semantically the same as synchronous hardware.

2.2.2 Operational Semantics of C/S

The operational semantics of C/S provides more insight into the meaning of C/S models. However, it is not as concise as its denotational semantics. We give the operational semantics for a subset of C/S models, where no combinational loops are allowed.

Definition A ***table graph*** is a directed graph where every node is a table. There is an arc from u to v if some output variable of table u is an input to table v. A cyclic table graph is said to contain a ***combinational loop*** (or ***cycle***).

Lemma 2.2.1 Let an acyclic table graph G be given. Then, table T corresponding to a root node either has no inputs or each one of its inputs is the output of some latch.

1. This is no restriction since it is always possible to replace an input by a table representing a non-deterministic constant.

Proof Follows from the definition of a table graph (QED).

We define the operational semantics, restricting models not to contain combinational loops. The operational semantics is defined in terms of a configuration or state graph and its transition relation. Every node in the configuration graph corresponds to a set of values assigned to the latches of the model. Since the domains of all variables are finite, the configuration graph is finite. The initial states of the configuration graph are those where an initial value is assigned to every latch. Let l denote a node (i.e. a state) in this graph. The following algorithm defines the transitions from l to other nodes.

1. *Fix a topological sort of the table graph.*
2. *Assign to the outputs of the latches the values given by l.*
3. *Assign values to the outputs of each table consistent with its inputs, processing the tables in the order given by the topological sort. More precisely, let a table T represent the relation $R(i, o)$, where i represents the inputs to the relation and o its outputs. Let the inputs have the assignment $v(i)$. Choose $v(o)$ such that $(v(i), v(o)) \in R(i, o)$.*

Note that in step 3 of the above algorithm (referred to as ***value propagation***), due to non-determinism, there may be more than one assignment to the output of a table given an assignment to its inputs. It is also possible that a table is not complete, i.e. there are inputs for which there are no outputs. Then, the set of values assigned to an output of a table may be empty. The empty values propagate, i.e. if one of the inputs to a table is empty, then the output is empty as well. Lemma 2.2.2 shows that step 3 of the algorithm is well-defined.

Lemma 2.2.2 When a table is processed in step 3 of the above algorithm, each one of its inputs has been assigned a value.

Proof We name the tables $T_1, \dots, T_n$ according to their positions in the chosen topological sort, where n is the number of tables. We proceed by induction on $1 \le i \le n$. T_1 either has no inputs or all of its inputs are outputs of latches (by lemma 2.2.1). The lemma holds in this case, since step 1 assigns values to the latches. Now, let table T_i and an input to it v be given. v is either an output of a latch or a table. If it is the output of a latch it has been assigned a value by step 1. If it is the output of a table, by the fact that the model is acyclic, and we have processed the table in a topological sort, it has been assigned a value (QED).

We prove below that the algorithm is well-defined, i.e. no matter what topological sort is chosen, the same transition relation is obtained.

Lemma 2.2.3 The above procedure assigns values to all next state variables (inputs of the latches).

Proof Every input to a latch is either an output of a table or a latch. In either case, steps 1 or 3 assigns it a value (QED).

Lemma 2.2.4 Let l be an assignment of values to all latch outputs. Let l' be an assignment to all latch inputs obtained from the above procedure given topological sort O_1. Then, given topological sort O_2, l' can be achieved from l by the above algorithm.

Proof This follows from noting that the order of processing of two tables in step 3 can be swapped without any change in the final result as long as neither table is reachable from the other one in the table graph. Since O_1 can be obtained from O_2 by doing such swaps, the result follows (QED).

We assume the edges in the configuration graph are labeled by the values of a user-define subset of the variables called ***output variables***. Therefore, the configuration graph is a safety automaton (defined below) over the alphabet of the Cartesian product of the domains of the output variables. This in turn defines a language, which is the language of a C/S model according to the operational model.

Definitions A ***safety automaton*** is a tuple (Σ, Q, T, I), where Σ is a finite alphabet, Q is a finite set of states, T is a transition relation on $Q \times \Sigma \times Q$, and I is a set of initial states. A ***run*** r of an ω-string x in A is an infinite sequence of states, such that the first state $r_0 \in I$, and for all i, $(r_i, x_i, r_{i+1}) \in T$. The set of states occurring infinitely often in a run r is called the ***infinitary set*** of r, and is denoted by $inf(r)$. Since the set of states Q is finite, $inf(r)$ is always a non-empty finite set. The set of all ω-strings which have a run in A is called the ***behavior*** or ***language*** of A, and is denoted by $L(A)$. A ***finite state machine (FSM)*** is a safety automaton where $\Sigma = \Sigma_I \times \Sigma_O$, Σ_I is the input alphabet, and Σ_O is the output alphabet.

To handle combinational loops, one may proceed by breaking the loops arbitrarily, allowing the newly created variables take any values in their domain, and following the above procedure. Only those values should be accepted which are stable, i.e. all instances of the broken-up variables take on the same value. The denotational semantics is simple and concise, and handles combinational loops without any special treatment.[1]

2.2.3 Denotational Semantics of C/S

The denotational semantics is given by describing what C/S models denote and what the syntactic and semantic algebras are. A C/S model denotes an ω-language, defined over the output variables. The language of a C/S model can be represented by two relations: transition and initial state relations. The ***transition relation*** is defined over all variables and is denoted by $T(x, i, y)$, where x denotes the present state variables, y the next state variables, and i the rest of the variables. To define the denotational semantics, we assume the set of present and next state variables are distinct:. Therefore, if in a C/S model a present state variable is also a next state variable, a buffer is inserted to create two distinct variables. The initial state relation is defined over present state variables and is denoted by $I(x)$.

To define the denotational semantics, we need to define what the syntactic and semantic objects are, and how semantic objects are composed ([GT93]). There are two syntactic objects in the syntactic domain: tables and initial values. There is only one semantic object: relation. Composition of semantic objects is the join operation of relations, which corresponds to taking the Boolean AND of the characteristic functions of the relations. The correspondence between the syntactic and semantic domains is as follows. Each table (a syntactic object) represents a relation (a semantic object), with the composition operation being the join operation of relations. The initial values for each latch (a syntactic object) define a relation over the present state of that variable (a semantic object). The composition between initial state relations is again the join operation of relations. Therefore, a C/S model denotes two relations: transition and initial state relations. This defines a safety automaton over the Cartesian product of the output variables, which in turn denotes an ω-language. It is not hard to show that the operational and Denotational semantics define the same language, by showing that for purely combinational circuits the two agree.

2.2.4 Representation Using BDDs

The denotational semantics can be easily implemented using BDDs. Since each table describes a relation over a finite domain, it can be represented by a BDD. In our verification tool, we represent each table by a multi-valued decision diagram or MDD ([Kam92]), since MDDs provide a convenient multi-valued interface to BDDs. The product of all the BDDs is a transition relation

1. Our semantics for combinational loops discards those behaviors where some variable is oscillating, even though this variable may have no impact on the outputs. [SSBS96] provides algorithms for discovering such cases.

$T(x, i, y)$ which describes the (stable) behavior of the whole system. This transition relation is used for simulation and debugging, where information about output variables is needed. For verification, we existentially quantify out all non-state variables, and get a transition relation denoted by $T(x, y)$. In chapter 3, we introduce efficient algorithm for this problem.

2.3 Combinational/Sequential versus Other Concurrency Models

We will be comparing the expressive powers of different modeling schemes. To do so, we need a notion of equivalence between languages. Since equality turns out to be too restrictive, we use the following definition.

Definition Let language L_1 defined over Σ be given. A language L_2 defined over $\Sigma \times \Sigma'$ is ***projection-equivalent*** to L_1 if the projection of L_2 onto Σ is equal to L_1. Note that this notion of equivalence is one-way, i.e. L_2 may be equivalent to L_1, but the reverse may not be true.

Definition A concurrency model C_1 has as much ***expressive power*** as another concurrecny model C_2 if given a system in C_1 with language L_1 there is another system in C_2 with language L_2, where L_2 is projection-equivalent to L_1.

In this section, we first define product of safety automata, and then introduce several popular modeling schemes, and compare their relative expressive powers.

2.3.1 Product of Safety Automata

Definition Let $A = (Q_A, \Sigma, I_A, T_A)$ and $B = (Q_B, \Sigma, I_B, T_B)$ be given. Then, $P = A \times B = \left(Q_A \times Q_B, \Sigma, I_A \times I_B, T\right)$, where $T((q_A, q_B), a, (q'_A, q'_B))$ iff $T(q_A, a, q'_A)$ and $T(q_B, a, q'_B)$.

Lemma 2.3.1 Let $P = A \times B$. Then, $L(P) = L(A) \cap L(B)$.

Proof Let $x \in L(P)$. Then, there is a run $r = \begin{pmatrix} r_{0_A} & & r_{n_A} & \\ & , \dots, & & , \dots \\ r_{0_B} & & r_{n_B} & \end{pmatrix}$ in P for x. Now consider the sequence of states $r_{0_A}, \dots, r_{n_A}, \dots$ in A. By definition, this is a run in A. Hence, $x \in L(A)$. Similarly, $x \in L(B)$. Now, let $x \in L(A)$ and $x \in L(B)$. We need to show that $x \in L(P)$. Consider the runs $r_{0_A}, \dots, r_{n_A}, \dots$ and $r_{0_B}, \dots, r_{n_B}, \dots$ for x in A and B, respectively. Consider the sequence of states $\begin{pmatrix} r_{0_A} & & r_{n_A} & \\ & ,\ldots, & & ,\ldots \\ r_{0_B} & & r_{n_B} & \end{pmatrix}$ in P. By checking the definition, this is a run for x in P (QED).

2.3.2 Selection/Resolution Model

In this section, we define the selection/resolution model, which is a simple synchronous modeling scheme. We then prove some simple properties of the S/R model. We finally show that S/R and C/S have the same expressive power, although C/S may be more concise. With the exception of theorem 2.2 proving this fact, all results can be found in [Kur90].

Definition A ***process*** is a 6-tuple $P = (Q, \Sigma_I, \Sigma_O, I, T, O)$, where Q, I, and T are as before. Σ_I denotes the input alphabet (all of our alphabets are finite unless otherwise stated), and Σ_O denotes the output alphabet. The output relation O is defined over $Q \times 2^{\Sigma_O}$; intuitively a state can have many outputs. The transition relation T is defined over $A \times \Sigma_I \times \Sigma_O \times 2^{Q}$; intuitively given a state, an input symbol, and an output symbol, there may be more than one transition. The ***language of a process*** is generated as follows.

1. *Start at an initial state.*
2. *At the current state,*

    Produce an output non-deterministically.

    Based on the output and input, choose a next state.

One can think of a process as a non-deterministic output non-deterministic transition Moore machine. The only difference with a Moore machine is that a process can look at its output from its current state to select its next state. More formally, $x = \begin{pmatrix} x_{0_I} & & x_{n_I} & \\ & ,\ldots, & & ,\ldots \\ x_{0_O} & & x_{n_O} & \end{pmatrix} \in L(P)$ if there exists a sequence of states in A, $r = (r_0, \ldots, r_n, \ldots)$ such that r_0 is an initial state, and $\left(r_i, x_{i_I}, x_{i_O}, r_{i+1}\right) \in T$ for all i. Note that the language of a process is defined just in terms of the edge labels. The outputs are introduced only for book-keeping purposes, and are not used in the formal definitions. Hence, in effect a process is just a safety automaton defined over the alphabet $\Sigma_I \times \Sigma_O$. In figure 2.1, an example process is shown, whose inputs are from the set a, b and whose outputs are $0, 1$.

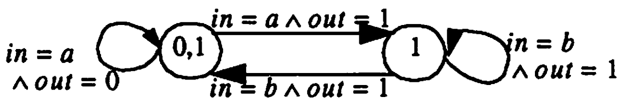


Figure 2.1: Example of a process in S/R

Sometimes the language of P is defined over some larger alphabet $\Sigma_I \times \Sigma_O \times \Sigma$. In this case, the edge symbols i/o are changed to $i/o/-$ accordingly to reflect this, i.e. for each symbol we assume there is an additional element which can take any value in Σ. In many cases, as opposed to giving the symbols of the alphabet, a Boolean combination of variables ranging over Σ_I and Σ_O is used. In this case, enlarging an alphabet is the same as enlarging the set of variables of the alphabet. For example, the statement $(x = 0) \wedge (y = 1)$ specifies only one symbol if the alphabet is defined over two binary variables x and y, but it refers to two symbols if the alphabet contains three binary variable x, y and z.

Definition Let n interacting processes $P_1, \ldots, P_n$ be given such that the outputs of all processes are distinct, and each input to a process is the output of some other process. In other words, the system is closed, and each output is driven by exactly one process. Since every input is an output, the input and output variables are collectively called the ***selection variables***. The operational semantics of the ***selection/resolution model*** is generated as follows:

1. All processes start at one of their initial states.

2. At every point,

Each process non-deterministically chooses one of the outputs possible from its state (selection).

Based on the global selection, each process chooses a next state (resolution).

The language of $P_1, \ldots, P_n$ according to the selection/resolution model is the set of values the selection variables take in the above process.

Alternatively, a string $x = (x_1, \ldots, x_n, \ldots)$ defined over the product alphabet of the selection variables is in the language of the set of processes $P_1, \ldots, P_n$ according to selection/resolution model if there exists a sequence of global states $s_o, \ldots, s_n, \ldots$ such that,

1. s_o is a global initial state.

2. For all i, the transition (s_i, x_i, s_{i+1}) is possible, i.e. for each process P_j, $(s_i[j], x_i, s_{i+1}[j])$

is a transition in P_j, where $s_i[j]$ denotes the state of P_j in s_i.

This views S/R as an ***edge-synchronous model.*** At every point in time, each process is at some local state. The edge labels are interpreted as predicates or restrictions on the alphabet symbols. To reach the next global state, a set of edges out of the current global state (one edge per process) is chosen if their assignment to selection variables is consistent. This defines the transition relation of a safety automaton, whose language is the same as the operational semantics of S/R.

Definition Let $P_1, \ldots, P_n$ according to the selection/resolution model be given. Then, the ***product process*** is defined to be the product of the corresponding safety automata, where the alphabet is the product of the alphabets of the selection variables.

Lemma 2.3.2 Let $P_1, \ldots, P_n$ according to the selection/resolution model be given. Then, the language of the processes according to the selection/resolution model is the same as the language of the product process.

Proof The proof follows directly from the edge-synchronous view of S/R and the definition of the product automaton (QED).

Theorem 2.1 Let $P_1, \ldots, P_n$ according to the selection/resolution model be given. Let L_1 denote the language of the processes according to the selection/resolution model, L_2 the language of the product process, L_3 and the intersection of the languages of the P_i's. Then, we have $L_1 = L_2 = L_3$.

Proof Follows from the definition of the product process, and lemmas 2.3.1 and 2.3.2 (QED).

Theorem 2.2 The selection/resolution and combinational/sequential models have the same expressive power.

Proof Let an S/R model with n processes $P_1, \ldots, P_n$ be given. To construct an equivalent C/S model, proceed as follows.

1. For each process P_i, create a latch taking values from the set of states of P_i.
2. For each process P_i, have a table which encodes the outputs of P_i.
3. For each process P_i, create a table which encodes the transition of P_i. Inputs to each such table may be the outputs of all tables created in step 2.

The equivalence follows from the operational semantics of S/R and C/S.

Conversely, let a C/S model be given. To get a corresponding S/R model, do the following:

1. Define the ***transitive fanin of a variable*** x as all the relations which can be reached by tracing

backward from the relation whose output is x. The leaves of this transitive fanin are present state variables. For each latch, create a process whose transitions are given by the intersection of all the relations in the transitive fanin of the next state variable of the latch, after the intermediate variables have been quantified out. Have each process output its state (hence each process is Moore). Call these variables state-output variables.

2. For each output variable o, create a one state process whose output is o, and its edge labels are defined by taking the intersection of all relations in the transitive fanin of o. Note that the edge labels may involve state-output variables.

To get an equivalent language, project out the state-output variables. The equivalence follows from the edge-synchronous view of S/R (QED).

Note that the translation from S/R into C/S can be done in small polynomial time and space. However, the reverse might incur an exponential blow-up, since multiplying tables can blow up. In languages (or systems) which support the S/R model, the use of some functions corresponding to a high-level language like C is provided. However, such functions are deterministic. It is possible that one can compactly describe a non-deterministic relation using a set of non-deterministic relations, where the translation into a language supporting S/R is still exponential.

2.3.3 Asynchronous Shared Memory (ASM) Model

In this section, we describe the Asynchronous Shared Memory (ASM) model which forms the basis of some verification tools, such as Murphi ([DDHY92]) and Spin ([Hol91]). A model in ASM consists of a set of n state variables $q_1, \ldots, q_n$, a set of m memory locations $m_1, \ldots, m_m$, an output variable o, and a set of p guarded commands (transitions) $e_1, \ldots, e_p$. The state variables and memory locations are collectively called ***shared variables***. A ***guard*** is a predicate on the shared variables, whereas a ***command*** is a new assignment to the shared variables and the output variable. The output associated with a guarded command e is sometimes called its action. The operational semantics of ASM can be summarized as follows. Initially, all processes are at some initial state, with all memory locations and state variables taking one of their initial values. At every point, a transition whose guard is active (evaluates to 1) is chosen, and its command is executed. The language of the system is defined over the values of the shared variables.

More precisely, let $\Sigma_1^q, \ldots, \Sigma_n^q$ denote the domains of the state variable, $\Sigma_1^m, \ldots, \Sigma_m^m$ the domains of the memory locations, and Σ^o the domain of the output variable. The ***language of this system***

according to ASM is defined over $\prod_i \Sigma_i^q \times \prod_j \Sigma_j^m \times \Sigma^o$ and is given as follows. A string $x = (x_1, \dots, x_i, \dots)$ belongs to the language of the above system if the shared variables in x_0 take one of their initial values, and for each i, there exists some guarded command e_j such that its guard is true in x_i, its command is true in and x_{i+1}, and the value assigned to o in x_{i+1} is the output of e_j.

The ***product automaton*** of the system according to ASM is a safety automaton (Σ, Q, T, I), where $Q = \prod_i \Sigma_i^q \times \prod_j \Sigma_j^m$, $\Sigma = Q \times \Sigma^o$, and $T \subseteq Q \times \Sigma \times Q$. $(q, (q, \alpha), q') \in T$ if there exists a guarded command e whose guard and command are true in q and q' respectively, and whose action is α. $q \in I$ if all shared variables take one of their initial values.

Lemma 2.3.3 The language of the system according to ASM is equal to the language of its product automaton.

Proof Follows from the definitions (QED).

Theorem 2.3 The ASM and selection/resolution models have the same expressive power.

Proof Given an S/R model, create the product process, which can be viewed as a system in ASM with one state variable, no memory, and the guarded commands being the edges. The complexity of this translation is exponential. Conversely, let a system in ASM be given. We create a S/R model with $m + n + 2$ processes as follows.

1. Create a one-state "scheduler" process, which non-deterministically outputs j if the guard of e_j is true. Also create a one-state "output" process, which outputs the action of the guarded command chosen by the scheduler.
2. For each shared variable v_i with domain Σ_i, create a process with state space Σ_i. The edge (α, β) is labeled by a predicate restricting the output of the scheduler being j, if the command of e_j changes the value of v_i from α to β.

The complexity of this translation is linear. To get an equivalent language, the state variables of the scheduler and output processes are projected out. The equivalence follows from the edge-synchronous view of S/R and the operational semantics of ASM (QED).

Remark Some interleaving models, such as the one defined in [God94], distinguish between state variables and memory locations, and associate outputs with transitions. The partial order methods of [God94] rely on such information.

2.3.4 Node Synchronous

In this subsection, we describe node-synchronous models, which sometimes come up in verification using theorem-provers. The ***node-synchronous model*** is similar to the edge-synchronous model, except that the predicates are on the nodes as opposed to edges (there are no edge labels), where a predicate is an assignment of values to variables. At every point the system is at some global state. The system makes a transition to the next global state if

1. there is a global transition to this next state from the current state, and
2. all predicates in the next state are consistent, i.e. no variable is assigned two different values.

The language of a node-synchronous model is defined in terms of the global assignment to the variables given by the above process.

Let (u, l_u) denote a node u with label l_u in the node-synchronous model. To translate an automaton N in the node-synchronous model to an automaton M in the edge-synchronous model, label each edge in N by the label of its starting vertex. Hence, an (unlabeled) edge $((u, l_u), (v, l_v))$ in N becomes the labeled edge (u, l_u, v) in M. This translation has linear complexity. Conversely, to translate an automaton M in the edge-synchronous model to an automaton N in the node-synchronous model, for each labeled edge (u, α, v) in M, create a node (u, α), and edges $((u, \alpha), (v, \beta))$ for all pairs $(v, \beta) \in N$ (created from edges $(v, \beta, v') \in M$ for some v'. This translation has quadratic complexity, i.e. $O(|M|^2)$.

2.3.5 Kripke Structures

In this subsection, we describe Kripke structures which are used in CTL model checking (see section 4.4.1). A ***Kripke structure*** is a deterministic-output, non-deterministic-transition process. Therefore, Kripke structures are a subclass of S/R processes, and hence inherit the concurrency models of S/R. To get a Kripke structure K from a S/R process P, if a state s has l non-deterministic outputs $o_1, \ldots, o_l$, create l copies of s in K, $s_1, \ldots, s_l$, where the output of each s_i is o_i. If there is an edge (s, o_i, t) in P, add to K all edges (s_i, o_i, t_j), for each $t_j \in K$ associated with t. This gives a translation procedure from Kripke structures into S/R, with complexity $O(n|P|)$, where n is the maximum number of non-deterministic outputs from any state in P. Since Kripke structures are a subset of S/R models, we conclude that Kripke structures and S/R are polynomially transformable into each other. Since S/R can be polynomially translated into C/S, we conclude that Kripke structures can be polynomially translated into C/S.

2.3.6 Summary of Concurrency Models

We studied several concurrency models which form the basis of most verification tools and paradigms. The selection/resolution model is a synchronous model, where a set of non-deterministic Moore machines, described by giving their transition structures explicitly, move in lock-step. One can view S/R as an edge-synchronous model. We showed that S/R can be polynomially translated into C/S, whereas the reverse translation may be exponential. We next introduced the asynchronous shared memory model, where a set of transition are fired one at a time. ASM can be translated into C/S polynomially, whereas the reverse translation is exponential. Node-synchronous models are similar to the edge-synchronous ones, and there is a polynomial translation among them. Kripke structures are a subset of S/R models, where the outputs are restricted to be deterministic. S/R models can be polynomially translated into Kripke structures.

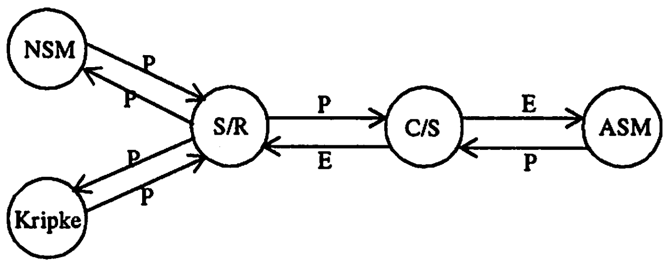


Figure 2.2: Translation complexity between various models

2.4 BLIF-MV

An intermediate format must address the following issues.

1. *Well-defined and simple semantics.* The semantics of the intermediate format should be well-defined, so that user's specifications are translated unambiguously. BLIF-MV achieves this by having C/S as its concurrecny model.

2. *User-friendly.* The interaction with the user must feed back variable names defined by the user. Many high-level languages allow for symbolic or multi-valued variables. BLIF-MV accommodates this by allowing multi-valued (MV) variables. Also, it is useful to display error traces, returned by the verifier, in terms of the user's original source code. For this, the application-specific attribute mechanism of BLIF-MV can be used.

3. *Expressive.* The generated code for the intermediate format should remain relatively small, so that parsing time is reasonable. BLIF-MV uses relations to describe (possibly non-deterministic) behavior. Various constructs are allowed to make the specification of the relations more compact. Also facilities for easily dealing with integers are introduced, since functions defined on integers, such as addition, cannot be described compactly using multi-valued relations.

4. *Structure preserving.* It is important that the original structure of the user's specification is preserved as much as possible for two reasons. First, interfacing with the user becomes easier. Second, some algorithms may be able to take advantage of the initial structure. BLIF-MV preserves the structural aspects using hierarchical specification, multi-level logic, intermediate variables, and pre-defined functions.

A first version of BLIF-MV was presented in [BMV91]. The version described here is the result of experience with HSIS, and has several additions and changes to make BLIF-MV suitable for compilation from high-level languages.

2.4.1 Basic Syntax

In this section we describe the basic syntax of BLIF-MV. The primitives of the language are: multi-valued variable declarations, tables for describing relations, multi-valued latches, models for describing a process, and subcircuits for instantiating models.

2.4.1.1 Variables

A multi-valued variable is declared as follows.

```
.mv <variable-list> <range> [<value-list>]
```

where < > implies a necessary argument, and [ ] an optional one. A variable-list is a set of comma-separated variables. A value-list is a set of space-separated names. The default on value list is 0,..., range-1. Examples are,

```
.mv var1, var2 3 apple orange banana
.mv four-bit-integer 16
```

2.4.1.2 Tables

Tables are used in BLIF-MV for describing relations. In this section, we first give the basic method of defining a table, with more advanced features introduced next.

Table Definition. A table describes a relation between a set of variables. A table has a set of inputs and a set of outputs. A table is defined as follows.

```
.names <in-1> ... <in-n> => <out-1> .... <out-m>
<row-1>
```

```
.
<row-p>
```

If there is only one output, then the symbol "=>" can be omitted. A table may have no inputs, but must have at least one output. The rows of the table describe a relation between the variables. Each row has $m + n$ components. Each entry describes a set of values, which is a subset of the values of the corresponding variable. Each row denotes the (cartesian) product of these entries. An entry can be one the following:

1. "-", which denotes the universal set, i.e. the set of all values for a variable.
2. A single value, for example 3.
3. A list of values, for example (1, 3, 5).
4. A range of values, for example 5-9.

Each entry can be complemented using "!".

As an example, assume variables x and y can each take 100 values, from 0-99. Consider the following table.

```
.names x y
(10,21) !3
5  -
3-8 5-15
```

This first entry describes a relation between 10 and 21 and any number between 0 and 99 except for 3. The second entry says that 5 is in relation with any number. The third entry says that numbers between 3 and 8 are in relation to numbers between 5 to 15. In this case, x is the input and y is the output.

Symbolic Entries. One can also use ***symbolic entries*** in table specifications. Assume under a column corresponding to variable x, we have the variable y (the syntax is =y). Then, the relation corresponding to that entry is multiplied by the relation $y = x$. For example, consider a multi-valued multiplexer, which has the following semantics: if $x = 0$, then $z = y$, otherwise $z = w$, where x, is a binary variable controlling the multiplexer, and y, z, and w are multi-valued variables of the same type. The table for this multiplexer is described as follows.

```
.names x y w z
0  -  -  =y
1  -  -  =w
```

The only restriction with symbolic entries is that the two variables which are made equal must have the same domain, i.e. be of the same type.

Default Values. With every table, a default value for the outputs can be given, which is assigned to all the minterms in the input domain which were not assigned in the table. An example is as follows.

```
        .names x1 x2 => y2 y2
        .def 0 0
        1 - 1 1
```

Now, the two unspecified minterms, i.e. 00 and 01, are assigned 00.

2.4.1.3 Latches

Latch Definition. A latch is declared as follows.

```
        .latch <latch-input> <latch-output>
```

where latch-input (next state) and latch-output (present state) are variables of the same type. An example is as follows.

```
        .latch ns ps
```

Latch Initialization. A latch is initialized (in general with a set of values) as follows.

```
        .r <in-1> ... <in-n> <p.s.>
        <row-1>
        .
        <row-m>
```

where p.s. is the present state variable of the latch we want to initialize, and in-1,...,in-n are any set of variables. The syntax for describing the entries is the same as the syntax for describing tables. An example is,

```
        .r initial-state-input state
        - =initial-state-input
```

This example says that the variable state is equal to initial-state-input. Hence, it is possible to write models where the initial state of the latches are parametrized. A latch may have more than one initial state. Below, we show three ways to specify that the binary latch "ps" has two initial states, 0 and 1.

```
.r ps                .r ps                .r ps
0                    (0,1)                -
1          or                  or
```

2.4.1.4 Hierarchical Development

We describe how hierarchies are defined, referenced, and what restrictions are placed on them.

Module Definition. In BLIF-MV, hierarchy is defined as a syntactic notion. This allows one to take a portion of a system and put a box around it. This portion can then be re-used. The unit of hierarchical development is called a ***model*** or ***module***, and is declared as follows.

```
.model <model-name>
.inputs <variable-1> ... <variable-n>
.outputs <variable-1> ... <variable-m>
<command-1>
.
<command-p>
.end
```

A ***command*** is any one of the following: variable declaration, table definition, module instantiation (or reference), latch definition, or initial state declarations. In BLIF-MV, we distinguish between four sets of variables: inputs (in), outputs (out), present state (ps), and next state (ns). Table 2.1 shows the restrictions placed on variables inside a module's definition.

Table 2.1: Variable equalities inside models

	in	out	p.s.	n.s.
in	▨	no	no	yes
out	▨	▨	yes	yes
p.s.	▨	▨	▨	no

The justification for these rules are

1. $in \neq out$, which means an input and an output variable cannot be the same inside a module. However, as we will see, one can connect the inputs and outputs of a module instantiation.

2. $in \neq ps$. This rule requires the output of a latch defined in a model not to be an input to it.

3. $in = ns$. This is allowed because we may want to input the next state of a latch.

4. $out = ps$. We may want to directly communicate the present state value of a latch defined in one of the subcircuits with another one, without having to introduce another variable "out" with the relation $out = ps$[1].

5. $out = ns$. Once the next state value is calculated, we may want to communicate that value with another block.

1. This is also an efficiency issue in verification, since computational efficiency is related to the number of variables in the system.

6. $ps \neq ns$. In order to build a transition relation for the FSM corresponding to a model, the present and next state variables must be distinct.

Module References (Instantiations). A model is instantiated as follows.

```
        .subckt <model-name> <instance-name> <formal-actual-list>
```

The formal-actual-list defines the correspondence between formal and actual variables. This list is a sequence of assignments, each separated by a space. The declaration is as follows.

```
        formal-1=actual-1 ... formal-n=actual-n
```

The actual variables are the ones declared in the current model, whereas the formal variables are declared in the referenced model.

A hierarchy in BLIF-MV is always a tree, which implies that a subcircuit cannot be referenced from any other subcircuits except for its parent. The first model encountered in a BLIF-MV file is the root of the hierarchy. Let N be the parent module of M. The ***actual variable*** of a variable x of M is x if either M is the root process or x is not an input or output variable. Otherwise, the actual variable of x is the actual variable of the variable corresponding to x in the formal-actual list of N's instantiation. Table 2.1 shows the restrictions placed on actual variables in a hierarchy.

Table 2.2: Actual variable equalities in hierarchies

	in	out	p.s.	n.s.
in	▨	yes	yes	yes
out	▨	▨	yes	yes
p.s.	▨	▨	▨	no

We give justification for two entries in table 2.1 which are different than the ones in table 2.1. The justification for the rest of the entries is as before. Let $ac(x)$ denote the actual variable of x.

1. $ac(in) = ac(out)$, which means an input and an output variable can correspond to the same actual variable. An example is where the user connects an output of a model to one of its inputs.

2. $ac(in) = ac(ps)$. The only situation where this is allowed is when a ps variable is an output of a model's instantiation M which is in turn an input to M. Another situation where this come up, but is illegal in BLIF-MV is when $ac(ps)$ is driven by more than one source.

2.4.1.5 Comment Lines

A comment is anything after "# " to the end of the line.

2.4.2 Additional Features

In this section, additional features of BLIF-MV are described.

2.4.2.1 Integers

There are some arithmetic operations which are very expensive to describe in terms of symbolic variables. Assume we want to specify the function $c = a + b$, where a and b are 16-bit integers. The symbolic table will be of size $2^{16} \times 2^{16} = 2^{32}$! However, one can easily describe a 16-bit adder using the binary-valued components of the symbolic variables.

To handle such situations, the "bundle" construct is used. A ***bundle*** is a set of binary variables associated with an integer variable. A bundle is declared as follows.

```
.bundle variable-name bit-n ... bit-1
```

where bit-n is the most significant bit. As an example, to describe the relation $c = a + b$, we use the following reference.

```
.subckt adder16 myadder16 x=a y=b z=c
```

At some later point, we define adder16 as follows.

```
model adder16 x y z
.inputs x y
.outputs z
.bundle x x15 x14 ... x0
.bundle y y15 y14 ... y0
.bundle z z16 z15 ... z0
/* a bit-wise description of a 16-bit adder */
.end
```

2.4.2.2 General Attributes

There is some information which is specific to certain tools. To allow for such information, called ***attributes***, BLIF-MV uses the following general-purpose attribute construct.

```
%attribute-name[information]%
```

One can associate an arbitrary number of attributes with BLIF-MV's basic objects. Specifically, attributes can be associated with tables, variables, latches, subcircuits, and models. The attributes associated with an object must be specified in pre-defined positions. For tables, attributes must be given right after ".names" keyword; for variables, right after ".mv" keyword;

for latches, right after ".latch" keyword; for latch initialization, right after ".r" keyword; for model references right after ".subckt" keyword; for models, right after ".model" keyword. Examples of attributes are temporary variables recognized by some tools, and source-code debugging information. For instance, the line number where a variable is declared may be defined as follows.

```
        .mv %line=10% templ  8
```

2.4.2.3 *Temporary Subcircuits*

Sometimes during translation from high-level languages, temporary subcircuits are created. For example, to translate an if-statement, a multiplexer module may be created. These extra subcircuits are not part of the user's hierarchy. The ".macro" command can be used to identify such subcircuits. This construct is different than the macro construct used in programming languages to denote in-line expansion of code[1]. A temporary subckt is declared exactly the same way as a ".subckt", except the word ".subckt" is replaced by ".macro". An example is as follows.

```
        .macro mymux mux2 control=a in1=b in2=c out=z
```

This construct is useful in specifying a technology mapped circuit, by using macros for standard cells.

2.4.2.4 *Pre-Defined Functions*

Definition To improve the efficiency of parsing and to better preserve the structure of user's specifications, some tools may define a set of pre-defined models. A pre-defined model is used like any other model, and can be referenced using both ".subckt" and ".macro" constructs. The following is an example.

```
        .subckt HSIS_ADDER16 myadder16 x=a y=b z=c
```

which references a 16-bit adder defined and understood by HSIS. If an example is to be distributed to tools which do not define these pre-defined models, a definition of the model in BLIF-MV must be included. For example, in the case above `HSIS_ADDER16` would need to be defined using the bundle construct and a bit description of the adder. Hence, there must be a BLIF-MV model associated with every pre-defined relations.

2.4.2.5 *Miscellaneous*

The ".include" command is used to include files. An example is as follows.

```
        .include utilities.mv
```

1. The difference between .macro and in-line expansion is that for .macro, we have the choice of building the transition relation once and then using variable substitution to get new instances.

2.4.3 Possible Future Changes and Extensions

In this subsection, we describe some further extensions being considered.

1. *Typed variables.* Variable types can be described using ".type" command using the following format.

```
        .type type-name range [value-list]
```

This would be used in a ".mv" construct to replace the range and value-list.

2. *Bit ranges.* To describe the range of integers, the number of bits can be given using the syntax, "*Bn*", where *n* is the number of bits. The following example defines the type large-integer to be a 64-bit integer.

```
        .type large-integer B64
```

3. *Behavioral constructs.* We are thinking of introducing behavioral constructs such as arrays and iterators, and memory read/write operators. The need for these constructs arises from abstraction techniques which require more information about the circuit structure (see chapter 8).

Acknowledgment

The material presented in sections 2.2 and 2.3 are based mainly on the author's lecture notes given for the course EE290H, UC Berkeley, fall of 1993. The material presented in section 2.4 is based mainly on the unpublished manuscript [BMV93].

Chapter 3

Early Quantification and Partitioned Transition Relations

3.1 Introduction

Design Verification is accomplished by specifying a system at a suitable level of abstraction using a set of interacting FSMs, and then proving a set of properties that should hold for the system. Binary Decision Diagrams (BDDs, [Bry86]) have been used successfully in formal verification to represent transition relations and manipulate sets of states. The first step of many verification algorithms is computing the set of reachable states of the product machine. Indeed, a significant subset of properties, the so-called safety properties, can be verified by checking that the set of reachable states is contained in the set of "good" states. To compute the set of reachable states, one first builds the BDD for the transition relation of the ***product machine*** by conjuncting (or multiplying) the BDDs corresponding to each individual FSM. However, to compute the set of reachable states of the product machine, only the state variables of the product machine are needed; the set of non-state variables, i.e. the input and output variables, can be eliminated using existential quantification. The resulting graph is referred hereto as the ***product graph***.

More precisely, let $T_j(x_j, i_j, y_j)$ be the transition relation of the j-th machine, where x_j, y_j and i_j represent its present state, next state, and non-state variables respectively. The transition relation of the product graph is computed by $T(x, y) = \exists i (T_1(x_1, i_1, y_1) \wedge \ldots \wedge T_n(x_n, i_n, y_n))$, where x, y, and i represent the set of all present state, next state, and non-state variables. This computation can be sped up by making two observations.

1. Existentially quantifying a variable from a relation usually results in a smaller size BDD (even though existential quantification can be exponential in the BDD size [McM94]).
2. $\exists v (R_1(u, v) \wedge R_2(u)) = (\exists v R_1(u, v)) \wedge R_2(u)$, where R_1 depend on u and v, and R_2 depends only on u. This operation of moving a quantifier inside a product is called the ***early quantification principle***. In general, quantification and conjunction do not commute.

The ***early quantification problem (EQ)*** is to find a schedule for multiplication of relations and quantification of variables so that the maximum size of any BDD encountered is minimized. This schedule can be represented by a binary tree, called a ***quantification tree (or schedule)***, where every node specifies which variables are quantified after the left and right children have been multiplied. The leaves of the quantification tree are the original relations, referred to as ***leaf*** or ***original relations*** . The term resulting at each node of the quantification tree is called a ***partial product***, and corresponds to multiplying a set of leaf relations and quantifying a set of variables.

Since it is hard to estimate the size of a BDD resulting from an operation such as multiplication or quantification, we use the number of variables the BDD depends on (***support*** of the BDD) as an easily computable approximation. Two cost functions for approximating the cost of a node t in a quantification tree can be defined. The ***support cost function*** is the number of variables in the relation represented at t, whereas the ***un-quantified support cost function*** is the number of variables in the relation obtained by multiplying the left and right children (and before quantifying the variables to be quantified at t). We show that the problem of finding a quantification tree minimizing the un-quantified support cost function is NP-complete. We then suggest two heuristic algorithms for solving the early quantification problem.

Our first algorithm amounts to a "good" implementation of a greedy strategy. The idea is to have a set of trees, representing partial products. A ***merge*** represents multiplying two partial products and quantifying out all variables which do not appear in any other partial product. The support cost function is used in this algorithm, i.e. the cost of a merge is the size of the support set which results after the merge. A merge minimizing this cost is chosen, and the merge is performed. If there are k partial products, then there are $O(k^2)$ possible merges. The challenge here is to design data structures so that choosing the best merge, and then updating the data structures after a merge can be done efficiently. We provide one solution to this.

The time required to find a good schedule becomes important when the size of the problem, i.e. the number of relations or variables, is large. This come up in an environment where HDL descriptions of non-deterministic FSMs are compiled into an intermediate format, such as BLIF-MV ([HSIS94], [Che94]). The compiler may create many relations and intermediate variables by representing a FSM as a multi-level netlist of relations. To find the transition relation of a FSM, one can multiply all of the relations and quantify out the intermediate variables. This application gives rise to an instance of the early quantification problem, with possibly hundreds or thousands of relations and variables.

In general, this problem can arise in any verification system where a high-level description of a large non-deterministic FSM is compiled into an intermediate format. Since there are many input languages for verification tools, such as VHDL, Verilog, protocol languages, LOTOS, and graphical languages, intermediate formats are indispensable for verification tools. Since the descriptions can be non-deterministic, the relations are not functions, and hence functional composition of BDDs cannot be used directly to compute the transition relation of the FSM. The most general formulation we know of for computing the transition relation is in terms of early quantification. Even in the absence of non-determinism, the early quantification formulation has more degrees of freedom compared to the functional approaches such as the one in [Sis92], which may lead to more efficient methods.

The second algorithm is geared to smaller problems. It uses the dynamic programming principle to build an optimal quantification tree given a linear order for the leaf relations. Finding a good linear order is an important sub-problem, and remains a challenge. However, one application of this algorithm is to optimize the greedy algorithm as follows.

1. Run the greedy algorithm to get a quantification tree. Then, derive a linear order from it.
2. Use the dynamic programming algorithm to build an optimal tree for that linear order.

The early quantification appears in other application areas besides formal verification. We are aware of applications in the following areas.

1. In synthesis, in computing don't care sets ([Sav94]).
2. In implicit state minimization using BDD's ([KBVS94]).
3. In testing, where the satisfiability of a set of Boolean equations can be checked by testing whether the BDD resulting from multiplying all Boolean equations and quantifying all variables is 1.

In most verification algorithms, the set of reachable states of the product graph needs to be computed. This set is the least fixed-point of $R_k(y) = R_{k-1}(y) + \exists x (R_{k-1}(x) \wedge T(x, y))$, where R_k represents the set of states reachable in k steps, R_0 the set of initial states, and $T(x, y)$ is the transition relation of the product graph. Often, in large problems, the BDD for $T(x, y)$ cannot be built due to space limitations. In such cases, since $T(x, y) = \exists i (T_1(x_1, i_1, y_1) \wedge \ldots \wedge T_n(x_n, i_n, y_n))$, we can write the fixed point computation as $R_k(y) = R_{k-1}(y) + \exists x \exists i (R_{k-1}(x) \wedge T_1(x_1, i_1, y_1) \wedge \ldots \wedge T_n(x_n, i_n, y_n))$, thereby avoiding computing $T(x, y)$. The trade-off with this approach is that computing the set of reachable states usually takes longer.

To speed-up this computation, partial collapsing can be used, i.e. individual transition relations can be clustered (pre-multiplied) to create fewer relations to be multiplied at every step of the reachability computation. Note that all variables in i which appear in only one cluster can be pre-quantified out. Now, each step of the reachability computation is an instance of the early quantification problem. The ***partitioned transition relations problem (PTR)*** is to find a clustering of the transition relations, so that the time for computing the set of reachable states is minimized.

Our solution for this problem is related to first find a quantification tree for computing $T(x, y)$ using algorithms for early quantification. Then, the multiplications and quantifications given by the quantification tree are performed until the sizes of the BDDs for the partial products become larger than some threshold value. Assume that l partial products $P_1, \ldots, P_l$ result, and that a subset $\bar{i}$ of i remains to be quantified out. At each reachability step, we have to solve an instance of the early quantification problem with the $l+1$ terms, where the current set of reachable states R_k is the additional term. We make two observations to avoid calling early quantification at each step.

1. In most cases, R_k involves all of the x variables.

2. Any further collapsing of the partial products $P_1, \ldots, P_l$ might result in a blow-up.

Hence, using observation 1, early quantification is called with a dummy set representing R_k involving all x variables, relations $P_1, \ldots, P_l$, and variables $\bar{i}$ and x to be quantified. Using observation 2, the schedule is further restricted to be a linear chain with the first element being the current set of reachable states. This schedule has the property that no partial products P_i and P_j are directly multiplied. We take this as a fixed schedule for all iterations.

Four related papers should be mentioned.

1. [TSLBS90] used a balanced binary tree at each step of the reachability computation to compute $R_k(y) = R_{k-1}(y) + \exists x \exists i (R_{k-1}(x) \wedge T_1(x_1, i_1, y_1) \wedge \ldots \wedge T_n(x_n, i_n, y_n))$. Early quantification was not addressed, balanced binary trees were used as quantification trees in each step of the reachability computation, and no partial collapsing was performed.

2. In [BCL91] partial collapsing and then use of linear chain was proposed. However, no automatic algorithms were given, and the partial collapsing and linear order were chosen manually.

3. [CC93] introduces a new "exist" generalized cofactor which allows for distribution of conjunction and quantification. Their technique for the early quantification problem is quadratic in

the number of relations, and hence is not practical when the number of relations is large, as may be the case when HDL descriptions of non-deterministic FSMs are translated into intermediate formats ([Che94]). However, it appears that for the partitioned transition relation problem one can use both our techniques and theirs to achieve better results. Using our techniques the number of transition relations is reduced by automatically clustering highly correlated FSMs. Then using the techniques of [CC93] the reachability step can be sped up. We have not performed any experiments with this idea.

4. [GB94] addressed the problem of partition transition relations and gave a simple greedy algorithm for it, without addressing the early quantification problem.

Another technique for reducing the sizes of BDDs is dynamic variable reordering (*DR*, [Rud93]). BDDs are very sensitive to the order of the variables. DR changes this order dynamically during each BDD operation in order to reduce the size of BDDs. EQ and PTRs appear to be orthogonal to DR, i.e. the application of one technique does not render another ineffective.

This chapter is organized as follows. Section 3.2 presents the early quantification problem. Section 3.3 describes our greedy algorithm for early quantification, whereas in section 3.4 the exact algorithm given a linear ordering of the relations is described. Section 3.5 gives our experiments with early quantification, whereas section 3.6 discusses our experiments with the partitioned transition relations. Future directions are given in section 3.7.

3.2 The Early Quantification Problem

In this section, we formulate the early quantification problem, and prove an abstraction of it is NP-complete.

Assume a relation $R(x)$ is given, where the variable x ranges over a finite domain of size n. The characteristic function of R can be represented by a Boolean equation over $\log n$ variables. In general, if $R(x_1, \dots, x_k)$ is a relation over finite domains with $n_1, \dots, n_k$ elements respectively, then the characteristic function of R can be represented by a Boolean equation over $\log(n_1) + \dots + \log(n_k)$ variables. We denote all $\log n$ Boolean variables corresponding to a finite domain x of size n by x, knowing that x represents a set of Boolean variables. A convenient multi-valued interface to the BDD package is described in [Kam91].

Definition Let n relations $R_1(u_1), \dots, R_n(u_n)$ be given, where each u_i represents a set of variables on which R_i depends. Let $u = \cup u_i$, i.e. the set of all variables. Let $x \subseteq u$. The ***early quantification problem (EQ)*** is to compute in minimum space the BDD corresponding to $\exists x (R_1(u_1) \wedge \dots \wedge R_n(u_n))$, where each R_i is represented by a BDD. We call $R_1, \dots, R_n$ the ***leaf relations***, x the ***quantifying variables***, and u the ***occurring variables***.

Definition A ***quantification tree*** is a rooted binary tree, where the leaves are the R_i's. Each node t contains a set of variables $q(t) \subseteq x$, which occur only in the relations contained in the subtree rooted at t. The set $q(t)$ represents the set of variables which should be quantified after the left and right children have been multiplied. $R(t)$ denotes the relation represented at t, obtained by multiplying the children of t and quantifying the variables $q(t)$. The cost of a quantification tree is the maximum of the costs of all nodes (to be defined below) in the tree.

Definition A ***partial product*** is of the form $P = \exists x_{l_1} \dots \exists x_{l_k} \left(R_{i_1} \wedge \dots \wedge R_{i_j} \right)$, where $R_{i_1}, \dots, R_{i_j}$ are leaf relations, $x_{l_1}, \dots, x_{l_k}$ are quantifying variables not occurring in any other leaf relations except for $R_{i_1}, \dots, R_{i_j}$. The partial product P can be represented by a quantification tree for $\exists x_{l_1} \dots \exists x_{l_k} \left(R_{i_1} \wedge \dots \wedge R_{i_j} \right)$, and can be part of a quantification tree for $\exists x (R_1 \wedge \dots \wedge R_n)$. Conversely, each node in a quantification tree represents a partial product.

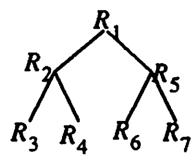
Since it is hard to get a good a priori bound on the size of a BDD resulting from a multiplication or quantification, we use the size of the support set of a BDD as an approximation. In this context, two cost functions for a node t can be defined.

1. ***Support cost function***, which is the size of the support of the relation $R(t)$ at node t, i.e. the size of the support of $R(t)$. Note that we assume that if two BDDs R_1 and R_2 with support sets S_1 and S_2 are multiplied, the resulting BDD depends on $S_1 \cup s_2$. This is an approximation to reality since the real support is a subset of $S_1 \cup s_2$. This multiplication and quantification can be done in one BDD operation, called ***BDD-and-smooth***. Unless otherwise stated, by cost of a node, we mean the support cost.

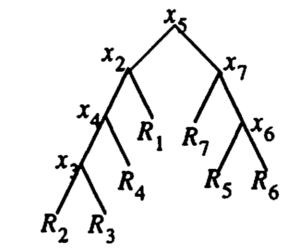
2. ***Un-quantified support (UQS) cost function***, which is the size of the support of the relation obtained by multiplying the children (but before the variables $q(t)$ are quantified). For example, the support cost of $\exists x (R_1(x, y) \wedge R_2(y, z))$ is 2, whereas its un-quantified support cost is 3. This cost function makes sense since the space taken by BDD-and-smooth may be correlated to

the size of the BDD resulting from multiplying the two functions.

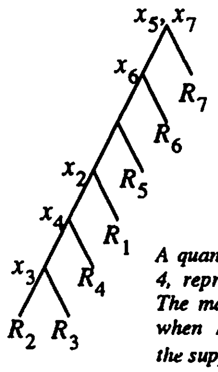
It can be shown that if early quantification is restricted to linear chains, i.e. arranging the relation in an array and multiplying and quantifying from left to right, then the cost (support cost) may increase by a factor of $\log n$, where n is the number of relations. The following example shows an instance of early quantification with 7 relations and 7 variables. The variable dependency of each relation can be defined using the binary tree shown in the left part of figure 3.1. Each node of this tree is labeled by some R_i. Each R_i is dependent on x_i, and any other x_j such that R_i and R_j are neighbors in the tree. For example, R_2 is dependent on x_3, x_4, and x_1, and R_3 is dependent on x_2 and x_3. Let $\exists x_2 \ldots \exists x_7 (R_1 \wedge \ldots \wedge R_7)$ be computed. The minimum cost over all quantification trees is 3, whereas that of linear chains is 4. By generalizing this example, one can build an example with $2^n - 1$ relations and variables, such that the minimum cost over all quantification trees is 3, whereas that of linear chains is $\log n + 1$.



The tree of variable dependency structure for the set of 7 relations and 7 variables.



A quantification tree of cost of 3.



A quantification tree of cost of 4, representing a linear chain. The maximum cost is achieved when R_5 is multiplied, where the support is x_1, x_5, x_6 x_7.

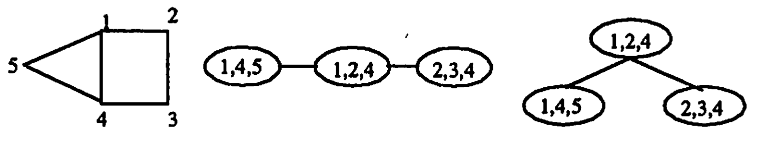
Figure 3.1: Examples of quantification tress

Theorem 3.1 Answering the question of whether a quantification tree of un-quantified support cost less than c exists is NP-complete.

Proof To see the problem is in NP, just guess the quantification tree and verify that it is a valid one, and has cost less than c. To show the problem is NP-hard, we reduce the ***minimum width tree decomposition (MWTD) problem***, described in [RS86] and proven NP-complete in [ACP87], to it. Given an undirected graph $G = (V, E)$, a ***tree decomposition*** is an undirected tree T, where every node i in the tree is marked by some $X_i \subseteq V$, and the following conditions are satisfied.

1. Every node in V appears in some X_i, i.e. $\cup X_i = V$.
2. Every edge $(u, v) \in E$ lies in some X_i, i.e. there is some X_i such that $u \in X_i$ and $v \in X_i$.
3. If there is path from i to j passing through k in T, then $X_i \cap X_j \subseteq X_k$.

The ***width*** of a tree decomposition is $max(|X_i|)$. Answering the question of whether a tree decomposition of width less than c exists is NP-complete. Figure 3.2 gives an example of a graph with one of its tree decompositions. The figure on the right shows a rooted tree, derived from the undirected tree in the middle.

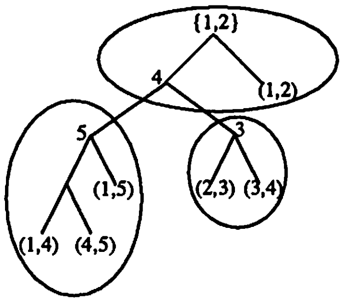


The figure on the left shows a graph on 5 nodes. The middle figure shows a tree decomposition of width 3, and the right figure is a rooted tree corresponding to the undirected tree in the middle.

Figure 3.2: Example of tree decomposition

Let $V = \{v_1, ..., v_n\}$, and $E = \{e_1, ..., e_m\}$. The reduction to early quantification is as follows. Create n variables $v_1, ..., v_n$. For each edge $e_k = (v_i, v_j)$, create a relation $e_k(v_i, v_j)$, and ask whether a quantification tree of UQS cost less than k for $\exists v_1 ... \exists v_n (e_1 \wedge ... \wedge e_m)$ exists. Assume a tree decomposition T of width less than c exists. Make T rooted. Starting from the leaves, recursively create the quantification tree as follows. Let node $n \in T$ be marked by X_n, and have children $c_1, ..., c_k$. Let $Q_1, ..., Q_k$ be the quantification trees for $c_1, ..., c_k$. Let $e_1, ..., e_l$ be those edges which lie inside X_n, but do not appear in $Q_1, ..., Q_k$, or anywhere else in the quantification tree built so far. Build an arbitrary quantification tree T_n for node n on the leaves $\{Q_1, ..., Q_k\} \cup \{e_1, ..., e_l\}$, quantifying a variable u if all edges involving u appear in $Q_1, ..., Q_k$ or are among $e_1, ..., e_l$. Figure 3.3 shows the quantification tree for the tree decomposition in figure 3.1, where the groupings show what happens after different nodes of the tree decomposition are processed.

To see that the cost of the resulting quantification tree built by this procedure is less than c, we proceed by induction on the above recursive procedure. If n is a leaf in the tree decomposition T, then the un-quantified supports in the quantification tree Q_n are contained in X_n. Hence, UQS cost of n is less than c. If n is an intermediate node with children $c_1, \ldots, c_k$, by the inductive assumption, the UQS costs of $Q_1, \ldots, Q_k$ are less than c. Let $S_1, \ldots, S_k$ denote the supports at $Q_1, \ldots, Q_k$ after the variables to be quantified have been quantified. We need to show $S_j \subseteq X_n$ for each S_j. Let $u \in S_j$. Then, there is some edge e involving u which does not appear in Q_j. Since n will be on the path from u to the node in T in which e lies, by condition 3 of tree decompositions, we have $u \in X_n$.



The quantification tree built by the above algorithm for the tree decomposition in figure 3.2. Note that the leaves correspond to the edges of G. The un-quantified support cost of this quantification tree is 3.

Figure 3.3: Quantification trees and tree decompositions

We now show that a quantification tree of cost less than c leads to a tree decomposition of width less than c. Assume a quantification tree Q with UQS cost less than c for $\exists v_1 \ldots \exists v_n (e_1 \wedge \ldots \wedge e_m)$ is given. Define T to have the same tree structure as Q, and let X_n for node $n \in T$ be the un-quantified support at $n \in Q$. We need to show T satisfies the three conditions of a tree decomposition, and has width less than c. Conditions 1 and 2 are satisfied since the leaves of Q are the edges of G. Let k be on the path from i to j in T. Let $u \in X_i \cap X_j$. Then, u can be first quantified from the common root r of i and j. If $k = r$, we are done, since u belongs to the un-quantified support of r. Otherwise, then, k is a descendent of r, and hence contains u. In both cases, $u \in X_k$. The width of T is less than c, since the size of the set of variables at node $n \in T$ is equal to the size of the un-quantified support at $n \in Q$ (QED).

3.3 A Greedy Algorithm for Early Quantification

In this section, we describe a greedy algorithm for early quantification. Assume an instance of early quantification of the form $\exists x_1 \ldots \exists x_m (R_1 \wedge \ldots \wedge R_n)$ is given.

Definition Two partial products are ***disjoint*** if they have no relations in common. Let P_1 and P_2 be disjoint. We say partial product P is the result of ***merging*** P_1 and P_2 if the set of relations of P is the union of the relations of P_1 and P_2, and the set of quantifying variables of P contains those of P_1 and P_2 and any other variable which appears only in P_1 and P_2. Let u_1, u_2, and u be the supports of P_1, P_2, and P, respectively. The ***cost of merging*** P_1 and P_2 is $|u| - MAX(|u_1|, |u_2|)$. If P_1 and P_2 are represented by quantification trees t_1 and t_2, merging them amounts to forming a new quantification tree t whose children are t_1 and t_2, and whose set of quantifying variables are those variables which appear only in t_1 and t_2 and are not quantified in t_1 and t_2.

In some applications, such as compilation of high-level languages to intermediate formats, the set of BDDs to be multiplied contain many constants. Let a ***constant BDD*** be one whose support includes only one variable, and the number of minterms it represents is 1. An example is the BDD for the relation $x = 5$. ***Constant propagation,*** which repeatedly multiplies the constant BDDs into every relation in which they appear, is effective in reducing the size of the leaf relations. We assume constant propagation is performed before our algorithms for early quantification are called, although this does not change our algorithms in any way. Our greedy algorithm is described below.

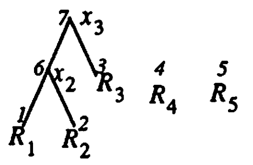
```
1. Start with each relation, and quantify those variables which appear in only one relation. Set done to
   false.
2. Until there is only one quantification tree left or done is true,
      2.1 Choose a minimum cost merge, among a "subset" of possible merges.
      2.2 If the cost of the chosen merge is less than some user-defined value, perform the
          merge. Otherwise set done to true.
```

After describing the data structures needed for efficiently implementing this algorithm, the details of the algorithm are given.

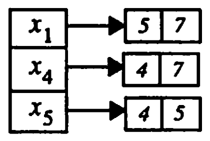
3.3.1 Data Structures for the Greedy Algorithm

The greedy algorithm maintains a forest F of disjoint quantification trees $T_1, \ldots, T_k$. ***Active variables*** are a subset of the variables to be quantified, which have not been quantified in

$T_1, ..., T_k$. The ***connections*** of an active variable are those T_i's (among $T_1, ..., T_k$) in which it appears. Each node in every T_i is given a unique integer identifier. The ***variable connection table*** for a variable v is a hash table which contains the set of connections of v. The ***connection table*** is a hash table which points to the variable connection table for every variable v. Insertion into and deletion from both of these tables can be done in constant time. Figure 3.4 gives an example of a forest and its connection table, where are five leaf relations $R_1(x_1, x_2)$, $R_2(x_2, x_3)$, $R_3(x_3, x_4)$, $R_4(x_4, x_5)$, $R_5(x_5, x_1)$, and $\exists x_1 ... \exists x_5 (R_1 \wedge ... \wedge R_5)$ is to be computed.



A forest of 3 quantification trees, where every node has a unique identifier.



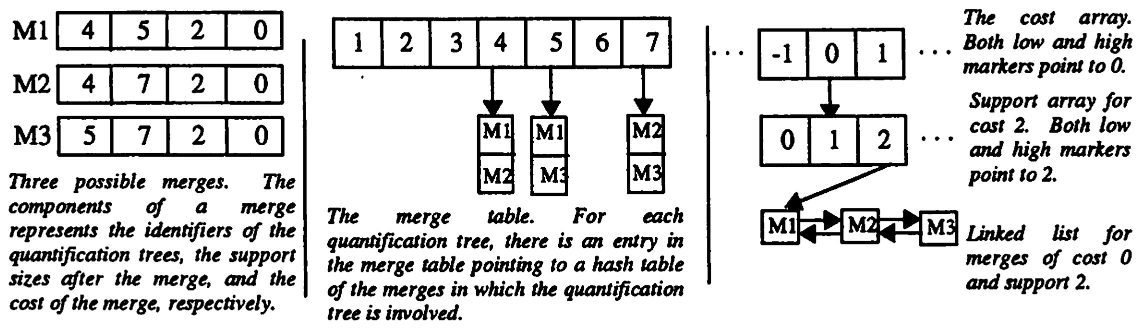
The connection table for the forest in the left figure.

Figure 3.4: A forest and its connection table

Definition A ***merge*** M is a tuple (t_1, t_2, c, s), where t_1 and t_2 are the two quantification trees to be merged, c is the cost of M, and s is the size of the support after the merge. Note that in computing a merge no BDD operations are needed.

All merges are managed in a ***cost array***, where merges of the same cost are in the same ***bucket***. A low and high marker keep track of the lowest and highest cost merges. The bucket for a cost c points to a ***support array***. A position s in a support array A for some cost c points to a doubly linked list containing all merges of support size s (and cost c). Hence, given c and s, the linked list $L(c, s)$, linking all merges of cost c and support s, can be found in constant time. Since a merge can appear in only one such linked list, the next and last pointers are maintained inside the data structure for a merge. For a support array A, the high and low markers keep track of the smallest and largest support sizes contained in A. These are the first and last positions in A for which the linked list of merges in not empty.

The cost array is of fixed size $(-C_L, C_H)$, where C_L and C_H are non-negative constants. A merge whose cost is lower than $-C_L$ or greater than C_H is inserted in the bucket for $-C_L$ and C_H, respectively. Similarly, the support arrays are of fixed size $(0, S_H)$. In our implementation, $C_L = C_H = 100$, and $S_H = 200$ since a BDD with more than two hundred variables is rarely built. The ***merge table*** is an array indexed by the identifiers of the quantification trees, pointing to the set of merges in which the quantification tree is involved. Figure 3.5 gives the merges, cost array, and merge table for the forest in figure 3.4.



Three possible merges. The components of a merge represents the identifiers of the quantification trees, the support sizes after the merge, and the cost of the merge, respectively.

The merge table. For each quantification tree, there is an entry in the merge table pointing to a hash table of the merges in which the quantification tree is involved.

The cost array. Both low and high markers point to 0.

Support array for cost 2. Both low and high markers point to 2.

Linked list for merges of cost 0 and support 2.

Figure 3.5: Data structures for the greedy algorithm

In what follows, we will prove some complexity results of the data structures. We assume $o = O(m)$ where o is the total number of variables, and m is the number of quantifying variables. This assumption is reasonable since we have observed that in large instances of early quantification, a large proportion of the variables are quantified. If this is not the case, then m has to be replaced by o in all of our results below. In our implementation of quantification trees, at every node t, we store two sets of BDD variables: $q(t)$ and $s(t)$, which are the sets of quantifying and support variables at node t, respectively. Each BDD variable is represented by a unique integer identifier, and the sets $q(t)$ and $s(t)$ are in increasing order of BDD variable identifiers.

Lemma 3.3.1 A merge $M = (t_1, t_2, c, s)$ can be computed in time $O(m)$.

Proof To compute M, s has to to be computed. Let r_1 and r_2 be the roots of t_1 and t_2. Since $s(r_1)$ and $s(r_2)$ are in sorted order, scan them in increasing order. If a variable v occurs in both $s(r_1)$ and $s(r_2)$, and has only two connections, mark it as quantified. Otherwise, v will be in the support after the merge. The results follows by noting that the number of connections of a variable can be looked up in constant time from the variable connection table, and the assumption that $o = O(m)$ (QED).

Lemma 3.3.2 A merge $M = (t_1, t_2, c, s)$ can be inserted into or deleted from the merge table in constant time.

Proof Let H_1 and H_2 be the hash tables containing the merges for t_1 and t_2. H_1 and H_2 can be looked up from the merge table in constant time. The lemma follows by noting that insertion into and deletion from a hash table is a constant time operation (QED).

Lemma 3.3.3 A merge $M = (t_1, t_2, c, s)$ can be inserted into or deleted from the cost array in constant time.

Proof Insertion can be done by inserting M at the beginning of $L(c, s)$. Deletion is done by adjusting the next and last pointers, maintained in M. If M is the first link in $L(c, s)$, then $L(c, s)$ is set to point to the second one (QED).

Definition A merge $M = (t_1, t_2, s, c)$ is ***performed*** by following the procedure below.

1. Create a new quantification tree t with children t_1 and t_2, and set $q(t)$ to contain those quantifying variables which have not been quantified in t_1 and t_2, and occur only in t_1 and t_2.

2. Update the connection table by deleting the variables in $q(t)$. Update the variable connection table for a variable v occurring in t_1 (in t_2) by deleting the connections to t_1 (to t_2) and adding a connection to t.

3. Delete M from the merge table and cost array.

4. Delete each merge $M' = (t_1, t'_2, s', c')$ involving t_1 from the merge table and cost array, and insert a merge $M'' = (t, t'_2, s'', c'')$ into the merge table and cost array.

5. Update merges involving t_2 as in the previous step.

Lemma 3.3.4 A merge can be performed in $O(nm)$ time.

Proof The set $q(t)$ in step 1 can be computed by visiting each variable in $s(t_1)$ and $s(t_2)$ once (recall the variables in $s(t_1)$ and $s(t_2)$ are in sorted order). Hence, step 1 takes $O(m)$ time. Step 2 takes $O(m)$ time since insertion into and deletion from the connection table and variable connection tables can be done in constant time. By lemmas 3.3.2 and 3.3.3, step 3 can be performed in constant time. There are at most n merges involving t_1, since the number of quantification trees is less than n. By lemmas 3.3.1, 3.3.2 and 3.3.3, steps 4 and 5 take $O(nm)$. The lemma follows (QED).

3.3.2 The Greedy Algorithm in More Detail

A main point left unresolved in the greedy algorithm is how a subset of possible merges is chosen. If there are k quantification trees in a forest F, there are $O(k^2)$ possible merges. If k is large, it may not be possible to build the data structures for all merges. Our approach, given below, is to perform low-cost merges (to be defined precisely below) first on a small subset of possible merges. This will reduce the size of F. On this smaller forest, and for higher cost merges, a bigger subset of possible merges is considered.

1. Create the small merge table: for an active variable v with two connections, create a merge between the two quantification trees in which v appears.

2. Perform merges with a small maximum cost M_S, i.e. choose a minimum cost merge, and perform the merge if its cost is not greater than M_S.

3. Create the full merge table: for an active variable v whose number of connections is less than some maximum value E_M, create all possible pairwise merges between the quantification trees in which v appears. Arrange in an arbitrary ring, all quantification trees containing some variable u whose number of connections is larger than E_M. Create a merge between any neighboring quantification trees in this ring.

4. Perform merges, i.e. choose a minimum cost merge and perform the merge, until left with only one quantification tree (if the merge table is empty, choose two arbitrary subtrees).

For our implementation, we let $M_S = 0$ and $E_M = 50$. We have the following results.

Lemma 3.3.5 The number of merges created by the small merge table is bounded by $O(m)$.

Proof There are at most m active variables having two connections. For each such variable, at most one merge is created in the small merge table (QED).

Lemma 3.3.6 The number of merges created by the full merge table is bounded by $O(m+n)$.

Proof The number of merges created due to active variables whose number of connections are less than E_M is bounded by $O(m)$. Since there are at most n quantification trees (there are no more quantification trees than leaf relations), at most $O(n)$ merges are created due to the ring in step 3 (QED).

3.3.3 Complexity of the Greedy Algorithm

Theorem 3.2 The running time of the algorithm is bounded by $max(O(n^2 m), O(m^2))$.

Proof The time to create merges is bounded by $O(m^2 + mn)$ by lemmas 3.3.1, 3.3.5 and 3.3.6. Since at most $O(n)$ merges are performed, by lemma 3.3.4, the total time for performing merges is $O(n^2 m)$. The theorem follows (QED).

When the number of relations is large, the early quantification problem may be (and is the case in our experience) ***sparse***, i.e. every variable appears in $O(1)$ relations, and every relation has $O(1)$ variables. Hence, for sparse problems, $O(n) = O(m)$.

Theorem 3.3 If the early quantification problem is sparse, then the running time of the algorithm is $O(n)$.

Proof There are at most $O(n)$ merges created by the small and full merge tables. Performing a merge takes constant time, since every quantification tree is involved in a constant number of merges, and the size of the support of the quantification trees is bounded by a constant. Hence, performing the merges takes $O(n)$ time (QED).

3.4 An Exact Algorithm for Early Quantification Given A Linear Order

Let an instance of early quantification $\exists x_1 \dots \exists x_m (R_1 \wedge \dots \wedge R_n)$, and an ordering $R_{i_1}, \dots, R_{i_n}$ of the relations be given. Without loss of generality, we assume $i_1 = 1, \dots, i_n = n$. Given a quantification tree t, its ***linear realization*** is an ordering of the leaf relations obtained by a depth-first search visiting left child, right child, and the root, respectively. Intuitively, the linear realization is the left-to-right ordering of the leaves of t. In this section, we give an algorithm which returns in polynomial time a minimum cost quantification tree over all those quantification trees whose linear realization is $R_1, \dots, R_n$.

Let $p[i,j]$ be the partial product $\exists \hat{x} (R_i \wedge \dots \wedge R_j)$, where $\hat{x}$ is the subset of quantifying variables which occur only in $R_i, \dots, R_j$. Let $\text{cost}[i,j]$ be the minimum cost over all quantification trees for $p[i,j]$ whose linear realization is $R_i, \dots, R_j$. If t is a quantification tree whose linear realization is $R_i, \dots, R_j$, then there exists k, $i \le k < j$, such that the linear realization of the left child of the root is $R_i, \dots, R_k$, and the linear realization of the right child of the root is

$R_{k+1}, \ldots, R_j$. Index k is called the ***separator*** of t. The idea of the algorithm is to calculate the best way of computing partial products of the form $p[i,j]$. In the algorithm given below, first the cost of the original terms is computed. Then, the best way of computing $p[i,j]$ for increasing $j-i$ is calculated.

```
0. Preprocessing step: quantify out any variables present in only one relation.
1. For $i = 1, \ldots, n$, initialize cost$[i,i]$.
2. For $d = 1, \ldots, n-1$,
     For $i = 1, \ldots, n-l$
       $j = i+l$
       cost$[i,j] = \infty$
       For $k = i, \ldots, j-1$
            $q = ComputeCost(i,j,k)$ (described below).
            If $q <$ cost$[i,j]$, then cost$[i,j] = q$, and $seperator[i,j] = k$.
```

3.4.1 Cost Functions

In the above algorithm, $ComputeCost(i,j,k)$ is a function returning the cost of computing $p[i,j]$ in a quantification tree where the left (right) child of the root is a quantification tree of minimum cost whose linear realization is $R_i, \ldots, R_k$ ($R_{k+1}, \ldots, R_j$). Let t be a node in a quantification tree, representing partial product p, and having children l and r. We have experimented with the following two cost functions.

1. The un-quantified support cost function returns $MAX(|s(l) \cup s(r)|, |s(l)|, |s(r)|)$, where $|s(n)|$ denotes the size of the support at some node n. Note that $|s(l) \cup s(r)|$ denotes the size of the support at t before the variables $q(t)$ are quantified.

2. The ***variable duration cost function***, motivated in [BCL91], measures the cost of quantifying a variable v as the number of multiplications that has to be performed starting from the original relations to form the partial product from which v is quantified. Intuitively, it enforces the principle "the earlier you quantify a variable the better." More precisely, let $|t|$ be the number of intermediate nodes in t, which is also the number of multiplications needed to form the partial product p (represented by t) starting from the original relations. Let $|q(t)|$ denote the number of variables to be quantified at t Then, $cost(t) = cost(l) + cost(r) + |q(t)||t|$. Note that the

cost of t can be determined in both cases from the costs of l and r.

3.4.2 Correctness and Complexity

Theorem 3.4 Given a linear ordering of the relations, the above algorithm returns a minimum cost quantification tree for the given cost function.

Proof A quantification tree of cost $\mathrm{cost}[1,n]$ can be built using the separators. Hence, it suffices to show $\mathrm{cost}[i,j]$ is the minimum cost of computing $p[i,j]$, subject to the linear order. To show this, we use induction on the number of terms in a partial product $p[i,j]$. The base case, i.e. $p[i,i]$, holds since there is only one way to compute $p[i,i]$. The claim follows from the inductive assumption and by noting that the algorithm tries all possible ways of computing $p[i,j]$ from smaller sub-partial products subject to the linear ordering (QED).

Theorem 3.5 The above algorithm runs in $O(n^3 m)$, where n is the number of relations and m is the number of variables to be quantified.

Proof The algorithm computes $\mathrm{cost}[i,j]$ for $i = 1, \ldots, n-1$ and $j \geq i$. There are $O(n^2)$ such calculations. To compute $\mathrm{cost}[i,j]$, for all k, $i \leq k \leq j-1$, the support and quantifying variables of a quantification tree t whose left and right children are the partial products $p[i,k]$ and $p[k+1,j]$ have to be computed. For a quantifying variable v, let $l(v)$ and $r(v)$ denote the smallest and largest indices of the relations in which v occurs. v is in the support of t iff $l(v) < i$ or $j < r(v)$; v is quantified at t iff $i \leq l(v) \leq k < r(v) \leq j$. By pre-computing $l(v)$ and $r(v)$ for all quantifying variables, the support and quantifying variables at t can be determined in $O(m)$ time. To determine $\mathrm{cost}[i,j]$, $O(n)$ such calculations are needed. The theorem follows (QED)

Remark In theorem 3.5, as before, we assume $o = O(m)$, where o is the total number of variables.

3.4.3 Obtaining Linear Ordering

A minimum cost quantification tree can be built by finding a suitable linear order and running the above algorithm. Linear realizations of minimum cost quantification trees are among linear orders on which the algorithm will return a minimum cost quantification tree. Currently, we use the linear realization of the quantification tree returned by the greedy algorithm. Finding good linear orders for this algorithm remains a challenge.

3.5 Experimental Results for Early Quantification

Since our package is integrated into HSIS, we have experimented with a set of academic and industrial verification examples. The input language to HSIS is Verilog, extended to handle non-determinism. Most verification examples (and, in general, high-level designs) are described as a set of interacting and possibly non-deterministic FSMs. Instances of early quantification arise in two places:

1. As each Verilog process (representing a FSM) is translated into BLIF-MV, a multi-level structure consisting of many small non-deterministic relations is created. To form the transition relation of a FSM, all relations have to be multiplied and the intermediate variables quantified. In this application, it is not uncommon to call the routine with thousands of relations.

2. After the transition relation for each FSM is built, to form the product graph, all these relations are multiplied and non-state variables are quantified. In this application, the input usually consists of less than 100 relations, where each relation is the transition relation for one of the FSMs.

The next two sections summarize our experiments within each of these applications. The examples we used are: ***Gigamax***, a multi-processor cache coherency protocol distributed with the tool SMV([McM93]); ***Scheduler***, an academic example from [Mil89]; ***2mdlc***, a message data link controller obtained from industry.

3.5.1 Experiments Building the Transition Relation of a Single FSM

In this section, we present our experiments with the largest FSMs for each of our three examples. We experimented with three algorithms. The first one (G) is the greedy algorithm, presented in section 3.3. Our second algorithm (B) returns a balanced binary tree built from the original relations, quantifying variables as soon as possible. Our third algorithm (L) is a greedy algorithm building a quantification tree representing a linear chain, where relations are multiplied from left to right, quantifying a variable as soon as possible. It is obtained by restricting our greedy algorithm to choose a minimum cost merge involving the quantification tree resulting from the previous merge.

Table 3.1 shows the results of our experiments. Columns 2 and 3 of the table give the number of relations and variables before and after constant propagation. Column 4 shows the total time to find the schedule and perform the multiplications and quantifications (for these experiments, the time to find the schedule was negligible). Column 5 contains the number of nodes for the largest BDD seen, whereas column 6 gives the maximum support set encountered. In column 7, ***G*** stands for the greedy algorithm, ***B*** for the balanced binary tree algorithm, and ***L*** for the greedy algorithm restricted to return linear chains. Our experiments were run on a DECsystem 5900 with 440MB of RAM.

Table 3.1: Experiments building transition relation of one FSM

Examples	# of relns/after const. prop.	# of vars/after const. prop	time in secs.	largest BDD	max support	alg
Diners	122/80	128/86	0.1 0.08 0.12	40 145 205	12 22 26	G B L
Gigamax	153/100	148/95	0.29 0.89 3.63	202 3370 3014	14 30 33	G B L
Scheduler	106/69	104/67	0.25 0.40 0.61	253 1074 502	12 23 22	G B L
2mdlc-fsm1	212/144	187/119	1.50 2.60 40.1	8982 11261 23731	37 42 49	G B L
2mdlc-fsm2	820/556	809/545	3.02 20.1 177.5	274 62468 62468	24 46 46	G B L
2mdlc-fsm3	3188/2194	2179/1373	11.58	1644 space-out space-out	38 space-out space-out	G B L

Our experiments can be summarized as follows.

1. The support cost function tracks well with the sizes of the BDDs.

2. In all examples, the greedy algorithm performs better than the other two. The difference becomes much larger as the size of the problem grows. In our largest example, the greedy algorithm is the only one which can finish building the BDD.

3. The running-time of all three algorithms to find a schedule were a fraction of the time needed for the BDD operations. Hence, they are not reported here.

4. Constant propagation is effective in reducing the number of relations in this application. We think that this an impact on the total time, although this was not measured.

3.5.2 Experiments Building the Product Graph

For experiments building the product graph, we included the dynamic programming algorithm as well as the algorithms of the previous section (greedy, balanced binary tree, and linear chain). We experimented with two dynamic programming algorithms. The first one, referred to as ***DS***, uses the un-quantified support cost function on a linear order obtained from the quantification tree returned by the greedy algorithm. The second one, referred to as ***DV***, uses the variable duration cost function on a linear order obtained by restricting the greedy algorithm to linear chains.

Table 3.2 shows the result of our experiments. Columns 2 and 3 give the number of relations and quantifying variables, respectively. Column 4 gives the size of the maximum BDD encountered, whereas column 5 gives the cost of the quantification tree. Note for DV, this cost, given in parentheses, is as described in section 3.4.1. In column 6, the time for performing the multiplications and quantifications is given. For DS and DV, the time to find the schedule is also given. The last column shows the different algorithms.

Table 3.2: Experiments building the product graph

Example	# of relns.	# of quant. vars.	largest BDD	cost	time in secs sched/BDD ops.	algorithm
Gigamax	11	22	8507	31	2.36	G
			5693	21	0.04/1.48	DS
			4256	(127)	0.01/2.77	DV
			15042	31	6.42	B
			176163	36	6.87	L
Scheduler	12	39	49999	194	20.29	G
			49999	166	0.06/21.03	DS
			49999	(1143)	0.01/27.47	DV
			102691	194	44.0	B
			102691	194	95.61	L
2mdlc	20	18	2393	40	0.32	G
			2393	40	0.12/4.22	DS
			2718	(120)	0.02/2.86	DV
			-	space-out	space-out	B
			-	space-out	space-out	L

Our experiments can be summarized as follows.

1. The greedy algorithm gives good results in this application also.
2. Although the dynamic programming algorithm always returns a lower cost quantification tree when given the linear realization of the quantification tree returned by the greedy algorithm, this does not always translate into shorter processing time for the BDD operations. For the case of 2mdlc, it performs substantially worse.
3. For our examples, where the number of relations is less than 40, the time taken by the dynamic programming algorithm is a fraction of a second.

3.6 Experiments with Partitioned Transition Relations

The partitioned transition relations (PTR) problem was defined in section 1, and our solution based on early quantification was given. In this section, we describe our experiments with this problem. The example we worked with is a model of the memory system for the SPARC8 multiprocessor architecture ([SUN90]). To get a good BDD variable ordering, the description was flattened. This description when translated into BLIF-MV results in 1800 relations and 1783 intermediate variables , which need to be quantified. The transition relation for this model cannot be built. However, if the set of reachable states R is computed, and the relations are restricted to R, then the transition relation can be built. The reason is that the un-restricted transition relation represents transitions in both the reachable and unreachable parts of the state space. As Ken McMillan has also observed, if the unreachable part of the state space in not "well-behaved," as is the case with our example, then the BDD for the un-restricted transition relation might blow up.

Our algorithm for PTR first calls early quantification to get a quantification tree, and then performs the multiplications and quantifications until the sizes of the BDDs become larger than a threshold value. At this point, a set of partially collapsed BDDs is obtained, which are arranged in a linear chain and are used to compute the set of reachable states. Table 3.3 shows the result of our experiments, where the columns represent various threshold values, the time taken to form the clusters, the number of clusters, the time for computing the set of reachable states, and the total time, respectively. All times are in CPU seconds.

Table 3.3: Experiments with partitioned transition relations

threshold	collapse (time)	# clusters	reach (time)	total time
100	9.98	59	43.83	53.81
1000	9.05	24	13.96	23.01
5000	9.07	21	12.77	21.84
15000	10.46	19	5.60	16.06
50000	15.25	17	1.79	17.04
200000	50.83	9	3.83	54.66

Our experiments can be summarized as follows.

1. PTRs are vital to verify this example.
2. Partial collapsing is very useful to reduce the time taken to compute the set of reachable states.

3. Although partial collapsing with low or high threshold values gives poor results, there is a range for which partial collapsing does not appear to be too sensitive to the threshold value used.

3.7 Conclusions

Formal verification tools based on BDDs are being used more and more in industrial applications. The central problem for these tools is computing the set of reachable states of the product graph. To do so, a representation of the transition relation of the product graph is needed. In this paper, we have attacked this problem. Our contributions are as follows.

1. We have formulated the early quantification problem, which is the most general formulation we know of for computing the BDD for transition relations of individual FSMs and the product graph. We have shown an abstraction of this problem is NP-complete.

2. We have given a greedy algorithm for heuristically solving the early quantification problem. This algorithm is fast, and has produced very good experimental results on all of our examples. We have also given a dynamic programming algorithm for solving the problem optimally when restricted to a given linear ordering of the relations.

3. For large examples, the BDD for the transition relation of the product graph may be too large to build. In such cases, partitioned transition relations can be used. We have followed the formulation of [BCL91], but have given automatic algorithms, based on our algorithms for early quantification, for solving this problem.

A few directions for further research can be suggested.

1. We have shown the early quantification problem is NP-complete when the un-quantified support cost function is used. Complexity of early quantification with the support or variable duration cost functions needs to be investigated, although it is reasonable to expect the problem to remain NP-complete in these cases as well.

2. Experiments using the greedy algorithm with un-quantified support cost function, and the dynamic programming algorithm using support cost function need to performed.

3. Algorithms for finding good linear orders for the dynamic programming algorithm are needed. Algorithms for one-dimensional placement may be used as a starting point.

Acknowledgment

A first version of this work appeared in [HKB94]. The version presented in this chapter is based mainly on an unpublished manuscript written by the author, S. Krishnan, and R. Brayton.

Ken McMilllan pointed us to the minimum-width tree decomposition problem for the reduction to show early quantification with un-quantified support is NP-complete. General discussions with Tom Shiple were useful.

Chapter 4

Symbolic Computations for Language Containment and Fair CTL

4.1 Introduction

In providing a system for design verification, there are many issues which have to be addressed. In this chapter, we look at several problems: specification of systems, specification of properties, checking of properties, early failure detection, and hierarchical verification.

Specifying the System. In chapter 3, we introduced the combinational/sequential (C/S) concurrency model, to which HDL's can be compiled. However, non-determinism and abstraction may result in unwanted behaviors. An example is in modeling unbounded delays, where we want to allow the system to stay at a state s for some finite but unknown amount of time. The behaviors where the system is in s forever should be excluded. An easy abstraction would allow infinite delays, but then all properties of the system should only be checked on traces, where the system gets out of s infinitely often if it enters s infinitely often. A second example of unwanted behavior is in modeling schedulers. Assume that a set of processes is executed in a system controlled by a ***fair scheduler***: one which disallows starvation of processes. An example of a fair scheduler is a round robin scheduler. To avoid having to model the details of a particular scheduler, system constraints can be used to disallow looking at traces where one process is blocked forever. Such constraints on a system's traces are called ***fairness constraints*** (***FCs***). Therefore, a hardware system consists of a (possibly abstract) model of the hardware plus a set of fairness constraints. In this chapter, we introduce various forms of specifying FCs.

Specifying Properties. We use two methods for specifying properties, CTL [CES86] and edge-Rabin automata (an extension of Rabin automata, [Rab72]). Combined with the fairness constraints (also called acceptance conditions in the context of specifying properties), this provides the ability of specifying any property describable by "fair CTL" plus any "omega-regular" property. Although these two sets overlap, they are not strictly comparable. Hence, not only does this provide a larger set of properties that can be specified than either alone, it also benefits the user by offering a choice according to which is more convenient, efficient or succinct. Since the symbolic methods for proving these two types of properties have many common features, it is not necessary to have to choose between the two in designing a formal verification system. We provide both without too much additional effort.

Checking Properties. Checking that a system satisfies a CTL formula is called ***model checking*** (MC). A set of symbolic methods for model checking a subset of fair CTL was first given in [BCMDH92]. These are based on fixed point computations which have efficient BDD ([Bry86]) implementations. The CTL formula is unrolled creating a parse tree, and the states for which each sub formula holds is computed successively.

Checking that the system satisfies a property specified as a Rabin automaton is done by language containment (LC). Here one needs to prove that the language of the system (set of its traces satisfying the fairness constraints, known as ***fair traces***) is contained in the language of the property automaton (the set of input sequences accepted by the automaton). This is usually done by complementing the property automaton and taking the product of the system with the result. If the resulting product has no fair traces (its language is empty), then the property is proved. An interesting property of our language containment environment, known as the ***edge-Streett/edge-Rabin (eSeR) environment***, is that the language containment check for this environment is polynomial, whereas this check is NP-complete for the next natural (in some sense) extension to this environment ([HSB94]).

For both LC an MC we need to compute a set of states called *fair reachable states*, denoted $Fair+$, which can reach a "fair cycle", a cycle whose traversal satisfies all the fairness constraints. We present several algorithms for computing $Fair+$ [HTKB92]. These are based on the ideas of [EL85] on how to translate fair CTL formulas into μ-calculus. In addition, for LC we give an alternative algorithm, which views LC as a sequence of transformations which trim portions of the state space where no fair behavior can occur.

Early Failure Detection. In practice, a verification tool is more often used when there are bugs in the specification. So a good debugging environment associated with formal design verification is essential. We provide two aids, early failure detection and short error traces. Early failure detection (EFD) refers to finding "easy" bugs quickly, that is without having to go through the complete MC or LC process. In this we try to emulate what simulation is good at, where bugs are discovered easily by simulating relatively random inputs. We provide a set of EFD LC algorithms which although not complete, empirically seem to find most bugs. Hence the entire LC computation often can be avoided.

Short error traces are useful in providing the user with a more easily understood counter-example to the property. An error trace is simply a simulation trace that does not display the required property. One advantage with symbolic methods is that when an error occurs, internally the set of all counter-examples is effectively available. The task of finding a short error trace is to find a fair trace of the system which is as short as possible. Since an error trace involves an initial segment plus a cycle, we provide algorithms to find a shortest path to a fair cycle and then a short fair cycle. Although we do not guarantee that the trace produced is shortest, experience is that they are quite short and these short traces are very useful for debugging purposes. Techniques for finding short error traces are described in chapter 5.

Hierarchical Verification. Even though symbolic BDD methods are quite powerful, eventually the system to be implemented becomes too complex and the state space becomes too large. Then it is necessary to use abstraction to describe the system in order to verify a property. It is desirable that if the property holds for the abstracted system then it also holds for the refined detailed version. It is known, for all omega-regular properties and for some CTL properties, that if the language of the abstract system contains the language of the refined system, then such properties holding for the abstract system also hold for the refined system. We give algorithms for checking language containment between two systems with edge-Streett fairness conditions, where for example one may be the detailed system which should be contained in the other abstract representation.

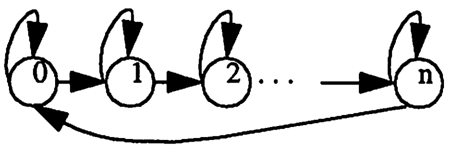
Most methods described in this paper have been implemented in HSIS. The outline is the chapter is as follows. Section 4.2 describes eight types of fairness conditions and their translations into edge-Streett conditions. Section 4.3 presents the algorithms for computing the fair reachable states, *Fair+*. Sections 4.4 and 4.5 give symbolic algorithms for MC and LC respectively. Section 4.6 discusses the advantages for providing both LC and MC capabilities. Section 4.7 describes some of the techniques used for early failure detection. Section 4.8 discusses hierarchical verification strategies and presents some algorithms for this.

4.2 Fairness Constraints

To restrict the behavior of abstract FSM's, fairness constraints are introduced. Fairness constraints put restrictions on which runs are ***accepting***. An ω-string is accepted, i.e. is part of the behavior of the system, if and only if it has an accepting run, where the acceptance condition comes from the fairness constraints. In this section, we first give eight different forms to describe fairness constraints, and although all are not necessary for completeness, they can make it easier for the user to specify the total set of fairness constraints. We then show that all these fairness constraints can be translated into a more restricted format, known as edge-Streett fairness constraints. Fairness constraints can be divided into negative and positive constraints. Within each category, various kinds of fairness constraints are allowed.

4.2.1 Positive Fairness Constraints (PFC)

We motivate positive fairness constraints by an example. Consider the scheduler given below, which schedules processes $0, \dots, n$ in a round robin fashion. If no FCs are imposed, then the scheduler can schedule one of the processes forever. To impose fairness, we restrict the behavior of this machine, by asserting that only those behaviors are accepting, where the edge $(n, 0)$ is taken infinitely often (i.e. a positive fair edge constraint). Intuitively, this means that the scheduler keeps cycling through all the processes.



This scheduler schedules processes in a round rabin fashion, allowing each process to continue for an arbitrary amount of time. To ensure fairness, the edge *(n,0)* is declared as a positive fair edge, one which must be taken infinitely often.

Figure 4.1: Example motivating fairness constraints

In what follows, let r be a run. There are four types of PFCs.

1. ***Positive Fair Nodes***. A set S of states one of which must be visited infinitely often (∞-often), i.e. $inf(r) \cap S \neq \emptyset$.
2. ***Positive Fair Edges***. A set of edges one of which must be traversed ∞-often.
3. ***Positive Fair Subset***. A set S of states which must be visited ∞-often and exited only finitely many times, i.e. $inf(r) \subseteq S$ for a run r.
4. ***Positive Canonical Fairness Constraints (PCFC)***. A constraint of the form $F^{\infty}(S) + G^{\infty}(T)$, where S and T are subsets of states. This constraint is satisfied by a run r iff $inf(r) \cap S \neq \emptyset$ (r visits the set S ∞-often) or $inf(r) \subseteq T$ (after some point, r consists only of states in T).

A set of PFCs are satisfied if all are satisfied.

4.2.2 Negative Fairness Constraints (NFC)

Similarly, there are four types of NFCs.

1. ***Negative Fair Node***. A state q which must visited only finitely many times, i.e. it should not

be traversed ∞-often ($inf(r) \cap q = \varnothing$).

2. ***Negative Fair Edge.*** An edge which must be visited only finitely many times.

3. ***Negative Fair Subset.*** A set of states which either must be visited only finitely many times, or exited ∞-often, i.e. $inf(r) \cap \bar{S} \neq \varnothing$.

4. ***Negative Canonical Fairness Constraints (NCFC).*** A constraint of the form $F^{\infty}(S) \wedge G^{\infty}(T)$, where S and T are subsets of nodes. This constraint is satisfied by a run r iff $inf(r) \cap S = \varnothing$ or $inf(r) \cap \bar{T} \neq \varnothing$.

Note that a run is accepting iff all the positive and negative fairness constraints are satisfied, i.e. $\prod_{i=1}^{k} P_i \wedge \prod_{j=1}^{l} N_j$ holds, where P_i is a positive fairness constraint, and N_j is a negative one.

4.2.3 Edge-Streett Automata

To make algorithm design easier, we translate a FSM with the above fairness constraints to a more restricted form, known as edge-Streett automata.

Definition An ***edge-Streett automaton*** is a FSM with the following fairness constraints: a set of negative fair edge constraints, a set of positive fair edge constraints, and a set of positive canonical fairness constraints. A run is accepting if all of the fairness constraints are satisfied.

Lemma 4.2.1 A system with the above positive and negative fairness constraints can be expressed as an edge-Streett automaton, so that the set of accepting runs is the same in both systems.

Proof We show how each positive or negative fairness constraint can be converted to one of the fairness constraints allowed by edge-Streett automata. Since, negative fair edge constraints, positive fair edges constraints, and PCFC's are allowed by edge-Streett automata, we only need the following transformations.

1. Positive fair nodes. Let S be a set of positive nodes, one of which must be traversed ∞-often. Create a constraint of the form $F^{\infty}(S) + G^{\infty}(\varnothing)$.

2. Positive fair subset. Let S be a positive fair subset. Create a constraint of the form $F^{\infty}(\varnothing) + G^{\infty}(S)$.

3. Negative fair node. Mark each incoming and outgoing edge q of a negative node as negative edges. Alternatively, we can create a CFC of the form $F^{\infty}(\varnothing) + G^{\infty}(\bar{q})$.

4. Negative fair subset. For each negative fair subset S, create a PCFC of the form

$F^{\infty}(\bar{S}) + G^{\infty}(\varnothing)$.

5. NCFC. Transform a NCFC $F^{\infty}(S) \wedge G^{\infty}(T)$ to a PCFC $F^{\infty}(\bar{T}) + G^{\infty}(\bar{S})$ (QED).

The following figure shows the conversions of lemma 4.2.1.

Positive

Fair node S	$F^{\infty}(S) + G^{\infty}(\varnothing)$
Fair edges	N/A
Fair subset S	$F^{\infty}(\varnothing) + G^{\infty}(S)$
PCFC	N/A

Negative

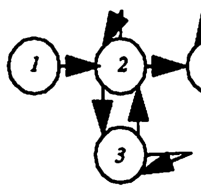
Fair node q	$F^{\infty}(\varnothing) + G^{\infty}(\bar{q})$
Fair edges	N/A
Fair subset S	$F^{\infty}(\bar{S}) + G^{\infty}(\varnothing)$
NCFC $F^{\infty}(S) \wedge G^{\infty}(T)$	$F^{\infty}(\bar{T}) + G^{\infty}(\bar{S})$

Figure 4.2: Translating fairness constraints into edge-Streett conditions

4.3 Computing Fair States

At the heart of MC and many LC algorithms, is a function which computes the set of all fair reachable states, i.e. those from which there is an accepting run. An accepting run according to fairness constraints is called a ***fair path***. In this section, we present an algorithm for this task, which is a modification of the Emerson-Lei computation, presented in [EL85], and re-formulated in the language containment environment by [TKB91]. In [HTKB92], several other ways of computing this set are given. In what follows, let G be a graph, V the set of its vertices, $A \subseteq V$, and $T(x, y)$ its transition relation, i.e. its set of edges.

Definition A ***cyclic strongly connected component (CSCC)*** of G is an SCC of G which contains at least one cycle. The remaining SCC's are single node SCC's with no self-loops. These are called ***acyclic strongly connected components (ASCCs)***.



This graph contains three SCC's: {1}, {2,3}, {4}. The first one is acyclic, the last two cyclic.

Figure 4.3: Cyclic and acyclic SCCs

Definition Let F be a monotone <u>increasing</u> k-ary predicate transformer. Define the ***least fixpoint*** of F given Q, denoted by $\mu(X, Q).FX$ where Q is a k-ary predicate over sets $D_1, \ldots, D_k$, by the set $F^i(Q)$, for the least i such that $F(F^i(Q)) = F^i(Q)$, where $F^0(Q) = Q$. Intuitively, X is the variable we are recurring on, and Q is its initial value. Similarly, define the ***greatest fixpoint*** of a monotone decreasing k-ary predicate transformer F given Q, denoted by $\nu(X, Q).F(X)$, by the set $F^i(Q)$, for the least i such that $F(F^i(Q)) = F^i(Q)$, where $F^0(Q) = Q$. We restrict attention to finite domains for which the fixpoints of monotone predicate transformers are well-defined.

Definition Let $S_1(A, y) = \exists x(A(x) \wedge T(x, y))$ and $S_1(x, A) = \exists y(A(y) \wedge T(x, y))$. Thus, $S_1(A, y)$ are the (one-step) successors of $A(x)$ and $S_1(x, A)$ are the (one-step) predecessors of $A(y)$.

Definition Let $R_1(A, y) = A(y) \vee S_1(A, y)$, $R_1(x, A) = A(x) \vee S_1(x, A)$, $R^*(A, y) = \mu(X, A).R_1(X, y)$, $R^*(x, A) = \mu(X, A).R_1(x, X)$. Note that the first two are "one-step" operators, while the last two are fixed-point computations. One should read $R^*(A, y)$ as the set of points reachable from A. Similarly $R^*(x, A)$ is the set of points that can reach A. Note that $R_1(A, y)$ and $R_1(x, A)$ are monotone increasing predicate transformers.

Definition Let $U_1(A, y) = S_1(A, y) \wedge A(y)$ and $U_1(x, A) = S_1(x, A) \wedge A(x)$. Thus, $U_1(A, y)$ denotes the set of states in A which have predecessors in A, and $U_1(x, A)$ denotes the set of states in A which have successors in A. Note that $U_1(A, y)$ and $U_1(x, A)$ are monotone decreasing.

Definition We define two "stable set" operators, $S^*(A, y) = \nu(X, A).U_1(X, y)$ and $S^*(x, A) = \nu(X, A).U_1(x, X)$. One can think of the first as calculating the ***backward stable set*** contained in A, i.e. the set of all vertices in A which are reached by some vertex involved in some cycle in A. The second is the ***forward stable set*** of A, i.e. the set of all vertices which can reach some vertex involved in some cycle in A)[1].

1. The backward operators introduced in this section will be used in the next section.

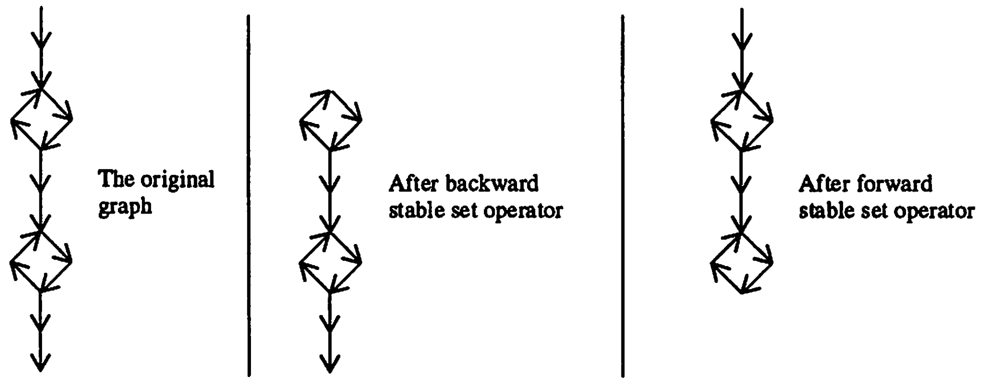

The original graph
After backward stable set operator
After forward stable set operator

Figure 4.4: Effect of stable set operators

Definition Given a set of positive fair edges constraints $\{E_j\}$, n CFC's of the form $F^{\infty}(S_i) + G^{\infty}(T_i)$, and a current set of states S, with $S_i \subseteq S$ and $T_i \subseteq S$, the ***forward fair-path operator*** returns a set of states $M \subseteq S$ such that for each $x \in M$:

1. for each E_j, there is a path in M, starting at x, and reaching some edge E_j, and
2. for each condition $F^{\infty}(S_i) + G^{\infty}(T_i)$, either $x \in T_i$, or there is a path in M, starting at x, and reaching a state in S_i.

The computation for this operator is $FP(S) = \left(\prod_{i=1}^{n}\left(R^{*}(x, S_i) + T_i\right)\right) \wedge \left(\prod_{j=1}^{p} R^{*}\left(x, E^{x}_{j}\right)\right)$, where n is the number of CFCs, S_i and T_i are the sets in the i-th CFC $F^{\infty}(S_i) + G^{\infty}(T_i)$, p is the number of positive edge constraints, E^{x}_{j} represents the set of head states of the j-th set of positive fair edges E_j after these edges are restricted to S, and $FP(S)$ is the result of the forward fair path operator. The corresponding ***backward fair-path operator*** is similar, except that the direction of the path is the reverse. The computation for the backward fair-path operator is $BP(S) = \prod_{i=1}^{n}\left(R^{*}(S_i, y) + T_i\right) \wedge \prod_{j=1}^{p} R^{*}\left({}^{y}E_j, y\right)$, where ${}^{y}E_j$ represents the set of tail states of the j-th set of positive fair edges E_j after these edges are restricted to S.

Definition Let a ***fair cycle*** be a cycle whose infinite traversal satisfies all fairness constraints. Let a ***fair state*** be a state involved in some fair cycle. Let $Fair$ denote the set of all fair states. Let a ***fair reachable state*** be a state which can reach some fair state. Let $Fair+$ denote the set of fair reachable states.

The algorithm for computing the set of all fair reachable states is described as follows.

```
1. Restrict the transition relation $T(x, y)$ to the set of reachable states.
2. Remove negative fair edges from $T(x, y)$.
3. Let $A_0$ be the set of reachable states. Repeat until convergence,
    3.1 Apply the forward stable set operator; $A_{i+1/2} = S^*(x, A_i)$.
    3.2 Apply the forward fair-path operator; .$A_{i+1} = FP(A_{i+1/2})$
4. Let the set computed in step 3 be $U$. Return $R^*(x, U)$, i.e. the set of states which can reach $U$
```

Theorem 4.1 The above algorithm computes $Fair+$.

Proof Denote the set returned by the above algorithm by W. To show $Fair+ \subseteq W$, it suffices to show $Fair \subseteq U$. Let $x \in Fair$. Then, x is in some fair cycle. So, it is not deleted by the forward fair-path and stable set operators. Since, x is a reachable state, we conclude $x \in U$.

To show $W \subseteq Fair+$, it suffices to show $U \subseteq Fair+$: since any $x \in W$ can reach U which can then reach a fair cycle, it follows that x can reach a fair cycle. Now, let $x \in U$. We will show $x \in Fair+$. Assume to the contrary that x cannot reach a fair cycle. Then, there are two cases:
Case 1. x cannot reach a cycle, in which case, x is deleted by the forward stable set operator.
Case 2. x can only reach non-fair cycles. Consider the subgraph reachable from x. Consider any one of the leaf SCC's (one which cannot reach any other SCC) of this subgraph. Call it C. Since C was not deleted by the forward fair-path operator, it must be that all positive edge constraints are satisfied, i.e. for each set of positive edges, one of the edges is included in C. Also for the i-th CFC, every state in C is either included in T_i or can reach some S_i. Since, states in C can only reach themselves, and since vertices in C only have edges to vertices in C, it follows that C is either included in T_i or contains some node of S_i. Hence, a cycle γ which goes through all nodes and edges of C satisfies the i-th fairness constraint. We conclude that γ satisfies all positive edge constraints and all CFCs (γ satisfies the negative fair edges constraints since all the negative fair edges were removed). Hence, γ is fair. But this is a contradiction to x not being able to reach any fair cycles (QED).

There is another operator which can be used in step 3 of the above algorithm to quickly eliminate portions of the state space in which no fair reachable states exist. Given a set of states S, the ***forward trim operator*** is defined by the following.

Perform the following until there are no more changes:

For each $F^{\infty}(S_i) + G^{\infty}(T_i)$, if $R_1\left(y, S \cap \overline{S}_i \cap \overline{T}_i\right) \subseteq S \cap \overline{S}_i \cap \overline{T}_i$, let $S = S \cap (T_i \cup S_i)$, where S is the current set of states.

The backward trim operator is defined by replacing $R_1\left(\overline{S}_i \cap \overline{T}_i, y\right)$ by $R_1\left(x, \overline{S}_i \cap \overline{T}_i\right)$. One can easily show that the trim operators does not delete any fair reachable states. In chapter 6, we extend the trim operator by analyzing the graph induced by fairness constraints. This analysis can result in deletion of portions of the state space where no fair behavior exists. These techniques are especially useful when the fair states are being computed as part of a language emptiness check.

Remark Computing the set $Fair$ when BDDs are used is more difficult than computing $Fair+$ since it involves a transitive closure computation rather than just reachability. One way to compute it is to modify the above algorithm as follows. Let $C(S)$ denote the set of all cyclic strongly connected components containing some state of S. For any set S, $C(S)$ can be computed easily from the transitive closure of the underlying graph. In the forward stable set and fair-path operators, replace the reachability computations of the form $R^*(x, S)$ by $C(S)$. By following the same line of reasoning as before, one can prove that the set U computed by the above algorithm in step 3 is the set $Fair$.

4.4 Fair CTL

We first define Computation Tree Logic (CTL) ([CES86]), and then introduce fair CTL ([EL85]), and give a symbolic model checking algorithm for this logic.

4.4.1 CTL

Definition Let ***AP*** denote a set of atomic propositions, i.e. a set of symbols. The ***syntax of CTL*** is described as follows.

1. Any atomic proposition P is a CTL formula.
2. If f and g are CTL formulas, so are the following: $\overline{f}$, $f + g$, EXf, $E[fUg]$, and EGf.

Definition A ***Kripke structure*** on AP is a finite directed node-labeled graph, where the node labels are subsets of AP (see also section 2.3.5). We assume that every state in a Kripke structure has some successor. If a state s is labeled by $P \in AP$, we say that P is true at s. *If* a node is not labeled by some $P \in AP$, it is assumed that $\bar{P}$ is true there. Intuitively, the atomic propositions are the outputs produced at the states.

Definition The semantics of a CTL formula φ is defined in terms of a state s in a Kripke structure S. We say φ is true at s if one of the following holds:

1. $\varphi = P$ for atomic proposition P, and P is true at s.

2. $\varphi = \bar{f}$ for CTL formula f, not satisfied at s.

3. $\varphi = f + g$ for CTL formulas f and g, such that either f or g is satisfied at s.

4. $\varphi = EXf$ for CTL formula f, such that there is some (infinite) path $\pi(0), \pi(1), \ldots$ starting at x, where f holds at $\pi(1)$.

5. $\varphi = E[fUg]$ for CTL formulas f and g, such that there exists a path π in S starting at s, where the first time g is true along π, f has been true for all previous times. Intuitively, this means that f is true until g becomes true. Note that g has to become true along π.

6. $\varphi = EGf$ for CTL formula f, such that there exists a path π starting at s, where f is true everywhere along π.

We say φ holds for S if it holds at some initial state.

For ease of readability, we introduce the following syntactic abbreviations, with their intended meanings.

a. $AXf \equiv \overline{EX\bar{f}}$, meaning f is truc at all successors of s.

b. $EFf \equiv E[true\,Uf]$, meaning there exists a path starting at s along which f finally becomes true.

c. $AFf \equiv \overline{EG\bar{f}}$ meaning for every path π starting at s, f is true at some state along π, i.e. it becomes true along π.

d. $AGf \equiv \overline{EF\bar{f}}$, meaning for every path π starting at s, f is true along all states of π, i.e. f is true at all states reachable from s.

e. $A[fUg] \equiv \overline{E[\bar{g}U\bar{f}\,\bar{g}]} \wedge \overline{EG\bar{g}}$ meaning that for all paths π starting at s, the first time g is true, f has been true for all previous times.

4.4.2 Fair CTL

Definition Let $F^{\infty}f$ (***infinitely often*** f) hold for a path π in S iff the CTL formula f holds for an infinite number of times along π. Let $G^{\infty}g$ (***almost everywhere*** g) hold for a path π iff after a finite amount of time the CTL formula g holds forever along π. These operators are sometimes called the ***infinitary operators***.

Assume a Kripke structure S, a formula φ of the type $F^{\infty}f$ (or $G^{\infty}f$) where f is a CTL formula, and a path π in S, are given. One can calculate the set of states in which f is true, create a new atomic proposition Q, and assign Q to all nodes in which f is true. Now $F^{\infty}f$ (or $G^{\infty}f$) is true of π iff $F^{\infty}Q$ (or $G^{\infty}Q$) is true of π. Hence, we assume that only atomic propositions are used in conjunction with the infinitary operators.

Definition A ***general fairness constraint*** is any Boolean combination of infinitary operators. A ***canonical fairness constraint (CFC)*** is a fairness constraint of the form $\sum_{i=1}^{m}\prod_{j=1}^{n}\left(F^{\infty}P_{ij}+G^{\infty}Q_{ij}\right)$, where P_{ij} and Q_{ij} are atomic propositions.

[EL85] showed that any general fairness constraint can be expressed as a CFC, but, the translation may be exponential; however, they argued that most practical fairness constraints can be represented efficiently using CFCs.

Definition A formula in the logic ***fair CTL*** is a pair (φ, Φ), where φ is a CTL formula and Φ is a canonical fairness constraint. The truth of (φ, Φ) is determined at a state s by interpreting the quantifiers only over those (infinite) paths which satisfy the fairness constraint Φ (i.e. the fair paths).

Example Let s be a reachable state such that atomic proposition P is true at s, but s is not fair reachable (i.e. s cannot reach a fair cycle). Then, the formula P holds at s, but EFP does not.

4.4.3 Fair CTL Model Checking

[EL85] introduced Fair CTL and gave an algorithm to model check fair CTL formulas. This algorithm relied on finding a ***fair strongly connected component*** of the graph, i.e. a SCC which completely contains a fair path. The first step in detection of fair SCC's finds all SCC's of the graph. Unfortunately, this operation is very time-consuming (on large graphs), even when BDDs are used ([HTKB92]). [BCMDH92] first presented a BDD-based algorithm for a subset of fair

CTL, where the fairness constraint is of the form $\prod_{i=1}^{n}\left(F^{\infty}P_i\right)$. In this paper, we present an algorithm for full fair CTL, which uses the fair reachable state computation of section 4.3 as a subroutine.

Remark To compute fair-states for a system S with fairness constraints of the form $\sum_{i=1}^{m}\prod_{j=1}^{n}\left(F^{\infty}P_{ij}+G^{\infty}Q_{ij}\right)$, we compute fair-states for each constraint $\prod_{i=1}^{n}\left(F^{\infty}P_j+G^{\infty}Q_j\right)$, and then take their union. So, for the remainder of this section, we assume the fairness constraints are of the form $\prod_{j=1}^{n}\left(F^{\infty}P_{ij}+G^{\infty}Q_{ij}\right)$.

To model check a fair CTL formula, we start from the leaves of the parse tree of the formula (which are atomic propositions), and compute the set of states satisfying each subformula. If some initial state is in the final set of states (i.e. at the root of the parse tree for the formula), then the original formula is satisfied. For example to compute the set of states satisfying $f+g$, we first compute the set of states satisfying f, the set of states satisfying g, and take their union. If an initial state is in this union, the formula is satisfied.

In order to present a MC algorithm for fair CTL, we need to present algorithms for six cases: AP P, $f+g$, $\bar{f}$, EGf, EXf, and $E[fUg]$. In general, f and g are CTL formulas, and we assume the sets of states satisfying f and g are known. Let F be the set of fair reachable states of the system ($Fair+$). The following algorithm performs the task. Let $S(f)$ denote the set of states satisfied by a CTL formula f. Let $T(x,y)$ denote the transition graph of the Kripke structure.

1. AP P. Return $S(P)$.

2. $f+g$. Return $S(f)\cup S(g)$.

3. $\bar{f}$. Return $\overline{S(f)}$.

4. EGf. Compute the set of fair reachable states, satisfying EGf, using the following algorithm.

4a. Restrict $T(x,y)$ to $S(f)$. Call the result $T'(x,y)$.

4b. Return the set computed by the fair reachable state computation using $T'(x,y)$ with the same fairness constraints.

5. EXf. Return the intersection of the set of states whose next states satisfy f, with F:

$$U(x) = S_1(x, S(f)\wedge F)$$

6. $E[fUg]$. Compute the set of fair-states satisfying $E[fUg]$, using the following fixed-point compu-

tation.

$$U_0 = S(g) \cap F$$
$$U_i(x) = U_{i-1}(x) + (S_1(x, U_{i-1}) \wedge S(f))$$

To prove the correctness of the above algorithm, we need one lemma.

Lemma 4.4.1 Assume a fair path π is given. Then, every state along π is a fair reachable state.

Proof The set of infinitely occurring states of the path satisfies the fairness constraints. For any state s along π, consider the path π', obtained by restricting π to start at s. The set of infinitely occurring states of π and π' are the same. Hence, π' satisfies the fairness constraints, and therefore s is a fair reachable state (QED).

Theorem 4.2 Our algorithm correctly model checks fair CTL.

Proof We prove the correctness of the last three operations.

1. EGf. Let s satisfy EGf. Then, there exists a path π starting at s, along which f is always true. If we restrict the graph to states where f is true, π is still a fair path of this new graph. Hence, the algorithm returns all states s which satisfy EGf. To see that it does not return any other state, note that it returns fair reachable states of a graph, where f holds everywhere. Hence, all fair paths it considers satisfy EGf.

2. EXf. If s satisfies EXf, then there exists a fair path starting at s whose next state t satisfies f. By the above lemma, $t \in F$. Conversely, if s is such that for one of its next states t, $t \in F$ and $t \in S(f)$, then s satisfies EXf. So, states whose next states are in $S(f) \cap F$ are exactly the states satisfying EXf.

3. $E[fUg]$. Let s satisfy $E[fUg]$. Then, there is a path from s to a fair reachable state t such that g holds at t, and f holds at the states between s and t along the path. Therefore, the above computation includes s, since $t \in S(g) \cap F$, and s reaches t along a path on which f is always true. Conversely, any state returned by the computation satisfies $E[fUg]$ (QED).

Remark All of the steps of the above algorithm can be efficiently implemented using BDDs.

4.5 Language Containment

In this section, we are concerned with ω-automata specifications. Using language containment for formal verification, one represents a super-set of the desirable behaviors of the system using an ω-automaton T. One can think of T as specifying acceptable patterns for the traces of the system. An example is: only those traces are acceptable where each request is eventually followed by a service. This property can be specified by a two-state automaton. One then checks that the language of the system, which is represented by an ω-automaton S, is included in the language of the property automaton T, $L(S) \subseteq L(T)$. Doing so, the user is guaranteed that the system is incapable of producing traces which do not have the desired pattern. This is called verifying a system with respect to a property. This verification continues until the user is convinced that the intersection of the languages of the properties is equal (or maybe very close) to the desired set of behaviors.

The above scheme first appeared in [VW86], who suggested specifying both the system and the property using Buchi automata, and gavc an algorithm for language containment in this environment. The usual method to verify that $L(S) \subseteq L(T)$ is by checking that $L(S) \cap \overline{L(T)}$ is empty (known as ***language emptiness check***). However, complementing an ω-automaton is a PSPACE-complete problem, and the best known algorithms have exponential complexity ([SVW87], [Saf89]).

[Kur87b] introduced an environment, where the acceptance conditions of the system and the property are complementary. The system is modeled by an L-process, whereas the property is modeled by a deterministic L-automaton. An L-process is a process (section 2.3.2) with the fairness constraints being negative fair subsets and edges. The negative fair subsets are called ***cycle sets***, whereas the negative fair edges are called ***recur edges***. A run is accepting if 1) it does not traverse some recur edge ∞-often, and 2) its set of infinitely occurring states is not contained in any cycle set. An L-automaton is syntactically the same, except that a run is accepting if some recur edge is traversed ∞-oftcn, or if the set of infinitely occurring states is contained in some cycle set. We rcfer to this method for vcrification as the ***L-environment***. Computing the complement of the language of a property is trivial: since the acceptance conditions of L-processes and L-automata are complementary, one can just think of the L-automaton as an L-process. Based on this paradigm, the first software tool for automatic verification of finite-state systems using language containment was built ([HK90]). The advent of BDDs allows for handling very large state spaces.

In this section, we introduce a new language containment based formal verification environment, which contains the L-environment as a subset. The advantage of this environment is that specifications can be more compact, however, the algorithms remain as efficient as the algorithms for the L-environment. Specifically, if the specifications come from the L-environment, the algorithms reduce to those for the L-environment ([HTKB92]).

In this environment, the system is modeled using edge-Streett automata, whereas the properties are modeled using edge-Rabin automata. We call this environment the ***eSeR-environment***. Edge-Rabin automata have two useful properties. First, the set of deterministic edge-Rabin automata accept the whole set of ω-regular languages[1]. In general, if an automaton is deterministic, complementation is much easier[2]. The second property is that deterministic edge-Rabin automata can be compactly complemented into edge-Streett automata. Hence, the language containment check is reduced to checking that the language of an edge-Streett automaton, formed by taking the product of the complement of the property and the system, is empty. We first define edge-Rabin automata, and show how one can complement a deterministic edge-Rabin automaton into an edge-Streett automaton. We then present the language emptiness check for edge-Streett automata. We conclude by discussing some expressiveness results about our environment.

4.5.1 Edge-Rabin Automata

Definition An ***edge-Rabin automaton*** R is an automaton A augmented with a set of fairness constraints, each of the following form:

1. Positive fair edge constraint, which is an edge of A.
2. Negative fair edges constraint, which is a set of edges of A.
3. Canonical fairness constraint (CFC), which is of the form $F^{\infty}(S) \wedge G^{\infty}(T)$, where S and T are sets of states.

An ω-string x of A is accepted ($x \in L(A)$) if there is a run r of x in A such that at least one of the fairness constraints is satisfied, i.e.

1. r traverses some positive fair edge ∞-often, or
2. for some set of negative fair edges E, r does not traverse any of the edges in E ∞-often, or
3. for some CFC $F^{\infty}(S) \wedge G^{\infty}(T)$, $inf(r) \cap S_i \neq \emptyset$ and $inf(r) \subseteq T_i$.

Lemma 4.5.1 For every deterministic edge-Rabin automaton R, there is a deterministic edge-Streett automaton S with the same transition structure and no more fairness constraints, which accepts the complement of the language of R.

1. Deterministic L-automata cannot express all ω-regular languages, however, one can define any ω-regular language by taking an intersection of a finite (possibly large) number of languages expressed by deterministic L-automata ([Kur87b]).
2. [CDK93] gives the complexity of language containment between various types of automata when the specification automata are deterministic.

Proof For the set of all positive edge constraints in R, create one set of negative edges in S. For each negative fair edges constraint in R, create a set of positive fair edges in S. For each $F^{\infty}(S_i) \wedge G^{\infty}(T_i)$, create a constraint $G^{\infty}\left(\overline{S_i}\right) + F^{\infty}\left(\overline{T_i}\right)$. Let $x \in L(S)$. Since S and R are both deterministic, x has a unique run r in both S and R, which is rejected in R, i.e. $x \notin L(R)$. Thus, $L(S) \subseteq \overline{L(T)}$. One can similarly show that if $x \in L(R)$ then $x \notin L(S)$ (QED).

Remark When BDDs are used for representing sets of states (like S_i and T_i), computing the complement of an edge-Rabin automaton by the above procedure is trivial, since complementing a BDD can be done in constant time.

4.5.2 Language Emptiness Check for Edge-Streett Automata

The language of an edge-Streett automaton is empty iff there are no fair paths. Hence, one way to check for language emptiness is to use the fair reachable states computation: this returns a non-empty set precisely when the language of the automaton is not empty. In language containment, we are only interested in the set of fair states (denoted *Fair*, section 4.3). However, the fair reachable state computation returns a super-set of the fair states. We now define the set *Fair*± which is a better approximation to *Fair* than *Fair+*.

Definition Let *Fair*± denote the set of states which can reach some fair cycle and can be reached from some fair cycle. We have $Fair \subseteq Fair\pm \subseteq Fair+$. Figure 4.5 shows the difference between these sets.

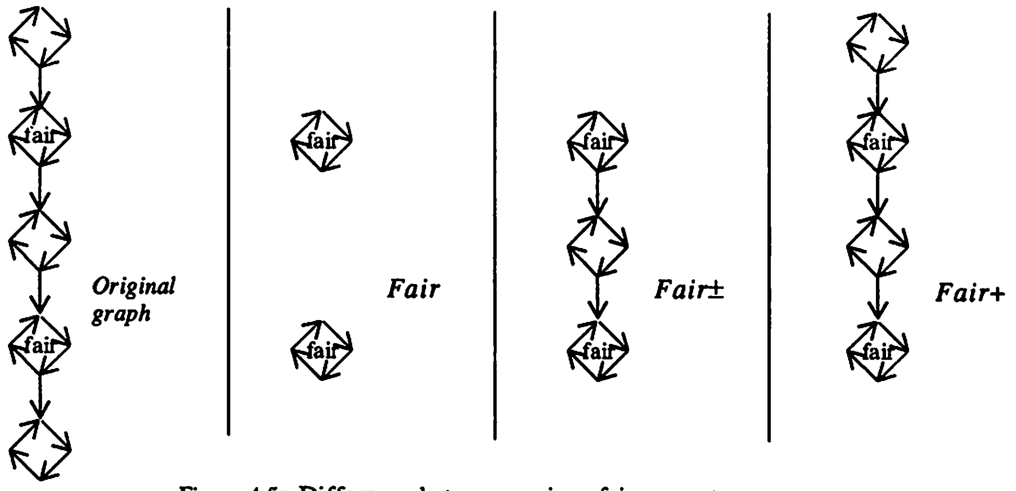


Figure 4.5: Difference between various fairness sets

The following algorithm computes a subset of *Fair*±, containing *Fair* (the proof is similar to the proof of the correctness of the algorithm for *Fair+* computation).

1. *Remove negative fair edges.*
2. *Start with reachable states.*
3. *Repeat the following operators until convergence:*
    - 3.1.a *Apply the forward fair-path.*
    - 3.1.b *Apply the forward stable set.*
    - 3.1.c *Apply the forward trim.*
    - 3.2.a *Apply the backward fair-path.*
    - 3.2.b *Apply the backward stable set.*
    - 3.2.c *Apply the backward trim.*

Step 3 of the above algorithm is known as the ***main computation***. In a later section, we will modify this algorithm by checking for easy failures.

4.5.3 Expressiveness of Edge-Streett Automata

In this subsection, we present two expressiveness results. First, we discuss why the eSeR-environment enjoys a sense of maximality. Second, we compare the compactness of specifications in the eSeR-environment to those in the L-environment. The heart of the language containment check is an emptiness check for edge-Streett automata. One may ask if it is possible to make the environment even more powerful without sacrificing too much efficiency (the language emptiness check for edge-Streett automata can be done in small polynomial time). In some sense, the next most natural extension is to allow acceptance constraints of the form of a product of sums of an arbitrary number of F^{∞}'s and G^{∞}'s. Since the sum of a set of $F^{\infty}(S_i)$'s is just equivalent to $F^{\infty}\left(\bigcup_i S_i\right)$, only one F^{∞} suffices. It remains to answer whether allowing more G^{∞}'s will sacrifice efficiency. [EL85] basically answered this question (they were looking at a slightly different problem).

Theorem 4.3 The problem of determining whether the language of an automaton with acceptance conditions of the form $\prod_i \left(G^{\infty}(S_i) + G^{\infty}(T_i)\right)$ is empty, is NP-complete (in the number of states and fairness constraints).

Proof (sketch) The problem is in NP. The answer is a set of states which can be traversed infinitely often, such that all fairness constraints are satisfied. Just guess this set of states, and check that this set is contained in S_i or T_i for all i. To prove the problem is NP-complete, just use the reduction of [EL85], which was employed to show the fair state problem with fairness constraints of the above type is NP-complete. [EL85] reduces 3SAT to a graph with fairness constraints of the above type. Now, the language of the automaton is non-empty iff the original formula is satisfiable (QED).

Since the language emptiness check for Streett automata is done in polynomial-time, and the next natural extension makes the language emptiness check NP-complete, the eSeR-environment enjoys a sense of maximality.

Remark [HSB94] discusses several compactness issues of the eSeR-environment. The most important result is that translating an edge-Street automaton into an L-process is exponential in the worst case. To facilitate Streett automata in the L-environment, [Kur94] translates each Streett fairness constraint into an L-process. Checking the language emptiness of the original Streett automaton now reduces to checking the language emptiness of the intersection of the original transition structure and the new L-processes. The regular emptiness check is exponential in this case. However, [Kur94] presents a polynomial emptiness algorithm (mimicking our emptiness algorithm for edge-Streett automata), which takes advantage of the special structure of this problem.

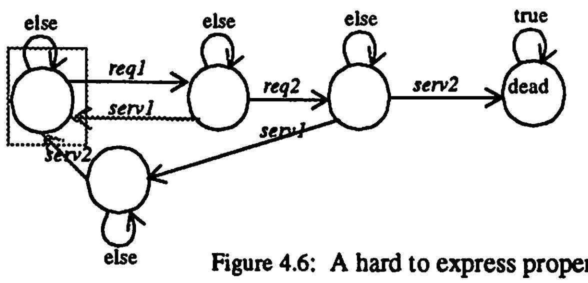
4.6 Complementary Nature of MC and LC

In this section, we give a classification of properties, which constitutes a different way of looking at property specification methods, such as CTL and edge-Rabin automata. We argue that, using this view, LC and MC are somewhat complementary.

4.6.1 A Property Classification

The properties one might want to verify about a system can be divided into three sets: string properties, language properties, and system properties.

1. ***String Properties.*** These properties must be satisfied by every trace of the system, i.e. every ω-string in the language of the system. An example is: for every trace of the system, if request 1 is pending when request 2 is generated, then request 1 is serviced before request 2. The edge-Rabin automaton recognizing this property is given in figure 4.6.



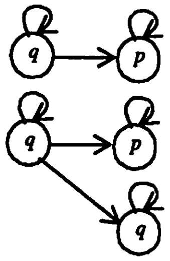
"Else" means under the remaining conditions, "true" means always. The highlighted edges denote positive fair edges and the highlighted box a positive fair subset.

Figure 4.6: A hard to express property for CTL

Much is known about the hierarchy of specification languages for traces. At the highest level is the set of ω-regular languages, which is recognized by non-deterministic Buchi automata as well as deterministic Rabin automata ([Cho74]). ***Linear Temporal Logic (LTL)*** is another method to specify string properties, which is properly contained in ω-regular languages ([Wol83] shows that the property p is true at every even state is not expressible in LTL). [Eme90] gives a survey of such expressiveness results.

2. ***Language Properties***. These are properties of the languages, not expressible as string properties. For example, one might wish to verify that if strings x and y are in the language then string z is also in the language. Consider the CTL property $P = AGEFp$, which says for all reachable states s, there exists a path from s to some state t where atomic proposition p holds. Any system having the above property P has the following language property Q: Q is the set of languages L such that if $yz \in L$, where y is a finite string and z is an infinite one, then there exists a string w, with p occurring in w, such that $yw \in L$.

3. ***System Properties***. These properties distinguish between transition structures, possibly having the same language. An example is the above CTL property $P = AGEFp$, which distinguishes between the systems shown in figure 4.7, having the same language.



The CTL property *AGEFp* holds for the first system, but it does not hold for the second. Note that both systems have the same language. Hence, CTL has the power to distinguish between different implementations of the same behavior.

Figure 4.7: Ability of CTL to recognize structures

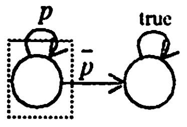
4.6.2 MC versus LC

To compare MC and LC, we first need to define expressiveness of a specification language.

Definition A ***specification language (P, A)*** is a set P (usually infinite) of finite strings over some alphabet, and an algorithm A to decide whether a system S satisfies some property $p \in P$. An example is the set of all CTL formulas as P, and a CTL model checking algorithm as A. In this case, the alphabet is the set of all outputs of all systems, plus symbols used in CTL formulas. We assume that systems are given in the C/S model. Other common finite state models can be used instead. Let $A(S, p)$ be the result returned by algorithm A on system S on property p.

Definition Given two specification languages (P_1, A_1) and (P_2, A_2), we say P_2 is at least as ***expressive*** as P_1, if $\forall p_1 \in P_1$, $\exists p_2 \in P_2$ such that $\forall S (A_1(p_1, S) = A_2(p_2, S))$. The intuition is that given a property p_1 in P_1, we can come up with a corresponding property p_2 in P_2, such that for any system S, p_1 holds of S iff p_2 holds of S, i.e. p_2 is true on exactly the same set of systems S that p_1 is true..

An important subclass of trace properties, called invariance properties, is expressible in both MC and LC. ***Invariance properties*** are of the type "no bad event happens in any reachable state of the system." Invariance properties are expressed by formulas of type AGp in CTL (p may be a Boolean formula of atomic propositions) or as edge-Rabin automata of the type shown in figure 4.8.



This edge-Rabin automaton expresses the invariance property that in all states p is true. If we ever visit the second state, we will stay there forever, and the resulting string is rejected.

Figure 4.8: Automata for expressing invariances

Moreover, some simple ***liveness properties***, i.e. properties of the form "some good event will happen in the future", are also expressible by both methods. However, there are string properties expressible by ω-automata, but not expressible by CTL. An example ([Wol83]) is the property "at all even times p holds." There are other properties, usually involving sequencing of events, which are very hard or impossible to express in CTL. An example is "if $req1$ happened before $req2$, then $serv1$ should happen before $serv2$ " (figure 4.6). Properties depending on sequencing of events occur often in practice. Examples are memory systems and pipeline computers. It is our experience that even when a property involving sequences of events is expressible in both CTL and edge-Rabin automata, it is harder to find the correct CTL formula for it.

On the other hand, no language or system properties are expressible by ω-automata. In general, properties involving the existential quantifier of CTL are not expressible by ω-automata. These properties are significant, because sometimes we want to ensure that a certain behavior is possible, e.g. it is always possible to reach a safe state from any state of the system. The CTL formula $AGEFp$, which involves existential quantification, expresses this property.

We believe LC and MC provide different, useful capabilities to the user, and therefore, are somewhat complementary. By combining the two environments, we gain more expressive power and ease of expression; a larger set of properties can be more easily specified and verified. It remains to determine how much of an overlap there is in the useful expressive powers of CTL and ω-regular languages.

Remark It may also be the case that debugging with CTL properties is easier, since the debugger unfolds the formula one operator at a time. Language containment, however, gives all the debugging information back to the user in one bunch. Hence, at every point in debugging CTL formulas, less information is fed back to the user. Another possible advantage is that with CTL, one can do one reached state computation and then check any number of formulas. Finally, one can always directly write the property as part of the system by forming a monitor which represents the complement property. Then model checking $EFtrue$ is the same as language emptiness. However, this requires the user to explicitly complement the property automaton. Also, it may not be possible to take advantage of all optimizations tailored for language containment in this setting (see chapter 6).

4.7 Early Failure Detection

We present techniques for early failure detection (***EFD***). We first present a classification of errors for LC, and show how each can be computed. In most cases where a property fails, it is not necessary to compute the full set of reachable states to find the error. Indeed, since the notion of an easy error in many cases corresponds to an error which is reproducible in a few steps, we expect to compute only a few reachability steps before the error is found. We combine this idea with a classification of fair cycles to give an algorithm for EFD during LC. The idea of avoiding the full reachability set is used then to give a EFD algorithm for ACTL, i.e. CTL formulas involving only universal quantification, and with complementation applied only to atomic propositions.

4.7.1 Early Failure Detection for LC

In LC, a failure is an ω-string accepted by an edge-Streett automaton. Given the transition relation with negative fair edges removed, and a subset R of the reachable states, we define the following set of fair cycles of R. Let $S = \bigcap_i S_i$, where the S_i's are the cycle sets..

1. ***Cycles of the first kind.*** Cycles entirely contained in S, and satisfying the positive edge constraints. Any such cycle is fair.
2. ***Cycles of the second kind.*** Cycles not of the first kind but intersecting S, and satisfying the positive edge constraints.
3. ***Cycles of the third kind.*** Any fair cycle not of the first two kinds.

To find the cycles of the first kind, run the stable set operator on S. If the resulting set is not empty, run the main computation (given in subsection 4.5.2) on this set with the positive edge constraints[1]. To find cycles of the second kind, run the main computation on R, with the fairness constraints being the positive edge constraints, and a constraint of the form $F^{\infty}(S)$. If a non-empty set is returned, some fair behavior exists.

Theorem 4.4 If there are no cycles of the first or second kind, the main computation can be restricted to $\bar{S} \cap R$.

Proof We can delete the states in S from consideration, because there are no cycles of the first or second kind, and hence there cannot be any fair cycles involving states in S (QED).

To avoid computing the full reachability set, as we compute the set of reachable states, we can perform a cheap test which may find some errors. The following is an implementation of this idea. We start with the current set equal to the set I of initial states.

1. Using breadth-first search (BFS), compute the set of states R_D, reachable in D or fewer steps from the current set.

2. Check if there are any cycles of the first kind in R_D. If so, stop.

3. If none is found, set the current set equal to R_D, go to step 1, continuing the BFS for another D steps

1. One can similarly check for cycles entirely contained in $\bigcap_i T_i$, since any such cycle is fair, and these cycles can be easily detected using stable set operators.

or until all reachable states are computed.

Intuitively, the algorithm takes a few BFS reachability steps (D steps), and then checks whether there are any (easily found) fair cycles of the first kind in this set. Since easy errors are almost always of the first kind, and since checking for these is less computationally expensive than checking cycles of the second kind, we only invoke EFD here with cycles of the first kind (although obviously other fair cycles could be checked as well). The process continues until an error is found, or the full set of reachable states is computed.

4.7.2 Early Failure Detection in Fair CTL

We do not know how to extend EFD techniques to general CTL formulas. However, the technique described in subsection 4.7.1 can be applied to ACTL formulas, where the idea is to take a few reachability steps, and then check whether the formula is satisfied. The trouble with formulas involving existential quantification is that the reachable states not yet visited may be "witnesses" for the existential quantification. This problem does not arise with ACTL formulas.

Definition A CTL formula is in ***positive normal form*** (PNF) if all negations are applied to atomic propositions, and the only Boolean operators used are "AND" and "OR". Any CTL formula can be written in this form.

Definition A CTL formula is a ***for all CTL (ACTL)*** formula if when written in PNF it does not involve existential quantification.

Lemma 4.7.1 Let S be a subset of the reachable states R. Let ϕ be an ACTL formula, interpreted over CFC Φ. Let $s \in S$. If ϕ does not hold at s (over Φ) in S, then ϕ does not hold at s (over Φ) in R.

Proof We use induction over the size of CTL formulas. Assume ϕ does not hold at s in S. Note that ϕ is interpreted over fair paths, and the fair paths in S are a subset of the fair paths in R.

1. $\phi = P$, P some proposition. Since P is not a label of s, ϕ does not hold in S, and hence not in R either.

2. $\phi = f + g$, where f and g are CTL formulas. By the same reasoning as in case 1, neither f nor g hold at s in S. By the inductive assumption, neither f nor g hold of s in R. The conclusion follows.

3. $\phi = f \wedge g$. Similar to case 2.

4. $\phi = AGf$. Since ϕ does not hold in S, there is some state t reachable from s along a fair path π, such that f does not hold at t. By the inductive assumption, f does not hold of t in R. Since π is a fair path in R, ϕ does not hold of s in R.

5. AXf. Proof is similar to case 4.

6. $AfUg$. Proof is similar to case 4 (QED).

Since we do not have any way of quickly determining whether an ACTL formula is not satisfied, the check should be applied more sparingly than for edge-Rabin automata. The problem of extending quick failure checks to ACTL formulas, analogous to the cycles of the first kind, remains open.

4.8 Hierarchical Verification

The design process can be done hierarchically. For large designs, this may be the only way to verify it. The designer may first abstract the design, and prove some properties about it. Next, the designer may add more detail. For example, a module may be implemented by three more detailed ones. By definition, a subsystem implements another if the behavior of the implementation is contained in the behavior of the specification at a higher level. This means that the language of the lower level implementation must be contained in the language of the higher level specification.

Since systems are specified using edge-Streett automata, this requires checking language containment between two edge-Streett automata. [Kur92] presents an algorithm for language containment of two L-processes (this comes up in the L-environment because of his use of the homomorphic reductions). We present an algorithm with the same flavor for our problem. Assume two edge-Streett automata A and B are given, where B is deterministic. Using the following two lemmas, the problem of containment of $L(A)$ in $L(B)$ is reduced to a finite set of language containments between edge-Streett and edge-Rabin automata.

Lemma 4.8.1 Let A and B be edge-automata with B being deterministic. $L(A) \subseteq L(B)$ iff $L(A) \subseteq L(B_i)$, for all $1 \leq i \leq n$, where n is the number of fairness constraints of B, and each B_i has the same transition structure of B, but only one of the fairness constraints of B (the set of negative edges is considered as one fairness constraint, whereas each positive edge and each constraint $F^{\infty}(S_i) + G^{\infty}(T_i)$ is a separate fairness constraint).

Proof (=>) Assume $L(A) \subseteq L(B)$. Let x be a trace of A. We have to show x is a trace of each B_i. Assume x is not a possible trace of some B_i. Then, that fairness constraint is not satisfied on x. Hence, x is not a trace of B. So, $L(A) \not\subset L(B)$, which is a contradiction.

(<=) Assume to the contrary that $L(A) \not\subset L(B)$. Let x be a string in $L(A)$ and not in $L(B)$. Then, some fairness constraint of B is not satisfied on x. Let it be constraint j. Then, x is not a trace of B_i (QED).

Lemma 4.8.2 If a deterministic edge-Streett automaton S has only one fairness constraint, then there exists a deterministic edge-Rabin automaton, with the same transition structure and at most two fairness constraints which accepts the same language.

Proof We have to show the transformation for each kind of fairness constraint. We have three cases:

1. A negative fair edge. Create an edge-Rabin automaton with the edge being a negative fair edge constraint.
2. A set of positive fair edges. Create an edge-Rabin automaton with the edges being a positive fair edges constraint.
3. A constraint of the form $F^{\infty}(S_i) + G^{\infty}(T_i)$. Create an edge-Rabin automaton with two CFCs: $F^{\infty}(True) \wedge G^{\infty}(T_i)$ and $F^{\infty}(S_i) \wedge G^{\infty}(True)$ (QED).

Note that the algorithm for language containment between two edge-Streett automata is rather efficient. For checking $L(A) \subseteq L(B)$, we have to solve n problems (n being the number of constraints of B), each of which can be solved very efficiently. Since, these fairness constraint are given by the user, we expect n to be small.

Remark One can also use the algorithm reported in [Saf92] for the above task. Given a deterministic Streett automaton with n states and h CFCs, [Saf92]'s algorithm returns a deterministic Rabin automaton with $n2^{h\log h}$ states and $h+1$ CFCs accepting the same language. Although this algorithm is exponential in h, if h is small, as it is expected to be in practice, this algorithm may be practical.

If B is non-deterministic, then the language containment check is more expensive. [Saf92] proposed an (exponential) algorithm for determinizing a non-deterministic Streett automaton into a deterministic Rabin automaton. One can translate an edge-Streett automaton into a Streett automaton, using techniques similar to the node-recur transform of [Kur87a]. Hence, the problem of testing for language inclusion of an edge-Streett automaton into another edge-Streett automaton can be reduced to testing the language inclusion of an edge-Streett automaton into a deterministic Rabin automaton, for which we have presented algorithms. The requirement that B be a non-deterministic automaton may not be too severe since it appears that in practice many specifications may be effectively deterministic, i.e. have deterministic transition structures, since Moore machines with non-deterministic outputs and deterministic transitions (as is the case with L-processes in [Kur92]) can be used for the necessary abstractions in many cases. In those cases, where the finite state machine is viewed as an automaton the output non-determinism is simply seen as another input which can take any value. For cases where the associated automaton is non-deterministic, [THB95] presents semi-implicit implementation of determinization procedures for ω-automata.

Acknowledgment

This material presented in this chapter is based mainly on [HB95a].

Chapter 5

Generating Short Error Traces

5.1 Introduction

An important part of any verification system is generating error traces. To make it easier for the user to locate the bug, the error trace should be short. In general, a short error trace is easier to understand and follow. In this chapter, we study how short error traces can be generated for both language containment and fair CTL model checking. We first build a debugging environment for generating short error traces for language containment, and then use this environment and additional ideas to generate short error traces for fair CTL model checking. Since, as we will prove, the problem is NP-hard in both cases, heuristics are needed. However, all of our algorithms are based on BDDs, hence, they are capable of handling large state spaces.

An overview of the algorithms to come is as follows. An error trace in LC consists of two parts: a path from an initial state to some fair state x, and a fair cycle starting at x. We present BDD-based algorithms which guarantee that the initial path of the error trace is minimum among all error traces. Then, a heuristic algorithm is given which returns a short fair cycle starting at x. This problem is NP-complete, and it is the source of why debugging for LC is NP-hard. As for CTL, we unwind the CTL formula one operator at a time, giving debugging information about the topmost operator. This has the advantage of giving the debugging information to the user incrementally. For ACTL formulas the debugger can return the debug structure automatically, whereas when existential quantification is used, the user has to interact with the debugger to find the source of error. Recall that the quantifiers in the CTL formulas are interpreted over the fair paths.

The flow of this chapter is as follows. In section 5.2, we present some BDD-based utilities. In section 5.3, the debugging problem for LC is proven NP-complete and heuristic algorithms for it are given. Section 5.4 describes techniques for debugging of fair CTL formulas, using the algorithm of section 5.3. A comparisons between our algorithms and those of [CGMZ95] is given in section 5.5. Almost all of the algorithms presented in sections 5.3 and 5.4 have been implemented in HSIS.

5.2 Utilities

In this section, we present BDD-based algorithms for finding the strongly connected components (SCC) of a vertex, and the shortest path between two sets of states. The definitions for cyclic and acyclic SCC's were given in section 4.3. In what follows, we freely interchange set operations (such as intersection and union) with Boolean operations (such as AND and OR) on the BDDs representing these sets.

5.2.1 Finding The SCC of A State

Lemma 5.2.1 Let s be a state. Let $R^*(s, y)$ denote the set of states reached by s, whereas $R^*(x, s)$ is the set of states which can reach s.

a. The SCC of s is computed by the formula $SCC_s = R^*(s, y) \wedge R^*(x, s)$

b. The computation $C_s = R^*(R_1(s, y), y) \wedge R^*(x, s)$ returns the empty set if s is an ASCC. Otherwise, it returns the CSCC of s.

Proof Part a follows from the definition of a SCC. Now, assume that s is an ASCC. Then s is not reachable from $R_1(s, y)$, i.e. the set of states reachable from s in one step. Hence $R^*(R_1(s, y), y)$ and $R^*(x, s)$ cannot have a point in common, implying $C_s = \emptyset$. Now, assume s belongs to some CSCC S. Then, all states in S are forward reachable from $R_1(s, y)$ and backward reachable from s. Hence, $CSCC_s \subseteq C_s$. Conversely, all states in S are reached by s and can reach s. Thus, $C_s \subseteq CSCC_s$. Hence, $CSCC_s = C_s$ (QED).

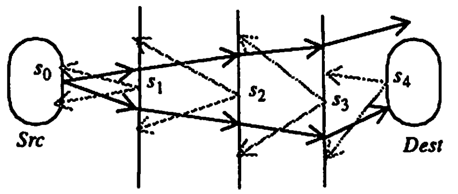
5.2.2 Finding A Shortest Path Between Two Sets

Let Src and $Dest$ be two subsets of vertices V. We want to find a shortest path from Src to $Dest$. The following algorithm returns a path $(s_0, \ldots, s_n)$, where $s_0 \in Src$ and $s_n \in Dest$, and n is minimum. The algorithm works in two stages. In the first stage, using breadth-first search, all states distance $1, 2, \ldots$ from Src are generated until $Dest$ is reached. This search mechanism guarantees that the path length is shortest. During the second pass, one such shortest path from

Src to *Dest* is found, by searching backward from *Dest* until *Src* is reached.

```
1 Let $i = 0, F_0 = src$.
2 While $F_i \cap Dest = \emptyset$,
      $F_{i+1} = R_1(F_i, y)$, and $i = i + 1$
3 Let $n = i$ and $s_n \in F_n \cap Dest$.
4 For $j = n - 1$ down to 0,
      Choose $s_j$ arbitrarily from $F_j \cap R_1(x, s_{j+1})$,
5 Return $(s_0, s_1, \ldots, s_n)$.
```

This algorithm is applied in several places in the debugger, and is a useful utility. The following figure depicts how the algorithm works.



During the forward search, the set of reachable states are computed until Dest is reached. Then, the backward search starts by picking a state in the intersection of the reachable states and Dest. One backward reachability step is performed, the intersection with reachable states in that level is taken, and an arbitrary state is chosen. The process continues until Src is reached.

Figure 5.1: Finding a shortest path between two sets

5.3 Debugging Algorithms for Language Containment

In this section, we present BDD-based algorithms for finding an error trace for LC. The first BDD-based algorithm for this problem was presented in [Tou91] for the L-environment, and was implemented in [Ste92]. [Tou91]'s algorithm depended on having the transitive closure of the transition relation. However, as [HTKB92] points out the transitive closure computation is expensive. Here, we present a debugger for the eSeR-environment, which contains the L-environment as a subset. Our debugger needs very little pre-processed information, i.e. only a set B containing a subset of fair states. The algorithm consists of two parts: finding a fair state x closest to an initial state, and returning a short fair cycle starting at x. In this section, we first show how a path to a fair state is found. The algorithm for finding a short fair cycle is given first for the L-environment, and then for the eSeR-environment.

5.3.1 Finding a Path to a Fair State

The algorithm starts from the set of initial states, and computes the set of reachable states $\hat{R}$ until the set B (assumed to contain some fair states) is reached. Then, it arbitrarily chooses a state x in $\hat{R} \cap B$, and calls the subroutine *is-Fair* to determine whether x is fair. If x is fair, *is-Fair* returns a fair connected component containing x. A ***fair connected component*** (***FCC***) is a connected component (where there is a path between any two states) where negative fair edges are removed, and where a cycle traversing all its vertices and edges satisfies all fairness constraints. If x is not fair, the subroutine *is-Fair* deletes from $\hat{B}$ a set of unfair states including x. This process of choosing a state, checking whether it is fair, and deleting some unfair states, continues until a fair state is found or $\hat{R} \cap \hat{B} = \emptyset$. In the latter case, more reachability steps are performed until $\hat{R} \cap \hat{B}$ is non-empty again, in which case we fall back into the loop of choosing a state arbitrarily and checking whether it is fair. The algorithm is as follows.

Path Finding Algorithm

```
1 Let $\hat{R}$ be the set of initial states and $\hat{B} = B$.
2 While a fair state has not been found
    2.1 While $\hat{R} \cap \hat{B} \neq \emptyset$
        2.1.1 Let $x \in \hat{R} \cap \hat{B}$.
        2.1.2 Let $C = \text{is-Fair}(\hat{B}, x)$. If $C \neq \emptyset$, return $x$ and $C$.
    2.2 Let $\hat{R} = R_1(\hat{R}, y)$, and go back to 2.1.
```

The subroutine $\textit{is-Fair}(\hat{B}, x)$ consists of a main loop which computes the fair connected component of x with respect to the CFC's (i.e. $F^{\infty}(S_i) + G^{\infty}(T_i)$'s). The main loop computes W, the SCC of x in the current set of states *cur-R*, where *cur-R* is initially set to be the set of all reachable states. It then calls $\textit{get-Fair}(W, \text{CFC's})$ to get the fair subset of W with respect to the CFC's, *fair-W*. If $\textit{fair-W} \neq W$, the states in $W - \textit{fair-W}$ are removed from $\hat{B}$. If x was not deleted, i.e. $x \in \textit{fair-W}$, the main loop is repeated with $\textit{cur-R} = \textit{fair-W}$. If $W = \textit{fair-W}$, W satisfies the CFC's. It is then checked that the positive fair edges constraints can also be satisfied in W. If so, W is returned. Otherwise it is removed from $\hat{B}$. The algorithm is as follows.

```
is-Fair ($\hat{B}, x$)
  0 Let $W$ = cur-R = $R$, where $R$ is the set of reachable states. Consider the transition relation $\hat{T}$ with
    the negative fair edges removed.
  1 While done is false
    1.1 Let $W$ be the SCC of $x$ in cur-R using the transition relation $\hat{T}$.
    1.2 If $W$ is a single state with no self-loops, let $\hat{B} = \hat{B} - x$, and return $\emptyset$.
    1.3 Otherwise, let fair-W = get-Fair(W, CFC's), cur-R = fair-W, and $\hat{B} = \hat{B} -$ (W - fair-W).
    1.4 If $x \notin$ fair-W, return $\emptyset$.
        else if W = fair-W, set done to true.
        else go back to 1.1.
  2 If each positive fair edges constraint $E_j$ is satisfied in $W$, i.e. some edge in $E_j$ is present in $\hat{T}$
    restricted to $W$, return $W$. Otherwise, let $\hat{B} = \hat{B} - W$, and return $\emptyset$.
```

The idea for computing the fair subset of W is to check, for each CFC $F^{\infty}(S_i) + G^{\infty}(T_i)$, whether W intersects S_i. If not, W is restricted to T_i, and the process continues until all CFC's can be satisfied. The CFC's which cause such a restriction can be marked as inactive, since further restrictions will not affect the satisfaction of these CFC's. The algorithm is as follows.

```
get-Fair(W, CFC's)
  0 Mark all CFC's as active. Set cur-W = W.
  1 For each CFC $F^{\infty}(S_i) + G^{\infty}(T_i)$
    if cur-W $\cap S_i = \emptyset$
      1.1 Let cur-W = cur-W $\cap T_i$, and mark the i-th CFC as inactive.
      1.2 For all active $F^{\infty}(S_j) + G^{\infty}(T_j)$ 's processed so far,
        1.2.1 if cur-W $\cap S_j = \emptyset$, let cur-W = cur-W $\cap T_j$, and mark the j-th CFC as inactive.
  2 Return cur-W.
```

Definition A ***fair set*** U with respect to the CFC's is a set of states which satisfies all CFC's, i.e. for each $F^{\infty}(S_i) + G^{\infty}(T_i)$, either $U \cap S_i \neq \emptyset$ or $U \subseteq T_i$.

Lemma 5.3.1 For a set W, $S = get\text{-}Fair(W, \text{CFC's})$ is the largest fair subset of W.

Proof The algorithm maintains the invariance that for all CFC's $F^{\infty}(S_i) + G^{\infty}(T_i)$ processed so far, either $cur\text{-}W \cap S_i \neq \emptyset$ or $cur\text{-}W \subseteq T_i$. Therefore, $cur\text{-}W$ satisfies all CFC's at the end, and hence is a fair subset of W. We show maximality of S by showing that for any fair set $V \subseteq W$, the algorithm maintains the invariance $V \subseteq cur\text{-}W$ at all times. Since initially $cur\text{-}W = W$, $V \subseteq cur\text{-}W$ holds initially. Now assume $V \subseteq cur\text{-}W$ before step 1.1. Since $cur\text{-}W \cap S_i = \emptyset$, we have $V \cap S_i = \emptyset$. Therefore, by V being a fair set $V \subseteq T_i$. It follows that $V \subseteq cur\text{-}W \cap T_i$. So, $V \subseteq cur\text{-}W$ after step 1.1. By a similar reasoning, one can show that if $V \subseteq cur\text{-}W$ before step 1.2.1, it holds afterwards also (QED).

Lemma 5.3.2 Let x be a fair state, and V a fair connected component containing x. Then, $is\text{-}Fair(\hat{B}, x)$ maintains the invariances $V \subseteq W$ and $V \subseteq cur\text{-}R$ at all times.

Proof Initially $V \subseteq cur\text{-}R$ and $V \subseteq W$ since $W = cur\text{-}R = R$. Assume before step 1.1, $V \subseteq cur\text{-}R$ and $V \subseteq W$. Then, since V is a fair connected component containing x, and $V \subseteq W$, we have that V is a subset of $SCC(x)$ in the graph restricted to $cur\text{-}R$. Therefore, $V \subseteq W$ after step 1.1 (note that step 1.1 does not modify $cur\text{-}R$). Now, assume $V \subseteq W$ and $V \subseteq cur\text{-}R$ before step 1.3. By lemma 5.3.1 and V being a fair subset of W, we have $V \subseteq fair\text{-}W$. Hence, $V \subseteq cur\text{-}R$ after step 1.3 (note that step 1.1 does not modify W) (QED).

Lemma 5.3.3 If $is\text{-}Fair(\hat{B}, x)$ returns a non-trivial set, then it is a fair connected component containing x.

Proof The only way step 1 returns a non-trivial set is if $W = fair\text{-}W$ at step 1.4. That means that W satisfies all CFC's, and it is a SCC containing x in $cur\text{-}R$. This implies that W is a fair connected component with respect to the CFC's and the negative fair edges. The only way step 2 returns a non-trivial set is if all positive edges constraints can be satisfied. It follows that W is a fair connected component containing x with respect to all fairness constraints (QED).

Lemma 5.3.4 Step 1 of $is\text{-}Fair(\hat{B}, x)$ terminates (and hence the algorithm terminates).

Proof This follows by noticing that at step 1.3 either $W = fair\text{-}W$, in which case step 1 terminates, or $fair\text{-}W$ is a proper subset of W (and hence $cur\text{-}R$), in which case the states in $W - fair\text{-}W$ are removed from $cur\text{-}R$ (QED).

Lemma 5.3.5 If x is a fair state, then *is-Fair*$(\hat{B}, x)$ returns a maximal fair connected component containing x.

Proof By lemma 5.3.4 step 1 terminates, and by lemma 5.3.2, at the end of step 1, for any fair connected component V containing x, we have $V \subseteq W$. Since x is fair, each positive fair edges constraint can be satisfied in V and hence in W. Therefore, step 2 returns W. By lemma 5.3.3, W is a fair connected component. By $V \subseteq W$ for any other fair connected component V containing x, W is a maximal fair connected component (QED)

Lemma 5.3.6 If *is-Fair*$(\hat{B}, x)$ returns a non-trivial set, then x is fair.

Proof Follows from lemma 5.3.3 (QED).

Lemma 5.3.7 x is not fair if and only if *is-Fair*$(\hat{B}, x)$ returns the empty set.

Proof Let x be unfair. By lemma 5.3.4 the algorithm terminates. Since x is unfair, by lemma 5.3.6, *is-Fair*$(\hat{B}, x) = \emptyset$. The reverse direction follows from lemma 5.3.5 (QED).

Lemma 5.3.8 If x is unfair, then x is deleted from $\hat{B}$.

Proof By lemma 5.3.7, *is-Fair*$(\hat{B}, x)$ returns the empty set. This can only happen in steps 1.2, 1.4, and 2. In all these cases, x is deleted from $\hat{B}$ before $\emptyset$ is returned (QED).

Now, we will prove that the subroutine *is-Fair* only deletes unfair states. To prove this, we need the following lemma.

Lemma 5.3.9 Let Γ be a fair cycle. Step 1 of *is-Fair*$(\hat{B}, x)$ maintains two invariances: if $\Gamma \cap W \neq \emptyset$, then $\Gamma \subseteq W$, and if $\Gamma \cap cur\text{-}R \neq \emptyset$, then $\Gamma \subseteq cur\text{-}R$.

Proof Both invariances are true after step 0. Now, assume they are true before step 1.1, and let $\Gamma \cap W \neq \emptyset$ after step 1.1 (note that step 1.1. does not modify $cur\text{-}R$). We need to show $\Gamma \subseteq W$. Since $\Gamma \cap W \neq \emptyset$, and since $W \subseteq cur\text{-}R$, we have $\Gamma \cap cur\text{-}R \neq \emptyset$. By the second invariance, it follows $\Gamma \subseteq cur\text{-}R$. Since Γ is a cycle containing x, and since $\Gamma \subseteq cur\text{-}R$, it follows that $\Gamma \subseteq SCC(x)$ in $cur\text{-}R$, i.e. $\Gamma \subseteq W$, as was desired. Now, assume the invariances hold before step 1.3. Let $\Gamma \cap cur\text{-}R \neq \emptyset$ after step 1.3 (note that step 1.1. does not modify W). We need to show $\Gamma \subseteq cur\text{-}R$. Since $\Gamma \cap cur\text{-}R \neq \emptyset$, and since $cur\text{-}R \subseteq W$, we have $\Gamma \cap W \neq \emptyset$. By the first invariance holding, it follows $\Gamma \subseteq W$. By lemma 5.3.1 and by Γ being a fair subset of W, we have $\Gamma \subseteq$ *get-Fair*$(W,$ CFC's$)$, which implies $\Gamma \subseteq cur\text{-}R$, as was desired (QED).

Lemma 5.3.10 The subroutine $is\text{-}Fair(\hat{B}, x)$ only removes unfair states from $\hat{B}$.

Proof We have to consider steps 1.2, 1.3, and 2 of the subroutine. In step 1.2, only state x can be deleted, in which case since *is-Fair* returns the empty set, and by lemma 5.3.7, x is unfair. Now, consider step 1.3, and let $y \in W - fair\text{-}W$ be a fair state, i.e. $y \in \Gamma$ for some fair cycle Γ. We have $\Gamma \cap W \neq \emptyset$, which by lemma 5.3.9 implies $\Gamma \subseteq W$. Now, by lemma 5.3.1 and by Γ being s fair subset of W, we have $\Gamma \subseteq fair\text{-}W$. But, this is a contradiction since $y \notin fair\text{-}W$. We conclude all states in $W - fair\text{-}W$ are unfair. Now, consider step 2, and assume some positive fair edges constraint E_i cannot be satisfied in W (i.e. $E_i \cap W = \emptyset$), but there exists some fair $y \in W$. Let $y \in \Gamma$ for some fair cycle Γ. We have $\Gamma \cap W \neq \emptyset$, which by lemma 5.3.9 implies $\Gamma \subseteq W$. By $E_i \cap W = \emptyset$, we have $E_i \cap \Gamma = \emptyset$. This means Γ cannot be a fair cycle, which is a contradiction. We conclude W only contains unfair states (QED).

Theorem 5.1 The path finding algorithm reports a fair connected component closest to the initial states.

Proof The algorithm terminates since at every point either a fair connected component is found (lemma 5.3.5), or some states are deleted from $\hat{B}$ (lemma 5.3.8). The correctness of the algorithm follows from the fact that the subroutine *is-Fair* only deletes unfair states (lemma 5.3.10). To prove the minimality, let x be the fair state reported by the above algorithm, whose distance from the initial states is l. The lemma follows by noting that all states in B of distance less than l have been determined to be unfair, and were removed from $\hat{B}$ (QED).

Some remarks are in order.

1. Some properties of the set B can make the debugging task easier. For example, if the set B contains only fair states, then the first x chosen will be fair, and no further search is needed.

2. If the fairness constraints come from the L-environment, i.e. are of the type $\prod_{i=1}^{n} F^{\infty}(S_i)$, then W either contains no fair cycle, or its fair subset is equal to it. Thus, the main loop of *is-Fair* ends after one iteration.

5.3.2 Finding A Fair Cycle in the L-Environment

Now that we have found a fair state x and a fair connected component W containing it, the next step is to find a short fair cycle starting at x in W. We first study the problem for the L-environment, i.e. for fairness constraints of the type $\prod_{i=1}^{n} F^{\infty}(S_i)$. Note that the negative fair edges are assumed to have been removed from W. As we show next, the complexity of returning a short error trace is NP-complete.

5.3.2.1 Complexity

Lemma 5.3.11 Let G be a graph on m vertices $V = \{v_1, \ldots, v_m\}$, and $V_1, \ldots, V_n$ be subsets of V. Then, answering the question of "whether there exists a cycle of length less than k visiting each V_i" is NP-Complete.

Proof 1. Proof of membership in NP. Just guess the cycle, and check whether the cycle visits each V_i. This check can be done in poly-time.

2. Proof of NP-Hardness. We reduce "Min Cover" ([GJ79]) to our problem. Let a set $S = \{s_1, \ldots, s_m\}$, and subsets of S, $U_1, \ldots, U_n$ be given where a set U_i is covered by s_j if $s_j \in U_i$. Create a complete graph G with m vertices, $v_1, \ldots, v_m$. Create n subsets of vertices of G, $V_1, \ldots, V_n$, as follows: $v_j \in V_i$ iff $s_j \in U_i$. Note that the size of G is polynomial in the size of the original min covering problem. Now assume cycle C with vertices $\{v_{i_1}, \ldots, v_{i_k}\}$ of length k visits each V_i. The cover $s_{i_1}, \ldots, s_{i_k}$ visits every V_i, and is of size k. Conversely, if there is a cover of size k, there is a set of vertices in G visiting each V_i. Since G is complete, there is a cycle among these vertices (QED).

Theorem 5.2 Let the ***short error trace problem*** be the problem of finding a fair path in a graph with fairness constraints of the form $\prod_{i=1}^{n} F^{\infty}(S_i)$ such that the total length of the path is minimum. The decision problem corresponding to the short error trace problem is NP-complete.

Proof The fair path consists of two segments: a path from initial states to a fair state x, and a fair cycle starting at x. Membership in NP follows by guessing the fair path and checking that it is indeed a fair path, which can be done in polynomial time. By making all states initial, we get a reduction from the problem of lemma 5.3.11 to the short error trace problem (QED).

5.3.2.2 The Basic Heuristic Algorithm

Let W be a FCC (fair connected component). The following algorithm finds a short fair cycle in W. It constructs a fair cycle gradually by extending an existing path. Let $P = (x_1, \ldots, x_m)$

be the current path. All the $F^{\infty}(S_i)$'s intersecting P are marked as inactive. If all $F^{\infty}(S_i)$'s are inactive, we augment P by a path from x_n to x_1, and report this as the fair cycle. Otherwise, let $F^{\infty}S_{i_1}, \ldots, F^{\infty}S_{i_k}$ be the current set of active $F^{\infty}(S_i)$'s, ordered in some arbitrary manner (more experiments may show that some orderings give better results than others). We heuristically try to find a state which causes many of active fairness constraints to become inactive. Let l be the maximum number such that $W \cap \left(\bigcap_{j=1}^{l} S_{i_j} \right) \neq \emptyset$. Let $Q = S \cap \left(\bigcap_{j=1}^{l} S_{i_j} \right)$. We choose a state $x \in Q$ such that the distance between x_m (the last state of P) and x_{m+1} is minimum, using the general shortest path finding algorithm. Note, by augmenting P by this path, we can mark $F^{\infty}(S_{i_1}), \ldots, F^{\infty}(S_{i_l})$'s as inactive. The following algorithm implements this idea.

Path Extension Algorithm

```
1 Let $Q = W \cap S_{i_1}$ and $l = 2$.
2 While $k \neq l$ and $Q \cap S_{i_l} \neq \emptyset$
      Let $Q = Q \cap S_{i_l}$, and $l = l + 1$.
3 Return a shortest path from $x_m$ to $Q$.
```

The following algorithm uses the above path extension method and starting vertex x to find a short bad cycle in W.

Short Cycle Algorithm

```
1 Let $x_1 = x$ and $P = (x_1)$.
2 While not done
  2.1 Mark as inactive all the $F^{\infty}(S_i)$ 's which intersect $P$.
  2.2 If there are no more active $F^{\infty}(S_i)$ 's, find a path from the last state of $P$ to $x_1$, and quit.
      Else extend $P$ by calling the above path extension routine.
```

5.3.2.3 Some Optimization

In this subsection, we present some optimizations which can be applied to the algorithm of the previous subsection.

Optimization 1. When the path finding algorithm is called in step 2 of the path extension algorithm, one can try to choose the states so that more fairness constraints become satisfied. For example, assume a path from x to some set Q is wanted with $F^{\infty}\left(S_{i_{l+1}}\right), \ldots, F^{\infty}\left(S_{i_{k_l}}\right)$ being still active. Let R_j's be the states reached during the forward search of the path finding algorithm. During the backward pass, if $R_j \cap S_{i_q} \neq \emptyset$ for some active $F^{\infty}\left(S_{i_{ql}}\right)$, then choose a state from $R_j \cap S_{i_q}$, thereby making $F^{\infty}\left(S_{i_{ql}}\right)$ inactive.

Optimization 2. To further decrease the cycle's length, the following algorithm can be used as a post-processing step.

1. *Start with all nodes in the cycle, and find a minimum cover, where a cover is a set of vertices such that each $F^{\infty}(S_i)$ is visited.*
2. *Arrange the points in the cover in the same order as the original fair cycle.*
3. *Find a shortest path between consecutive points of the cover. Use optimization 1 during this process, eliminating points in the cover if they become inactive.*

It is easy to see that the length of the new cycle is guaranteed to be no longer than the original length (note that the vertices in the cover are arranged in the same order as the original cycle). Step 3 of the above algorithm can be improved by forming a new covering problem after the shortest path between two points in the new cover is calculated; some vertices in the new cover may be eliminated since the vertices in the path between two points can cause some active $F^{\infty}(S_i)$ to become inactive.

5.3.3 The edge-Streett Case

The algorithm of subsection 5.3.1, which finds a path to a fair state, returns a fair connected component W containing some fair state x which serves as the starting point for the fair cycle. It also guarantees that $S_i \cap W \neq \emptyset$ for the "active" CFC's, and that for the inactive CFC's, $W \subseteq T_j$ and hence are already satisfied. Hence, we can restrict our attention to S_i's of the active CFC's. The algorithm of subsection 5.3.2 finds a fair cycle, when the fairness constraints are of the form $\prod F^{\infty}(S_i)$. If we disregard the positive edge constraints, since all inactive CFC's are already satisfied and for all active CFC's $S_i \cap W \neq \emptyset$, we can use the algorithm of subsection 5.3.2 on the S_i's of the active CFC's. Note that this solution may be sub-optimal since we have not considered the T_i's of the active CFC's. For example, assume W is a complete graph on 2 vertices, and there is an active CFC $F^{\infty}\{2\} + G^{\infty}\{1\}$, and the starting state is 1. Our algorithm will return the cycle $(1, 2)$, whereas the cycle consisting of the self-loop on 1 is a shorter fair cycle.

Also, the positive edge constraints may not be satisfied by the solution returned by the algorithm of subsection 5.3.2. To satisfy a given positive edge constraint, consisting of the set of edges $E_i(x, y)$ and not satisfied by the current cycle $C = (c_1, \ldots, c_{m+1})$, we look for a closest state c_j in C to the starting vertices of the edges in E_i. Let the chosen starting vertex be u. We then choose a closest path from some vertex v to $c_{(j+1) \bmod n}$, such that $(u, v) \in E_i(x, y)$. We call this operation ***patching the cycle*** with respect to the positive edge constraint E_i. We repeat this process for all positive edge constraints. The algorithm to find a fair cycle is, thus, summarized as:

1. Find a cycle by calling algorithms of subsection 5.3.2 on the S_i's of active $F^{\infty}(S_i) + G^{\infty}(T_i)$'s.

2. Patch the cycle with respect to each positive fair edges constraint.

Note that the above algorithm always works since W is a fair connected component, i.e. it contains at least one edge from each positive fair edges constraint.

Remark One can improve the quality of the algorithm for the edge-Streett case by modifying the algorithm of subsection 5.3.2, so that it only considers those active CFC's whose T_i's are violated, i.e. there is some state in the current path which does not lie in T_i.

5.4 Debugging for Fair CTL

In this section, we describe a debugger for fair CTL formulas based on the debugging environment of section 5.3 for LC. Since CTL formulas are interpreted over computation trees, a witness for falsity of a formula ϕ is a computation tree, called a ***debug tree***, on which ϕ does not hold. For ACTL formulas, it is possible to finitely describe a class of debug traces. For other formulas involving existential quantification, user interaction is required to explore the debug trees.

As an example, consider the ACTL formula $AGAFp$. A class of debug trees can be described by giving two fair paths π_1 and π_2, such that π_1 starts at some initial state, p never holds along π_2, and the initial state of π_2 is some state belonging to π_1. Intuitively, π_1 and π_2 are witnesses to the formulas $AGAFp$ and AFp not holding, respectively. Now, consider the formula $AGEFp$. One can present a path π_1 along which there exists some state s_i such that EFp does not hold at s_i. However, to prove that EFp does not hold at s_i involves presenting all computation trees starting at s_i. Hence, user interaction is needed to explore a subset of the debug trees.

Definition For an ACTL formula ϕ not holding at x, a ***debug structure*** is defined recursively as follows.

1. $\phi = P$, for some atomic proposition P. The state x.
2. $\phi = f \wedge g$. A debug structure for f or g, whichever does not hold at x.
3. $\phi = f + g$. Debug structures for both f and g at x.
4. $\phi = AXf$. A fair path $\pi = \pi(0), \ldots, \pi(n), \ldots$, such that f does not hold at $\pi(1)$, and the debug structure for f at $\pi(1)$.
5. $\phi = AGf$. A fair path $\pi = \pi(0), \ldots, \pi(i), \ldots, \pi(n), \ldots$, such that f does not hold at $\pi(i)$ for some i, and the debug structure for f at $\pi(i)$. Recall a fair path is represented finitely by an initial segment and a fair cycle.
6. $\phi = AFf$. A fair path $\pi = \pi(0), \ldots, \pi(n), \ldots$, such that f is false at all $\pi(i)$'s, and the debug structures for f at all $\pi(i)$'s.
7. $\phi = A(fUg)$. There are two possibilities in this case.

a. A fair path $\pi = \pi(0), \ldots, \pi(i), \ldots, \pi(n), \ldots$ such that f is false at $\pi(i)$, and g is false at all $\pi(j)$'s for $j \leq i$. The debug structure for f at $\pi(i)$, and the debug structures for g at all $\pi(j)$'s for $j \leq i$ are also given.

b. A fair path $\pi = \pi(0), \ldots, \pi(n), \ldots$ such that g is false at all $\pi(j)$'s, and the debug structures for g at all $\pi(j)$'s.

Theorem 5.3 Let the size of a debug structure be the number of states in it. Determining the debug structure (for ACTL formulas) of minimum size in the presence of fairness constraints of the form $\prod_{i=1}^{n} F^{\infty}(S_i)$ is NP-complete.

Proof From the definition of debug structures, it follows that a debug structure is polynomial in the size of the formula and system, and it can polynomially be checked whether a debug structure is valid. Hence, the problem is in NP. To see the problem is NP-hard, reduce the short error trace problem of subsection 5.3.2.1 to our problem, with the CTL formula being $AF(false)$. Note that any fair path satisfies this formula (QED).

Remark The complexity of finding a debug structure of minimum length for ACTL formulas in the absence of fairness constraints is unknown.

We now present a debugger for fair CTL. Let ϕ be a CTL formula in positive normal form (PNF). The formula ϕ does not hold for the system if it does not hold at some initial state s_0. In this case, we have to give s_0 and provide a proof of why it does not hold at s_0. The debugger works incrementally, giving debugging information one operator at a time, and leaving it to the user to direct the debugger as to what other information is needed next. At every point, the input is a CTL formula ψ, and a fair state x in which ψ is false. The formula ψ and state x constitute the state of the debugger. Every state output by the debuggers is given a unique identifier, and the user can refer back to any previously visited state. This feature is used to backtrack and explore other paths to get further debugging information.

As an example, consider the formulas $AGAFf$. The debugger first gives a path from some initial state s_0 to some state x where AFf is false. Then, the user can either ask for a fair path from x, or a proof of why AFf is false at x. In the latter case, the debugger presents a fair path π along which f is always false. All states along π are given a unique identifier, and the user is given the option to explore why f is false at any of these states. The user can pop back from such explorations, and explore why f does not hold at other states of π. The algorithm is as follows.

1. $\psi = P$, for some atomic proposition P. Report x.

2. $\psi = f+g$. In this case, ask the user which of the formulas to pursue further, since both are false. Assume f is chosen. The debugger is recursively called with f at x. The user can come back to this state, and call the debugger with g at x later on.

3. $\psi = f \wedge g$. Choose (arbitrarily if there is a choice) one of the formulas f or g which does not hold at x, and call the debugger with this formula at x.

4. $\psi = AXf$. Choose (arbitrarily if there is a choice) a fair next state of x in which f does not hold. Give the user the option of having a fair path starting at x, or debugging f at x.

5. $\psi = AGf$. Call the shortest path algorithm of section 5.2 to find a shortest path between x

and the set of fair states satisfying $\bar{f}$. Let y be a closest state in $Fair+$ to x where f does not hold. Report the path from x to y, and give the user the option of having a fair path starting at y, or debugging f at y.

6. $\psi = AFf$. In this case, we have to find a fair path from x along which f never becomes true. Call the LC debugger with $S(\bar{f})$ as the reachable set, $\bar{f} \cap Fair+$ as the bad set, and x as the initial state. To speed up the LC debugger, the forward and backward stable set operators can be applied to $\bar{f} \cap Fair+$, to delete states which cannot reach or be reached by cycles. Report the error trace, and allow the user to ask why f does not hold at any state along the error trace.

7. $\psi = A[fUg]$, where $f \neq true$. There are two possibilities: there is a fair path along which either g never becomes true, or f does not hold until g becomes true. Consider the set of states $S = \bar{f} \cap \bar{g} \cap Fair+$. Find a shortest path π from x to S, along which $f \wedge \bar{g}$ holds. Let $y \in S$ be the endpoint of π. Report π, and ask the user whether a fair path from y, or a proof of why f and g do not hold at y, is desired. If such a path π does not exist, call the LC debugger with x as the initial state, $S(\bar{g})$ as the reachable set, and the set $U = \bar{g} \cap Fair+$ as the bad states. Give the user the option to explore why g does not holds along the error trace returned by the LC debugger. To guarantee minimality of the initial segment, one can get an error trace for both possibilities, and choose the one whose initial segment is smaller.

8. $\psi = EXf$. Tell the user how many next states x has, and ask for user constraints on the set of next states of interest. The constraints are a set of state variables and values for them (recall that each state is a product state). Enumerate those next states which satisfy the user constraints and are in $Fair+$ until the user chooses a next state y. Call the debugger recursively with f at y. In this case, the user had thought f holds at y, whereas it does not.

9. $\psi = EGf$. If f does not hold at x, call the debugger with f at x. Otherwise, let $N(x)$ denote the set of next states of x, subject to some user constraints. Let $S_1 = N(x) \cap f \cap Fair+$, and $S_2 = N(x) \cap \bar{f} \cap Fair+$. Ask the user to choose S_1 or S_2. If S_1 is chosen, enumerate all states in S_1 until the user chooses a next state y. In this case, the user is following the path expected to have satisfied EGf. Call the debugger recursively on EGf and y. If S_2 is chosen, enumerate all states in S_2 until the user chooses a next state y, and call the debugger recursively on f at y. In this case, the user had thought f holds at y, whereas it does not.

10. $\psi = EFf$. In this case, we know for all next states of x, ψ does not hold. Let $S = N(x) \cap Fair+$, where $N(x)$ denotes the set of next states of x, subject to user constraints. Enumerate S until the user chooses a next state y. Ask the user to choose between proving why EFf or f does not hold at y. Call the debugger recursively with the chosen formula and y.

11. $\psi = E[fUg]$, where $f \neq true$. If x satisfies $\bar{f} \wedge \bar{g}$, print x, and ask whether the user wants to see a fair path starting at x, or prove why f and g are false at x. Otherwise, f holds at x. Ask the user if a proof of why g does not at x is desired. If not, tell the user the number of next states satisfying f and $\bar{f} \wedge \bar{g}$, and ask which set to enumerate next. Enumerate the corresponding set of states, subject to user constraints and being in $Fair+$, until the user chooses a next state y. Call the debugger with $E[fUg]$ at y.

Remark For ACTL formulas in absence of fairness constraints, one can prove a minimality result by slightly modifying the algorithm. For each formula of the type AGf, AXf, and $A(fUg)$, an infinite path showing the falsity of the formula is returned. An infinite path consist of an initial segment and a cycle. Now, it is easy to see that at each step the witness path returned has the property that its initial segment is of minium length, and the cycle is a shortest cycle containing the last element of the initial segment.

5.5 Algorithms of [CGMZ95]

In this section, we compare our algorithms to those of [CGMZ95].

5.5.1 Debugging For EGf Formulas

[CGMZ95] describes an algorithm for finding an error trace for the formula EGf under fairness constraints of the form $\prod_i F^{\infty} p_i$. This algorithm can then be used as the basis for debugging fair CTL formulas. The algorithm picks a state $s \in Fair+$ (initially, s is some initial state), and tries to find a fair cycle containing s, by satisfying the fairness constraints one at a time. To satisfy a fairness constraint of the form $F^{\infty} p_i$ from a state u, the algorithm finds a shortest path from u to some state in p_i. Note that the algorithm may leave the SCC of u in this process. When all fairness constraints are satisfied, a path from the last state to s is found. If no such path exists, then the procedure is repeated with last state of the path as the new seed. Note that the cycle may not be completed due to the algorithm having left the SCC of s. The algorithm is guaranteed to find an error trace since the seed will eventually end up in a leaf SCC of $Fair+$. This leaf SCC is guaranteed to contain a fair cycle.

Finding a short witness for the formula EGf under the fairness constraints $\prod_i F^{\infty} p_i$ can be accomplished using the LC debugger by restricting the set of reachable states to $S(f) \cap R$, where $S(f)$ denotes the set of states satisfying f, and R is the set of reachable states. Our algorithm for this task, presented in section 5.3, finds a state $s \in Fair\pm$ which is closest to the initial states. If the SCC of s is fair, then the algorithm finds a fair cycle in it. Otherwise, the states in SCC of s are deleted, and the search is re-started. Hence, it can guarantee that in the error trace, the path from the initial state to the fair cycle is minimum. Note that if the set $Fair$ (the set of states involved in some fair cycle) is used instead of $Fair\pm$, we would be guaranteed that the SCC of s is fair. But computing $Fair$ is difficult. Hence, the set $Fair\pm$ is used as an approximation.

The algorithm of [CGMZ95] may be faster, since it does not compute the SCC of any state (note that this can be done using only reachability computations). However, it is expected to produce worse results, or run even longer. For example, assume s belongs to some fair cycle. If the graph consists of several SCC's, then [CGMZ95]'s algorithm may leave the SCC of s, and hence never find a fair cycle involving s. In this case, a new state has to be picked and the computation re-started. Also, the error trace will become unnecessarily long, due to the fact that an error trace in the first SCC was missed. Incidentally, for the algorithm of [CGMZ95], if the set $Fair$ was used in place of $Fair+$, the situation does not change much, i.e. it might still produce long error traces.

5.5.2 Debugging for Streett Conditions

The second main contribution of [CGMZ95] is giving an algorithm for debugging under Streett conditions, i.e. fairness constraints of the form $\prod_{i=1}^{n} F^{\infty}(p_i) + G^{\infty}(q_i)$. Since there is a fair cycle, we are guaranteed that there exists a set of $O_i^{\infty}(r_i)$, where $O_i^{\infty}(r_i)$ is $F^{\infty}(p_i)$ or $G^{\infty}(q_i)$, such that there is a fair cycle for the fairness constraint $\prod_{i=1}^{n} O_i^{\infty}(r_i)$. This fair cycle is a fair cycle of the original problem. To find the $O_i^{\infty}(r_i)$'s, the algorithm makes n calls to the language containment routine. First, the language emptiness routine is called with $F^{\infty}(p_1) \wedge \prod_{i=2}^{n} F^{\infty}(p_i) + G^{\infty}(q_i)$ as the fairness constraint. If there is a fair cycle, then $O_1^{\infty}(r_1) = F^{\infty}(p_1)$. Otherwise, $O_1^{\infty}(r_1) = G^{\infty}(p_1)$. Next the language emptiness routine is called with $O_1^{\infty}(p_1) \wedge F^{\infty}(p_2) \wedge \prod_{i=2}^{n} F^{\infty}(p_i) + G^{\infty}(q_i)$ as the fairness constraint. If there is a fair cycle, then $O_2^{\infty}(r_2) = F^{\infty}(p_2)$. Otherwise, $O_2^{\infty}(r_2) = G^{\infty}(p_2)$. This process continues until all $O_i^{\infty}(r_i)$'s are determined. Let $p_{i_1}, \ldots, p_{i_j}$ be those r_i 's for which $O_i^{\infty} = F_i^{\infty}$. Let $q_{k_1}, \ldots, q_{k_l}$ be those r_i 's for which $O_i^{\infty} = G_i^{\infty}$. The algorithm for finding counter-examples for the formula EGf under the fairness constraint $\prod_i F^{\infty} p_i$ is now called with $f = q_{k_1} \wedge \ldots \wedge q_{k_l}$, and the p_i 's being $p_{i_1}, \ldots, p_{i_j}$. This error trace is an error trace for the graph under the Streett fairness conditions.

Since each language containment check is expensive, this algorithm is expected to have high complexity cost. Note that our algorithm presented in section 5.3 does not need to call the language containment routine. Also, [CGMZ95]'s algorithm can generate long error traces due to the fact that for some choices for $O_i^{\infty}(r_i)$'s, there may be shorter error traces.

Acknowledgment

This material presented in this chapter is based mainly on [HBK93] and [HSB94].

Chapter 6

Improving Language Emptiness Check Using Fairness Graphs

6.1 Introduction

We present techniques to speed up the main computation for language containment. Recall that language containment reduces to checking whether the language of the product automaton is empty. To do so, the set of reachable states is computed, and early failure detection is performed. If no errors are found, a computation called the main computation is started, which returns a superset of fair states (section 4.5). In some cases, the main computation is a significant portion of the running-time of the algorithm. Our hope is to make language containment with fairness constraints essentially as efficient as language containment without fairness constraints, i.e. reduce the complexity of verification to that of computing the reachable set. The techniques we present in this chapter apply to the L-environment, which is an important subset of the eSeR-environment. Hence, we assume we are given a graph with negative fair subset constraints, or cycle sets. A cycle is fair if it is not entirely contained in any cycle set. Note that at this stage, all negative fair edges have been removed[1].

We propose to take advantage of a graph induced by the fairness constraints (i.e. cycle sets), known as the ***fairness graph***. If this graph is acyclic, then we prove that there are no fair runs. Hence, the main computation can be bypassed. If it contains cycles, then in some situations we can combine sets of fairness constraints, and hence reduce the number of fairness constraints in the main computation. The problem of minimizing the number of fairness constraints at this step is equivalent to the ***partition into forest problem***, which is NP-complete (GJ79]). So we use a greedy heuristic for it. We have implemented our techniques in the formal verification system HSIS using BDD's. Based on limited experience, we observed that our techniques can reduce the run-time of the main computation.

The chapter's flow is as follows. Section 6.2 defines the fairness graph and states its proper-

1. This formulation is equivalent to checking language emptiness for a graph with the product of Buchi conditions as the fairness constraint. If $c_1, c_2, \ldots, c_n$ are the cycle sets, $\overline{c_1}, \overline{c_2}, \ldots, \overline{c_n}$ are the Buchi constraints.

ties. Section 6.3 presents our BDD-based method for building the fairness graph. Section 6.4 formulates the optimization problem associated with clustering the cycle sets, states its NP-completeness, and describes our heuristic. Section 6.5 presents some experimental results.

6.2 The Fairness Graph and Its Properties

In what follows, we assume Q is the state graph of an automaton with cycle sets $c_1, \ldots, c_n$, such that every state is contained in at least one cycle set (which is the case after early failure detection). Recall that early failure detection (section 4.7) checks that there are no fair cycles containing a state of $\prod \bar{c}_i$.

Definition The ***fairness graph*** Q_f induced by the cycle sets is a graph on n nodes $\tilde{c}_1, \ldots, \tilde{c}_n$ which has an edge from vertex $\tilde{c}_i$ to $\tilde{c}_j$ iff there exists an edge (u, v) in Q such that $u \in c_i$, $v \in c_j$, and $v \notin c_i$.

Examples In figure 6.1, three examples of graphs and their fairness graphs are given. The original graphs are on top, and the fairness graphs are at the bottom. Each oval around the vertices signifies a cycle set. Note that in example b, there is no edge from the first cycle set to the second one, since there is no edge from c_1 to c_2 whose endpoint is not in c_1. Also, note that in example c, although there is no cycle in the original graph, there is a cycle in the fairness graph.

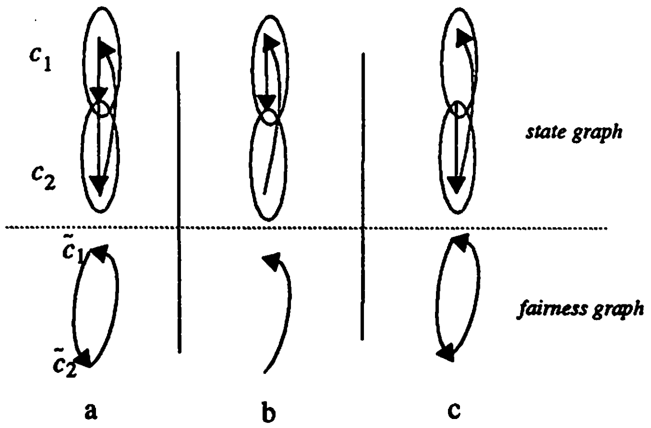


Figure 6.1: Examples of fairness graphs

Definition Let a path in Q $x_1, \ldots, x_p$ be given. Let $c_1, \ldots, c_p$ be a sequence of cycle sets define recursively from $\{x_i\}$ such that $x_1 \in c_1$ for all i, $x_i \in c_i$, and for $i \geq 2$ if $x_i \in c_{i-1}$, then

$c_i = c_{i-1}$. We call this sequence a ***projection*** of a path. Let a ***shadow of a path*** be obtained from a projection by deleting consecutive duplicate copies of a cycle set.

Intuitively, as we follow a path we go through a sequence of cycle sets. A shadow of a path is obtained by only counting those cycle sets which are different than the current one. Note that a shadow of a path is a sequence of nodes in its fairness graph. Figure 6.2 gives an example of a state graph and its fairness graph, which shows that the shadow of a path may not be unique, since the projection of a path may not be unique.

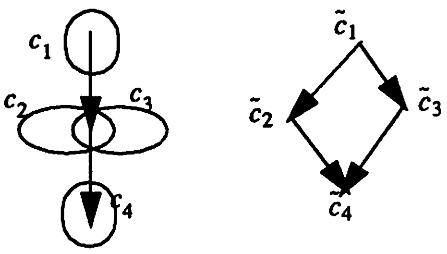


Figure 6.2: Shadow of a path may not be unique

We have the following properties for Q_f, the fairness graph, which follow directly from the definition of the fairness graph.

Lemma 6.2.1 There are no self-loops in Q_f.

Lemma 6.2.2 Let $\Gamma = x_1, \ldots, x_n, x_1$ be a cycle of states, a shadow of which visits $c_{i_1}, \ldots, c_{i_k}, c_{i_1}$ in order. Then, there is a corresponding cycle $\tilde{c}_{i_1}, \ldots, \tilde{c}_{i_k}, \tilde{c}_{i_1}$ in Q_f.

To motivate the next central lemma, consider the example shown in figure 6.3. A state graph Q is shown on the left with its fairness graph on the right. There is one fair cycle in Q, one of its projections is (c_1, c_2, c_3), which is not a cycle in the fairness graph. Another projection, however, is (c_3, c_2, c_3) which is a cycle in the fairness graph. The next lemma proves that for every fair cycle in Q, there is a cyclic projection, and hence a cycle in the fairness graph.

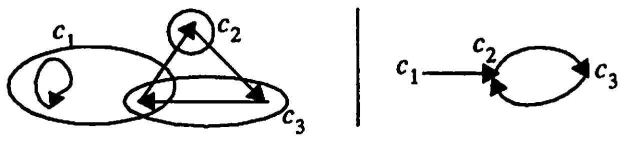


Figure 6.3: A motivating example for lemma 6.2.3

Lemma 6.2.3 Every fair cycle in Q has a projection which is a cycle containing more than one node (recall that a fair cycle is a cycle not contained in any of the cycle sets).

Proof Let $\Gamma = (x_1, \ldots, x_p, x_1)$ be a fair cycle in Q. Every projection of Γ has more than one node by the definition of a fair cycle, i.e. Γ is not contained entirely in any cycle set. Now, consider a projection of Γ, $P_1 = \left(\bar{c}_1, \ldots, \bar{c}_p, \bar{c}_{p+1}\right)$, where $\bar{c}_1 \neq \bar{c}_{p+1}$. If $x_1 \notin \bar{c}_p$, then $\bar{c}_1, \ldots, \bar{c}_p, \bar{c}_1$ is also a projection, and we are done. Assume $x_1 \in \bar{c}_p$, which implies $\bar{c}_{p+1} = \bar{c}_p$. Since Γ is not contained in any cycle set, there exists $1 < l < p-1$ such that $x_{l+1} \notin \bar{c}_p$ and $x_i \in \bar{c}_p$ for $1 \leq i \leq l$. Consider the projection P_2 obtained from P_1 by replacing the first l elements by $\bar{c}_p$. Since P_2 is a cycle, the lemma follows (QED).

Lemma 6.2.4 Every fair cycle in Q has a shadow which is a cycle containing more than one node.

Proof Follows from lemma 6.2.3 (QED).

Lemma 6.2.5 There may be a cycle $\bar{c}_{i_1}, \ldots, \bar{c}_{i_k}, \bar{c}_{i_1}$ in Q_f, but no corresponding cycle in Q that visits $c_{i_1}, \ldots, c_{i_k}, c_{i_1}$.

Proof Figure 6.1c is an example (QED).

Theorem 6.1 If Q_f is acyclic, then there are no fair cycles in Q.

Proof Assume that there is a fair cycle Γ in Q. By lemma 6.2.4, there exists a shadow of Γ which is a cycle containing more than one node. By lemma 6.2.2, there exists a cycle in Q_f (QED).

6.3 Building the Fairness Graph

Building the fairness graph is the most time-consuming part of our algorithm. Note that without using BDD's, building this graph would be very time-consuming, since possibly all edges in the product L-process may need to be visited. There are many ways to build this graph using BDD's, some performing better than others. In section 6.3.1, we first present a naive algorithm, and then present one which improves the performance of the first one. In section 6.3.2, we discuss ways of better reflecting the structure of Q in Q_f, by deleting states which cannot contain any fair cycles.

6.3.1 Algorithms

To build the fairness graph using BDD's, we introduce two new variables which take values from 1 to n, where n is the number of cycle sets. The first variable, denoted by x', corresponds to a present state of Q_f; the second one, denoted by y', corresponds to a next state. We eventually build a transition relation $R(x', y')$, which represents Q_f. Let $A(x)$ represent the active set of states, which is originally the set of reachable states with the states in $\bigcap_i \bar{c}_j$ deleted (these states are deleted after early failure detection). We assume the transition relation of Q, $T(x, y)$, is restricted to $A(x)$, and that the negative fair edges have been deleted from T. The following lemma provides a straight-forward way to build the fairness graph.

Lemma 6.3.1 $R(x', y') = \sum_i \sum_j \exists x \exists y \Big(T(x, y) \wedge c_i(x) \wedge c_j(y) \wedge \overline{c_i(y)} \wedge (x' = i) \wedge (y' = j) \Big)$

computes Q_f. This computation does $2n^2$ quantifications, $5n^2$ Boolean AND's, and n^2 Boolean OR's.

Proof For every pair of cycle sets c_i and c_j the above computation checks whether there is an edge (i, j) in Q_f. Hence, R represents Q_f. The number of operations follows by expanding the above sums, which has exactly n^2 terms (QED).

The improved version involves re-arranging the order of various operations.

Lemma 6.3.2 $\sum_j \exists y \Big(\sum_i (\exists x (T(x, y) \wedge c_i(x))) \wedge \overline{c_i(y)} \wedge (x' = i) \Big) \wedge c_j(y) \wedge (y' = j)$ computes Q_f. Moreover, it takes $2n$ quantifications, $5n$ Boolean AND's, and $2n$ Boolean OR's.

Proof It suffices to show that the above equation is algebraically equal to the one in the previous lemma. This follows by noticing that existential quantification and Boolean OR commute. To count the number of operations, note that $\sum_i (\exists x (T(x, y) \wedge c_i(x))) \wedge \overline{c_i(y)} \wedge (x' = i)$ is independent of j and has to computed only once. It requires n quantifications, $3n$ Boolean AND's, and n Boolean OR's. The outer computation takes n quantifications (once for each j), $2n$ Boolean AND's, and n Boolean OR's. The bounds follow (QED).

Algorithmically, we can think of this expression as consisting of two passes over the cycle sets, with the first pass computing $S(x', y) = \sum_i (\exists x (T(x, y) \wedge c_i(x))) \wedge \overline{c_i(y)} \wedge (x' = i)$.

```
I. Let $S(x, y) = 0$.
For each cycle set $c_i$
    1. $U_i(y) = (\exists x (T(x, y) \wedge c_i(x))) \wedge \overline{c_i(y)}$.
    2. $S(x', y) = S(x', y) + (U_i(y) \wedge (x' = i))$.
```

The second pass takes $S(x', y)$ and returns the fairness graph $R(x', y')$. The algorithm is as follows.

```
II. Let $R(x', y') = 0$.
For each cycle set $c_j$
    1. $W_j(x') = \exists y (S(x', y) \wedge c_j(y))$
    2. $R(x', y') = R(x', y') + (W_j(x') \wedge (y' = j))$.
```

We remark that $U_i(y)$ is the set of states in $\overline{c_i}$ which are reached by some states in c_i, and $W_j(x')$ is the set of cycle sets c_k that have an edge to a state in $\overline{c_k} \cap c_j$. We use these observations in the next section. Note that the U_i's can all be computed in parallel. Once S is computed, all the W_j's can also be computed in parallel.

6.3.2 Improving the Structure of the Fairness Graph

As the fairness graph is being built, sets of states where no fair behavior can exist are deleted. This process will change the fairness graph. In this section, we study what states can be deleted, and how inaccurate the fairness graph can become.

Definition A cycle set c_i is said to be a ***sink cycle set***, if there is no edge (u, v), where $u \in c_i$ but $v \notin c_i$. Similarly, a cycle set c_i is said to be a ***source cycle set*** if there is no incoming edge (u, v), where $u \notin c_i$ but $v \in c_i$. No states of a sink and source cycle sets can be involved in a fair cycle.

Consider example a in figure 6.4, where s_1, s_2, s_3, s_4 are the states and c_1, c_2, c_3, c_4 are the cycle sets. Assume in the first pass we process c_1, c_2, c_3, c_4 in that order. When processing c_4, we find that $U_4(y) = 0$, which means c_4 has no outgoing edges, i.e. it is a sink cycle set. No state of a sink cycle set can be involved in any fair cycles. Hence, s_4 can be deleted from Q, and c_4 from Q_f. Now, s_2 has no outgoing edges, and can be deleted by the forward stable set operator (recall this operator deletes any states which cannot reach a cycle). Let ***optimization 1*** denote the deletion of sink cycle sets as well as the application of forward and backward stable set operators in the first pass. Optimization 1 in this example deletes s_2, s_4 and c_4 from Q.

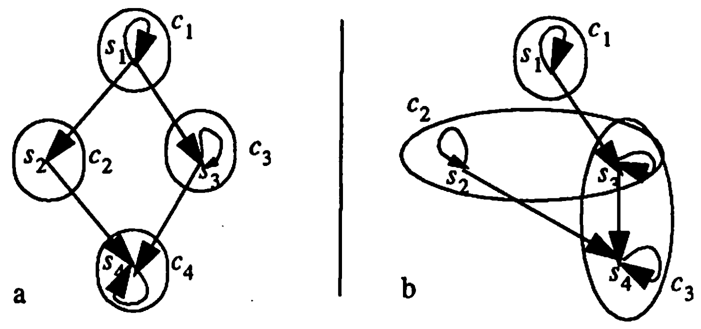


Figure 6.4: Concept of fake edges

More precisely, with optimization one enabled, pass 1 becomes as follows.

1. Let $S(x, y) = 0$*, and* $A = R$*, where* R *is the set of reachable states..*

For each cycle set c_i

1. $U_i(y) = (\exists x(T(x, y) \wedge c_i(x))) \wedge \overline{c_i(y)}$.
2. *If* $U_i \neq 0$, $S(x', y) = S(x', y) + (U_i(y) \wedge (x' = i))$.
3. *If* $U_i = 0$, $A = A \wedge \bar{c}_i$, *apply the forward and backward stable set operators to* A *, and let* $T(x, y) = T(x, y) \wedge A(x) \wedge A(y)$.

Now, consider example b in figure 6.4. Assume in the first pass c_1, c_2, c_3 are processed in order. Since c_3 is a sink cycle set, states s_3 and s_4 are deleted. The stable set operators cannot delete any more states in this case. When we processed c_1, we added the pair $(1, s_3)$ to $S(x', y)$, which means c_1 has an edge to s_3. But since s_3 is deleted from the graph, this is no longer true. The question is whether we will have an edge $(1, 2)$ in $R(x', y')$. We call such edges in $R(x', y')$ which should not exist in the final fairness graph, ***fake edges***. Assume we modified our algorithm in step 1 of pass 2 to $W_j(x') = \exists y(S(x', y) \wedge c_j(y) \wedge A(y))$, where

$A(y)$ is the current active set of states in Q. Note that in the first pass, we always restrict $T(x,y)$ to $A(x)$. Also assume that the stable set operators are originally applied to Q, before the building of the fairness graph is started. We have the following important theorem.

Theorem 6.2 If optimization 1 is applied during the first pass, then $R(x',y')$ contains no fake edges.

Proof Let (u,v) be an edge in Q such that $u \in c_i$, $v \in c_j$, and $v \notin c_i$ (figure 6.5 a). Then, (u,v) gives rise to a fake edge (i,j) in $R(x',y')$ iff at least one of u or v is deleted by optimization 1. We distinguish between three cases.

Case 1. v is deleted by optimization 1. Then $v \notin c_j$ in the second pass, and the pair (i,v) gets deleted from $S(x',y)$ in pass 2, step 1, and does not gives rise to the edge (i,j) in $R(x',y')$.

Case 2. Only u is deleted in the first pass, and this occurs because u belongs to some sink cycle set c_k. We have that $v \notin c_k$, otherwise it would have also been deleted. But, we have an edge from c_k to c_j because $u \in c_k$, $v \in c_j$, and $v \notin c_k$. Therefore, c_k is not a sink cycle set, which is a contradiction.

Case 3. Only u is deleted in the first pass, and this occurs because of an application of one of the stable set operators after some sink cycle set c_k is deleted (figure 6.5 b). Since v is not deleted, v can reach a cycle. Hence, u can reach a cycle, after c_k was deleted. Thus, u is not deleted by the forward stable set operator. So, u is deleted by the backward stable set operator. Since, before deletion of c_k, u could be reached by some cycle, there is a cycle in c_k which can reach u. But, since $u \notin c_k$, there is a path from c_k to some state not in c_k. This implies that c_k has an out-going edge contradicting that c_k is a sink cycle set (QED).

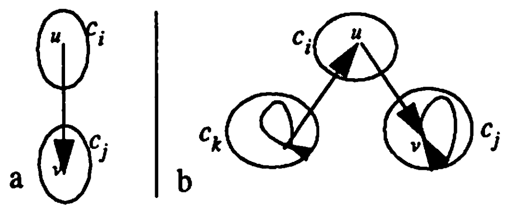


Figure 6.5: Different cases in proof of theorem 6.3

Now, consider example a in figure 6.7. Assume in the second pass, we process c_1 and c_2 in that order. When processing c_2, we find that $W_2(x') = 0$. Note that we may have $W_j(x') = 0$, but c_j is not a source cycle set, as example a in figure 6.7 shows where c_1 is not a source cycle set but $W_1(x') = 0$ (figure 6.3 and c_1 is another example). Similarly, we may have $W_j(x') \neq 0$, but c_i is a source cycle set, as example b in figure 6.7 where c_2 is a source cycle set but $W_2(x') \neq 0$. However, if $W_j(x') = 0$, we can conclude those states which are unique to c_j cannot be involved in any fair cycle.

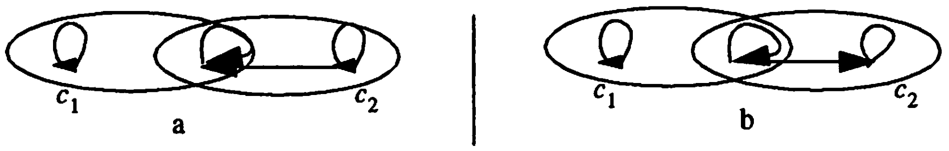


Figure 6.6: Examples motivating optimization 2

Lemma 6.3.3 If $W_j(x') = 0$, then the states unique to c_j are not fair.

Proof Assume to the contrary that x_1 is unique to c_j, and is fair. Let $C = (x_1, \ldots, x_n, x_1)$ be a fair cycle containing x_1. Then, by lemma 6.2.3, there exists a projection of C, $P = \left(c_{i_1}, \ldots, c_{i_n}, c_{i_1}\right)$, which is a cycle and contains more than one node. Since x_1 is unique to c_j, $c_{i_1} = c_j$. Let $1 < l \leq n$ be the largest such that $c_{i_l} \neq c_j$. Then, $x_l \in c_{i_l}$, $x_{l+1} \in c_j$, and $x_{l+1} \notin c_{i_l}$. Therefore, the edge (x_l, x_{l+1}) gives rise to the edge $\left(c_{i_l}, x_{l+1}\right)$ in $S'(x, y)$, and to the node c_{i_l} in $W_j(x')$. This is in contradiction to $W_j(x') = 0$ (QED).

Let ***optimization 2*** be the deletion of states unique to c_j where $W_j(x') = 0$, and the application of the stable set operators in the second pass. Hence, pass 2 with optimization 2 applied is as follows.

II. Let $R(x', y') = 0$.

For each cycle set c_j

1. $W_j(x') = \exists y (S(x', y) \wedge c_j(y))$
2. *If* $W_j \neq 0$, $R(x', y') = R(x', y') + (W_j(x') \wedge (y' = j))$.
3. *If* $W_j = 0$, $A = A \wedge \overline{u_j}$ *where* $u_j = c_j \cap \overline{\bigcup_{i \neq j} c_i}$ *are the unique state of* c_j, *apply the forward and*

backward stable set operators to A , and let $T(x, y) = T(x, y) \wedge A(x) \wedge A(y)$.

Lemma 6.3.4 Applying optimization 2 can result in fake edges.

Proof Consider figure 6.7. In pass 2 of the algorithm, after c_2 is processed the edge $(1, 2)$ is added to $R(x', y')$. Then, in processing c_3, we find that $W_3(x') = 0$, so we can delete state 1 which is unique to c_3. State 2 cannot be reached by a cycle, and will be deleted by the stable set operators, which makes the edge $(1, 2)$ a fake edge (QED).

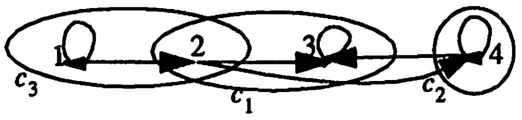


Figure 6.7: Fake edges after optimization 2

Consider figure 6.8. After the fairness graph is built, we notice that c_3 is not involved in any cycles of the fairness graph. The unique states of such a cycle set cannot be involved in any fair cycle, and can be deleted from the state graph (by the same reasoning as the proof of lemma 6.3.3). Let ***optimization*** *3* denote the deletion of unique states of such cycle sets and the application of the stable set operators, after the fairness graph is built.

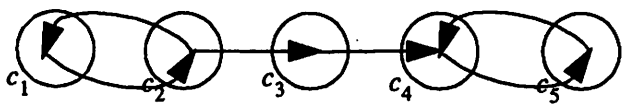


Figure 6.8: Example motivating optimization 3

Lemma 6.3.5 Application of optimization 3 can result in fake edges.

Proof Figure 6.9 gives an example, where the state graph Q is on the left and the fairness graph Q_f on the right. Optimization 3 deletes state 2. The stable set operators delete state 1, which in turn makes the edge (c_1, c_3) in Q_f fake (QED).

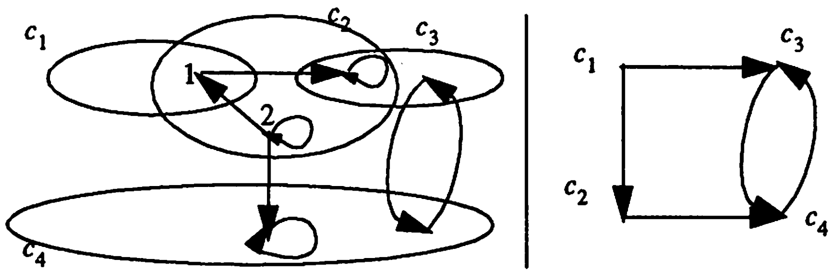


Figure 6.9: Fake edge after optimization 3

In summary, optimization 1 does not create fake edges, whereas optimizations 2 and 3 might. If optimizations 2 and 3 are included when the fairness graph is being built, two algorithms can be implemented to deal with fake edges. The first algorithm is as follows.

1. *Do the first pass with optimization 1.*
2. *Do the second pass with optimization 2. If any states are deleted, go back to step 1, and restart with those states deleted. Otherwise, go to step 3.*
3. *Perform optimization 3 on the fairness graph. If any states are deleted, go back to step 1, and restart the computation with those states deleted.*

Note that the first algorithm re-starts the computation as soon as any state is deleted by optimization 2 and 3. The second algorithm is given below.

1. *Do the first pass with optimization 1.*
2. *Do the second pass with optimization 2.*
3. *Perform optimization 3 on the fairness graph.*
4. *If any states are deleted in steps 2 or 3, go back to step 1 when all states deleted so far remain deleted.*

This algorithm carries out the computation to the end, ignoring possible fake edges. At the end, if there is a possibility of fake edges (because of deletion of states by optimizations 2 or 3), the computation is re-started.

6.4 Clustering Cycle Sets

In this section, we formulate the problem of clustering the cycle sets to minimize the number of fairness constraints. We show that the problem is NP-complete, and propose a heuristic solution.

6.4.1 The Clustering Problem

Definition An ***acyclic cluster*** is a set of vertices in Q_f which does not contain a cycle.

Theorem 6.3 Let $S_1, \ldots, S_p$ be a partition of vertices of Q_f such that every S_i is an acyclic cluster. Let Q' have the same states as Q, but with p cycle sets $c'_1, \ldots, c'_p$, where c'_i is formed by taking the union of all cycle sets in S_i. Then, a cycle is fair in Q iff it is fair in Q'.

Proof Let Γ be a fair cycle in Q. The only way Γ is not a fair cycle in Q' is for Γ to be entirely contained in some c'_i. Assume this is the case. Let $Q'' = c'_i$, and let the cycle sets in Q'' be the cycle sets in S_i. Γ is still a fair cycle in Q'', and by lemma 6.2.4, it has a shadow which is a cycle in Q''_f. But this cycle is also a cycle in Q_f, which is contradictory to the assumption that each S_i is an acyclic cluster. Conversely, assume Γ is a fair cycle in Q'. If Γ is not a fair cycle in Q, then Γ is contained entirely in some c_j. Let's say c_j is in some partition S_i. Since Γ is fair in Q', it is not entirely contained in S_i. Hence it cannot be contained in c_j (QED).

Note that the above theorem can be used to reduce the number of fairness constraints. We call this technique ***clustering***. The associated optimization problem is to decompose a directed graph (Q_f) into a minimum set of acyclic clusters. This problem is NP-complete, since it is equivalent to the *Partition into Forest* problem described in [GJ79]. We now show how this problem can be reduced to the decomposition of each strongly connected component (SCC) of Q_f.

Lemma 6.4.1 Let the SCC's of Q_f be given. Assume each is partitioned separately into a set of acyclic clusters. The union of a set of acyclic clusters each from different SCC's is again an acyclic cluster.

Proof This follows from the definition of SCC's (QED).

Theorem 6.4 Assume each SCC of Q_f can be decomposed into a minimum of m_i acyclic clusters. Let $M = max\{m_i\}$. Then, M is the solution of the clustering problem.

Proof Clearly by lemma 6.4.1, a solution with M clusters can be constructed by clusters with at most one element from each SCC. Now assume M is not minimum, and let a set of $M' < M$ acyclic clusters $S_1, \ldots, S_{M'}$ be given. We can build a clustering $W_1, \ldots, W_{M'}$ for each SCC W with at most M' clusters by taking $W_i = S_i \cap W$. This contradicts that m_i is a minimum solution for each SCC (QED).

The above theorem gives us a way of clustering vertices of Q_f by processing each SCC separately. Further, it tells us which SCC to work on the hardest.

Example Consider the example in figure 6.10, which shows a fairness graph with three SCC's: $\{1,2,3\}$, $\{4,5,6\}$, and $\{7,8,9,10\}$. The first SCC can be decomposed into $\{1, (2,3)\}$, the second into $\{4, (5,6)\}$, the third into $\{(7,8), (9,10)\}$. Thus, $M = 2$. The final decomposition is $\{(1,4,7,8), (2,3,5,6,9,10)\}$, and we have reduced the cycle sets from 10 to 2.

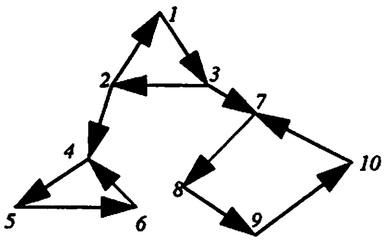


Figure 6.10: Clustering in fairness graphs

6.4.2 Heuristic Clustering

We use a simple recursive, greedy heuristic for the clustering of each SCC, which is described below.

1. *If two vertices do not form a cycle, merge them into a hyper node.*
2. *If any merges were performed in step 1, call the routine recursively on the graph formed by taking each cluster as a node.*

This algorithm allows many clusters to grow concurrently, as opposed of starting with one and continuously adding to it. Another heuristic is to maximally grow a cluster until no more nodes can be added, and then continue with thc next one. We propose to process the SCC's in decreasing size. Since we do not need to decompose a small SCC into less than $M = max\{m_i\}$ seen so far, we can use this as an early bounding step to speed up processing.

6.5 Experiments

We have implemented our algorithms in HSIS. To test our algorithms, we used an abstracted description of SUN's Sparc-8 memory model [SUN90], [DNP93]. The memory model is modeled abstractly as a single port device that grants write access to the attached processors by randomly selecting one.

For our experiments we mapped an assembly code sequence (from [SUN90]) that implements spin locking to an automaton representing a processor. To model general computation, both the automata for the processor and the memory have several states with self loops: the processor can stay in an idle state indefinitely long before attempting to acquire the lock; it can also wait indefinitely long returning to its computation loop, and stay in its critical section (after acquiring the lock) for an unspecified period of time. Each of these states becomes a one-element cycle set, since we want to exclude the behavior from the system where processors remain in one state forever. Similarly, we require that the memory does not always choose to service the same processor, i.e. the memory is "fair".

We verified the liveness property that a processor will eventually be able to acquire the lock, for systems with 3 and 4 processors. The results are summarized in table 1. All times are in seconds and are measured on a DECsystem 5000/260 with 128 MB of memory.

Table 6.1

num of processors	3	4
reachable states num	10685	143228
reachability	5.35	116.7
main comp. w/o f.g.	5.47	61.45
total time w/o f.g.	15.10	221.98
building the f.g	1.45	14.81
main comp. with f.g	0.0	0.0
total time with f.g.	11.11	175.1
improvement factor for main computation	3.77	4.15

Note that in this example, we were able to decide the language containment check by only analyzing the fairness graph. Hence, the main computation time with the fairness graph analysis is 0. The last row is obtained by taking the time of main computation without our technique, divided by the sum of building the fairness graph and the time for the main computation with our technique.

Acknowledgment

This material presented in this chapter is based mainly on [HMB94].

Chapter 7

Computing Fair Simulation Relations

7.1 Introduction

Simulation relations, which are approximations to language containment, come up in several places in verification.

1. *Non-deterministic property checking*. In property checking, simulation relations can be used in place of language containment. If the property is deterministic, then a simulation relation exists iff the language of the system is contained in the language of the property. In this case, there seems to be no computational advantage in using simulation relations instead of language containment. If the property is non-deterministic, then existence of a simulation relation implies that the language of the system is contained in the language of the property, although the converse does not hold. However, if the property is non-deterministic, and if determinization proves to be costly, then simulation relations may be an economical alternative.

2. *State Minimization*. Practical systems have many regular structures. Therefore, many states in the product machine of such systems are equivalent. Minimization reduces the size of a system by deleting equivalent states. Minimization does not change the behavior of a system. In an automaton A, if two states have exactly the same set of traces (i.e. are ***trace equivalent***), they can be merged. This process is known as state minimization. However, checking whether two states are trace equivalent is PSPACE-complete. As an approximation, simulation relations between an automaton and itself can be computed. If two states simulate each other, we have that they are trace equivalent, and can be merged.

3. *Hierarchical Verification*. Hierarchical verification is the process of verifying properties at a high-level of abstraction, then refining the abstract models according to some criteria, and being ensured that the properties still hold at lower levels of abstraction. In hierarchical verification, simulation relations can be used in two places. If ACTL properties are proved of the abstract model, then the existence of a simulation relation between the detailed model and the abstract one, guarantees that the properties continue to hold at lower levels of abstraction ([GL91]). If automata are used for property specification, we only need to prove that the lan-

guage of the detailed model is contained in the abstract model to ensure that the properties proved at the abstract levels continue to hold at detailed levels. If the abstract model is non-deterministic, then checking this language containment can be very expensive. In such case, it may be possible to prove the abstract model simulates the detailed model, which would imply the language of the detailed model is contained in the abstract one.

Simulation relations were first introduced in [Mil71], and have been used in formal verification in state minimization and property verification. Computing simulation relations in the presence of fairness constraints, referred to as fair simulation relations (FSRs), is a difficult and challenging problem. In the eSeR-environment, there are two situations where we need to know whether there is a FSR from A_2 to A_1. The first comes up in property verification, where A_1 is an edge-Streett automaton and A_2 is an edge-Rabin automaton. The second comes up in state minimization and hierarchical verification, where A_1 and A_2 are edge-Streett automata.

In this paper, we study a notion of equivalence for FSRs, called existential FSRs (EFSRs) introduced in [GL91], and argue that it is a natural notion for FSRs. However, the existence problem for EFSRs even for Buchi conditions is PSPACE-complete. We then define a weaker equivalence, called UFSRs, which is a generalization of "dynamic-live-cycles Buchi simulation relation" of [DHW91] to general fairness constraints. Using a result of [DHW91] we prove that if A_1 is edge-Streett, and A_2 is edge-Streett or edge-Rabin, then the problem of existence of a USFR from A_1 to A_2 is NP-complete. Finally, we give approximation algorithms, based on BDDs and language containment algorithms, for computing UFSRs for the two problems which come up in our environment.

The flow of the chapter is as follows. In section 7.2, the classical theory of simulation relations is reviewed. In section 7.3, we define EFSRs and UFSRs, and prove or review some results about them. In section 7.4, algorithms for computing UFSRs are given. Section 7.5 gives some comments on experiments.

7.2 Simulation Relations

In this section, we review the classical theory of simulation relations, known as safety simulation relations. Assume FSMs A_1 and A_2, with state spaces Q_1 and Q_2 are given.

Definition We say A_2 ***safely simulates*** A_1, or there is a ***safe simulation relation (SSR)*** from

A_2 to A_1, if there exists a relation $R(s_1, s_2)$ between A_1 and A_2 such that the following conditions hold.

1. ***(Transition Relation Condition)*** $\forall s_1 \in S_1, \forall s'_1 \in S_1, \forall \alpha \in \Sigma, \forall s_2 \in S_2$ if $T_1(s_1, \alpha, s'_1)$ and $R(s_1, s_2)$ hold, then $\exists s'_2 \in S_2$ such that $T_2(s_2, \alpha, s'_2)$ and $R(s'_1, s'_2)$ hold.

2. ***(Initial State Condition)*** $\forall s_1 \in I_1, \exists s_2 \in I_2$ such that $R(s_1, s_2)$ holds.

Intuitively, condition 1 states that if A_1 can do a move, A_2 is able to match it. Condition 2 states that each initial state of A_1 is simulated by some initial state of A_2.

Definition Let $P = A_1 \times A_2$ denote the product machine of A_1 and A_2. The ***product relation*** R_P is defined by the reachable states of P, i.e. $R_P(s_1, s_2)$ iff (s_1, s_2) is reachable in P.

7.2.1 Safety Simulation Relations and Language Containment

In this section, the relationship between SSRs and language containment (LC) is studied. Most of these results are known in the literature. Unless noted otherwise, the results apply to languages on both finite and infinite strings. There are no fairness constraints.

Definition Let A_2 safely simulate A_1 by the SSR R. Let string σ have a run r_1 in A_1. r_2 is a ***simulating run of*** r_1 ***on*** σ ***in*** A_2 if:

a. $r_2(0)$ is an initial state of A_2, where $r(i)$ denotes the i-th state of r.

b. For all i, $(r_1(i), r_2(i)) \in R$, and there is a transition $(r_2(i), \sigma(i), r_2(i+1))$ in A_2.

Lemma 7.2.1 (easy) If A_2 safely simulates A_1, then for every run in A_1, there is a simulating run in A_2.

The following two lemmas show that although existence of SSRs implies language containment, the converse is not true.

Lemma 7.2.2 (literature) If A_2 safely simulates A_1, $L(A_1) \subseteq L(A_2)$.

Proof Let r_1 be a run for $\sigma \in L(A_1)$. Then, a simulating run of σ in A_2 is an accepting run for σ in A_2. Hence, $\sigma \in L(A_2)$ (QED).

Lemma 7.2.3 (literature) If A_2 is non-deterministic and $L(A_1) \subseteq L(A_2)$, there may be no simulation relation.

Proof The following is an example.

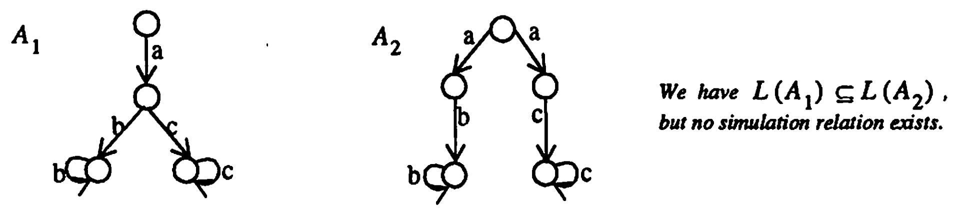

A_1
a
b
c
b
c
A_2
a
a
b
c
b
c
We have $L(A_1) \subseteq L(A_2)$,
but no simulation relation exists.

Figure 7.1: Simulation relation versus language containment

The following two lemmas will be used in constructing algorithms for computing SSRs.

Lemma 7.2.4 ([DHW91]) Let $L(A_1) \subseteq L(A_2)$, A_1 be ***non-deadlocking*** (i.e. all states have some outgoing edge), and A_2 be deterministic. Let $P = A_1 \times A_2$. Then, the product relation R_P is a SSR from A_2 to A_1.

Proof The initial state condition is trivially satisfied, since every initial state of A_1 is related to the initial state of A_2. Let (s_1, s_2) be reachable in P, and $T_1(s_1, \alpha, s'_1)$ hold. To prove the transition relation condition holds, we will show there exists s_2' such that (s_1', s_2') is reachable in P and $T(s_2, \alpha, s_2')$ holds. Let $\hat{r}$ be a path in P from some initial state to (s_1, s_2). Let $\hat{\sigma}$ be a string corresponding to $\hat{r}$. Let $\hat{r}_1$ be the projection of $\hat{r}$ in A_1. Since A_1 is non-deadlocking, complete $\hat{r}_1$ to an infinite run r_1 which coincides with $\hat{r}_1$ initially, goes through the edge (s_1, s'_1), and does something arbitrary after that. Let σ coincide with $\hat{\sigma}$ initially, take the value α, and then take values consistent with the run r_1. We have that r_1 is a run for σ. By $L(A_1) \subseteq L(A_2)$ and determinism of A_2, there is a unique run in A_2 for σ. This run coincides with the projection of $\hat{r}$ initially, and goes through some state s'_2 such that $T_2(s_2, \alpha, s'_2)$. We have that (s'_1, s'_2) is reachable in P, and $T(s_2, \alpha, s_2')$ holds, as was desired (QED).

Lemma 7.2.5 Let A_1 be non-deadlocking. A pair of states (x, y) cannot belong to any simulation relation from A_2 to A_1 if there is some finite string σ which has a run starting at x in A_1, but no run starting at y in A_2.

Proof Assume to the contrary that there is a SSR R from A_2 to A_1 which relates x to y. Form automata A'_1 and A'_2, with the same transition structures as A_1 and A_2, but with initial states x and y, respectively. Then R is also a SSR from A'_1 to A'_2. By lemma 7.2.2, $L(A'_1) \subseteq L(A'_2)$. But this is a contradiction to the assumption that there is some string which has a run from x in A_1, but no run from y in A_2 (QED).

7.2.2 Computing Safety Simulation Relations

Lemma 7.2.6 (literature) Safety simulation relations are closed under union.

Proof Let R' and R'' be SSRs. Let $R = R' \cup R''$. We need to show R is a SSR. Let x be an initial state of A_1. Then, since R' is a SSR, there exists y an initial state of A_2 such that $R'(x, y)$ holds. Since $R' \subseteq R$, we have that $R(x, y)$ holds also. Hence, R satisfies the initial state condition. To show the transition relation condition holds, let $R(s_1, s_2)$ and $T(s_1, \alpha, s'_1)$ hold. Since $R = R' \cup R''$, we have $(s_1, s_2) \in R'$ or $(s_1, s_2) \in R''$. Without loss of generality, assume $(s_1, s_2) \in R'$. Then, there exists s'_2 such that $(s'_1, s'_2) \in R'$, and $T(s_2, \alpha, s'_2)$ holds. Since $R' \subseteq R$, we have $R(s'_1, s'_2)$ and $T(s_2, \alpha, s'_2)$ hold, as was desired (QED).

Definition Let E_0 be a relation on state spaces of A_1 and A_2. The relation computed by the fixpoint $E_{i+1}(x, y) = \forall\alpha\forall z((E_i(x, y) \wedge T_1(x, \alpha, z)) \rightarrow \exists w(T_2(y, \alpha, w) \wedge E_i(z, w)))$ is the ***refinement of*** E_0 ***by the transition relation condition***.

The following algorithm finds the largest SSR from A_2 to A_1 (existence of "the largest SSR" is implied by lemma 7.2.6).

1. Let $E_0(x, y) = 1$, i.e. all states of A_1 are related to all states of A_2.

2. Let E be the refinement of E_0 by the transition relation condition.

3. Return E if it satisfies the initial state condition. Otherwise return the empty set.

The following two lemmas prove the correctness of the algorithm.

Lemma 7.2.7 (literature) Let $A_{1,x}$ and $A_{2,y}$ denote A_1 and A_2 with x and y as initial states respectively. Let E be the relation computed in step 2 of the above algorithm. Then, $(x, y) \in E$ iff $A_{2,y}$ safely simulates $A_{1,x}$, which implies the above algorithm computes a safety simulation relation.

Proof If $(x, y) \in E$, then both conditions of a SSR are satisfied. Hence, $A_{2,y}$ safely simulates $A_{1,x}$. Conversely, assume $(x, y) \notin E$, and let E_m be the relation from which (x, y) is first deleted. We prove by induction on m that if (x, y) is deleted from E_m then there is no SSR from $A_{2,y}$ to $A_{1,x}$. If $m = 1$, (x, y) is deleted from E_1 since there exists some symbol α such that only one of x and y has a transition on α. Hence, (x, y) cannot belong to any SSRs. Now, assume the claim holds for all pairs deleted up to iteration m. (x, y) is deleted from E_m since there is x', next state of x on some symbol α, such that no next states of y on α is related to x'. If some SSR relates x and y, then some next state of y on α can safely simulate x'. By inductive assumption, this is not the case (QED).

Lemma 7.2.8 (literature) The above algorithm computes the largest SSR.

Proof By lemma 7.2.7, we know that the computed relation is a SSR. To show it is the largest one, assume to the contrary that E' is a SSR, $(x, y) \in E'$, and $(x, y) \notin E$. $(x, y) \in E'$ implies that $A_{2,y}$ safely simulates $A_{1,x}$. But $(x, y) \notin E$ is in contradiction with lemma 7.2.7 (QED).

Consider the example of figure 7.2. The above algorithm will take 5 iterations to find the SSR $\{(0, 0'), (1, 1')\}$. One can speed up the above algorithm on this example by letting E_0 be the product relation. The computed relation is called the ***product safety simulation relation (PSSR)***. For this example, step 2 of the above algorithm when E_o is the product relation takes only one iteration. This speedup is achieved by not considering the un-reachable states in the product graph which slow down the convergence of the original algorithm.

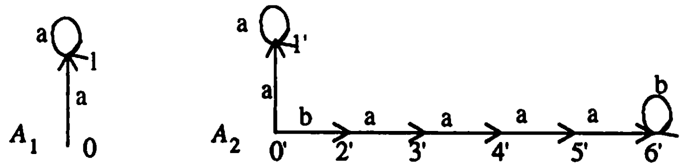


Figure 7.2: An inefficiency in computing SSRs

Definition Let R be a SSR from A_2 to A_1. The ***product graph induced by*** R, G_R, is defined to be the set of reachable states of the graph $\tilde{G}_R$ whose state space is R, whose initial states are all $(x, y) \in R$ such that x is an initial state of A_1 and y is an initial state of A_2, and there is a transition $((s_1, s_2), \alpha, (s'_1, s'_2))$ in $\tilde{G}_R$ if (s_1, α, s'_1) is an edge in A_1 and (s_2, α, s'_2) is an edge in A_2.

Lemma 7.2.9 Let R be a SSR from A_2 to A_1. Then $G_R \subseteq R_P$, i.e. the product graph induced by R is a subset of the product relation.

Proof First, note that if $(x, y) \in G_R$ and $(x, y) \in R_P$, then all outgoing edges of (x, y) in G_R are contained in R_P. The lemma follows by induction since all initial states of G_R are in R_P (QED).

Lemma 7.2.10 The PSSR exists iff there is a SSR from A_2 to A_1.

Proof Since the PSSR is a SSR, we only need to show that if a SSR R exists, then the PSSR S also exists. By lemma 7.2.9, we have $G_R \subseteq R_P$. Since all pairs in G_R satisfy the transition relation condition, we have $G_R \subseteq S$, i.e. they will not get deleted by step 2 of the algorithm. It follows that the PSSR S exists (QED).

7.2.3 Complexity of Checking and Computing

In this subsection, we give some upper bounds on the complexity of checking and computing SSRs. These bounds are not meant to be tight. In what follows, let n_1, m_1, n_2, m_2 be the number of nodes and edges of A_1 and A_2, respectively, and $|\Sigma|$ be the size of the alphabet. We assume $m_1 = O(n_1)$ and $m_2 = O(n_2)$ (note that we are not ruling out that m_1 and m_2 can be $\theta(n_1)$ and $\theta(n_2)$ respectively).

Lemma 7.2.11 Checking whether a given relation R is a SSR from A_2 to A_1 can be done in $O(m_1 |\Sigma| m_2)$.

Proof The following algorithm does the job.

```
1. For each edge $(u_1, v_1)$ in $A_1$ and symbol $\alpha$,
   2. For each node $u_2 \in A_2$ such that $R(u_1, u_2)$ holds,
         Ensure that there is an edge $(u_2, v_2)$ in $A_2$ on $\alpha$ such that $R(v_1, v_2)$ holds.
```

Note that given an edge (u_1, v_1) and symbol α, step 2 at most visits all edges of A_2. Assuming that checking whether $R(u, v)$ holds can be done in constant time, the lemma follows (QED).

Lemma 7.2.12 Complexity of finding the largest SSR is $O\left(|\Sigma| n_1 n_2 m_1 m_2\right)$.

Proof The algorithm for checking a SSR identifies all pairs which do not satisfy the transition relation condition. The number of times this procedure is called is bounded by $O(n_1 n_2)$, which is the number of pairs in E_0. The result follows by lemma 7.2.11. (QED)

Remark The algorithms of subsection 7.2.2 can be described by μ-calculus formulas. For example, $\mu((X_1, X_2), (I_1, I_2)).F((X_1, X_2))$ computes the product transition relation, where $F((X_1, X_2))$ is the monotone increasing predicate transformer $(X_1, X_2) + R_1((X_1, X_2), y)$, and $R_1((X_1, X_2), y)$ represents the set of next states of (X_1, X_2) in the product graph (see section 4.3 for notations of fixpoint operators). Similarly, one can define a fixpoint expression for the refinement of a relation by the transition relation condition. BDDs can be used to compute a relation described by a μ-calculus formula on finite graphs. [HTKB92] introduced a complexity class for such computations, which is roughly based on how many sets of state variables the relation being computed in a fixpoint expression depends on, and how many nested levels of fixpoint operations the expressions contains. According to this classification, the BDD complexity of these computations is $BDD_{1,1}$ for $A_1 \times A_2$, which says the relations being computed in the fixpoint expressions for these computations involve one set of state variables, and have one nested level of fixpoints. This is the same BDD complexity as performing reachability on $A_1 \times A_2$.

7.2.4 Output Determinization

In this section, we describe a language preserving transformation which increases the chance of existence of a SSR, at a slight computation time increase.

Definition Let a safety automaton $A = (\Sigma, Q, T, I)$ be given. Let $A^{OD} = (\Sigma, Q^{OD}, T^{OD}, I^{OD})$, the ***output determinization*** of A, be defined as follows. Let $Q^{OD} \subseteq Q \times \Sigma$, and be defined by $(q, a) \in Q^{OD}$ if $q \in Q$ and there is $q' \in Q$ such that $(q, a, q') \in T$. If $(q, a) \in Q^{OD}$, and $q \in I$, then $(q, a) \in I^{OD}$. Let $((q, a), a, (q', b)) \in T^{OD}$ if $(q, a) \in Q^{OD}$, $(q', b) \in Q^{OD}$, and $(q, a, q') \in T$. Note that all transitions from (q, a) are marked with a.

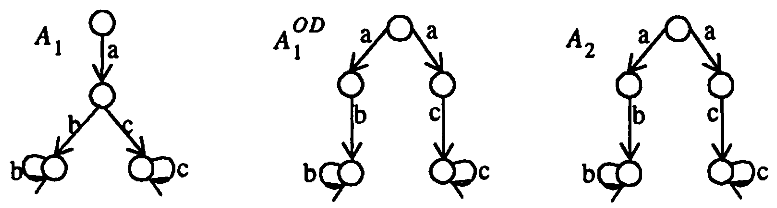
The following lemmas show that by output determinizing A_1, and then looking for a simulation relation from A_2 to A_1^{OD}, one can answer the question of $L(A_1) \subseteq L(A_2)$ more accurately. However, the extra complexity may not justify the improved accuracy. Incidentally, [DHW91] requires both A_1 and A_2 to be output deterministic.

Lemma 7.2.13 $L(A) = L(A^{OD})$, hence output determinization is language preserving.

Proof The lemma follows by noticing string $\sigma = \sigma_0, \sigma_1, \ldots$ has a run $r = r_0, r_1, \ldots$ in A iff σ has a run $(r_0, \sigma_0), (r_1, \sigma_1), \ldots$ in A^{OD} (QED).

Lemma 7.2.14 A_2 safely simulates A_1 implies A_2 safely simulates A_1^{OD}. However, the converse is not true.

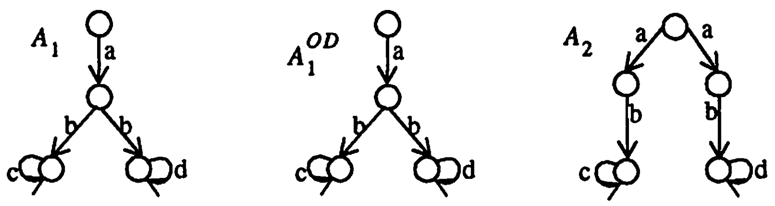
Proof Let R be a SSR from A_2 to A_1. If $(q_1, q_2) \in R$, let $((q_1, a), q_2) \in R^{OD}$ for each symbol a which is the label of some outgoing edge from q_1. It is easy to check that R^{OD} is a SSR from A_2 to A_1^{OD}. The example in figure 7.3 shows that the converse is not true, i.e. if $A2$ safely simulates A_1^{OD}, we cannot deduce A_2 safely simulates A_1 (QED).



Although A_2 does not safely simulate A_1, but after output determinizing A_1, A_2 safely simulates A_1^{OD}.

Figure 7.3: Output determinization and finding SR's

As figure 7.4 shows, it is easy to find examples of where $L(A_1) \subseteq L(A_2)$ but no SSR from A_2 to A_1^{OD} exists. The following lemma shows that output determinizing A_2 does not increase the chance of finding a SSR from A_2 to A_1.



$L(A_1)$ is contained in $L(A_2)$, but no SSR from A_2 to A_1 or A_1^{OD} exists.

Figure 7.4: Language containment vs. SSRs in output determinized FSMs

Lemma 7.2.15 A_2^{OD} safely simulates A_1^{OD} iff A_2 safely simulates A_1^{OD}.

Proof Let R be a SSR from A_2^{OD} to A_1^{OD}. Define R', a SSR from A_2 to A_1^{OD}, as follows. If $((q,a),(q',a)) \in R$, let $((q,a),q') \in R'$. Conversely, let R' be a SSR from A_2 to A_1^{OD}. Define R, a SSR from A_2^{OD} to A_1^{OD}, as follows. If $((q,a),q') \in R'$, let $((q,a),(q',a)) \in R$. Note that if $((q,a),q') \in R'$, then there is an outgoing edge from q' which is labeled by a (QED).

Lemma 7.2.16 The complexity of checking a SSR from A_2 to A_1^{OD} is $O(m_1|\Sigma|m_2)$.

Proof Apply the algorithm of lemma 7.2.11. Every edge of A_1^{OD} will be visited once since each such edge is labeled by exactly one symbol. There are $O(m_1|\Sigma|)$ edges in A_1^{OD}. Since step 2 takes $O(m_2)$ time, the lemma follows (QED).

Lemma 7.2.17 The complexity of checking a SSR from A_2 to A_1^{OD} is $O\left(|\Sigma|^2 n_1 n_2 m_1 m_2\right)$.

Proof Follows from lemma 7.2.12 by noting the number of nodes of A_1^{OD} is $|\Sigma|n_1$ (QED).

7.3 Fair Simulation Relations

In this section, we introduce simulation relations in the presence of fairness constraints. Our definitions are general and apply to any fairness condition. However, the complexity of the existence problem, which we also call the computation problem, depends on the type of the fairness constraint. Our first definition, called existential fair simulation relation (EFSR), makes the computation problem even for Buchi conditions PSPACE-hard, whereas the second definition, called universal fair simulation relations (UFSR), makes it NP-complete for edge-Streett conditions (which include Buchi conditions as a subset). EFSRs first appeared in [GL91], whereas UFSRs were defined originally in [DHW91] for Buchi automata, and were called dynamic-live-cycles Buchi simulation relations.

Both definitions have the property that if A_2 is deterministic and $L(A_1) \subseteq L(A_2)$, then a fair simulation relation from A_2 to A_1 exists. In what follows, we assume that from all states of A_2, there exists a fair path. We call such automata ***reduced***. This assumption is needed to prove the above property of our definitions. This is a minor restriction, since most systems are expected to satisfy this condition, and if not, language containment algorithms of chapter 4 can be used to easily find this set, and restrict the automata to this set. This operation does not change the language (i.e. behavior) of the system.

7.3.1 Preliminary Definitions

Definition A ***Buchi*** automaton A is a FSM with one fairness constraint of the form $F^{\infty}(S)$. The set S is called the set of final states of A. An ***ω-automaton*** denotes an edge-Streett, edge-Rabin or Buchi automaton.

Definition A fair constraint G is said to be the ***complement*** of fair constraint F iff whenever a run is accepting according to G it is not accepting according to F.

Lemma 7.3.1 (easy) The complement of a set of positive edges is a set of negative edges. The complement of $F^{\infty}(S) + G^{\infty}(T)$ is $G^{\infty}(\bar{S}) \wedge F^{\infty}(\bar{T})$. The complement of $F^{\infty}(S) \wedge G^{\infty}(T)$ is $G^{\infty}(\bar{S}) + F^{\infty}(\bar{T})$.

Lemma 7.3.2 Let A_1 be an edge-Streett automaton. Let A_2 be an edge-Rabin or edge-Streett automaton. Let A have the transition structure of $A_1 \times A_2$, and the fairness constraint $F_1 \wedge \overline{F_2}$. Then, the sets $Fair$ ($Fair+$, and $Fair\pm$) of A can be computed in poly-time.

Proof If A_2 is edge-Rabin, then A is just an edge-Streett automaton, and the lemma follows. If A_2 is edge-Streett, then build B_i's which each have the same transition structure as $A_1 \times A_2$, and the fairness constraint $F_1 \wedge \overline{F_2^i}$, where F_2^i is the i-th fairness constraint of A_2. The $Fair$ set of A is the union of the $Fair$ set of B_i's (similarly for $Fair+$ and $Fair\pm$) (QED).

Definition Let A^s be A without any fairness constraints, i.e. every run is accepting (a safety automaton).

We will now prove that if A_1 is reduced and $L(A_1) \subseteq L(A_2)$, then $L\left(A_1^{\,s}\right) \subseteq L\left(A_2^{\,s}\right)$. This result will be used in later sections.

Lemma 7.3.3 (easy) Every run in a reduced automaton can be extended to a fair run.

Lemma 7.3.4 (easy) A reduced automaton is non-deadlocking.

Lemma 7.3.5 If an ω-string σ does not have a run in a safety automaton A, then there exists an n such that $\sigma_1, \ldots, \sigma_n$ does not have a run in A.

Proof Assume no such n exists i.e. for all n, $\sigma_1, \ldots, \sigma_n$ has a run. Consider the tree of runs of σ in A, denoted by T_σ, where the root of the tree is the initial state of A, and there is an edge between nodes u and v at levels i and $i+1$ of the tree if there is a transition (u, v) in A on σ_i. Then, T_σ is unbounded, i.e. infinite. Since T_σ is infinite and finite branching, by Konig's lemma, there is an infinite path in T_σ. This path is an infinite run for σ in A. We have reached a contradiction (QED).

Lemma 7.3.6 If A_1 is reduced and $L(A_1) \subseteq L(A_2)$, then $L\left(A_1^{\ s}\right) \subseteq L\left(A_2^{\ s}\right)$.

Proof Assume there is a string σ which has a run r in $A_1^{\ s}$, but no run in $A_2^{\ s}$. By lemma 7.3.5, let n be such that $\sigma_1, \ldots, \sigma_n$ has no run in $A_2^{\ s}$. Since σ has a run in A_1, and since A_1 is reduced, $\sigma_1, \ldots, \sigma_n$ can be extended to an infinite string $\hat{\sigma}$ which is accepted by A_1. Since $L(A_1) \subseteq L(A_2)$, $\hat{\sigma}$ has an accepting run in A_2, and hence a run in $A_2^{\ s}$, contradicting that $\sigma_1, \ldots, \sigma_n$ has no run in $A_2^{\ s}$ (QED).

7.3.2 Existential Fair Simulation Relations (EFSR's)

Definition Assume automata A_1 and A_2 having fairness constraints are given. A_2 is said to ***existentially fair simulate*** A_1 (A_2 *ESF* A_1) if there exists a SSR R from S_2 to S_1 which satisfies the following additional condition:

Existential fair condition: For every accepting (fair) run $r = r_0, r_1, \ldots$ of string σ in A_1, there exists an accepting (fair) simulating run $r' = r'_0, r'_1, \ldots$ of r in A_2, i.e. $\forall i ((r_i, r'_i) \in R)$, and r_0 and r'_0 are initial states in A_1 and A_2 respectively.

EFSRs appear to be the natural extension of SSR's to fairness constraints. One indication is that there is a SSR from A_2 to A_1 iff for every run of every string in A_1 there exists some simulating run in A_2. Since for safety automata, every run is accepting (fair), then in the presence of fairness constraints, we expect the result to be "there exists a fair simulation relation from A_2 to A_1 iff for every accepting run of every string in A_1 there exists some accepting simulating run in A_2." This statement is true when EFSRs are used and A_1 is reduced. The following lemmas provide additional indications that EFSRs are the natural extension of SSR's.

Lemma 7.3.7 ([GL91]) EFSRs are closed under union.

Proof Let R' and R'' be two EFSRs from A_2 to A_1. Let $R = R' \cup R''$. R is a SSR since SSRs are closed under union, and we know R and R' are SSRs. Let r be an accepting run of A_1 on σ. Since R' is an EFSR, there is a fair simulating path r' in A_2 with respect to R'. Since $R' \subseteq R$, r' is also a fair simulating path with respect to R (QED).

Remark Note that by the proof of lemma 7.3.7, the union of an EFSR and a SSR is still an EFSR.

Lemma 7.3.8 If A_2 existentially fair simulates A_1, then $L(A_1) \subseteq L(A_2)$.

Proof Clear from definitions (QED).

Lemma 7.3.9 If A_1 is reduced, $L(A_1) \subseteq L(A_2)$, and A_2 is deterministic, then the product relation R_P defines an EFSR.

Proof By lemma 7.3.6, $L\left(A_1^s\right) \subseteq L\left(A_2^s\right)$. By lemma 7.3.4 A_1^s is non-deadlocking. By lemma 7.2.4, R_P is a SSR from A_2^s to A_1^s, which implies it is a SSR from A_2 to A_1. To see R_P satisfies the existential fair condition, assume σ has an accepting run r_1 in A_1. Since $L(A_1) \subseteq L(A_2)$ and A_2 is deterministic, σ has a unique accepting run r_2 in A_2. Since A_2 safely simulates A_1, every run of a string in A_1 has a simulating run in A_2. Since σ has a unique accepting run r_2 in A_2, r_2 is a simulating run. We conclude r_2 is a fair simulating run, as was desired (QED).

Lemma 7.3.10 If $L(A_1) \subseteq L(A_2)$, A_2 is deterministic, but A_1 is not reduced, then there may be no EFSRs.

Proof Let A_1 be a one-state automaton, with a self-loop on some symbol $a \in \Sigma$, and no fairness constraints (implying $L(A_1) = \emptyset$). Let A_2 be a one-state automaton with no edges. We have that $L(A_1) = L(A_2) = \emptyset$, but there are no SSRs, and hence no EFSRs from A_2 to A_1 (QED).

Lemma 7.3.11 Checking an EFSR is PSPACE-hard for Buchi conditions.

Proof We reduce universality of ω-automata (i.e. that all strings are accepted), which was proved PSPACE-complete in [SVW87] to our problem. Assume Buchi automaton A is given. Complete A by having undefined transitions go to a dead state. Call this automaton A_2. Let A_1 be the following one-state automaton which accepts Σ^{ω}.

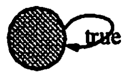


Figure 7.5: One state Buchi automaton accepting all strings

Let R be the relation where every state in A_2 is related to the single state of A_1. R is a SSR, and it satisfies the existential fair condition (i.e. it is an EFSR) iff $L(A_2) = \Sigma^{\omega}$. We have thus reduced universality of Buchi automata to checking EFSRs (QED).

Lemma 7.3.12 Existence of an EFSR is PSPACE-hard for Buchi conditions

Proof The same reduction as the above lemma can be used here as well (QED).

The question of whether checking or finding EFSRs is in PSPACE depends on the type of the fairness constraints. Since all relations between A_2 and A_1 can be generated in PSPACE, and since checking whether a relation is a SSR can be done in polynomial time (and space), the existence of an EFSR can be decided in PSPACE if the existential fair condition can be checked in PSPACE. Lemma 7.3.13 shows that checking this condition can be reduced to a language containment check.

Lemma 7.3.13 Let relation R be a SSR from A_2 to A_1. Then, there are automata $\tilde{A}_1$ and $\tilde{A}_2$, with the same transition relation and fairness constraints as A_2 and A_1 but with polynomially larger alphabet, such that R satisfies the existential fair condition iff $L(\tilde{A}_1) \subseteq L(\tilde{A}_2)$.

Proof Let $\tilde{\Sigma} = Q_1 \times \Sigma$. If (q, a, q') is an edge of A_1 create the edge $(q, (q, a), q')$ in $\tilde{A}_1$. If (p, a, p') is an edge of A_2 and $R(q, p)$ holds (p simulates q), create the edge $(p, (q, a), p')$ in $\tilde{A}_2$. Assume $L(\tilde{A}_1) \not\subseteq L(\tilde{A}_2)$. We need to show that R does not satisfy the existential fair condition. Let $\tilde{\sigma} = ((q_0, a_0), (q_1, a_1), \ldots)$, $q_i \in A_1$, $\tilde{\sigma} \in L(\tilde{A}_1)$ and $\tilde{\sigma} \notin L(\tilde{A}_2)$. Consider $\sigma = (a_0, a_1, \ldots)$, with a fair run $r = (q_0, q_1, \ldots)$ in A_1. If r has a fair simulating run in A_2, then $\tilde{\sigma} \in L(\tilde{A}_2)$, which is a contradiction. Conversely, assume R does not satisfy the existential fair condition. Then, there exists $\sigma = (a_0, a_1, \ldots)$ with a fair run $r = (q_0, q_1, \ldots)$ in A_1, but no simulating runs in A_2. Let $\tilde{\sigma} = ((q_0, a_0), (q_1, a_1), \ldots)$. We have that $\tilde{\sigma} \in L(\tilde{A}_1)$. If $\tilde{\sigma} \in L(\tilde{A}_2)$, then there is a fair run $(p_0, p_1, \ldots)$ in $\tilde{A}_2$ on $\tilde{\sigma}$. So $(p_0, p_1, \ldots)$ is a fair run for σ in A_2, and it simulates $(q_0, q_1, \ldots)$, contradicting the assumption that $(q_0, q_1, \ldots)$ has no fair simulating runs in A_2. We conclude $\tilde{\sigma} \notin L(\tilde{A}_2)$, and have $L(\tilde{A}_1) \not\subset L(\tilde{A}_2)$, as was desired (QED).

Theorem 7.1 The existence and checking problems of EFSRs for Buchi conditions are PSPACE-complete.

Proof Follows from lemmas 7.3.11, 7.3.12, and 7.3.13, and the fact that the language containment problem for Buchi automata can be decided in PSPACE ([SVW87]).

7.3.3 Universal Fair Simulation Relations (UFSR's)

We define universal fair simulation relations, which are generalizations to general fairness constraints of the notion of "dynamic-live-cycles Buchi simulation relations" defined for Buchi automata in [DHW91].

Definition Assume automata A_1 and A_2 having fairness constraints are given. A_2 is said to ***universally fair simulate*** A_1 (A_2 UFS A_1) if there exists a SSR R from S_2 to S_1, and satisfying the following additional condition:

Universal fair condition: For every accepting (fair) run r of string σ in A_1, all simulating runs of r in A_2 are accepting (fair).

Lemma 7.3.14 (easy) If A_2 UFS A_1, then $L(A_1) \subseteq L(A_2)$.

Lemma 7.3.15 If $L(A_1) \subseteq L(A_2)$, A_2 is deterministic, and A_1 is reduced, then a UFSR exists[1].

Proof By lemma 7.3.9, an EFSR exists. Since A_2 is deterministic, this EFSR is a UFSR (QED).

Lemma 7.3.16 ([DHW91]) UFSRs are not closed under union. Hence there may be no maximal UFSR.

Proof Figure 7.6 provides a counter-example, where A and A' are two Buchi automata (the

1. [DHW91] proved a similar theorem for their equivalence relation (dynamic-live-cycles Buchi simulation relations). However, the requirement that A_1 be reduced, was not stated. If A_1 is not reduced, the theorem is invalid, as shown by the counter-example of lemma 7.3.10.

shaded states are final states), having exactly the same transition graph. All edges have the same label a, and the initial states are 0 and 0'. The language of both automata is the string a^{ω}. Let $R = \{(0,0'), (1,1'), (2,2'), (3,1'), (4,2')\}$, and $R' = \{(0,0'), (1,3'), (2,4'), (3,3'), (4,4')\}$ be two UFSRs. $R \cup R'$ is not a USFR, since the run $\overline{(0,1,4)}$ (the cross bar denotes taking the set of states infinitely often) is fair in A but one of its simulating runs $\overline{(0',1',4')}$ is not fair in A' (QED).

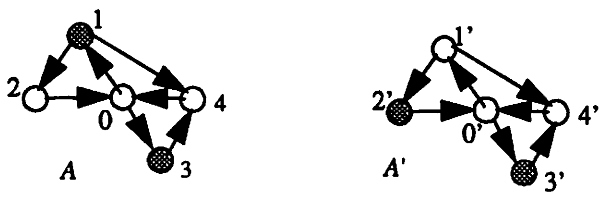


Figure 7.6: UFSR's are not closed under union

Lemma 7.3.17 Let A_1 and A_2 with fairness constraints F_1 and F_2 be given. Let R be a SSR from A_2 to $A1$. Let P be G_R (the product graph induced by R) with fairness constraints $F_1 \wedge \overline{F_2}$. Then, R satisfies the universal fair condition iff there are no fair (or fair reachable) states in P.

Proof Assume there is some fair state in P. That means there is a string σ with a run r in P, such that the projection of r to A_1, r_1, satisfies F_1, and the projection of r to A_2, r_2, does not satisfy F_2. Since r_2 is a simulating run for r_1, the universal fair condition is violated. Conversely, assume there are no fair states, and assume to the contrary that the universal fair condition is violated. Then, there is a run r_1 in A_1 on some string σ and a corresponding simulating run r_2 such that r_1 satisfies F_1 and r_2 does not satisfy F_2. The run r in P whose projections are r_1 and r_2 is a fair run in P, contradicting that there are some fair states in P (QED).

Lemma 7.3.18 If A_1 is edge-Streett, and A_2 is edge-Streett or edge-Rabin, then checking whether a relation R is a USFR from A_2 to A_1 can be done in polynomial time.

Proof We know that safe simulation conditions can be checked in polynomial time. Let P be as in lemma 7.3.17. By lemma 7.3.2, the fair reachable states of P can be computed in polynomial time in the size of P. But, the size of P is polynomial in the sizes of A_1 and A_2. Hence, by lemma 7.3.17, and the fact that computing the fair states of P is polynomial, the universal fair condition can also be checked in polynomial time (QED).

Lemma 7.3.19 If A_1 is edge-Streett, and A_2 is edge-Streett or edge-Rabin, then the problem of existence of a USFR is NP-complete.

Proof It is easy to see that the problem is in NP: just guess the relation, and check it (which takes polynomial time by lemma 7.3.18). To show the problem is NP-complete, we reduce 3-SAT to it[1]. Assume m clauses $C_1, \ldots, C_m$, and n variables $x_1, \ldots, x_n$ are given. Assume each variable is used in some clause. Build Buchi automata A_1 and A_2 as follows (figure 7.7).

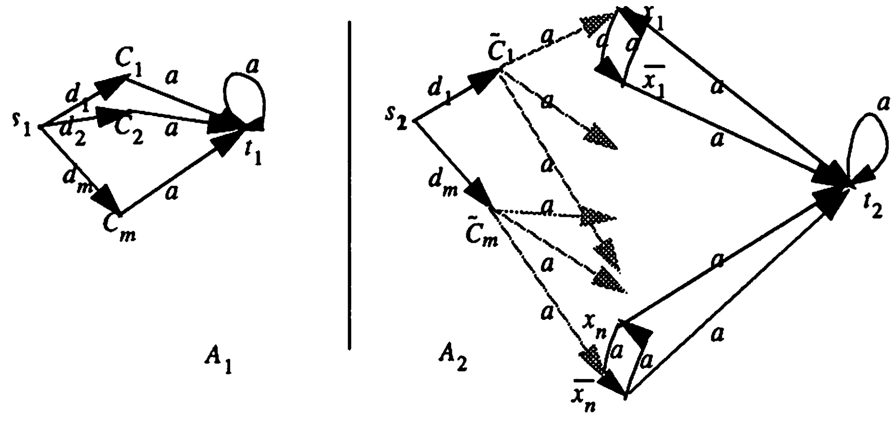


Figure 7.7: Reduction from SAT to finding UFSR's

The alphabet is $\Sigma = \{a, d_1, \ldots, d_m\}$, which are all newly introduced constants. The edges of A_1 are (s_1, d_i, C_i) and (C_i, a, t_1) for $1 \le i \le m$, and the edge (t_1, a, t_1). The edges of A_2 are $(s_2, d_i, \tilde{C}_i)$ for $1 \le i \le m$, from each $\tilde{C}_i$ to the corresponding literals in clause C_i (so there are three such edges for each $\tilde{C}_i$), the edges $(x_i, a, \bar{x}_i)$, $(\bar{x}_i, a, x_i)$, (x_i, a, t_2) and $(\bar{x}_i, a, t_2)$ for $1 \le j \le m$, and the edge (t_2, a, t_2). The final (accepting) state of A_1 is t_1, and that of A_2 is t_2. Several remarks are in order.

1. s_1 is matched to s_2 and C_i is matched to $\tilde{C}_i$ in every UFSR. If x_i and $\bar{x}_i$ are matched to anything, they are matched to t_1, but they may be unmatched.

1. This reduction appeared in an un-published version of [DHW91].

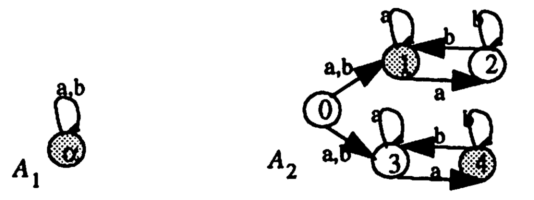
2. x_i and $\bar{x}_i$ are not both matched with t_1, because otherwise the cycle $(s_1, C_i, t_1, t_1, \ldots)$ can be matched by $s_1, \tilde{C}_i, x_i, \bar{x}_i, x_i, \bar{x}_i, \ldots$, where the first is accepting but not the second.

3. From remark 2, it follows that t_2 is matched with t_1.

Let a satisfying assignment $l_1, \ldots, l_n$ be given. Make the corresponding literal nodes in A_2 simulate (i.e. match) t_1, which gives us a simulation relation R. Now any cycle in A_1 is of the form $(s_1, C_i, t_1, t_1, \ldots)$, which is accepting. These cycles are simulated by (accepting) cycles of the form $\left(s_1, \tilde{C}_i, l_j, t_2, t_2, \ldots\right)$, where l_j is a literal satisfied in C_i. It follows that the R is a UFSR. Now, let a UFSR R be given. We want to find a satisfying assignment. By remark 2, a variable and its negation are not both matched to t_1. Take the induced assignment, i.e. those literals matched to t_1. If a variable does not appear in this assignment, give it an arbitrary value. Now for every clause C_i, there is a cycle of the form $\left(s_1, \tilde{C}_i, l_j, t_2, t_2, \ldots\right)$ to simulate the fair cycle $(s_1, C_i, t_1, t_1, \ldots)$, where l_j is the assignment we have chosen. This means that the assignment satisfies C_i. Hence, every clause is satisfied as desired (QED).

Lemma 7.3.20 Existence of EFSRs does not imply existence of UFSRs (although the reverse is easily seen to be true). Hence, EFSRs are better approximations to language containment than UFSRs.

Proof Figure 7.8 gives an example of where there is an EFSR (α in A_1 is related to every state of A_2) but no UFSR. The reason there are no UFSRs is as follows. Every SSR must contain $(\alpha, 0)$ to satisfy the initial state condition. It must also contain either $(\alpha, 1)$ or $(\alpha, 3)$. If it contains $(\alpha, 1)$ it will also contain $(\alpha, 2)$; if it contains $(\alpha, 3)$ it will also contain $(\alpha, 4)$. If it contains $(\alpha, 1)$ but not $(\alpha, 3)$, then the string aab^{ω} has no accepting simulating run in A_2. If it contains $(\alpha, 3)$ but not $(\alpha, 1)$, then the string a^{ω} has no accepting simulating run in A_2. If it contains both $(\alpha, 1)$ and $(\alpha, 3)$, then a^{ω} and aab^{ω} have both accepting and non-accepting simulating runs in A_2. Hence, in every case, the relation is not a UFSR (QED).



In this figure two Buchi automata are shown, where the highlighted states are final. The language of these automata is $\Sigma^{\omega} = (a+b)^{\omega}$. *There is a EFSR from the first to the second, but no UFSRs.*

Figure 7.8: EFSR's are more general than UFSR's

7.4 Algorithms for Approximating UFSR's

In this section, we give heuristic approximation algorithms for UFSRs. These algorithms are meant as initial starting points, which have to be refined by experimentation. Our algorithms can be applied to any type of fairness constraints, and only depend on the ability to compute the sets $Fair$, $Fair+$, and $Fair\pm$, given some fairness constraint. The efficiency of computing these sets depends on the notion of fairness used. For the problems which come up in the eSeR environment, the algorithms are polynomial.

7.4.1 A Fast Algorithm

Let reduced automata A_1 and A_2 with fairness constraints F_1 and F_2 be given. The following algorithm computes a UFSR R.

1. *Let* $R_0 = R_P$, *the product relation of* A_1 *and* A_2.
2. *Let* $Fair$ *be the set of fair states of* R_0 *with the fairness constraint* $F_1 \wedge \overline{F_2}$.
3. *Let* $R_1 = R_0 \cap \overline{Fair}$, *i.e. delete the states in* $Fair$ *from* R_0.
4. *Let* R *be the refinement of* R_1 *by the transition relation condition.*

The above algorithm first delctes all states which violate the universal fair condition in the product relation. It then refines the relation by the transition relation condition.

Lemma 7.4.1 If R is non-empty, then R is a UFSR.

Proof We only need to check whether R satisfies the universal fair condition. This follows from lemma 7.3.17, noting that every fair run of R is also a fair run of R_0, and the fact that the product graph induced by the product relation is the product relation (QED).

Remark The above algorithm computes the extension of the equivalence relation "live-cycles Buchi simulation relations" of [DHW91] to general fairness constraints.

We think of simulation relations as approximations to language containment. When fairness is used, we need to trade expressiveness (how well we approximate language containment) with complexity. One way to measure expressiveness ([DHW91]) is whether the approximation is exact when A_2 is deterministic. This is the case for both EFSRs and UFSRs. The following lemma shows that the above approximation algorithm to UFSRs preserves this property.

Lemma 7.4.2 If A_2 is deterministic, then the algorithm returns a UFSR if one exists.

Proof By lemma 7.3.14, if a UFSR exists, $L(A_1) \subseteq L(A_2)$. By the fact that A_2 is deterministic and $L(A_1) \subseteq L(A_2)$, the set *Fair* in step 1 of the algorithm is empty, which implies the set returned by the algorithm is the product relation R_P. By lemma 7.3.9, this relation is an EFSR, and hence a UFSR (QED).

Lemma 7.4.3 If A_2 is non-deterministic, the above algorithm may not find a UFSR even if one exists.

Proof Consider figure 7.6 in the proof of lemma 7.3.16 which shows UFSRs are not closed under union. Since, the cycle $((0,0'), (1,1'), (4,4'))$ violates the universal fair simulation condition, states $(0,0')$, $(1,1')$, and $(4,4')$ are deleted from the product graph. Since $(0,0')$ is part of every simulation relation, we conclude no simulation relation can exist. However, as we saw before, there are two UFSRs for this example (QED).

Lemma 7.4.4 (easy) If the set *Fair* can be computed in poly time, then the algorithm runs in poly-time.

Lemma 7.4.5 If A_1 is edge-Streett, and A_2 is edge-Streett or edge-Rabin, then the algorithm runs in poly-time.

Proof Follows from lemmas 7.3.2 and 7.4.4 (QED).

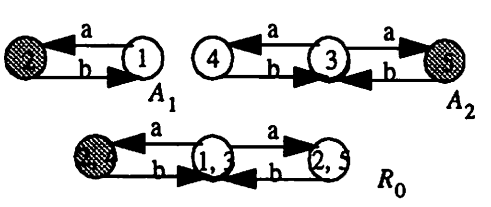
Lemma 7.4.6 If instead of *Fair* a set T where $Fair \subseteq T$ is deleted from R_0, then the relation R computed by the above algorithm is still a UFSR.

Proof As before R is a SSR. It also satisfies the universal fair condition, since all fair states of R_0 are deleted (QED).

Since, when BDDs are used *Fair* is hard to compute, one can use $Fair\pm$ as an easily computable approximation. In some cases, such as in non-deterministic property checking we may be willing to try harder in finding a fair simulation relation, since the alternative (in this case, determinization of the property automaton) may be very costly. In some cases, such as state minimization, the extra effort may not be justified. Since $Fair = \emptyset$ implies $Fair\pm = \emptyset$, if $Fair\pm$ is used in the algorithm, lemma 7.4.2 still holds.

7.4.2 Better Approximations

In this section, we present improvements to the algorithm of section 7.4.1. Figure 7.9 shows two automata A_1 and A_2, with their product relation having fairness constraint $F_1 \wedge \overline{F_2}$ (denoted by R_0). Thc states $(2, 4)$ and $1, 3$ in the product automata arc fair. Sincc $(2, 5)$ docs not satisfy the initial condition property, the algorithm of section 7.4.1 won't find a UFSR. However, if we only delete $(2, 4)$, then all other states become unfair. As this example suggests, in general, it may not be necessary to delete all fair states to restore the universal fairness condition.



This example shows two Buchi automata, with states 2 and 5 being final, and 1, and 3 being initial. State 3 can simulate 1 in a relation which includes (1,3), (2,5). However, in R_0 the cycle (1,3),(2,4) is a fair cycle. Hence, (1,3) and (2,4) are deleted by the algorithm of the previous section, and no UFSR is found (although one exists).

Figure 7.9: Motivating examples for FSR algorithms

Ideally, we want to delete a minimum number of states whose deletion will cause the set of fair states to become empty. We present methods to choose a heuristically good set. The modified algorithm refines the current relation by the safe simulation condition, computes fair states, and deletes a subset of the fair states. This process is repeated until convergence is achieved.

Definition Let an edge-Streett automaton A be given. Given a fairness constraint F of A, its ***restriction set*** $R_E(F)$ is defined as follows.

1. If $F = F^{\infty}P + G^{\infty}Q$, then $R_E(F) = P \cup Q$.

2. If F is a set of negative fair edges $\{ (u_1, v_1), \ldots, (u_n, v_n) \}$, then $R_E(F) = \prod_{k=1}^{n} \overline{(u_k \cup v_k)}$.

3. If F is a set of positive fair edges $\{ (u_1, v_1), \ldots, (u_n, v_n) \}$, then $R_E(F) = \prod_{k=1}^{n} (u_k \cup v_k)$.

The modified algorithm iterates over a two pass procedure until convergence is achieved. In the first pass, a subset of the fair states is chosen and deleted, whereas in the second pass the relation is refined with respect to the transition relation condition. The algorithm is as follows.

1. Let R_0 be the product relation of A_1 and A_2.

2. Until convergence is achieved,

2a. Compute the set of fair states of R_i, $Fair_i$, and let $D_i = Fair_i$.

2b. For each fairness constraint F_j, let $D_i = D_i \cap R_E(F_j)$, i.e. restrict D_i by the restriction set of F_i. If the result is empty, leave D_i unchanged, and continue with the next fairness constraint.

2c. Let $R_{i+1/2} = R_i \cap \overline{D_i}$, i.e. delete the states in D_i from R_i.

2d. Let R_{i+1} be the refinement of $R_{i+1/2}$ by the transition relation condition.

For the example in figure 7.9, the fairness constraint is $F^{\infty}(2,-) \wedge G^{\infty}(-,\{3,4\})$. The modified algorithm will only delete $(2,4)$. Hence, the UFSR will be found. We now present modifications for the case of state minimization, which guarantees states of the form (x,x) will not be deleted by the algorithm. It remains to be seen whether these changes improve the quality of the results in practice.

Let A be an edge-Streett automaton being minimized, with a set of n fairness constraints $F_1, \dots, F_n$. Let A' be a copy of A, with fairness constraints $F_1, \dots, F_n$.

Lemma 7.4.7 The relation $R = \{(x,x') : \}$ is a UFSR from A to A'.

Proof Since $(x,x') \in R$ for the initial states, the initial condition property holds. Let (x,a,y) be an edge in A with $(x,x') \in R$. Since $(y,y') \in R$, the transition relation condition is also satisfied. Let $x_1, x_2, \dots$ be a fair run in A. The only simulating run in A' is $x'_1, x'_2, \dots$, which is fair. Hence, the universal fair condition is also satisfied (QED).

In the case of state minimization, we modify the above algorithm as follows. The fairness constraints on $A \times A'$ is $\left(\prod_{i=1}^{n} F_i\right)\left(\overline{\prod_{j=1}^{n} F_j}\right) = \sum_{j=1}^{n}\left(\overline{F_j}\prod_{i=1}^{n} F_i\right)$.

1. Since every state can simulate itself, we let $D_i = Fair_i \cap (ps \neq ps')$ initially in step 2a, where ps and ps' represent the present states of A and A' respectively.

2. At every point in step 2a, we are dealing with a fairness constraint of the form $F_1 \wedge \dots \wedge F_n \wedge \overline{F_j}$ for some j. In this case, F_j and $\overline{F_j}$ are processed first in computing D_i.

Lemma 7.4.8 In state minimization, if $Fair_i \neq \emptyset$, then $D_i \neq \emptyset$ initially, i.e. if $Fair_i \neq \emptyset$, some state will be deleted.

Proof We need to show there is some fair state of R_i which is not of the form (x, x'). Assume to the contrary that all fair states of R_i are of the form (x, x'). Then, there is a fair cycle $((x_1, x'_1), \ldots, (x_l, x'_l))$. This implies that $\{x_1, \ldots, x_l\}$ and $\{x'_1, \ldots, x'_l\}$ satisfy F_j and $\overline{F_j}$, which is a contradiction (QED).

Lemma 7.4.9 In state minimization, refinement by the transition relation condition of a relation R which includes all states of the form (x, x'), does not delete any such states.

Proof Such states are deleted only if for some edge (x, a, y) the state y is not related to y'. This is not the case by the assumption (QED).

Lemma 7.4.10 Our modified algorithm does not delete any states of the form (x, x') when used for state minimization.

Proof Initially all such nodes are present. By construction, in state minimization, D_i does not include any states of the form (x, x'). The result follows from lemma 7.4.9 (QED).

7.4.3 Static Fair Simulations

[DHW91] gave another type of fair simulations for Buchi automata, called "final-final Buchi simulation relations." This equivalence is very crude, and appears to be only appropriate for state minimization, or Buchi conditions. We generalize this notion to edge-Streett automata for the case of state minimization.

Definition A relation R from edge-Streett automaton A to A is a ***static fair simulation relation (SFSR)*** if R is a SSR, and if $R(x, y)$ holds, then x and y are not distinguished by any fairness set. Two states x and y are ***distinguished by a fairness set*** if

1. for some fairness constraint $F^{\infty}P + G^{\infty}Q$, exactly one of x or y belongs to P (Q).
2. for some state z, only one of (x, z) or (y, z) is a positive (negative) fair edge.
3. for some state w, only one of (w, x) or (w, y) is a positive (negative) fair edge.

Let $S(x, y)$ be the relation where all states, not distinguished by any fairness set, are related. SFSRs are seen to be closed under union, and the largest SFSR can be built by starting from $S(x, y)$ and applying the safe simulation relation condition until convergence. Since computing $S(x, y)$ is simple, this equivalence relation is fast to compute. If A_1 and A_2 are two arbitrary edge-Streett automata, there does not seem to be any natural way of defining SFSRs. However, if they are Buchi, then one can define SFSRs by requiring the final states and non-final states to be related. In this case, it seems that SFSRs do not give us very good approximations to language containment. For example, if A_1 and A_2 consist of two cycles whose only difference is in the position of final states, no SFSR exists. This example shows that SFSRs are not complete with respect to deterministic right-hand sides, i.e. it may be that $L(A_1) \subseteq L(A_2)$, and A_2 is deterministic, but no SFSR exists.

7.5 Comments on Experiments

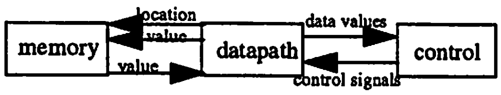
We have presented three methods for approximating UFSRs: static fair simulation relations, the algorithm of section 7.4.1, and the modified algorithm of section 7.4.2. Two parameters can affect the performance of these algorithms The first, called ***fair approximation***, determines to what degree the set $Fair$ is approximated. There are three choices: $Fair$, $Fair\pm$, and $Fair+$. The second parameter called ***output determinization*** indicates whether A_2 should be output determinized. Combining the three algorithms with the the two parameters, we get a total of 18 different algorithms. Experiments with these algorithm are needed to determine theirs run-time versus quality trade-offs.

Chapter 8

Automatic Datapath Abstraction In Hardware Systems

8.1 Introduction

A hardware system can be divided into three major components: control, datapath, and memory (figure 8.1). The ***control*** part consists of a set of interacting FSM's which, depending on data values and their internal states, produce a set of control signals for the datapath. The ***datapath*** consists of functions, predicates, and registers, which based on the control signals, operate on data. The data often consists of integers. The ***memory*** acts as a container for values, and communicates with the datapath.



A hardware system consists of three major components: control, datapath, and memory.

Figure 8.1: Components of a hardware system

Properties can be classified as control, data, and data/control. ***Control properties*** are those which involve only the control signals, and many times can be verified without considering the datapath. An example is that no two control signals of a bus are asserted at the same time. Most existing automatic verification techniques are suitable only for verifying such properties. Therefore, a verification expert needs to manually abstract the datapath. This process is time-consuming and often involves a third-party's understanding of the design.

Data properties are those which the datapath must satisfy. For example, for a pipelined datapath, one verifies that values appear on time where they are needed. The most successful technique to attack such properties has been theorem-proving. The data properties, addressed so far, fall into those for which automatic or almost automatic theorem-proving is possible. The theorem-prover PVS ([PVS93]) has been used to prove such properties almost automatically (the proofs involve a few routine proof commands).

Data/Control properties involve both control signals and datapath variables. An example is "if an add command is issued, and no exceptions occur, then in 3 time steps, location c contains

$a+b$." In general, such properties are the most difficult to verify, and only ad hoc techniques have been used to verify them.

Abstraction is the process of reducing the proof of a property on an infinite or large state space (***concrete model***) to a proof on a smaller state space (***abstract model***). Based on how well the abstract model approximates the concrete one with respect to the property, abstractions can be divided into three categories. ***Exact abstractions*** are those, where the property holds for the concrete model iff it holds for the abstract one. [MPS92] provides an example of exact abstractions, where n-state counters whose data values are not used by other FSMs are reduced to 3-state FSMs. ***Conservative abstractions*** are those where if the property holds in the abstract model it holds in the concrete model. The homomorphism theory of [Kur92] is an example of (manual) conservative approximations. ***Aggressive abstractions*** are those where if the property does not hold in the abstract model then it does not hold in the concrete model. Aggressive abstraction are often used to find bugs. If a conservative abstraction fails, an aggressive abstraction may confirm the property does not hold. Similarly, if an aggressive abstraction holds, conservative abstractions may be used to possibly confirm the property holds.

Abstractions of hardware systems can also be classified by their level of granularity. ***Datapath abstractions*** are those which eliminate portions of the datapath or reduce the number of bits in the datapath. ***Control abstractions*** are those which reduce the number of output values or states of a state machine based on equivalence with respect to a property. For example, some property may only be sensitive to two values of a multi-valued variable. In such cases, other values may be collapsed together. Control properties may not be behavior-preserving. For example, [HKB94] presents a technique for reducing the states of a FSM which does not preserve the behavior: if the original machine can produce an output in n steps, then the reduced machine can also produce it in n steps.

The main problem with formal verification is capacity. Not counting the on-chip cache, current microprocessors contain in the order of tens of thousands of latches. For example, UltraSparc ([Sparc95]) contains 20,000 clocked elements. However, the current BDD-based techniques for verification can at best handle circuits with a few hundred latches. This ***verification gap*** can be made smaller by using ***modular verification***, in which different subsystems such as dispatch units, memory subsystems, bus interfaces, and functional units are verified separately. When verifying a subsystem, other subsystems are replaced (manually) by simple modules which contain the behavior of the detailed modules they replace. Our approach is to model the circuits using integer combinational/sequential (ICS) concurrency model, and use datapath abstraction to nullify the effect of large datapaths and memory elements on the state explosion problem.

ICS is based on the combinational/sequential (C/S) semantics (chapter 2), and extends it by 1) introducing integer variables, 2) interpreted and uninterpreted predicates and functions on integers, and 3) interpreted memory functions. ICS easily supports non-determinism and fairness constraints, and if the extensions are not used, it reduces to C/S. One can extend the general theory of verification using language containment ([Kur92] and chapter 4) to ICS models. Again if the ICS extensions are not used, the theory reduces to the theory of language containment on C/S models. Using ICS models, the datapath variables are modeled as integers, functional units as uninterpreted functions, and memory as infinite memory. ICS models can easily and automatically be extracted from descriptions in hardware description languages such as Verilog, with very little work on the part of the designers.

Given a property, verification of ICS models proceeds using two techniques. For a subset of circuits where certain structural properties hold, verification can be done using ***finite instantiations***, where the integer variables are replaced by variables a few bits wide, and the uninterpreted functions are given appropriate interpretations (meanings). By performing this reduction, we generally get a model which can be attacked using current verification techniques (e.g. SMV, HSIS, etc.). If this reduction is not possible, a symbolic simulation algorithm can be used to verify the property directly. Symbolic simulation does not always terminate, but termination is guaranteed in some cases, such as some correctness properties of pipelined circuits.

The first kind of circuits we study for abstraction using finite instantiations are ***data insensitive controllers***. Intuitively, these only move data around, and are not sensitive to the values of the data. It appears that many communication protocols fall into this category. We prove that for verifying certain types of safety properties, a single bit of data for each variable is sufficient. ***Data sensitive controllers***, on the other hand, interrogate data values, i.e. apply predicates to them, as well as move them around. A typical memory controller with respect to operations on memory addresses is an example. We show that, depending on the predicates applied (we allow comparisons, equality, sign, and mod), a few bits can suffice to check the property. We also prove a negative result which gives an example of where finite instantiations cannot be used. Since we use ICS models and the language containment paradigm, all our results unless otherwise stated, apply to liveness as well as safety properties. Algorithms for automatically recognizing data sensitive and insensitive controllers are given. In chapter 9, we use these results to verify a memory model by reducing integer data values to binary, and integer memory addresses to a small number .

In modern microprocessors, there are two especially hard-to-design components. One is the memory subsystem and its interfaces; the other is the fetch-issue-execution-writeback unit, which we refer to as the ***pipeline control***. Issues such as speculative execution, out-of-order completion, register re-naming, and interrupts make the design of pipeline controls difficult. Since pipeline control communicates with functional units, these must be modeled also for proving some correctness properties. For example, let $v(r)$ denote the value of register r, and assume we are interested in proving property P: when instruction $r_3 = ADD(r_1, r_2)$, is about to be retired, the value which is written back is $v(r_1) + v(r_2)$. To prove P directly, we need to model the integer functional unit. However, the fact that the functional unit does "addition" is irrelevant for the correctness of pipeline control: it can be modeled as an uninterpreted function. We extend our results for finite instantiations to the case of uninterpreted functions, and present exact and aggressive abstractions.

A symbolic simulation algorithm can also be used to verify data and control/data properties. The set of reachable states of the composition of the model and the complement of the property is then computed, and it is verified that no fair path exists. However, the set of reachable states may be infinite, in which case language containment is undecidable. Verification can then be approximated in the sense that it can be checked that no errors of a given length n exist. If all reachable states are computed within n steps, this verification is exact. We present a BDD-based algorithm for symbolic verification of ICS models.

The flow of the chapter is as follows. Section 8.2 presents the syntax of ICS, its operational semantics, its derivation from descriptions in hardware description languages (HDL's), how fairness constraints are specified, and how language containment is done. Data insensitive and sensitive controllers, theorems about their exact abstractions, and algorithms for recognizing them are described in sections 8.3 and 8.4. The extension of finite instantiations results to uninterpreted functions is presented in section 8.5. Section 8.6 gives a BDD-based algorithm for the symbolic execution of ICS models.

8.2 Integer Combinatorial/Sequential (ICS) Concurrency Model

In this section, the integer combinational/sequential concurrency model is described. ICS is designed to represent systems composed of control, datapath, and memory. Some machinery is given to reason about integers, and how they affect state spaces.

8.2.1 Syntax

The primitives are: variables, tables, interpreted functions and predicates, uninterpreted functions and predicates, constant creators, latches, and memory functions.

Variables. Variables are of two types: ***finite*** and ***integer***. Finite variables take values from some finite domain; integer variables take integer values $(0, 1, 2, \ldots)$.

Tables. A table is a relation defined over a set of finite variables, divided into inputs and outputs. A table is a ***function*** if for every possible input tuple there is at most one output tuple (incompletely specified functions are allowed). Otherwise it is a ***relation***. If a table has only one binary output, and is a function, then it is a ***predicate***.

Interpreted Functions and Predicates. A predefined set of functions and relations over integers is built in. The interpreted functions are: $y := x$, $y := if(b, x)$, $z := mux(b, x, y)$, $z := x + y$, $y := x + c$, where x, y, and z are integer variables, b a binary variable, and c a numeral $(0, 1, 2, \ldots)$. The interpreted predicates are $y = x$ (equality), $y < x$, $x = c$, $(x \bmod m) = r$, and $(x \bmod m) < r$,

Uninterpreted Functions and Predicates. These are a set of function and predicate symbols with their arities and domain variables given. For example, $f(x_1, x_2)$ may be the specification of an uninterpreted integer function defined over binary variable x_1 and integer variable x_2. Predicates of the form $x = term$, where x is an integer variable, and $term$ is an ICS term are also allowed. An ***ICS term*** is built recursively from numerals, constants, interpreted and uninterpreted functions. Therefore, numerals and constants are ICS terms, and if f is an n-ary function and $t_1, \ldots, t_n$ are ICS terms, then $f(t_1, \ldots, t_n)$ is also an ICS term. Note that ICS terms do not involve any variables or predicates. Examples of ICS terms are c_0, $f(c_0, c_1)$, $g(15, f(c_2, c_3))$, and $c_1 + c_2$.

Constant Creators. A constant creator is a special element with no input, and an integer output. Intuitively, it is a higher-order function, which creates a new constant (i.e. a function with no argument) each time called. Intuitively, a constant creator models an unconstrained integer input.

Latches. A latch is defined on two variables over the same domain: input (or ***next state***) and output (or ***present state***). Present and next state variables may overlap, e.g. when an output of a latch is an input to another. Every latch has a set of initial values, which are a subset of the domain of its variables. If the latch is integer-valued, then the initial value set can either be a finite set of numerals, or a given constant. Predicates can be used in combination with constants to create an infinite set of initial values. For example, to declare that the initial set of a latch is all integers greater than 5, we let the initial value be some constant c_0. In the first state, the output of the latch is input to a predicate $x > 5$, and the machine continues only if the predicate holds. Hence, only those behaviors are allowed where the initial value of the latch is an integer greater than 5.

Memory Functions. Two functions ***read*** and ***write*** are provided with their usual interpretation; *read* is a binary function of a memory and a location; *write* is a ternary function, whose arguments are a memory, a location, and a value. Location and value are variables in the model. Reading a location which has not been written, returns a new constant (like a constant creator).

Definition A ***generalized gate*** is a table, an interpreted or uninterpreted function or predicate, or a constant creator.

Definition ***Data movement operations*** are $x := y$, $z := mux(b, x, y)$, and $y := if(b, x)$, where x, y, z are integer variables, and b is binary.

Every model has only a finite number of variables, latches, and generalized gates. Every variable is the output of exactly one generalized gate or latch. Hence, every input to a generalized gate or latch is the output of some other generalized gate or latch; ICS models are closed. A variable can be input to many generalized gates or latches.

Definition A ***state*** is a triple $(latch, memories, predicates)$, where,

a. *latch* is an assignment of values to the latches. For finite valued latches, the value comes from the domain. For integer valued latches, the value is an ICS term.

b. *memories* is a set of memory elements, where a ***memory*** is a set of pairs of ICS terms, where the first denotes a location and the second a value.

c. *predicates* is a set of ***atomic formulas***, where an atomic formula is any interpreted or uninterpreted predicate applied to ICS terms. Examples are $c_0 < f(c_1)$ and $P(f(c_0, c_1), g(c_0))$. Note that $(c_0 < f(c_1)) \wedge (P(f(c_0, c_1), g(c_0)))$ is not an atomic formula. Intuitively, *predicates* at a state s, is the set of assumptions made to reach s.

Definition Given two ICS terms t_1 and t_2, and two sets of atomic formulas $P = \{P_1, \ldots, P_n\}$ and $Q = \{Q_1, \ldots, Q_m\}$, ***t_1 is equal to t_2 subject to P and Q,*** denoted as $t_1|_P = t_2|_Q$ iff the formula $(P_1 \wedge \ldots \wedge P_n \wedge Q_1 \wedge \ldots \wedge Q_m) \Rightarrow (t_1 = t_2)$ is valid. For example, if $t_1 = x$,

$P = \{x > 7, x \le 8\}$, $t_2 = 8$, and $Q = \varnothing$, then $t_1 = t_2$ subject to P and Q since $(x > 7 \wedge x \le 8) \Rightarrow (x = 8)$ is valid. The equality of two ICS terms can be decided using the algorithms of [Sho79] and [Sho82].

Notation If $P = \{P_1, \ldots, P_n\}$ is a set of predicates, we will use P to denote $P_1 \wedge \ldots \wedge P_n$. For example, $P \rightarrow (b = 1)$ is the formula $(P_1 \wedge \ldots \wedge P_n) \rightarrow (b = 1)$.

Definition Let states $s_1 = \left(L_1, \left(M_1^1, \ldots, M_n^1\right), P_1\right)$ and $s_2 = \left(L_2, \left(M_1^2, \ldots, M_n^2\right), P_2\right)$ be given. $s_1 = s_2$ if the following hold.

a. Let l_1^i and l_2^i denote the values of the i-th latch in L_1 and L_2, respectively. If the i-th latch is finite, then $l_1^i = l_2^i$; otherwise, $l_1^i\big|_{P_1} = l_2^i\big|_{P_2}$ must hold.

b. Let $M_k^1[i] = \left(a_k^1[i], v_k^1[i]\right)$ denote the i-th address/value pair in M_k^1, the k-th memory element of s_1. Then, for each $M_k^1[i]$ there exists $M_k^2[j]$ such that $a_k^1[i]\big|_{P_1} = a_k^2[j]\big|_{P_2}$ and $v_k^1[i]\big|_{P_1} = v_k^2[j]\big|_{P_2}$. Similarly, for each $M_k^2[j]$ there must exist $M_k^1[i]$ such that $a_k^1[i]\big|_{P_1} = a_k^2[j]\big|_{P_2}$ and $v_k^1[i]\big|_{P_1} = v_k^2[j]\big|_{P_2}$.

c. $P_1 \equiv P_2$, i.e. $P_1 \Rightarrow P_2$ and $P_2 \Rightarrow P_1$.

Definition An ***initial state*** is a state $(latch_{init}, \varnothing, \varnothing)$, where $latch_{init}$ is an assignment of an initial value to each latch.

Lemma 8.2.1 It is decidable whether two states are equal.

Proof The problem reduces to deciding whether two ICS terms are equal subject to predicate constraints, which can be decided using the techniques of [Sho79] and [Sho82] (QED).

8.2.2 Operational Semantics of ICS

The operational semantics of ICS describes how a transition between two states of an ICS model occurs.

Definition A ***gate graph*** G is a directed graph where each node is a generalized gate. $(u, v) \in G$ if some output variable of the generalized gate u is an input to the generalized gate v. A cyclic gate graph is said to contain a ***combinational loop (or cycle)***.

Remark For an acyclic gate graph G, a root node either has no inputs or is a latch.

Our operational semantics is restricted to acyclic graphs and is defined in terms of a ***configuration* (or *state*) *graph*** and its transition relation. Every node in the configuration graph is a pair (s, n), where s is a state of the model, and n represents the number of constants created so far. An initial state of the configuration graph is of the form (s_{init}, k), where s_{init} is an initial state of the model, and k the number of constants in s_{init}. An edge (transition) is defined by the following algorithm which, given a state $u = ((L, M, P), n)$ in the configuration graph, assigns a new value to all next state variables (creating L'), and creates a new memory M', a new set of predicates P', and a new counter n'. In this case, we say there is an edge (u, v) in the configuration graph, where $v = ((L', M', P'), n')$. State transitions are computed as follows.

0. *Let* $P' = P$, $n' = n$.
1. *Let a user-given total order on all memory operations be given. Choose a topological sort* O *of the gate graph consistent with this memory order.*
2. *Assign the values given by* L *to the outputs of the latches.*
3. *Assign values to the outputs of each generalized gate consistent with its inputs, processing the generalized gates in the topological order* O. *More precisely, let a generalized gate* $g(i, o)$ *be given, where* i *represents the inputs to the gate and* o *its output.*
   a. *If* g *is a table representing the relation* $R(i, o)$, *then* $(i, o) \in R$.
   b. *If* g *is an interpreted or uninterpreted function, then* $o = g(i)$, *where* o *and* i *are ICS terms.*
   c. *If* g *is an interpreted or uninterpreted predicate, if* $P' \rightarrow (g = 0)$ *is valid, let* $g = 0$; *if* $P' \rightarrow (g = 1)$ *is valid, let* $g = 1$; *otherwise let* $o = 0$ *or* $o = 1$, *and* $P' = P' \cup \{g = o\}$.
   d. *If* g *is a constant creator, then* $o = c_{n'}$, *where* $c_{n'}$ *is a fresh constant. Let* $n' = n' + 1$.

Step 3 is referred to as ***value propagation***. Note that the configuration graph is finite-branching, i.e. for every state, there are a finite number of next states. It is possible that a table is not complete, i.e. there are inputs for which there are no outputs. Then, the set of values assigned to an output of a table may be empty. The empty values propagate, i.e. if one of the inputs to a table is empty, then the output is empty as well.

8.2.3 Creating ICS Models From the HDL Verilog

In the HSIS environment, Verilog descriptions are compiled into BLIF-MV. BLIF-MV has a library of pre-defined functions and predicates on finite-sized integers. For example, $add4(x, y, z)$ may mean that two 4-bit integers x and y are added to get the 5-bit integer z. To extend this to be able to compile a Verilog description into an ICS model, we describe below how the various new constructs in ICS can be obtained.

1. Tables, finite variables, and finite latches can be specified as usual.
2. To get integer variables, latches, functions, and predicates, identify a set of variables as integers, and instruct the compiler to use BLIF-MV's library functions.
3. To get uninterpreted functions (predicates), use subcircuits returning integer (Boolean) values whose definitions are not given. These will translate into undefined subcircuits in BLIF-MV.

8.2.4 Fairness Constraints and Language Containment

In this section, we describe how to construct an automaton from an ICS description, how to define fairness constraints on this automaton, what its language is, how to specify properties, and how language containment is performed by checking language emptiness.

8.2.4.1 Fairness Constraints in ICS Models

Definition Let an acyclic ICS model M be given. The ***symbolic automaton*** of M, A_M, is the configuration graph defined by the operational semantics of ICS. The ***alphabet*** of A_M is the Cartesian product of the alphabets of the non-state variable, and is denoted by Σ_M. The alphabet of the integer variables is the set of the ICS terms of M (a countable set), whereas the alphabet of the finite variables is their domains. Let an ω-string x over the alphabet Σ_M be given. r is a run of x in A_M if r_0 is an initial state of A_M, and for all i, there is a transition from r_i to r_{i+1} assigning x_i to non-state variables.

If the model has inputs, we close it by allowing the finite inputs to take any value in their domains, while the integer inputs are driven by constant creators. Hence, the notion of symbolic automaton carries to models with inputs. We allow edge-Streett fairness constraints[1] to be placed only on finite valued latches. For example, one can restrict the acceptable runs to be such that a finite-valued latch l takes some value a in its domain infinitely often. Fairness constraints are generally placed on the control part. Since control circuits are finite state, the restriction of

1. One can allow other types of fairness constraints as well. The language emptiness check for edge-Streett fairness constraints is polynomial, while their next natural extension has an NP-complete languages containment check.

fairness constraints to finite latches is expected not to be severe in practice.

Definition A run r is fair if all fairness constraints are satisfied. More specifically, for each finite latch l, the set of infinitely occurring values which l takes in the run r satisfies all fairness constraints placed on l. The ***symbolic language*** (or just ***language***) of symbolic automaton M, $L_S(M)$, is the set of all strings which have a fair run in A_M.

Definition The ***concrete language*** of a symbolic language L with respect to an interpretation I (an interpretation of all uninterpreted functions and predicates), denoted by $L_C(L, I)$, is obtained by allowing the constants to take any possible integer value for all $x \in L$. The concrete language of M with respect to interpretation I is $L_C(L_S(M), I)$.

Definition Let symbolic languages L, L', L'' be given. L is ***contained*** in L', denoted as $L \subseteq L'$, if for every interpretation I, $L_C(L, I) \subseteq L_C(L', I)$. L is the complement of L', denoted as $L = \bar{L}'$, if for all interpretations I, $L_C(L, I) = \overline{L_C(L', I)}$. L'' is the ***intersection*** of L and L', denoted as $L'' = L \cap L'$, if for every interpretation I, $L_C(L'', I) = L_C(L, I) \cap L_C(L', I)$. Similarly ***union*** of symbolic languages is defined.

Definition Let models M and N be given. The ***composition*** $M \bullet N$ is defined as follows. If M and N have no variables in common, or the common variables are not outputs in both models, the composition of the two models is their syntactic composition. For each common variable which is an output in both M and N, rename one of them, and add a one state automaton which checks that the two outputs are equal.

Lemma 8.2.2 Let models M and N be given. Then, $L(M \bullet N) = L(M) \cap L(N)$, i.e. composition of models corresponds to taking the intersection of their languages.

Proof Let an interpretation I be given. We need to show $L_C(M \bullet N, I) = L_C(M, I) \cap L_C(N, I)$. Let $x \in L_C(M \bullet N, I)$. x gives interpretations for all constants of M and N. Also, the values assigned by x to the variables is consistent with the transition structures of M and N. Hence, $x \in L_C(M, I) \cap L_C(N, I)$. Conversely, let $x \in L_C(M, I) \cap L_C(N, I)$. x gives interpretations for all constants of $M \bullet N$. Note that since $x \in L_C(M, I) \cap L_C(N, I)$, the common variables of M and N are assigned the same value. Also, the values assigned by x to the variables is consistent with the transition structures of $M \bullet N$.

Hence, $x \in L_C(M \bullet N, I)$ (QED).

8.2.4.2 Property Checking

Lemma 8.2.2 implies that the classical results of verification using language containment are valid. For example, to do property checking, the complement automaton of the property can be composed with the system, and the language of the composed system can be checked for emptiness. We will define property automata so that complementing them is straight-forward.

Definition An (edge-Rabin) ***property automaton*** is a complete (i.e. for every symbol there is a transition) edge-Rabin automaton with no outputs. Note that property automata can have integer inputs. All latches of a property automaton are finite valued, i.e. a property automaton has a finite number of states. The only allowed integer operation are integer predicates which are applied to integer inputs, but the alphabet of a property automaton can take an infinite number due to integer inputs.

Definition Let A be a property automaton. The ***finite encapsulation*** of A, A_F, is obtained by replacing each integer predicate (applied to integer inputs) with a binary input variable. Note that A_F is a regular edge-Rabin automaton with finite alphabet and finite number of states.

Lemma 8.2.3 Property automata can be complemented.

Proof Let R be a property automaton, and R_F its finite encapsulation. Since R_F is a regular edge-Rabin automaton, it can be complemented into an edge-Streett automaton. Let the complement be $\overline{R_F}$. Replace each binary variable introduced in $\overline{R_F}$ with the predicate it replaced in R. Call the resulting automaton $\bar{R}$. We have that $L(\bar{R})$ is the complement of $L(R)$. To make presenting the proof simpler, assume R has only one integer input x, and one integer predicate P. Let $c_1, c_2, \ldots$ be the symbolic inputs to R and $\bar{R}$ An interpretation I assigns integer values $i_1, i_2, \ldots$ to $c_1, c_2, \ldots$, and a meaning to P. By construction, we have that R_F and $\overline{R_F}$ each have one binary input, R_F accepts $P(i_1), P(i_2), \ldots$, and $\overline{R_F}$ rejects $P(i_1), P(i_2), \ldots$. Since $i_1, i_2, \ldots$ has the same runs in $\bar{R}$ as $P(i_1), P(i_2), \ldots$ in $\overline{R_F}$, it follows that $\bar{R}$ rejects $i_1, i_2, \ldots$. Similarly, we one show that if $\bar{R}$ accepts $i_1, i_2, \ldots$, R rejects it (QED).

8.2.4.3 Hierarchical Verification

Besides property checking, language containment appears in hierarchical verification. Hierarchical verification is the process of checking that the language of an abstract model contains the language of a detailed model. Since we don't know how to complement general symbolic automata, we restrict hierarchical verification to a subset of symbolic automata, called control automata. Since control automata are where non-determinism (and hence refinement) occurs, this may not be too restrictive in practice.

Definition A ***control automaton*** is a property automaton which is allowed to have only finite outputs. The finite encapsulation of a control automaton can be defined similar to the case of property automaton.

Lemma 8.2.4 Control automata can be complemented.

Proof Since the outputs are finite-valued, the automaton can be complemented. The same procedure as in the proof of lemma 8.2.3 can now be applied to form the complement (QED).

8.2.4.4 Checking Emptiness of ICS Models

Verification using language containment involves complementing the property automaton, and checking whether there is a fair path in the composition of the system and the complement of the property. In order to do this, we need to compute the set of reachable states, and check whether there are any fair paths. However, the set of reachable states may be infinite. Hence, we compute the set of states reachable in at most n steps, for some given n. We then check that no fair path in this subset of the reachable states exists. In practice, since many error traces are short (if they exist), this technique should be quite effective.

Lemma 8.2.5 The language emptiness check of ICS model is undecidable.

Proof Theorem 8.2 in section 8.5 provides the proof (QED).

8.3 Data Insensitive Controllers

In this section, we formalize the notion of data insensitive controllers (DICs). We then prove a theorem which can be used to verify the property that DICs correctly move data around on a system where the integer variables are replaced by single bit binary variables. A practical application of this result is then described, and an algorithm for automatically recognizing DICs is given.

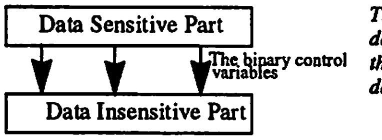
8.3.1 0-1 Theorem for Data Insensitive Controllers

Definition Let the ***data movement operations*** be $x := y$, $z := mux(b, x, y)$, and $y := if(b, x)$, where x, y, z are integer variables, and b is binary.

Definition Let an ICS model M be given. M is a ***data insensitive controller (DIC)*** with respect to a set of variables X, called ***data insensitive variables (DIV),*** if the following holds.

1. The only operations allowed on the variables in X are data movement operations and $x :=$ constant-creator where all variables involved in these operations belong to X.
2. If $l \in X$ is a state variable, then the initial value of l is a fresh constant.
3. If some constant creator C is input to some $x \in X$, then C is not input to any other variable $z \notin X$.

The (generalized) gates in a DIC are naturally divided into two disjoint parts. The first, called the ***data sensitive part***, contains the gates which drive the variables not in DIVs, whereas the second called the ***data insensitive part***, contains those gates which drive the DIVs. The binary variables which appear in the data insensitive part are driven by gates in the data sensitive part. No gate in the data sensitive part has a DIV as an input.



The data sensitive part generates the control signals for the data insensitive part. Changing the values of variables in the data insensitive part does not affect the values of the data sensitive part.

Figure 8.2: Data sensitive controllers

Definition Let M be a DIC with respect to X. The ***binary instantiation*** of M, M_2, is formed by replacing all variables in X by binary ones, replacing each constant creator driving a DIV by a gate which produces 0 or 1 non-deterministically, and replacing the initial values of all latches in X by the set $\{0, 1\}$.

Theorem 8.1 ***(0-1 Theorem for Data-Insensitive Controllers)*** Let an ICS model M be data insensitive with respect to a set of variables X. Let property P specify that when binary variable b becomes 1, then $x = y$, where $\{x, y\} \subseteq X$. Then, P holds for M iff P holds for M_2.

Proof If P holds for M, then P holds for M_2, since binary values are a subset of integer values. Now assume P does not hold for M. Then, there exists a sequence of states $s_1, \ldots, s_n$ and a sequence of assignments to non-state variables $a_1, \ldots, a_{n-1}$, such that s_1 is an initial state, s_{i+1} is reached from s_i in M on a_i, a_i for $i < n-1$ assigns 0 to b, a_{n-1} assigns 1 to b and different values to x and y. We will build $\tilde{s}_1, \ldots, \tilde{s}_n$ and $\tilde{a}_1, \ldots, \tilde{a}_{n-1}$, where all variables in X are assigned binary values in $\tilde{a}_i$ and $\tilde{s}_i$ for all i, $\tilde{s}_{i+1}$ is reached from $\tilde{s}_i$ in M_2 on $\tilde{a}_i$, $\tilde{a}_i$ for $i < n-1$ assign 0 to b, $\tilde{a}_{n-1}$ assigns 1 to b, and $\tilde{a}_{n-1}$ assigns 0 to x and 1 to y.

Let the data sensitive variables take the same values in $\tilde{a}_i$ and $\tilde{s}_i$ as in a_i and s_i's. This is possible since the values of the data insensitive variables do not affect the values of the data sensitive partition. It follows that the values of the binary control variables which are input to the data insensitive part also remain the same in $\tilde{a}_i$ and $\tilde{s}_i$'s.

Consider state s_{n-1} and how x and y are assigned at stage n. From the definition of DICs and the operational semantics of ICS models, it follows that either there exists a constant creator C such that $x = C(n-1)$, where $C(n-1)$ denotes the value of the constant creator assigned by a_{n-1}, or there exists an integer latch l such that $x = l(n-1)$, where $l(n-1)$ is the value of l at state s_{n-1}. If we trace this value to its source, we find the value of x assigned by s_{n-1} and a_{n-1} is equal to the value of some integer latch l' at s_1, or to the value of some constant creator C' assigned by some a_i, $i \leq n-1$. Assign the value 0 to this source, i.e. the initial value of l' or the value of C'. Similarly find l'' or C'' for y and assign it the value 1. Arbitrarily choose values for the rest of the constant creators and initial states. Since the values of the binary control variables is the same as before, the claim follows (QED).

Historical Remark [Wol86] introduced the notion of data-independent controllers for simple reactive programs, which are conceptually very similar to our data-insensitive controllers. However, the results of [Wol86], specifically theorem 5.4, seem to be essentially different than our 0-1 theorem. [Wol86]'s theorem 5.4 basically applies to properties which can be written as "for all tuples $i_1, \ldots, i_n$ with distinct values, $P(i_1, \ldots, i_n)$ should hold", where P is a linear temporal logic formula over the variables $i_1, \ldots, i_n$. Such a property can also be expressed as an infinite conjunction of linear temporal logic formulas of the form $P(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are value assignments to $i_1, \ldots, i_n$. However, our 0-1 theorem is equivalent to an infinite disjunction, i.e. "when b occurs either $x = y = 0$, or $x = y = 1$, or $x = y = 2$, etc."

One in general may be interested in proving the property "repeatedly whenever binary variable b becomes 1, then $x = y$." To do so, create M' from M by adding a two-state FSM, which outputs a variable b'. This FSM stays in its first state for an arbitrary amount of time, outputting $b' = 0$. It can non-deterministically make a transition to its second state, outputting $b' = 1$. It then stays in its second state forever outputting $b' = 0$. Let P' be the property "when binary variable b' becomes 1, then $x = y$." It is easy to see that P holds of M iff P' holds of M'. However, the 0-1 theorem applies to the latter case.

We also emphasize that this theorem also applies to finite state systems and can be used to abstract datapaths from a large number of bits to a single bit.

8.3.2 An Application of the 0-1 Theorem

A memory controller in a multi-processor takes load and store operations from the processors, and executes them. To gain more speed, it can do memory operations out of order. To make sure the memory works correctly, a property known as the value axiom is proved about the memory system (chapter 9). The value axiom says if a load is issued to location a, the value returned by the memory model when the load is serviced should be the value of the most recent pending store to a, or if there are no pending stores, the value stored at a currently.

To check the value property, a temporary latch can be added to the model, which stores the correct value of the load when the load is issued. This can be done by checking the memory's buffers to see if there are any pending stores to location a. When the load is serviced ($b = 1$), it is checked that the value returned by the memory is the same as the value stored in this temporary latch ($x = y$). The memory controller is a DIC with respect to the data variables giving the values to be stored at locations. Hence, by the 0-1 theorem and the remark in the previous section, it suffices to check the property only on binary values.

8.3.3 Recognizing Data Insensitive Controllers

DICs can be automatically recognized with a polynomial time algorithm (in the size of the integer partition) which finds the largest set of variables according to which a model is a DIC.

Lemma 8.3.1 If an ICS model M is a DIC with respect to X and a DIC with respect to Y, then it is a DIC with respect to $X \cup Y$. Hence, there is a largest set of variables $DIV(M)$ according to which M is a DIC.

Proof Follows directly from the definition of DIC's (QED).

The following algorithm finds $DIV(M)$. The algorithm proceeds by marking off any variable which cannot be a DIV. The remaining variables are in $DIV(M)$.

0. Let X be the set of all integer variables.

1. Delete from X those variables involved in any operations other than data movement, i.e. violating condition 1 of DICs.

2. Delete from X any variable violating condition 3 of DICs, i.e. fanning-out to an element not in X.

3. Delete from X those variables involved in some integer operation (including data movement) with variables not in X. Go back to step 1.

4. ReturnX as $DIV(M)$.

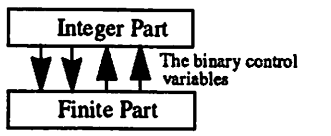
8.4 Data Semi-Sensitive Controllers

In this section, we present results about a class of circuits, called data semi-sensitive circuits (DSSC), which not only move data around, but also apply predicates to them. We identify subclasses of these circuits, where properties can be proved by replacing the ranges of integer variables by a small finite range. Recognizing DSSCs and any of the subsets we present in this section is straight-forward. In what follows, checking a property on a system M is done by checking for the language emptiness of the composition of the complement of the property and M.

Definition The predicates $x < c$, $x = c$, $x < y$, $x = y$, $(x \bmod m) = d$, and $(x \bmod m) < d$ are called ***integer predicates***. Note that the predicates $Even(x)$ and $Odd(x)$ are special cases of $(x \bmod m) = d$.

Definition An ICS model M is a ***data semi-sensitive controller (DSSC)*** if the only operations on integer variables are data movement, $x :=$ constant-creator, and integer predicates. The initial value of all integer latches is a fresh constant.

A DSSC can be pictured as shown in figure 8.3, where the model is divided into two parts: integer and finite. The integer part contains the gates which drive the integer variables, whereas the finite part contains those which drive the finite variables. The communication between the two parts is restricted to a set of binary control variables. The control variables produced in the integer part are the outputs of the integer predicates. The gates in the finite part take only finite variables as inputs.



The communication between the two parts is restricted to a set of binary control variables. The binary variables output from the integer parts are the outputs of the integer predicates.

Figure 8.3: Data semi-sensitive controllers

Definition Let M be a DSSC. The ***n-ary instantiation*** of M denoted by M_n, is formed by replacing all integers variables by finite variables taking values from $0, 1, \ldots, n-1$. The initial value of an integer is the set $0, 1, \ldots, n-1$.

8.4.1 Data Comparison Controllers

Data comparison controllers are a subclass of DSSCs which move data around, and compare them. We prove finite instantiations can be used to verify language emptiness for these circuits. A practical example of where data comparison controllers come up is given.

Definition A DSSC which only involves data movement operations, the predicate $x = y$, and constant creators is called a ***data comparison controller.***

Lemma 8.4.1 Let M be a data comparison controller with n integer variables, and M_n its n-ary instantiation. Then, $L(M) = \emptyset$ iff $L(M_n) = \emptyset$.

Proof If $L(M) = \emptyset$, it trivially follows that $L(M_n) = \emptyset$. Now assume $L(M) \neq \emptyset$. We want to show $L(M_n) \neq \emptyset$. Let $s_0, a_0, s_1, a_1, \ldots$ be a fair run in M, where s_i and a_i are assignment of values to state and non-state variables respectively. Let $x_1, \ldots, x_n$ be the integer variables. We will build a fair run $\tilde{s}_0, \tilde{a}_0, \tilde{s}_1, \tilde{a}_1, \ldots$ in M_n such that the following two conditions hold for all i.

1. The values assigned to the finite variables (including the binary variables) by $\tilde{s}_i, \tilde{a}_i$ are the same as those assigned by s_i, a_i.

2. Let $\alpha_{1,i}, \ldots, \alpha_{n,i}$ and $\tilde{\alpha}_{1,i}, \ldots, \tilde{\alpha}_{n,i}$ be the values assigned by s_i, a_i and $\tilde{s}_i, \tilde{a}_i$ to the integer values. Then, $\alpha_{k,i} = \alpha_{l,i}$ iff $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{l,i}$.

A string respecting the above two properties is accepting in M_n since 1) the fairness constraints of M and M_n are the same, 2) the fairness constraints are placed only on finite latches, and 3) for each such latch the set of infinitely occurring values in $s_1, s_2, \ldots$ is the same as $\tilde{s}_1, \tilde{s}_2, \ldots$. The existence of such a string follows from the following three claims.

Claim 1. There exists an initial state $\tilde{s_0}$ in M_n such that the above two properties are respected for the latch variables. Let the finite latches in $\tilde{s_0}$ have the same values as in s_0. Let $\alpha_1, \ldots, \alpha_p$ be the assignment given to the integer latches $l_1, \ldots, l_p$ by s_0, a_0. Choose $\tilde{\alpha}_1, \ldots, \tilde{\alpha}_p$ such that $\alpha_i = \alpha_j$ iff $\tilde{\alpha}_i = \tilde{\alpha}_j$.

Claim 2. Given a state $\tilde{s_i}$ where the latch variables satisfy the above two properties, there exists an $\tilde{a}_i$ such that all variables satisfy the two properties. Consider a topological sort of the gate graph, $g_1, \ldots, g_m$, where m is the number of gates. By induction assume the two properties hold for all latch variables and all variables which are outputs of gates g_k for $k < l$. If g_l is a finite table, by property 1, the output of g_l can be assigned the same value it was given by a_i. If g_l is an integer comparator or data movement element, by property 2, its output will have the same value as in a_i. If g_l is a constant creator, and if it was assigned a value equal to one of the integer variables processed so far by a_i, let's say x, then assign it the value given to x. If not, assign it a value distinct from all integer variables which have been assigned value so far in step i. Thus, the two properties also hold for g_l. Note that no more than n distinct integer values are needed.

Claim 3. If the two properties hold for $\tilde{s_i}, \tilde{a}_i$, then they hold for $\tilde{s}_{i+1}$. This follows by the fact the latch variables in $\tilde{s}_{i+1}$ have the same values as the next state variables assigned by $\tilde{s_i}, \tilde{a}_i$ (QED lemma 8.4.1).

A result similar to lemma 8.4.1 appeared in [ID93]. The main difference is that lemma 8.4.1 applies to both liveness and safety properties. The fact that lemma 8.4.1 applies to liveness properties comes basically for free as a consequence of dealing with ICS models. As an example, the property "if request and $x = y$, then eventually acknowledge and $z = w$" can be proved in our framework, but is not dealt with in [ID93].

Remark Adrian Isles [Isl95] has noticed that the bound for lemma 8.4.1 can be improved by letting n be the number of integer latches plus constant creators.

Lemma 8.4.2 There is a data comparison controller M having n integer variables such that

$L(M) \neq \emptyset$ and $L(M_i) = \emptyset$ for $i < n-2$.

Proof Let M have n integer variables $x_1, \ldots, x_{n-2}$, y_1, y_2, where $x_1, \ldots, x_{n-2}$ are integer latches with their initial states being fresh constants, and y_1 and y_2 are the inputs to a comparator. Let M also contain a binary latch l whose initial state is 1, a $\binom{n}{2}$-state counter, and enough gates to distinguish whether all initial values assigned to $x_1, \ldots, x_{n-2}$ are distinct. It does so, by using the comparator and comparing two numbers at every cycle (note that the values of $x_1, \ldots, x_{n-2}$ are unchanged throughout the computation). Hence, it needs $\binom{n}{2}$ cycles to do all the comparisons. Initially $l = 1$, and it is set to 0 if at any cycle $y_1 = y_2$. If $l = 0$, M falls into a dead state and does not accept any strings. Hence, M has an accepting run, when all the initial values of $x_1, \ldots, x_{n-2}$ are distinct. However, for $i < n-2$, at least two of the latches must have the same initial values. Hence, M_i does not have a run, i.e. $L(M_i) = \emptyset$ (QED).

One application of data comparison controllers is in memory systems of multi-processors. These memory controllers have a buffer for each processor where the memory instructions (load and store) are placed. To perform verification, we assume the sizes of the buffers and the number of processors are finite (chapter 9). However, it is assumed that the number of memory locations is infinite. If the memory controller only compares memory addresses to each other, which is what one might expect, then it is a data comparison controller with respect to the addresses. Hence, a small number of addresses suffice to prove the properties. This number is proportional to the sizes of the buffers. Combining this result with the 0-1 theorem for data insensitive controllers, we get that once the sizes of the instruction buffers are fixed, then a memory controller can be verified using binary data bits, and a finite (and small) number of memory locations without losing any accuracy in verification.

8.4.2 Other Types of Data Semi-Sensitive Controllers

We present results on circuits which use more sophisticated integer predicates (such as $x < c$, $(x \bmod m) = r$, and $x < y$). Our results show that the only situation where finite instantiations cannot be used is when both data movement and the predicate $x < y$ are used.

Lemma 8.4.3 There exists a DSSC controller M involving data movement operations and predicate $x<y$ such that $L(M) \neq \emptyset$ but $L(M_n) = \emptyset$ for all n.

Proof We will build an ICS model M whose only runs are on strictly increasing sequences of integers. Let M consist of an integer latch x, an integer variable y, a binary variable b, three gates and no fairness constraints (i.e. all runs are accepting). The first gate assigns y a fresh constant. The second gate does the comparison $x<y$, producing the binary variable b. The last gate is $x = if(b, y)$, which assigns x the value of y if $b = 1$. Let the initial value of x be a fresh constant. Intuitively, the circuit assigns x an initial value, checks whether y has been assigned a larger value, and if so sets x to the value of y. All runs of M assign strictly increasing values to x. Hence no finite instantiation can have a run (QED).

Lemma 8.4.4 Let M be a DSSC with n integer variables, p predicates of the form $x = c_l$, using only data movement operations and the predicate $x = y$. Then, $L(M) = \emptyset$ iff $L\left(\hat{M}_{n+p}\right) = \emptyset$, where $\hat{M}_{n+p}$ is M_{n+p} with the predicates of the form $x = c_l$ changed to $x = l$.

Proof The proof parallels the proof of lemma 8.4.1. The second property in the proof of lemma 8.4.1 is now modified to "$\alpha_{k,i} = \alpha_{l,i}$ iff $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{l,i}$, and $\alpha_{k,i} = c_l$ iff $\tilde{\alpha}_{k,i} = l$." The only other difference arises when a constant creator g, which is assigned a value v different than all variables assigned so far by (s_i, a_i), is processed. If $v = c_l$ for some l, then g is assigned l in $\bar{s}_i, \bar{a}_i$. Otherwise, a value $v \geq p$ and not equal to any of the variables created so far is assigned to g. Since there are n distinct variables, at most $n+p$ values are used (QED).

Remark In the proof of lemma 8.4.4, if there are no predicates of the form $x = y$, then $p+1$ values suffice. This is the core of [Wol86]'s techniques, where the predicates $x = c_i$ represent the atomic formulas of the temporal logic formula. Note that as [Wol86] argues, for such a circuit M (involving data movement and $x = c_l$'s), if emptiness holds for a set of distinct constants $c_1, \ldots, c_p$, then it holds for any circuit obtained from M by replacing $c_1, \ldots, c_p$ by another set of distinct constants $c_1, \ldots, c_p$.

Lemma 8.4.5 Let M be a DSSC with n integer variables, using only data movement operations, and the predicates $x = y$, $x = c_l$, and $x < d_j$. Let $C = MAX(MAX(c_l), MAX(d_j))$. Then, $L(M) = \emptyset$ iff $L\left(M_{\tilde{n}}\right) = \emptyset$, where $\tilde{n} = C+n+1$.

Proof The proof parallels that of lemma 8.4.1. The second property in the proof of lemma 8.4.1 is now modified to "$\alpha_{k,i} = \alpha_{l,i}$ iff $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{l,i}$, and $\alpha_{k,i} = c_l$ iff $\tilde{\alpha}_{k,i} = c_l$, and $\alpha_{k,i} < d_j$ iff $\tilde{\alpha}_{k,i} < d_j$." The only other difference is when a constant creator g, which is assigned a value v different than all variables assigned so far by (s_i, a_i), is processed. If $v \le C$, then g is assigned the same value as v in $\tilde{s}_i, \tilde{a}_i$. Otherwise, a value greater than C and not equal to any of the variables created so far is assigned to g. Since at most n values greater than C are used, the bound follows (QED).

One can get a bound for lemma 8.4.5 which is independent of the values of the constants. Let p and q predicates of the forms $x = c_l$ and $x < d_j$ respectively be given. Further assume $d_j \le d_{j+1}$. Let h_j be the number of c_i's which occur between d_j and d_{j+1}. Intuitively, between d_j and d_{j+1}, $n + h_j$ values are needed. By re-assigning value to $x < d_j$'s and $x = c_l$'s as in the proof of lemma 8.4.4, the emptiness check can be done on a system where the integer variables take on $(q+1)(n+1) + p$ values (there are $q+1$ intervals between d_j's each requiring $n + h_j$ values, and $\sum h_j = p$). We have the following result.

Lemma 8.4.6 Let M be a DSSC with n integer variables, using only data movement operations, p predicates of the form $x = c_l$, q predicates of the form $x < d_j$, and the predicate $x = y$. There exists a system N derived from M such that the integer variables take on $(q+1)(n+1) + p$ values, and $L(M) = \varnothing$ iff $L(N) = \varnothing$.

Proof Given above (QED).

Lemma 8.4.7 Let M be a DSSC with n integer variables, using only data movement operations, and the predicates $x = y$, $x = c_l$, $x < d_j$, $(x \bmod m_k) = r_k$, $\left(x \bmod m_{k'}'\right) < r_{k'}'$. Let $C = MAX(MAX(c_l), MAX(d_j))$, and N be the common multiple of all the m_k's and $m_{k'}'$'s. Then, $L(M) = \varnothing$ iff $L\left(M_{\tilde{n}}\right) = \varnothing$, where $\tilde{n} = C + Nn + 1$.

Proof The proof parallels that of lemma 8.4.1. The second property in the proof of lemma 8.4.1 is now modified to "$\alpha_{k,i} = \alpha_{l,i}$ iff $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{l,i}$, and $\alpha_{k,i} = c_l$ iff $\tilde{\alpha}_{k,i} = c_l$, and $\alpha_{k,i} < d_j$ iff $\tilde{\alpha}_{k,i} < d_j$, and $(\alpha_{l,i} \bmod m_k) = r_k$ iff $\left(\tilde{\alpha}_{l,i} \bmod m_k\right) = r_k$, and $\left(\alpha_{l,i} \bmod m'_{k'}\right) < r'_{k'}$ iff

$\left(\tilde{\alpha}_{l,i} \bmod m'_k\right) < r'_k$." The only other difference arises when a constant creator g, which is assigned a value v different than all variables assigned so far by (s_i, a_i), is processed. If $v \leq C$, then g is assigned the same value as v in s_i, a_i. Otherwise, a value $\tilde{v}$ greater than C and not equal to any of the variables created so far such that $(v \bmod m_k) = \left(\tilde{v} \bmod m_k\right)$ and $\left(v \bmod m'_k\right) = \left(\tilde{v} \bmod m'_k\right)$ for all k and k' is assigned to g by $\tilde{s}_i, \tilde{a}_i$. The bound follows by noting that 1) at most n values greater than C and with given residues with respect to m_k's and m'_k's are needed, and 2) for a set of residues with respect to m_k's and m'_k's, and for any o, there is a number between o and $o+N$ which has the same residues (QED).

The following lemma states that although the predicate $x<y$ cannot be used with data movement operations, but it can be used with all other predicates so long as there are no data movement operations.

Lemma 8.4.8 Let M be a DSSC with n integer variables using no data movement operations. Let m be,

a. n, if M involves only the predicates $x = y$ and $x<y$.

b. $C+n+1$, if M involves the predicates $x = y$, $x<y$, $x = c_l$, $x<d_j$, and $C = MAX(MAX(c_l), MAX(d_j))$.

c. $C+Nn+1$, if M involves the predicates $x = y$, $x<y$, $x = c_l$, $x<d_j$, $(x \bmod m_k) = r_k$, $\left(x \bmod m_k'\right) < r_k'$, where $C = MAX(MAX(c_l), MAX(d_j))$, and N is the least common multiple of the m_k's and m_k''s.

Then, $L(M) = \emptyset$ iff $L(M_m) = \emptyset$.

Proof a. The proof parallels the proof of lemma 8.4.1. The second property in the proof of lemma 8.4.1 is now modified to "$\alpha_{k,i} = \alpha_{l,i}$ iff $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{l,i}$, and $\alpha_{k,i} < \alpha_{l,i}$ iff $\tilde{\alpha}_{k,i} < \tilde{\alpha}_{l,i}$." Since there are no data movement operations in M, at every state s_i, the integer variables are assigned fresh constants. Let the constants be $c_{i,1}, \ldots, c_{i,l}$. Now choose the constants $\tilde{c}_{i,1}, \ldots, \tilde{c}_{i,l}$ such that the second property holds of the constants. Assign the rest of the gates as before.

b. Similar to part a, except that given constant $c_{i,1}, \ldots, c_{i,l}$ if $c_{i,j} \leq C$, let $\tilde{c}_{i,j} = c_{i,j}$.

c. Similar to proof of part b and lemma 8.4.7 (QED).

8.5 Verification Using Uninterpreted Functions and Finite Instantiations

8.5.1 Overview

In this section, we first prove that if ICS models are allowed to contain uninterpreted functions with predicates of the form $x = y$, where x and y are integers, then the ***reachability problem*** (whether a specific state s is reachable) becomes undecidable. This provides a limit for the applicability of finite instantiation techniques when uninterpreted functions are used. However, in verifying many properties of pipeline controls, where the functional units are replaced by uninterpreted functions, the predicates operating on data can be abstracted to be unconstrained inputs, i.e. taking any value 0 or 1. Therefore, data predicates are not needed in the model, but just in the property.

We prove that for ICS models with only uninterpreted functions, an important class of properties of the form "when binary variable b become true, $(x_1 = \alpha_1) \wedge \ldots \wedge (x_m = \alpha_m)$ ", where x_is are integer variables and α_i's are ICS terms, can be verified using finite instantiations. This result can be used, for example, to prove the following property Q, using two-bit integers, and appropriate interpretations for the uninterpreted functions. The property Q is: "Assuming r_1 and r_2 contain c_1 and c_2 initially, if the instruction $r_3 = ADD(r_1, r_2)$ (call it I) is issued after any stream of instructions which do not modify r_1 and r_2 , the result which is written to r_3 when I is retired is $ADD(c_1, c_2)$." Note that c_1 and c_2 can be any arbitrary integers, and all instructions before I cannot modify r_1 and r_2, but can read them. One can also prove that if any instruction after I has as one of its sources r_3 (and no intervening instruction has written to r_3), then it will receive the value $ADD(c_1, c_2)$, i.e. the bypass circuitry works correctly in this case.

Let $v(r)$ denote the value of register r, and assume we are interested in proving property P: when instruction $r_3 = ADD(r_1, r_2)$, is about to be retired, the value which is written back is $v(r_1) + v(r_2)$. Property Q is only an approximation to property P since no instruction before I is allowed to modify r_1 and r_2. To verify P, our approach is to run the unpipelined machine N in parallel with the implementation M. N halts after it executes I. After M retires I, it is checked that the r_3 registers of M and N are equal. This property has the form "when b becomes true, $x = y$", where x and y are integer variables.

Although we do not have techniques for exact verification using finite instantiations for this property (and there is some evidence that if such techniques exists, they may be expensive), we provide techniques which prove the property for a class of possible bugs. For example, they can catch any bug such that when b becomes true, the ICS terms stored at x and y differ in the number of times somc constant or uninterpreted function is used. For example, if x contains $f(a, b)$ and y contains $f(g(a), b)$, then this bug will be uncovered, since the function g does not occur in $f(a, b)$. Verifying this bug requires only one bit for each integer variable. This approach provides a combination of testing and verification, and should be effective in exposing bugs where little data variation is needed. [HYMD95] technique combines verification and simulation by building the graph of the "control part", and then producing a set of test vectors which ensure that all control arcs are traversed. This uncovered many subtle bugs, and confirms that, for exposing bugs in microprocessors, little data variation is needed. Our third technique is related to this in the sense that a very large set of test vectors is covered (specifically all control arcs will be checked), although 100% coverage is not obtained as with formally proving the property.

There is an extensive literature on verifying microprocessors. In most of these approaches ([Cyr93], [BD94], [Sri95]), a pipelined machine is compared to an unpipelined machine by proving that one step of the unpipelined machine is "equivalent" to one step of the pipelined one. These approaches require the user to provide a subset of the possible states of the implementation machine in which the correspondence can be proved. This set has to be large enough to contain all reachable states (otherwise verification is not complete), but it should not bc too large to contain unreachable statcs in which thc machine may behavc unprcdictably. Providing such an invariance, in theory, is equivalent to providing the set of reachable states, and therefore can be very challenging for complicated machines. These techniques have not so far been applied to the superscalar processors. [BaD94] also requires substantial user intervention, by asking the users to provide β-relations, and restricting the circuits to be k-definite. On the other hand, symbolic simulation algorithm of [Cor93] and [SS95]. However, [Cor93] does not give an algorithm for pipelined machines, and [SS95] requires some user intervention, and their algorithm may not terminate and can give false negative results. To our knowledge, there is no previous literature on verification using uninterpreted functions and finite instantiations.

The paper is organized as follows. Subsection 8.5.2 contains the technical part, proving the three main results: undecidability of ICS models with the predicate $x = y$, verification of properties involving $x = term$, and partial verification of properties involving the predicate $x = y$. Section 8.5.3 gives an application to the correctness of superscalar microprocessors. Section 8.5.4 discusses future research directions.

8.5.2 Main Results

In section 8.4, we showed that language emptiness of ICS models with only constant creators, data movement operations, and the predicate $x = y$ can be decided using finite instantiations. In this section, we prove the problem is undecidable if we allow, in addition, uninterpreted functions (and even disallow constant creators).

8.5.2.1 Undecidability of ICS Models with Predicate $x = y$

Theorem 8.2 The reachability problem for ICS models involving data movement operations, uninterpreted functions and the predicate $x = y$ is undecidable (note no constant creators are allowed).

Proof We reduce the halting problem for two-counter machines to our problem. A two-counter machine is a FSM with two integer variables called counters [HU79]. The counters can be incremented, decremented, and tested against zero. A transition can be represented as $(s, cond_1, cond_2, op_1, op_2, s')$, where,

a. s and s' are the current and next states respectively.

b. $cond_1$ and $cond_2$ are conditions on whether counters 1 and 2 are zero, not zero, or don't care. The transition is taken only if the conditions are satisfied in state s.

c. op_1 and op_2 are the operations performed on counters 1 and 2. Increment, decrement, and no-op are the possible choices.

For a two-counter machine M, we create an ICS model N with one uninterpreted function f representing increment, and a latch initialized to a constant c_0 and always holding that value.

1. For each counter, we define an increment circuit which operates as follows. Let I be the input to the increment circuit. When the circuit is enabled, it sets temporary variables y and z to $f(I)$ and c_0 respectively. The circuit continues only if $y \neq z$. This is accomplished by using an incompletely specified function which has an output only if $y \neq z$. If $z \neq I$, the circuit then lets

$z = f(z)$, and repeats the process. If $z = I$, the circuit outputs $f(I)$. Note that we have ensured that the only possible executions for N are those where no value between c_0 and I is equal to $f(I)$.

2. To decrement a counter, we define a circuit which operates as follows. When enabled, it first checks that its input I is not c_0. If so, it returns c_0, otherwise it sets a temporary variable y to c_0, and compares $f(y)$ to I. If the two are equal, it outputs y. If not, it lets $y = f(y)$, and repeats the process. It finally outputs a y for which $I = f(y)$.

3. To test a variable I against 0, N just checks whether $I = c_0$.

N simulates M by checking for a transition $(s, cond_1, cond_2, op_1, op_2, s')$, that $cond_1$ and $cond_2$ hold, and then performs op_1 and op_2 (using the subcircuits defined above), and moves to s'.

Let $f^i(I) = f(f^{i-1}(I))$. We note the following during the operation of N.

1. The increment subcircuit does not create an inconsistency, i.e. there are some interpretations for which all the predicates of the form $f^i(c_0) \neq f^k(c_0)$ (which this subcircuit creates) can be satisfied. One such interpretation is having integers as the domain, with f replaced by the increment operation on integers.

2. The decrement subcircuit always terminates. The reason is that the value at the input of a decrement subcircuit is always some formula $f^n(c_0)$, for some n, since these are the only values created by N. Also, note that if the input to a decrement subcircuit is $f^n(c_0)$, then the output of the circuit is $f^{n-1}(c_0)$, since when N created $f^n(c_0)$ it ensured that for all $0 \leq i \leq n-1$, $f^i(c_0) \neq f^n(c_0)$.

3. During the execution of N, the test $I = c_0$ returns true iff I is actually the formula c_0. To prove this, assume for some value i, $f^i(c_0) = c_0$. This is not possible since when $f^i(c_0)$ was created, the increment circuitry ensured that $f^i(c_0) \neq c_0$.

It follows that if

a. M is at some state s, and N is at its corresponding state s, and

b. if the two counters of M take the values i and k respectively, and the two counters of N take

$f^i(c_0)$ and $f^k(c_0)$ respectively, and

c. if M makes a transition $(s, cond_1, cond_2, op_1, op_2, s')$, with counters taking on i' and k' respectively,

then N will eventually end up in its state s' with counters taking $f^{i'}(c_0)$ and $f^{k'}(c_0)$ respectively. Since N merely simulates M, the reverse is also true. Hence, M falls into its halt state iff N reaches a corresponding halt state (QED).

Note that the proof of theorem 8.2 shows that the reachability problem for ICS models becomes undecidable when in addition to data movement operations and the predicate $x = y$, only one uninterpreted function (f), and one constant (c_0) are allowed.

8.5.2.2 Verification of Properties Involving Predicate $x = term$

Theorem 8.3 Let M be an ICS model with constant creators, data movement operations, and uninterpreted functions. Assume,

a. the integer latches in M take only constants as their initial values[1];

b. property P is "when b becomes true, $(x_0 = \alpha_0) \wedge \ldots \wedge (x_{m-1} = \alpha_{m-1})$ ", where b and x_i's are binary and integer variables of M respectively, α_i's are ICS terms, and the constants in the α_i's refer to the initial values of integer latches;

c. n is the number of all subformulas in the α_i's;

d. M_{n+1} is a finite instantiation of M where the uninterpreted functions and the constants in the α_i's are replaced by appropriate functions (constructed below) and constants on $n+1$ values (i.e. $\lceil \log(n+1) \rceil$ bits), and the constant creators are allowed to take values only from the set $0, \ldots, n$.

Let P_{n+1} be the property P which results from this interpretation. Then, P holds for M iff P_{n+1} holds for M_{n+1}.

Proof Recall that a state s in an ICS model is a tuple (L, Mem, P), where L is an assignment of values of all latches, Mem is a set of memory elements, and P is a set of predicates. Hence,

1. Not allowing numerals as initial values of integer latches is not a severe restriction in this case, since numerals are useful when interpreted integer functions are allowed, which is not the case here.

$f(a)$ may be equal to $f(b)$ in s if $a = b$ is a true predicate in s. Since there are no other integer predicates in M, two ICS terms are equal in any state of M iff they are exactly the same formula. The idea behind the proof is that if a formula β contains a subformula which does not occur in some other formula γ, then β and γ are not equal in any state of M. Hence, the set of all formulas can be divided into a finite set of equivalence classes: those which occur as a subformula in some α, and all other formulas.

Let $\beta_0, \ldots, \beta_{n-1}$ denote the set of all subformulas of α_i's, where $\beta_0, \ldots, \beta_{m-1}$ is $\alpha_0, \ldots, \alpha_{m-1}$. To define M_{n+1}, we need to give an interpretation $\bar{f}$ to each uninterpreted function f of M, and the constants which occur in the α_i's. If a constant c_l occurs as subformula β_k, then let c_l take the value k in M_{n+1}. For each uninterpreted function f, let $\bar{f}(k) = l$ if $f(\beta_k) = \beta_l$. Otherwise let $\bar{f}(k) = n$. Similarly define other uninterpreted functions of different arities, where if one of the argument is n, then the output is also n. Note that the value n propagates. Note that each predicate $x_i = \alpha_i$ in M becomes $x_i = i$ in M_{n+1}.

Assume P holds for M. Hence, by definition, P holds for all interpretations given to constants and uninterpreted functions. A set of such interpretations are represented by M_{n+1} and P_{n+1}. These interpretations are produced by defining the uninterpreted functions as we did above, and allowing the constants to take values only from the domain $0, \ldots, n$. Hence, it follows that P_{n+1} holds for M_{n+1}.

Conversely, assume P does not hold for M. Then, there exists a run $r = s_0, a_0, s_1, a_1, \ldots, a_{k-1}, s_k$ in M such that b holds in s_k but for some i, $x_i \neq \alpha_i$. We will build a run $\hat{r} = \hat{s}_0, \hat{a}_0, \hat{s}_1, \hat{a}_1, \ldots, \hat{a}_{k-1}, \hat{s}_k$ in M_{n+1} such that b holds in $\hat{s}_k$ and $x_i \neq i$, i.e. P_{n+1} does not hold for M_{n+1}. We do so by ensuring the following two invariants hold of all variables in any transitions (s_i, a_i, s_{i+1}) and $\left(\hat{s}_i, \hat{a}_i, \hat{s}_{i+1}\right)$.

a. All finite (non-integer) latches take the same values in (s_i, a_i, s_{i+1}) and $\left(\hat{s}_i, \hat{a}_i, \hat{s}_{i+1}\right)$.

b. If some integer variable z takes an ICS term β_k in (s_i, a_i, s_{i+1}), then z takes the value k in $\left(\hat{s}_i, \hat{a}_i, \hat{s}_{i+1}\right)$. Otherwise, z takes the value n.

We will show how s_0, a_0, s_1 can be constructed satisfying the above invariances; the other transitions are similar. Let O_0 be the topological ordering of the gates used in the transition (s_i, a_i, s_{i+1}) in M. We will use the same ordering to generate the transition $\left(\hat{s}_i, \hat{a}_i, \hat{s}_{i+1}\right)$ in M_{n+1}. The roots of O_0 are either latches (taking one of their initial values) or constant creators. For finite latches, let them take the same initial value in s_0 as in $\hat{s}_0$. Since the constants in the α_i's can only refer to the initial values of integer latches, if the initial value of a latch l is a constant and it occurs as a subformula β_k in one of the α_i's, assign it k. Otherwise, assign it n. Assign n to the constant creators. Now, by induction, assume the above two invariants hold, before some generalized gate g in M_{n+1} is processed. We show they hold for the output of g.

Case 1. If g is a finite table, then its inputs are the same as those in s_0, a_0, s_1. Hence, its output can be assigned the same value as in $\hat{s}_0, \hat{a}_0, \hat{s}_1$.

Case 2. If g is a data movement element, since the invariants hold for its input, they will hold for its output.

Case 3. If g is an uninterpreted function $\hat{f}$, by the second invariant, and the definition of $\hat{f}$, the second invariance will hold for the output of g.

It could be the case that processing gate g will affect whether the invariants hold for other variables besides the output of g. For example, if integer predicates were allowed, two formulas which were not equal before the predicate was processed, could become equal afterwards. This is not a problem in this case, since M does not have any integer predicates. Continuing by induction as above, the run $\hat{r} = \hat{s}_0, \hat{a}_0, \hat{s}_1, \hat{a}_1, \ldots, \hat{a}_{k-1}, \hat{s}_k$ can be constructed (QED).

8.5.2.3 Partial Verification of Properties Involving Predicate $x = y$

In the following, let M be an ICS model with constant creators, data movement elements, uninterpreted functions. Restrict M so that the initial values of the latches are constants. Let property P be "when the binary variable b becomes true, the two integer variables x and y are equal".

Definition Let α be an ICS term. Let $f_0, \ldots, f_m$ be the uninterpreted functions occurring in α. Given numerals $a_0, \ldots, a_m$, the ***linear expansion*** of α with respect to $a_0, \ldots, a_m$, $lin(\alpha)$, is obtained by adding all the symbols occurring in α, with each f_i having value a_i. For example, given uninterpreted function f and g with corresponding values $(1, 0)$, the linear expansion of ICS term $f((g(a, b)), f(a))$ is $1 + 0 + a + b + 1 + a$ which is $2 + 2a + b$.

Lemma 8.5.1 Let α be an ICS term, involving uninterpreted functions $f_0, \ldots, f_m$ and constants $c_0, \ldots, c_n$. Let $a_0, \ldots, a_m$ be a set of numerals. If each f_i is given the interpretation, which returns the sum of its arguments plus a_i, then the value of α under this interpretation is $lin(\alpha)$.

Proof Clear from the definitions (QED).

Lemma 8.5.2 Assume P does not hold for M, and let k be a given integer. Assume for some run $r = s_0, a_0, \ldots, a_{n-1}, s_n$,

a. in s_n, b becomes true, $x = \alpha$, $y = \beta$, $\alpha \neq \beta$,

b. for some constant c_i occurring in α or β, the coefficients of c_i in $lin(\alpha)$ and $lin(\beta)$ with respect to $0, 0, \ldots, 0$ (denoted by n_{i_α} and n_{i_β}) have different residues modulo k.

Let M_k be a finite instantiation of M, obtained by letting the constant creators take values 0 and 1, and each uninterpreted function f_i returning the sum of its arguments modulo k. Then, P does not hold for M_k either.

Proof Since there are no integer predicates, the values which the integer variables take have no impact on the finite variables. Hence, there exists a run $\hat{r} = \hat{s}_0, \hat{a}_0, \ldots, \hat{a}_{n-1}, \hat{s}_n$ in M_k which agrees with r on the values of finite variables. By the definition of f_i's and lemma 8.5.1, each integer variable z in $\hat{s}_j, \hat{a}_j$ can take the value $lin(z)$, where z denotes the value of z in s_j, a_j. Let all constants be 0, except for c_i, which is 1. It follows that x and y will take n_{i_α} and n_{i_β} respectively in $\hat{s}_n$. These values are not equal by assumption. Hence, P does not hold for M_k (QED).

An example of the application of lemma 8.5.2 is the following. Assume, for some run $x = \alpha$, $y = \beta$, $\alpha = f(a, b)$ and $\beta = f(a, c)$. Then $lin(\alpha) = a + b$, and $lin(\beta) = a + c$. Since the coefficients of b and c are not equal modulo 2, P will not hold of M_2.

To answer the question of how many k's should be tried, note that if two integers a and b have the same residues modulo three primes P_1, P_2, P_3, $a = b$ and $b < a$, then

$a = b + n(P_1 \times P_2 \times P_3)$ for some n. It is expected that even one bit integers, for which $k = 2$, will expose many of the bugs, since most bugs are control bugs, and should occur under many different data combinations.

Notation For an ICS term α, let α_f denote the number of times f occurs in α.

Notation For an uninterpreted function f, let $S_f(a)$ denote a function which adds a to the sum of the arguments of f. Hence, $S_f(1)$ denotes a function which adds 1 to the sum of f's arguments.

Lemma 8.5.3 Let P not hold for M. Let k be an integer. Let there be a run $r = s_0, a_0, \ldots, a_{n-1}, s_n$, such that in s_n, $b = 1$, $x = \alpha$, $y = \beta$, $\alpha \neq \beta$, and for some uninterpreted function f of M, $\alpha_f \neq \beta_f$ modulo k. Let M_k be a finite instantiation of M, obtained by letting the constant creators take value 0, f returning $S_f(1)$ modulo k, and each other uninterpreted function $g \neq f$ returning $S_g(0)$ modulo k. Then, P does not hold for M_k.

Proof The proof follows by noticing that in M_k, one can create a run in which when b becomes true x takes α_f modulo k and y takes β_f modulo k, which are not the same by assumption (QED).

Lemma 8.5.4 If techniques of lemmas 8.5.2 and 8.5.3 do not uncover any bugs in the verification of P, assuming that sufficiently many k's have been tried, the only possible bugs are where when $b = 1$, x and y take unequal ICS terms, but the number of occurrences of every symbol is exactly the same.

Proof Follows directly from lemmas 8.5.2 and 8.5.3 (QED).

An example of the situation in lemma 8.5.4 is $x = f(g(a, b), a)$ and $y = f(a, g(a, b))$. A simple technique which seems to produce good results is to have each uninterpreted functions return a weighted sum modulo k of its arguments, where the i-th argument is multiplied by i. For example, in this case, $x = (a + 2b) + 2a = 3a + 2b$ and $y = a + 2(a + 2b) = 3a + 4b$, where the coefficients of b are not the same. By creating finite instantiations where the constant creators are allowed to take values $(0, 1)$, and the uninterpreted functions are defined as above, such bugs can be uncovered.

8.5.3 Applications to Superscalar Microprocessors

We present an application of the results of the previous section to some correctness problems of superscalar microprocessors (see [Joh91] and [HP90] for background information). Superscalar microprocessors take advantage of instruction parallelism by allowing multiple instructions to be issued and completed in any cycle. Many commercial microprocessors designed in the past few years are superscalar. Throughout this section, we consider an imaginary processor M, which issues instructions out-of-order, allows out-of-order completion of instructions, does register renaming, allows speculative execution, and retires (writes back) the instructions in order. In all superscalar microprocessors we know off, results are retired in-order since precise interrupts and speculative execution (where a branch's outcome in predicted in advance) need to be supported.

When an instruction is issued, it enters a reorder buffer ([SP86]). The ***reorder buffer*** is a table consisting of a set of rows corresponding to different instructions and a set of columns corresponding to information about the instructions. The rows of the table are implemented as a circular FIFO. Hence, there are two pointers: head and tail, pointing to the first and last entries. The columns consist of an entry number (or tag), destination register, result, exception information, valid bit saying whether the result is valid, and the program counter of the instruction. When an instruction is issued, it is given the entry pointed to by the tail of the reorder buffer, and the instruction is sent to the reservation stations of the functional units for execution.

A ***reservation station*** is a buffer which contains all instructions waiting to be executed in a functional unit. Each entry in a reservation station contains the following fields: the instruction's tag, operation to be performed on the two sources, values of sources 1 and 2 or tags for the instructions which will produce them, and 2 bits stating whether sources 1 and 2 are valid. When an instruction is issued, if one of its arguments, say register r, has not been computed, then the tag for the last instruction in the reorder buffer which will write r is saved in the instruction's entry at its reservation station. When an instruction finishes, the result is written to any reservation station which is waiting for this result by comparing the tag of the instruction to the tags at the reservation stations.

When the instruction at the head of the reorder buffer contains valid information, and there are no exceptions, then the result is written to the destination register. If there is an exception associated with the instruction, then the writing of results is stopped, and the corresponding exception handler is called. The program counter of the instruction is used to resume execution after the exception handler returns. To implement speculative execution, if the wrong branch was taken after a jump instruction, all entries after the jump instruction are deleted from the reorder buffer.

Consider an instruction I, $r_3 = ADD(r_1, r_2)$, where r_1 and r_2 are source registers, r_3 the destination, and the values stored at r_1 and r_2 are constants a and b. A property P one may want to verify is that if I is issued with tag number t, when t is at the top of the reorder buffer, then the value stored under the result entry in the reorder buffer is $ADD(a, b)$, provided that there are no exceptions associated with I. To use the results of the previous section, several assumptions are made.

1. The number of registers and the size of the memory is finite.
2. The predicates operating on data can take any value (0,1) at any point in time. This abstraction might prevent us from proving P by allowing too much behavior only if the correctness proof somehow depends on always making the same choice for a predicate Q, i.e. if at some point $Q(x) = 1$, then at a later point, $Q(x)$ should be 1 as well. This would imply that the pipeline control remembers the value of $Q(x)$, which does not appear to be the case in general.
3. We assume that the instructions are generated non-deterministically, but no instruction before I is issued is allowed to modify r_1 or r_2. The initial values for r_1 and r_2 are set to be the constants a and b respectively. To catch errors associated with speculative execution, a program buffer is provided. The ***program buffer*** is implemented as a circular FIFO, and has a tail and a head pointer. The tail pointer marks the location where the next instruction is retired. The head pointer points to where the current instruction is being read. As soon as an instruction is retired, a new instruction is produced (non-deterministically) in its place. An instruction has four fields: operation, sources 1 and 2, and destination.
4. Each functional unit is replaced by an uninterpreted function.

The verification of this property can then be done by replacing each integer variable by two bit variables, since $ADD(a, b)$ has three subformulas: a, b and $ADD(a, b)$. Each functional unit should also be replaced by the function described in the proof of theorem 8.3.

The process of verification using a tool such as [HSIS94] would be as follows (we have not yet implemented the techniques presented in this paper). The Verilog description of the reorder buffer and its interface to the functional units are chosen for verification by the user. The description is compiled into the intermediate format BLIF-MV (chapter 2). The compiler is instructed to use library functions for integer operations such as addition (as opposed to creating a circuit implementing addition), and to leave the functional units undefined. For example, $z := x + y$ for 32 bit integers is compiled into $HSIS-add-32(x, y, z)$. For our Verilog compiler ([Che94]), the user needs to identify the uninterpreted functions. The user then inputs the property as an automaton or a CTL formula. The BLIF-MV file and the property are read, and are used to automatically extract an ICS model. It is then checked whether the property can be verified using finite instantiations. If so, the finite instantiation is created automatically and the standard verification routines are called.

For example, if the property is "when the result of instruction $ADD(r_1, r_2)$ is written back, $ADD(a, b)$ is saved", then a finite instantiation with two bits is created. Formulas a, b and $ADD(a, b)$ are represented by 0,1, and 2 respectively. Value 3 is used to denote all other formulas. If there is an error, the result has to be changed to be readable by the user. Specifically, the integer values need to be changed to formulas. One solution is to use formulas a, b and $ADD(a, b)$ for 0, 1, and 2 respectively, and use some special symbol such as *other* to denote other formulas. The user can then use this error trace to correct the design.

To get an idea of how large (in terms of the number of latches) the reduced systems can get, assume there are 4 registers, 4 memory locations, 4 instructions in the program buffer, 1 bit for exception information, 2 bits for data, and the tags of the reorder buffer are the locations of each entry in the buffer and therefore are not stored in the buffer, then the reorder buffer requires 8 bits per entry. If we further assume there is 1 bit for the operation in each functional unit and 2 bit tags, then each entry in the reservation stations requires 9 latches. Assuming that 8 different instructions are modeled, the program buffer requires 9 latches per entry. If there are 4 entries in the program and reorder buffers, and 2 reservation stations each holding 2 entries, we get a total of 128 latches: 8 for registers, 8 for memory, 40 for the program buffer (36 for entries and 4 for pointers), 36 for the reorder buffer (32 for entries and 4 for pointers), and 36 for reservation stations. In general, let m denote the number of bits for memory locations, r the number of bits for registers, d the number of bits for data, x the number of bits for exception information, f the number of functional units, e the number of nurtures in each reservation station, o the number of bits in the operations at each functional unit, i the number of bits required to represent an instruction type, p and b are the number of bits for the sizes of the program and reorder buffers. Then, the total number of latches is given by $[d(2^r + 2^m)] + [2^p(i + 3max(r, m)) + 2p] + [2^b(r + d + x + 1 + p) + 2b] + fe(\log b + 2d + 2 + o)$.

Another property one might verify of M is that when the result of I is computed, any instruction in the reservation stations waiting for the result of I, will receive $ADD(a, b)$. This property tests the by-pass circuitry. Yet, a stronger property to verify is that for any stream of instructions, even those which modify r_1 and r_2 before I is issued, when I is retired, the value written back is $ADD(v(r_1), v(r_2))$, where $v(r_1)$ and $v(r_2)$ denote the values of registers r_1 and r_2 when I was issued. To verify this property, one solution is to run an instruction level interpreter N in parallel with M. N merely executes the functional specification of each instruction. It duplicates the memory and registers of M, and stops right after it executes I. After M writes back I, it is checked that the r_3 registers of M and N are equal. To verify this property, techniques of section 8.5.2.3 are needed.

8.5.4 Conclusions and Future Directions

The finite instantiation techniques are a way of addressing state space explosion due to wide datapaths, by reducing the size of the datapaths to a few bits, with no or little sacrifice in verification's accuracy. We extended the results of sections 8.3 and 8.4 to uninterpreted functions, and applied the techniques to the correctness of pipeline controls of superscalar processors. We also showed the limits of the finite instantiation methods by proving that the problem becomes undecidable when uninterpreted functions and the predicate $x = y$ are used simultaneously. An important research area is studying the ramification of considering infinite memory. A simple case where the memory is never read by the machine but only read once by the property is dealt with in chapter 9. Also, in cases where finite instantiation cannot be applied, efficient symbolic execution algorithms are needed. We present such an algorithm next.

8.6 Symbolic Execution of ICS Models

The idea for symbolic execution of ICS models (which can be infinite) is to compute the set of states reachable in n steps, and check whether some infinite accepting path exists in this subset. An overview of a symbolic algorithm using BDDs for an acyclic ICS model M is given below.

1. *Compute the set of reachable states, or a subset of it reachable in n steps. Let this set be S, and represent it using BDDs*
2. *Let T_S be the transition relation of model M restricted to S. Represent T_S using BDDs.*
3. *Using techniques of chapter 4, check whether there are any accepting runs in T_S.*

Step 3 of the above algorithm is well-defined. In what follows, we explain steps 1 and 2 in detail.

8.6.1 Reachability Analysis of ICS Models

In this section, we present our algorithm for computing the set of reachable states of ICS models. To compute the set of reachable states, three continually expanding domains are used. The ***term domain*** H_I is a table of all ICS terms which have been stored in latches in states visited so far. The ***predicate domain*** H_P is a table of all predicate tables enumerated so far, where a predicate table contains a set of predicates. The ***memory domain*** H_M is a table of memories, where a memory is a table from ICS terms to ICS terms. One can encode a variable ranging over a finite domain, using a set of binary variables. We call a set of binary variables corresponding to a finite-valued variable also a variable. The set of states reachable in i steps is represented by a BDD, where there is a variable for each finite latch, a variable for each integer latch ranging over all terms in H_I, a variable p ranging over all predicate combinations in H_P, and a variable m whose domain is all memories in H_M. Note that the number of variables as well as their ranges are finite.

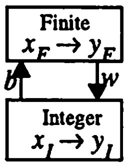


Figure 8.4: Finite and integer parts in ICS models

Let x_F, y_F, x_I, and y_I represent the present and next states of the finite and integer parts respectively, as shown in figure 8.4. Note that x_I and y_I contain variables for integer latches, as well as predicate and memory tables. Let $T_F(x_F, y_F, b, w)$ and $T_I(x_I, y_I, b, w)$ be the transition relations for the finite and integer parts. Although T_F can be computed easily by taking the intersection of all the finite tables, T_I may be infinite, and hence cannot be fully computed. Let $R_i(x_F, x_I)$ be the set of states reached after i steps. We have $R_{i+1}(y_F, y_I) = \exists b \exists w \exists x_I \exists x_F (T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I) \wedge T_I(x_I, y_I, b, w))$. By re-arranging the terms we get $R_{i+1}(y_F, y_I) = \exists b \exists w \exists x_I (\exists x_F (T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I)) \wedge T_I(x_I, y_I, b, w))$.

Let $U_i(y_F, x_I, b, w) = \exists x_F(T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I))$. The algorithm first computes $U_i(y_F, x_I, b, w)$, and then the portion of $T_I(x_I, y_I, b, w)$ corresponding to all minterms in $U_i(y_F, x_I, b, w)$. Note that this is exactly the part of $T_I(x_I, y_I, b, w)$ needed to compute $R_{i+1}(y_F, y_I)$. The algorithm maintains the invariance that in every state (l_F, l_I, p, m) the set of predicates in p imply that the addresses of the memory m are distinct. The algorithm is sketched first with more details given later.

1. Let $U_i(y_F, x_I, b, w) = \exists x_F(T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I))$.

2. For each $(x_I, b, w) \in \exists y_F U_i(y_F, x_I, b, w)$,

2a. Propagate values through the integer part, choosing the values given by b *to integer predicates if possible. If not, the tuple* (x_I, b, w) *is inconsistent, so stop this value propagation.*

2b. Add to H_I *the new ICS terms assigned to next state variables. Extend the range of all integer latches to accommodate this.*

2c. Add the new integer predicates to the predicate table. If the new predicate table is not in H_P*, add it to it, and extend the range of* p.

2d. If the new memory has not been encountered before, add it to H_M*, and extend the range of* m.

3. Let $T^I_{i+1}(x_I, y_I, b, w) = \sum\{b \wedge w \wedge (x_I, y_I)\}$ *, where* y_I *are the new values assigned to integer latches, predicate table and memory as the result of propagating* (x_I, b, w) *. Note that there may be several such choices.*

4. Let $R_{i+1}(y_F, y_I) = \exists b \exists w \exists x_I \left(U_i(y_F, x_I, b, w) \wedge T^I_{i+1}(x_I, y_I, b, w) \right)$.

The details of step 2 above are as follows.

1. In step 2a, given a predicate Q and a binary value (true or false) for it b_Q, the validity of both $Q_1 \wedge \ldots \wedge Q_k \rightarrow (Q = Q_b)$ and $Q_1 \wedge \ldots \wedge Q_k \rightarrow \left(Q = \overline{Q_b}\right)$ are checked, where $(Q_1, \ldots, Q_k)$ are all the predicates in the predicate table of x_I. If the former is valid, value propagation is continued. If the latter is valid, value propagation is stopped. If neither is valid, then Q is added to the predicate table, and is set to Q_b.

2. If the memory operation $read(x)$ is encountered during value propagation, find an address

y in M_H, such that $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x = y)$ is valid, and return the value pointed to by y. If no such address is found, for each address y in M_H, check whether $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x \neq y)$ is valid. For each address y where this is not valid, introduce a new predicate $x = y$, and make a case split based on whether this predicate is true. If it is true, then return the value pointed to by y. In the case where all such predicates are false, return a fresh new constant. This last situation corresponds to reading a location whose value is unknown.

3. If during value propagation, the memory operation *write* (x, d) (write the value d to address x) is encountered, check whether for any address y $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x = y)$ is valid. If such a y is found, change the value of y to d. If not, check the validity of $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x \neq y)$ for each address y. For each address y where this is not valid, introduce a new predicate $x = y$, and make a case split based on whether this predicate is true. If it is true, change the value of y to d. If all such predicates are false, add a new entry (x, d) to the memory. This last situation corresponds to writing to a new address.

4. If we disallow arithmetic functions and predicates (addition, mod, etc.), only congruence closure ([NO80]) (which is fast) is needed for deciding validity of logical formulas which arise during reachability analysis.

5. After value propagation, it is checked whether the created memory and predicate table has been encountered before. If not, the ranges of H_M and H_P are extended.

The complexity of the algorithm is based on the complexity of the finite partition, the number of predicates in the datapath, the amount of communication from the finite partition into integer partition (the set w), and the number of distinct pairs of (l_P, m) at every point in time.

Remark If only integer functions and predicates are used, there are no constant creators, and the integer latches are initialized to a finite set of values, then the ICS terms obtained during value propagation are integers, and can be evaluated using the computer's arithmetic functions. This situation might for instance come up when handling a complete micro-processor. In this case, the above algorithm can be used to generate all states reachable within n steps. If there is no non-determinism, then the algorithm reduces to a simulation algorithm, which uses BDDs to represent the control part, and uses the computer's arithmetic functions to compute the values encountered in the datapath.

8.6.2 Comparison With Previous Work

Three previous works are most relevant.

1. *Functional Equality.* [BD94] introduced techniques for checking equality of two functions: one corresponding to flushing the pipeline first and then running one instruction on the specification, and the other corresponding to running one instruction on the implementation and then flushing the pipeline. This type of verification can be approximated (by comparing the reached states along the two paths) using ICS models, and its verification is decidable, since the pipeline is executed only for a finite number of steps. The disadvantage of [BD94]'s method compared to the ICS approach is that it is very specialized, i.e., general property checking cannot be done. However, it might have a computational advantage on those problems where both techniques are applicable.

2. *Extended Temporal Logic.* [CN94] introduced an extended temporal logic, called ground temporal logic, with pretty much the same expressiveness as ICS models. They suggested that transition diagrams be translated into this logic, and the validity of the formulas be checked. This approach has the same drawbacks as when validity of linear temporal logics is used as the computational means for verification; there are no known efficient procedures for model checking a linear temporal formula by checking the validity of an LTL formula expressing both the transition structure and the property. In particular, one can expect that systems with large control circuitry cannot be efficiently model checked using this approach.

3. *Multi-Way Decision Graphs (MDGs, [Cor93]).* MDGs, which can represent finite and integer latches as one data structure, do not appear to be directly applicable since they do not provide any means for representing memories or predicates.

Acknowledgments

The material presented in this chapter are based mainly on [HB95] and [HIKB96].

Chapter 9

Verification of Memory Systems With Out Of Order Execution

9.1 Introduction

In order to achieve higher performance, a memory module may service requests issued by a processor out of order ([Dub88], [GLL90]). In general, the more relaxed the memory model, i.e. the more out-of-order execution is allowed, the higher the performance. However, this extra performance comes at the cost of more complicated programming ([AH90]). Recently, there have been efforts to develop formal models for such memory systems ([SFC91], [Col92]). These models concisely specify the memory system, and allow for manual or automatic verification. One such formal model is the partial order model ([SFC91]), which consists of a set of mathematical properties that the memory model has to satisfy.

In this chapter, we express these mathematical properties using ω-automata. Language containment can then be used to automatically verify that these properties hold for a hardware system. We refer to this type of verification as ***hardware verification***. In some cases, one may want to verify the correctness of programs written for some architecture, where an architecture is a family of memory systems satisfying a set of properties. We present a general framework based on language containment to perform such ***software verifications***. An approach used previously ([DPN93]) is to take an abstract implementation, and prove the properties of the programs running on this implementation. However, this technique cannot guarantee that the program will have the desired property running on any implementation of the memory model unless it is proven (possibly using a theorem-prover) that the abstract implementation is the most general implementation of the architecture. In contrast, our technique provides such a guarantee.

The size of the state space is a key limiting factor in automatic verification using state space exploration. The state space depends on the number of data values, lengths of the internal buffers, number of processors, and number of memory locations. Ideally, we would like to verify systems with all these parameters being unbounded. We use abstraction results from chapter 8 to reduce the number of data values to just two, and the number of memory locations to a small number without losing any accuracy in our verification, once the number of processors and the sizes of the buffers are fixed. As an example, we study SPARC's V8 memory model ([Sun90]). We describe the complete formal specification, and give some experimental results on a simple model. The BDD-based verification tool SMV ([McM93]) was used for our experiments.

The chapter is organized as follows. In section 9.2, we formulate various verification problems for memory models, and discuss some previous works on verifying memory systems. In section 9.3, the language containment paradigm is introduced, and formal specifications for ***total store ordering (TSO)*** and ***partial store ordering (PSO)*** memory models of the SPARC architecture ([Sun90]) are given. Section 9.4 explains how to deal with the complexity of verification by making simplifying assumptions and by developing a verification strategy. Section 9.5 applies some data abstraction results and symmetry arguments to simplify the verification task without losing accuracy. Section 9.6 gives our experimental results. Section 9.7 presents a verification strategy for memory models.

9.2 Verification Problems for Memory Models

A ***memory model*** or ***memory architecture*** is a set of properties (also called rules or axioms) which determine what set of behaviors is allowed. These rules restrict the amount of out-of-order execution the memory system can have. An example of a rule is: if a processor issues a store to a location followed by a load from the same location, then the store is performed first. We recognize three general categories of verification problems when dealing with memory models.

1. ***Hardware Verification*** amounts to verifying whether a given hardware system satisfies a set of properties. This verification arises in two different ways.

a. As hardware designers attempt to increase the performance of memory models, they come up with new designs. They might then try to formally specify their implementations by checking whether a given implementation satisfies a set of properties. Hence, verification is used to define the memory architecture.

b. Sometimes hardware designers are given an architectural specification, i.e. a set of properties, and they would like to verify that their implementation satisfies the architectural specification. In this case, verification is used to verify an implementation versus its formal specification.

2. ***Software Verification*** is the process of verifying that a program runs correctly on a given architecture. For instance, one may verify that a correctly functioning program on an old architecture behaves correctly on a new architecture. Given a program and an architecture, there are

two software verifications of interest:

a. ***Software Property Verification*** checks that a program running on a given architecture has a certain property, such as mutual exclusion.

b. ***Software I/O (Input/Output) Verification*** checks that given a set of inputs the program can or cannot produce some set of outputs ([Col92]). Usually, the user has some property in mind to check for correctness. The designer specifies some initial values (similar to test vectors) and some set of desirable final values. Software I/O verification is similar to simulation, whereas software property verification more closely follows the classical definition of formal verification.

3. ***Architectural Verification*** ([Col92]) investigates the relationship between different architectures. In particular, one may ask whether two architectures

a. are ***distinguishable***, i.e. there is some program and some input sequence for which the set of outputs generated by the two programs is different;

b. have some implication relation, i.e. whether a set of properties imply another set of properties. This kind of verification is known as ***property implication***.

From the above set of problems, hardware and software property verifications are the more practically significant problems. It appears that software I/O verification is in general used to check some property. Hence, by offering a solution for software property verification, software I/O verification does not have much practical use. In theory, software I/O verification can be done in our framework, however, we do not give any methods for doing so.

9.2.1 History of Verifying Memory Models

In this section, we describe some previous efforts in verifying memory systems.

9.2.1.1 Partial Order Approaches [SFC91]

Given a program, partial order methods describe the properties of memory systems by a set of axioms. Assume we are given a finite multi-processor program. Two partial orders on programs' instructions are defined: one on the set of requests (;), and one on the set of services (<). These two orders correspond to the relative timing of events. The axioms relate the order of requests to the order of services. For instance, one axiom might be of the form $(S, i, a, x, l, R) ; (S, i, b, x, l, R) \rightarrow (S, i, a, x, l, A) < (S, i, b, x, l, A)$, which is interpreted as: if a processor requests two memory stores to two different locations, then the first is serviced before the second. [SFC91] describes the axioms needed for modeling TSO and PSO.

The partial order models have been used only for formal specification, i.e. no automatic verification has been applied to them so far. The verifications performed using them have been program verifications, where one proves (with paper and pencil) that a small program produces the desired set of outputs, when run on a system satisfying the axioms of a given memory. For example, [SFC91] gives the code and proof of correctness for achieving mutual exclusion of two processes in their critical sections on a machine supporting PSO.

9.2.1.2 Graph Set Approaches [Col92]

In this approach, a graph is built where the nodes are the events in the system. The processors are allowed to have local memory. Hence, for every read/write event a set of read/write events (one for each processor) is generated. [Col92] describes two types of verification with this approach: program and (limited form of) architectural verification. For program verification, one proceeds as follows:

1. A set of rules defining the memory models is selected.
2. A program consisting of code, initial values of arguments for each memory location, and final values of arguments is chosen.
3. Each rule imposes a finite set of constraints on the relative ordering of events. These constraints are represented by a set of graphs, called ***graph set***, where the nodes are the set of events of the program. A graph set represents all possible behaviors of the program, running on the given memory model. If any of the graphs in the graph set is acyclic, then there is a possible execution of the program computing the final results from the initial ones. For architectural verification, [Col92] deals with the fact that a set of rules does not imply another set by giving counter-examples, and gives a few manual proofs for property implication problems ([Col92], chapter 15).

The methodology presented in [Col92] is close to ours, in the sense that all possible behaviors are represented. A major disadvantage is a lack of methods for dealing with H/W and property S/W verifications. Also, no efficient algorithms for automating software I/O verification is presented, although it may be possible to automate [Col92]'s methods for it.

9.2.1.3 The Operational Model Approach

[DPN93] describes an ***operational*** approach. A simple implementation of a given memory model is developed, which is later proved ([PD95]) to be equivalent to its partial order specification. What is meant by equivalence is that a memory system satisfies its partial order formal specification if and only if the behavior of the memory system is included in the behavior of the operational model. The form of verification reported is program verification, where the mutual exclusion for a fragment of a synchronization code is automatically verified. [GMG91] describes another operational approach to software verification using manual theorem-proving.

9.3 Automata Models for TSO and PSO

Assume we are given three memory operations load (*L*), store (*S*) and atomic load-store (*ALS*), *p* processors, *m* memory locations, *v* values, *l* program labels, and a flag indicating whether the operation is a request (*R*) or service (*A* for acknowledged). An event is represented by a 6-tuple *(operation, processor, memory location, value, label, service/request)*. Note that we have assumed that all processors see the effect of services at the same time; this property is known as ***atomicity of operations*** ([GLL90]). This is in contrast to the models described in [Col92], where each process has a local memory, and hence the effect of a load or store can be seen at different times at different processes. The models described in [Col92] can easily be accommodated by adding another field to the event tuple, specifying which processor is observing the effect of the operation. We ensure atomicity of operations by having a single shared global memory. If the processors have local caches, the equivalence of the machine with and without caches should be verified independently.

Since the 6-tuple notation is not very readable, we adapt the notation used in [SFC91] with one addition: for each operation, a flag is added to indicate whether the operation is a request (R) or a service (S). Therefore, L_a^i denotes a load from location a by processor i, $S_a^i n$ denotes a store to location a by processor i of value n, $[L_a^i;S_a^i]$ denotes an atomic load-store (note the stored value has been omitted, we will make such abbreviations when the meaning is clear), SO denotes a store operation, O denotes any operation, $O-R$ and $O-S$ represent request and service for operation O respectively, and (O, n) denotes operation O with label n.

The program labels are unique time stamps. Hence the same operations to the same location at different times will have different labels. If a request for an operation with label k occurs, we are guaranteed that next service with label k corresponds to that operation. To get a finite state system, one has to bound the number of labels of a program. These programs are called ***programs of finite character***, and have a constant bound on the number of un-serviced instructions. We re-cycle the labels so that we can verify infinite programs of finite character. However, we never re-cycle a label, until its corresponding operation has been serviced. This fact will reduce the number of automata we need, since sometimes we do not need to keep track of all the information in an instruction, and just its label. For each type of property, we specify the number of automata needed per processor. Note that for an atomic load-store, the services for the load and store parts have the same label.

9.3.1 Automata Specifications of PSO and TSO

Most axioms of a partial order specification exclude some set of behaviors and accept others. For example, in the model presented in [SFC91], all axioms, except for the order axiom, reject or accept some set of behaviors. The order axiom seems to be included for technical reasons, arising from dealing with partial orders. We directly translate each axiom of [SFC91] into an ω-automaton such that if a hardware system satisfies an axioms, then the system's language is contained in the language of the corresponding property.

Atomicity asserts that an atomic load-store happen atomically, i.e. no other stores can occur between the load and store parts of an atomic load-store. For each atomic load-store $[L^i;S^j]$, we define the following ω-automaton. We need $p \times l$ such automata, one per processor per label. In the graphical representations of ω-automata, dashed rectangles represent positive fair subsets, dashed edges represent positive fair edges, and text inside nodes represent names of states.

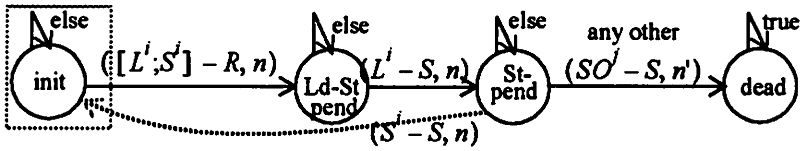
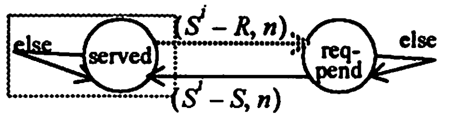


Figure 9.1: The atomicity property

Termination asserts that all stores and atomic load-stores terminate. This is a ***liveness property***, i.e. a property of the form "something good will happen eventually." Our specification of atomicity guaranteed that all atomic load-stores terminate; the behavior where an atomic load-store is requested and not serviced is not accepted. Hence, we only have to worry about stores now. For each store S^j, we define the following ω-automaton. $p \times l$ such automata are needed.



This automaton specifies that every store is eventually *serviced*.

Figure 9.2: The termination property

Value asserts that the value returned by a load issued by processor i from location a is the value written by the "last store" to location a. We will denote by $v\left[S_a^j\right]$ and $v\left[L_a^i\right]$ the values of a store or a load service to location a by processor i. The automaton for this axiom stays in its initial state until it receives the request $S_a^j - R$ or $S_a^j - S$ (servicing of a store by a different processor). At this point, the automaton remembers the value to be stored. When $L_a^i - R$ occurs the automaton sets a flag and waits for $L_a^i - S$, at which point it compares $v\left[L_a^i\right]$ with its remembered value. If they are not equal, the automaton goes to a dead state. The automaton requires $2v + 2$ states, where v is the number of data values. $p \times l \times m$ such automata are needed.

```
1. init: if $S_a^j - R$ goto $\left(v\left[S_a^j\right], 0\right)$
        else if $S_a^j - S$ goto $\left(v\left[S_a^j\right], 0\right)$
        else stay put
2. $(k, 0)$ : if $S_a^j - R$ goto $\left(v\left[S_a^j\right], 0\right)$
        else if $S_a^j - S$ goto $\left(v\left[S_a^j\right], 0\right)$
        else if $\left(L_a^i - R, n\right)$ goto $(k, 1)$
        else stay put
3. $(k, 1)$ : if $\left(L_a^i - S, n\right)$ and $v\left[L_a^i\right] = k$ goto init
        else if $\left(L_a^i - S, n\right)$ and $v\left[L_a^i\right] \neq k$ goto dead.
        else stay put
4. State dead : stay put.
```

Figure 9.3: The value property

LoadOp states that if a processor issues an operation after it has issued a load operation, the load is serviced first, i.e. loads are "blocking." For every load L^i and every operation O^i, we define an ω-automaton as follows. We need $p \times l$ such automata.

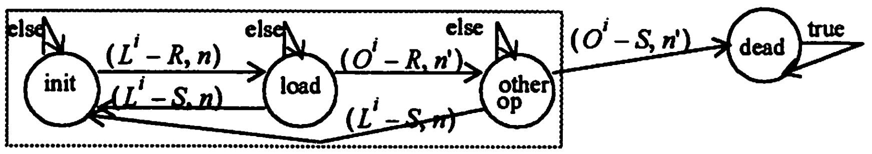


Figure 9.4: The LoadOp property

StoreStore (TSO) states that for every processor, SO's are serviced in the same order as requested. $p \times l \times (l-1)$ such automata are needed.

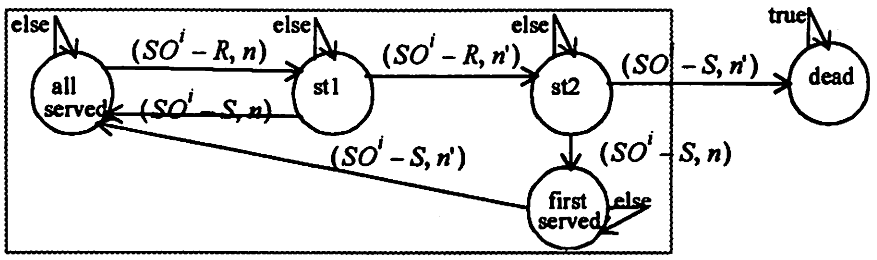


Figure 9.5: The StoreStore property for TSO

To specify PSO, we leave the first four axioms unchanged. We add the following two axioms.

StoreStore (PSO) states two SO's of a processor separated by an *stbar* are serviced in the same order as requested. $p \times l \times (l-1)$ such automata are needed.

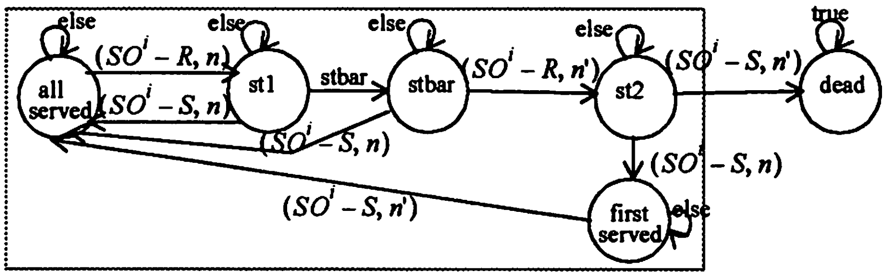


Figure 9.6: The StoreStore property for PSO

StoreStoreEq states two SO's of a processor to a location are serviced in the same order as requested. We need $p \times l \times (l-1) \times m$ such automata.

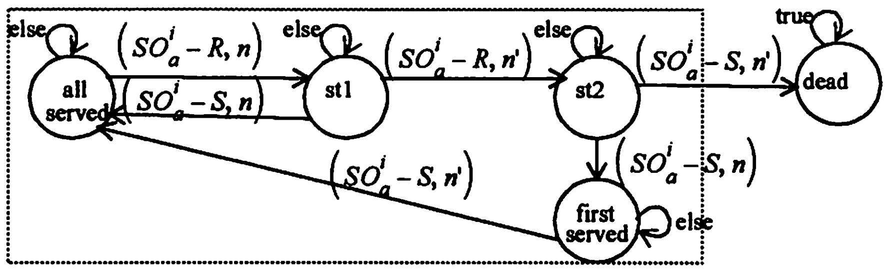


Figure 9.7: The StoreStoreEq property

9.4 Hardware and Software Verification

In this section, we discuss how one proceeds to verify memory systems using the language containment paradigm.

9.4.1 Hardware Verification

In formal verification, one models the system as well as its environment. The environment in our case is the set of all possible programs. To model the instructions, for each processor, we add a process which non-deterministically chooses any instruction to any location with any value. For each processor, there is a memory buffer which stores the store requests of the processor. If the memory buffer is full, the "noop" instruction is generated. The label of the generated instruction is any available label, where an available label is one which is not the label of any pending instruction.

9.4.2 Program Verification

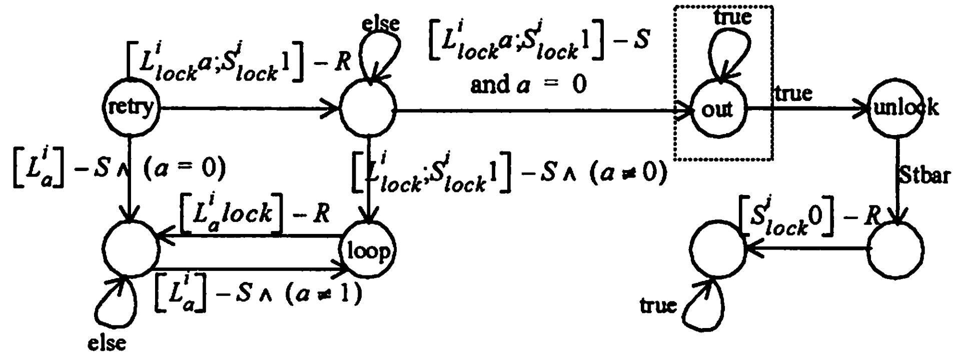
Assume we are given a program, a memory model, and a property on the program to check. As an example, assume we have a set of processes, running on a PSO memory model. To gain exclusive access to its critical section, each process attempts to set a lock using a spin lock procedure. If successful, it enters its critical section and after exiting it resets the lock. To implement lock and unlock procedures, one can use the following algorithm ([Sun90] with some modifications). Note that 0 means lock is free.

```
Lock(lock)
  retry: atomically load lock into a, and store 1 into lock.
         if $a = 0$ goto out
  loop: load a into lock
         if $a \neq 0$ goto loop else goto retry
  out:

Unlock(lock)
  Stbar
  store 0 in lock
```

Figure 9.8: Lock/Unlock algorithm

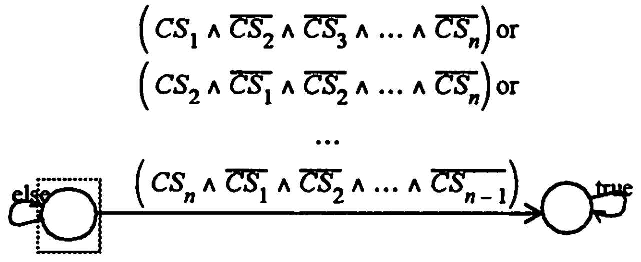
A safety property one might want to verify is that no two processes enter their critical sections at the same time. A liveness property of interest is that all processes eventually enter their critical sections. To deal with non-memory related operations such as test or jump, one can either model all operations ([DPN93]), or turn the programs into automata specifications using only loads, stores, and atomic load-stores. Note that by using an automaton to model the program, some details, such as the program counter, need not be modeled. Figure 9.9 presents an automaton for the above algorithms.



$[L^i_{lock}a;S^j_{lock}1]-R$
else
$[L^i_{lock}a;S^j_{lock}1]-S$ and $a=0$
true
retry
out
true
unlock
$[L^i_a]-S\wedge(a=0)$
$[L^i_{lock};S^j_{lock}1]-S\wedge(a\neq0)$
Stbar
$[L^i_a lock]-R$
$[S^j_{lock}0]-R$
loop
$[L^i_a]-S\wedge(a\neq1)$
true
else

Figure 9.9: Automaton corresponding to Lock/Unlock algorithm

The following automaton checks for the mutual exclusion of the processes.



$(CS_1\wedge\overline{CS_2}\wedge\overline{CS_3}\wedge\ldots\wedge\overline{CS_n})$ or
$(CS_2\wedge\overline{CS_1}\wedge\overline{CS_2}\wedge\ldots\wedge\overline{CS_n})$ or
...
else
$(CS_n\wedge\overline{CS_1}\wedge\overline{CS_2}\wedge\ldots\wedge\overline{CS_{n-1}})$
true

Figure 9.10: Automaton checking mutual exclusion

The following ω-automaton checks that processor i finally enters its critical section.

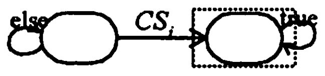


Figure 9.11: Automaton checking critical section is reached

We now show how one can verify these properties for an architecture using language containment. Let A be an architecture, P a program, and T a property which should hold of P running on A. $L(P)$ and $L(T)$ are well-defined. Let $L(A)$ be the intersection of the languages of all axioms which involve a memory request of P. For example, if $[L^i_{lock}a;S^j_a1]-R$ is a memory request generated by P, the atomicity axiom is intersected with $L(A)$. The number of such automata is finite and proportional to the number of memory requests of the program. We call the intersection of all such automata the ***most general implementation (MGI) of architecture A given program P***.

Lemma 9.4.1 Program P has the property T for any implementation I of the architecture A iff $L(A) \cap L(P) \subseteq L(T)$.

Proof Since $L(I) \subseteq L(A)$ for any I, if $L(A) \cap L(P) \subseteq L(T)$, we have that $L(I) \cap L(P) \subseteq L(T)$, i.e. P has the property T running on I. Now assume P has the property T for any implementation I. The set of all possible axioms constitutes an implementation. However, the axioms not involving memory requests of P do not constrain $L(A)$. It follows that $L(A) \cap L(P) \subseteq L(T)$ (QED).

Let O be an abstract operational model. It is easy to check that $L(O) \cap L(P) \subseteq L(T)$ and $L(O) \subseteq L(A)$. However, it may be the case that $L(A) \not\subseteq L(O)$, in which case for some implementation I, $L(I) \cap L(P) \not\subseteq L(T)$ may be true although $L(O) \cap L(P) \subseteq L(T)$. Hence, by verifying the program runs correctly on an operational model, one cannot guarantee that the program has the desired property for all implementations unless it is proved that the operational model and the architectural specification are equivalent The disadvantage of the MGI approach is that $L(A)$'s representation may be large (in terms of state space). A back of envelope computation is as follows. The current verification techniques are expected to handle about 20-100 automata, each having 4 states. If the architecture has 10 rules expressible by roughly 4 state automata, and for each instruction on average 4 apply, then the program is allowed 5-25 different memory request. This seems to cover a large number of small test programs.

9.5 Abstraction and Symmetry in Memory Models

A memory model has four parameters: processors, memory buffers (to queue up memory requests), data values, and memory addresses. One would like to verify the rules of the architecture for any system with any values assigned to these parameters. However, when using state space exploration tools, one needs to give finite instantiation to all parameters. In this section, by applying results of chapter 8, we show that once the number of processors and memory buffers is fixed, one can verify all of the properties with binary data values, and a small number of memory addresses without loss of accuracy in verification. Using the fact that our implementation of the memory system is symmetric, we then show that from each class of properties only one representative needs to be verified. For example, if the termination property is proved for a representative processor and label, then it holds for all processors and labels.

9.5.1 Reducing Memory Addresses and Data Values

Let p be the number of processors, m the number of variables holding memory addresses, and T any property except for the value axiom. The data values get produced, move through the memory buffers, but they are never looked at. Hence, the verification of T is independent of the data values, i.e. they can be deleted. Moreover, since the memory is never read, although it is written to, it can also be disregarded in proving T. The memory addresses get produced non-deterministically by the processes, and move through the buffers. The memory system may compare them against one another, but does not perform any other operation on them. Chapter 8 shows that for such systems, called data semi-sensitive controllers, T can be proved using only $m+1$ values for addresses.

Now let T be the value axiom for a load $L_a^i - R$. When memory services the load, i.e. $L_a^i - S$ is set, either the value is retrieved from the buffer, or from location a. Note that the values written and read from memory do not affect the model, since the model ignores the values read from the memory. But the property automaton is sensitive to the values written to location a. In this case, the memory cannot be disregarded. We show a simple generalization of results of chapter 8 tells us that only $m+1$ locations suffice. Assume there is an error trace e given unbounded memory. We will show there is an error trace $\tilde{e}$ where there are only $m+1$ memory addresses, the control and data values are the same as in e, and the values written to a in $\tilde{e}$ are the same as those written in e. Hence, $\tilde{e}$ is also an error trace. To obtain $\tilde{e}$ from e proceed as follows. Every address in the system is produced non-deterministically by the processes. If any such address is a, leave it unchanged. Otherwise, change it to one of the other m values so that two memory addresses are equal in $\tilde{e}$ iff they are equal in e. Note that since there are m buffers, there can be at most m outstanding requests, hence m distinct addresses. The control signals are left unchanged, and the values written to a are the same as e. We conclude $\tilde{e}$ is also an error trace. Since the number of memory locations is finite, the system is a data insensitive controller with respect to data values. By the 0-1 theorem of chapter 8, we have that two data values suffice to check the value axiom.

9.5.2 Reducing Number of Properties Using Symmetry

Consider a property such as termination. Given a processor, we have to check this property for every label. Assume we have checked the property for label 0. We will show the property holds for any other label i when the system is symmetric with respect to labels (the notion of symmetry can be made precise as is done for example in [IP93]). Assume not, and consider an error trace e. Replace every occurrence of label i with 0, and every occurrence of label 0 with label i. By symmetry with respect to the labels, the error trace still remains an error trace. This is in contradiction to the property passing for label 0. Hence, we conclude that the property holds for all labels. By using similar arguments for symmetric systems with respect to memory locations and processors , we conclude for every property, it suffices to check only one representative. This is a substantial decrease in the number of properties which have to be checked.

9.6 Experimental Results

In this section, we describe our experimental results for hardware and software verifications.

9.6.1 Hardware Verification Results

In our hardware model, there are two types of processes: one modeling the processors, and the other modeling the memory. The system consists of one memory process and two or more processors. Memory and processors communicate through store buffers: there is one store buffer per processor. A processor enqueues its store instructions (both store and load-stores). We make the assumption that only store instructions enter the store buffer. Hence, loads are modelled as direct reads from memory or the store buffer. When the memory chooses to service a processor, it dequeues the store instruction from the respective queue. The memory non-deterministically chooses which processor to serve next. For liveness properties (such as the termination axiom) this means that we need to introduce fairness constraints excluding the possibility of starvation for a process.

For verifying the memory axioms, we let each processor execute a non-deterministic stream of load and store instructions. All processes interact according to an interleaving model of computation: at each "clock tick" at most one process is active and updates its state variables while all state variables of the inactive processes hold their previous value. Again, to ensure progress of the whole system, we need to introduce the fairness constraint that each process is active infinitely often. Since the store buffer has limited depth, processors can only handle as many un-serviced store instructions as there are slots in the store buffer. This reflects the limitation that we can only verify infinite programs with finitely many open requests. In order to verify properties involving more than one un-serviced store in a row we need to attach labels to store instructions. The number of possible labels is the total number of un-serviced store requests allowed in the system. Each processor has the same number of possible labels; this number also determines the depth of the store buffer.

Some bookkeeping is required to "recycle" the labels as they become available. Every label has a one bit flag that keeps track of whether it is used or free. When a store request with a particular label is entered into the buffer, the label becomes used. When the memory services a request, the corresponding label becomes free. In order to use the symmetry argument introduced above, a processor needs to choose a free label arbitrarily instead of choosing it in a particular order. Note that in some cases, it may actually be advantageous to use an asymmetric model initially in order to reduce the state space and possibly obtain answers faster.

Table 9.1 summarizes our experiments for hardware verification. For each experiment, we combine the same model of a processor with either a TSO or PSO model of the memory to build the complete system. We verified the following axioms: termination, order of stores is preserved on the TSO model (*StoreStore-TSO*), order of stores to the same address is preserved on the PSO model (*StoreStoreEq*) and the value axiom (*value*). *term* was verified on the TSO model for simplicity. The number behind a property's name stands for the two different system configurations we examined to see how the verification scales with larger models: configuration 1 has two processors and store buffers of depth two and configuration 2 has three processors and store buffers of depth two. For our experiments, the number of addresses are the same as the depth of the store buffer. All experiments were run on a Sun Sparc-10 machine with 128 MB of memory. No optimizing options were used for SMV. The table lists the number of state variables in the system, the number of reachable states, the time it took to compute the set of reachable states, the time spent in property verification, the total time including compilation, and the amount of memory used. Compilation includes parsing, semantic analysis, and building the transition relation. Note that the verification time is only the time spent for verifying one representative set of parameters, since we are using the symmetry argument. Configuration 2 of the *value* property took much more space and time than all other configurations. The table shows that we can verify axioms for systems of interesting sizes in reasonable time. It also demonstrates the state explosion for larger systems.

Table 9.1: Hardware verification results

Prop.	# of state vars	# of reach states	reach state (sec.)	verif. (sec.)	total time (sec.)	mem (MB)
termination1	40	1.59×10^8	23.4	10.2	44.4	3.9
termination2	57	1.29×10^{12}	410	282.3	722.9	25.8
StoreStore-TSO1	42	1.67×10^8	40.5	0.13	51.6	3.7
StoreStore-TSO2	59	1.35×10^{12}	671.8	1.0	704.4	34
StoreStoreEq1	43	6.04×10^{10}	31.2	0.03	44.9	3.6
StoreStoreEq2	60	6.04×10^{15}	557.1	0.03	588.9	34
value1	46	3.36×10^9	229.4	0.28	255.9	14.76
value2	65	1.08×10^{14}	19421.6	4.9	19451.8	201.8

9.6.2 Program verification results

In this experiment, we modeled the synchronization code from section 9.4.2, and verified the mutual exclusion property: no two processors can be in their critical sections at the same time. The model consists of one automaton per processor that models the synchronization code as well as a number of automata representing the axioms needed to make the property pass (the most general implementation of section 9.4.2). The two configurations we verified (crit1, crit2) have two and three processors respectively. The verification time is the time needed to verify that no two processors can be in their critical sections at the same time. Compared to hardware verification the size of the state space here is much smaller, leading to shorter run times and smaller memory requirements.

Table 9.2: Software property verification results

Prop	# of state vars	# of reach states	reach. state (sec.)	verif. (sec.)	total time (sec.)	mem (MB)
crit1	22	3376	5.1	2.5	18.6	4.67
crit2	32	1.16×10^5	79.5	87.7	177.8	7.32

9.7 A Verification Strategy for Memory Models

We propose the following strategy to design and simultaneously verify memory systems with out-of-order servicing of requests.

1. *Design a formal specification for the system.* First develop a set of small programs, and

specify their correctness criteria. Come up with a set of properties of the system which is believed to characterize the system, and write them as automata specifications. Verify, using the most general implementation (MGI) concept, that these properties are sufficient to imply the correctness of the sample programs. Given a large and general enough set of programs, this procedure may be used to derive the axiomatic specification of the system incrementally.

2. *If needed, design an operational model.* For example, for applications programmers, it may be easier to work with an operational model. If so, design such a model and verify that it satisfies all of the properties. If the operational model is simple enough, it may be possible to verify that it has exactly the same set of behaviors as the MGI using a theorem-prover.

3. *Verify the hardware system satisfies its formal specification.* Perform the verification hierarchically, i.e. verify the hardware system before irrelevant details are added. For example, if it can be proved that the system with and without local processor caches has exactly the same behavior, prove the correctness of the system with respect to out-of-order execution with no caches.

Bibliography

[AH90] S. V. Adve, M. D. Hill, "*Weak Ordering -- A New Definition*", Proceedings of the 17th International Symposium on Computer Architecture, 1990.

[Agg83] S. Aggarwal, R. P. Kurshan, K. Sabnani, "*A Calculus for Protocol Specification and Validation*", in Protocol Specification, Testing and Verification III (193) North-Holland, pp. 19-34.

[Alu92] R. Alur, A. Itai, R. P. Kurshan, "*Timing Verification by Successive Approximation*", Fourth Workshop On Computer-Aided Verification, 1992.

[Alu91] R. Alur, "*Techniques for Automatic Timing Verification of Real Time Systems*", Ph. D. Thesis, Department of Computer Science, Stanford University, Report No. STAN-CS-91-1378, August 1991.

[ACP87] S. Arnborg, D. G. Corneil, and A. Proskurowski, "*Complexity of Finding Embeddings in a k-tree*", SIAM Journal of Algebraic and Discrete Methods, 8:277-284, 1987.

[AC94] P. Ashar, M. Cheong, "*Efficient Breadth-First Manipulation of Binary Decision Diagrams*", In proceedings of International Conference on Computer-Aided Design, pages 622-627, 1994.

[Bal92] Felice Balarin, A. Sangiovanni-Vincentelli, "*A Verification Strategy for Timing-Constrained Systems*", Fourth Workshop On Computer-Aided Verification, 1992.

[BRB90] K. S. Brace, R. L. Ruddell, and R. E. Bryant, "*Efficient Implementation of a BDD Package*", in Proc. of the 27th ACM/IEEE Design Automation Conference, pp. 40-45, 1990.

[Bry86] R. E. Bryant, "*Graph Based Algorithms for Boolean Function Manipulation*", IEEE Trans. on Computers, C-35(8):677-691, August 1986.

[BD94] J. Burch, D. Dill, "*Automated Verification of Pipelined Microprocessors*", Computer-Aided Verification, 1994.

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long, "*Symbolic Model Checking with Partitioned Transition Relations*", in Proc. of the 28th ACM/IEEE Design Automation Conference, 1991.

[BCMDH92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. Dill, L. J. Hwang "*Symbolic Model Checking:* 10^{20} *states and beyond*", Information and Computation, June 1992.

[BLIF87] Intermediate format used in MIS-II logic synthesis system of UC Berkeley.

[BMV93] Unpublished manuscript prepared by R. Hojati, T. Shiple, and R. K. Brayton, "*Revisiting BLIF-MV, An Intermediate Format for Verification and Synthesis of Hierarchical Networks of FSMs*", 1993.

[BMV91] Robert K. Brayton, Massimiliano Chiodo, Ramin Hojati, Timothy Kam, Kolar Kodandapani, Robert P. Kurshan, Sharad Malik, Alberto L. Sangiovanni-Vincentelli, Ellen M. Sentovich, Thomas R. Shiple, Kanwar J. Singh, Huey-Yih Wang, "BLIF-MV: *An Interchange Format for Design Verification and Synthesis*", Electronics Research Laboratory, College of Engineering, University of California, Berkeley, UCB/ERL M91/97, Nov 1991.

[CC93] G. Cabodi, P. Camurati, "*Exploiting cofactoring for efficient FSM symbolic traversal based on the transition relation*", in proc. IEEE ICCD'93, October 1993, pp. 299-303.

[Che94] S. T. Cheng, "*Compiling Verilog into Automata*", M.S. Thesis, University of California, Berkeley, 1994.

[Cho74] Y. Choueka, "*Theories of Automata on w-Tapes: A Simplified Approach*", Journal of Computer and System Sciences 8, 117-141, 1974.

[CGMZ] E. M. Clarke, O. Grumberg, K. McMillan, X. Zhao, "*Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking*", in Proc. of the 32nd ACM/IEEE Design Automation Conference, 1995.

[CDK93] E. M. Clarke, I. A. Draghicescu, R. P. Kurshan, "*A unified approach for showing language inclusion and equivalence between various types of automata*", Information Processing Letters 46 (1993) 301-308, Elsevier.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "*Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*", ACM Transactions on Programming Languages and Systems. 8(2), pp.244-263, 1986.

[Col92] William Collier, "*Reasoning About Parallel Architectures*", Prentice-Hall, 1992.

[CN94] D. Cyrluk, P. Narendran, "*Ground Temporal Logic: A Logic for Hardware Verification*", Computer-Aided Verification, 1994.

[Cyr93] D. Cyrluk, "*Microprocessor Verification in PVS: A Methodology and Simple Example*", Technical report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.

[PD95] S. Park, D. Dill, private communication, 1995.

[DPN93] D. L. Dill, S. Park, A. G. Nowatzyk, "*Formal Specification of Abstract Memory Models*", Research on Integrated Systems: Proceedings of the 1993 Symposium, G. Borriello and C. Ebeling, Eds., MIT Press, 1993.

[DDHY92] D. Dill, A. J. Drexler, A. J. Hu, C. H. Yang, "*Protocol Verification as a Hardware Design Aid*", ICCD 92.

[DNP93] D. L. Dill, A. G. Nowatzyk, S. Park, "*Formal Specification and Verification of Abstract Memory Models*", Conference on Hardware Description Languages and Their Applications, 1993.

[DHW91] D. L. Dill, A. J. Hu, H. Wong-Toi, "*Checking for Language Inclusion Using Simulation Relations*", Third Workshop On Computer-Aided Verification, 1991.

[DSB88] M. Dubois, C. Scheurich, and F. A. Briggs, "*Synchronization, Coherence, and Event Ordering in Multiprocessors*", IEEE Computer 21, 2 (February 1988), 9-21.

[ES95] E. A. Emerson, A. P. Sistla, "*Utilizing Symmetry When Model Checking Under Fairness Assumptions: an Automata-Theoretic Approach*", Computer-Aided Verification, pp. 309-325, 1995.

[Eme90] E. A. Emerson, "*Temporal and Modal Logic*", Handbook of Theoretical Computer Science, editor J. van Leeuven, Elsevier Science Publishers B. V., pp. 995-1072, 1990.

[EL87] E. A. Emerson, C. L. Lei, "*Efficient Model Checking in Fragments of the Propositional Mu-Calculus*", Logic in Computer Science, 1987.

[EL85] E. A. Emerson, C. L. Lei, "*Modalities for Model Checking: Branching Time Logic Strikes Back*", 12th Annual Symp. on Principles of Programming Languages, 1985.

[EC81] E.A. Emerson and E. C. Clarke, "*Characterizing Properties of Parallel Programs as Fixpoints*", In Proceedings of the 7th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, 85, Springer Verlag, New York, 1981.

[FOH93] H. Fujii, G. Ootomo, C. Hori, "*Interleaving Based Variable Ordering Methods for*

Ordered Binary Decision Diagrams", pp. 38-41, Proc. of the IEEE International Conference on Computer-Aided Design, 1993.

[GJ79] R. Garey, D. S. Johnson, "*Computers and Intractability, A Guide to the Theory of NP-Completeness*", W. H. Freeman and Company, 1979.

[GB94] D. Geist and I. Beer, "*Efficient Model Checking by Automated Ordering of Transition Relation Partitions*", Computer-Aided Verification, 1994.

[GLL90] K. Gharacharloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "*Memory Consistency and Event Ordering in Scalable Shared-Memory Multi-Processors*", Proceedings of the 17th International Symposium on Computer Architecture, 1990.

[GT93] D. Gifford, F. Turbak, "*6.821 Course Notes*", MIT, 1993.

[God94] P. Godefroid, "*Partial Order Method for the Verification of Concurrent Systems: An Approach to the State Explosion Problem*", Ph. D. thesis, Universite de Liege, 1995.

[GMG91] P. B. Gibbons, M. Merritt, K. Gharacharloo, "*Proving Sequential Consistency of High-Performance Shared Memories*", AT&T Technical Memorandum, May 1991.

[GL91] O. Grumberg, D. E. Long, "*Model Checking and Modular Verification*", In J. C. M. Baeten and J. F. Groote, editors, Proceedings of CONCUR 91: 2nd International Conference on Concurrency, volume 527 of Lecture Notes in Computer Science, August 1991.

[Har90] Z. Har'El and R. P. Kurshan, "Software *for Analytical Development of Communication Protocols*", AT&T Journal, January 1990, 45-59.

[HIKB96] R. Hojati, A. Isles, D. Kirkpatrick, R. K. Brayton, "*Verification Using Uninterpreted Functions and Finite Instantiations*", Submitted to Formal Methods in Computer-Aided Design, 1996.

[HB95a] R. Hojati and R. K. Brayton, "*An Environment for Formal Verification Based on Symbolic Computations*", Journal of Formal Methods in System Design, 6, 191-216, 1995.

[HB95b] R. Hojati, R. K. Brayton, "*Automatic Datapath Abstraction In Hardware Systems*", Computer-Aided Verification, 1995.

[HMLB95] R. Hojati, R. Mueller-Thuns, P. Loewenstein, R. K. Brayton, "*Automatic Verification of Memory System Using Language Containment and Abstraction*", Conference on Hardware Description Languages and Their Applications, 1995.

[HMB94] R. Hojati, R. Mueller-Thuns, R. K. Brayton, "*Improving Language Containment Using Fairness Graphs*", 6th International Conference on Computer-Aided Verification, Stanford, June 1994.

[HSB94] R. Hojati, V. Singhal, R. K. Brayton, "*Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment*", Memorandum No. UCB/ERL M94/12, UC Berkeley, 1994.

[HKB94] R. Hojati, S. Krishnan, R. K. Brayton, "*Heuristic Algorithms for Early Quantification and Partial Product Minimization*", ERL Memorandum M94/11, March 1994, UC Berkeley.

[HSIS94] A. Aziz, F. Balarin, S. T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, T. R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "*HSIS: A BDD-Based Environment for Formal Verification*", in Proceedings of the 31st ACM/IEEE Design Automation Conference, pp. 299-310, 1994.

[HSBK93] R. Hojati, T. P. Shiple, R. K. Brayton, R. P. Kurshan, "*A Unified Approach to Language Containment and Fair CTL Model Checking*", in Proceedings of the 30th ACM/IEEE Design Automation Conference, June 1993.

[HBK93] R. Hojati, R. K. Brayton, R. P. Kurshan, *"BDD-Based Debugging of Design Using Language Containment and Fair CTL"*, Proceedings of the Conference on Computer-Aided Verification, Elounda, Crete, Greece, June 1993.

[HTKB92] Ramin Hojati, Herve Touati, Robert P. Kurshan, Robert K. Brayton, *"Efficient ω-Regular Language Containment"*, Fourth Workshop On Computer-Aided Verification, 1992.

[Hol91] G. J. Holzmann, *"Design and Validation of Computer Protocols"*, Prentice Hall Software Series, 1991.

[ID93] C. N. Ip, D. Dill, *"Better Verification through Symmetry"*, Symp. on Computer Hardware Description Languages and Their Application, 1993.

[IHB96] A. Isles, R. Hojati, R. K. Brayton, *"Reachability Analysis of ICS Models"*, Submitted to SRC Techcon, 1996.

[Isl95] A. Isles, personal communication, summer 1995.

[KB91] T. Kam, R. K. Brayton. *"Multi-valued Decision Diagrams"*, Electronics Research Laboratory, University of California, Berkeley, Memorandum No. UCB/ERL, M90/125, 1990.

[KVBS94] T. Kam, T. Villa, R. K. Brayton and A. L. Sangiovanni-Vincentelli, *"A Fully Implicit Algorithm for Exact State Minimization"*, in Proceedings of the 31st ACM/IEEE Design Automation Conference, 1994.

[KR96] N. Klarlund, T. Rauhe, *"BDD Algorithms and Cache Misses"*, BRICS Report Series RS-96-5, Department of Computer Science, University of Aarhus, 1996.

[Kur94] R. P. Kurshan, *"Models Whose Checks Don't Explode"*, 6th International Conference on Computer-Aided Verification, Stanford, June 1994.

[Kur87a] R. P. Kurshan, *"Complementing Deterministic Buchi Automata in Polynomial Time"*, Journal of Computer and System Sciences, volume 35, 1987, 59-71.

[Kur87b] R. P. Kurshan, *"Reducibility in Analysis of Coordination"*, In LNCIS, volume 103, pages 19-39, Springer-Verlag, 1987.

[Kur92] R. P. Kurshan, *"Automata-Theoretic Verification of Coordinating Processes"*, UC Berkeley notes, 1992.

[MPS92] E. Macii, B. Plessier, F. Somenzi, *"Verification of Systems Containing Counters"*, IEEE/ACM International Conference on Computer-Aided Design, 1992.

[McM94] K. McMillan, *"290H Course Notes, Week 6,"*, University of California, Berkeley, fall 1994.

[McM93] K. McMillan, *"Symbolic Model Checking"*, Prentice Hall, 1993.

[Mil89] R. Milner, *"Communication and Concurrency"*, Prentice Hall, New York, 1989.

[Mil71] Robin Milner, *"An Algebraic Definition of Simulations Between Programs"*, Proceedings of the 2nd International Joint Conference on Artificial Intelligence, British Computer Society, 1971, pp. 481-489.

[OYY93] H. Ochi, K. Yasouka, and S. Yajima, *"Breadth-First Manipulation of Very Large Binary-Decision Diagrams"*, IEEE/ACM International Conference on Computer-Aided Design, pages 48-55, November 1993.

[PSP94] S. Panda, F. Somenzi, B. F. Plessier, *"Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams"*, IEEE/ACM International Conference on Computer-Aided Design, pages 628-631, November 1994.

[Pnu77] Amir Pnueli, *"The Temporal Logic of Programs"*, In Proceedings of the 18th IEEE

Symposium on Foundations of Computer Science". pages 46-77, 1977.

[QS81] J. P. Quielle, J. Sifakis, "*Specification and Verification of Concurrent Systems In CESAR*", In proceedings of the 5th International Symposium on Programming. Lecture Notes in Computer Science 137, Springer-Verlag, New York, 1981, 337-350.

[Rab72] M. O. Rabin, "*Automata on Infinite Objects and Church's Problem*", Regional Conference Series in Mathematics, volume 13, 1972, American Mathematical Society, Providence, Rhode Island, 19-39.

[RS86] N. Robertson and P. D. Seymour, "*Graph Minors. II. Algorithmic Aspects of Tree-Width*", Journal of Algorithms, 7:309-322, 1986.

[Rud93] R. L. Ruddell, "*Dynamic Variable Ordering for Ordered Binary Decision Diagrams*", Proc. of the IEEE International Conference on Computer-Aided Design, pages 42-47, 1993.

[Saf89] Shmuel Safra, "*Complexity of Automata on Infinite Objects*", The Weizmann Institute of Science, March 1989.

[Saf92] S. Safra, "*Exponential Determinization for w-Automata with Strong-Fairness Acceptance Condition*", Symposium on Theory of Computation, 1992.

[Sav94] H. Savoj, Personal Communication, 1994.

[PVS93] N. Shankar, S. Owre, J. M. Rushby, "*The PVS Specification and Verification System*", SRI International, 1993.

[SSBS96] T. R. Shiple, V. Singhal, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "*Analysis of Combinational Cycles in Sequential Circuits*", in Proceedings of International Symposium in Systems and Circuits, 1996.

[SHBS94] T. R. Shiple, R. Hojati, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "*Heuristic Minimization of BDDs Using Don't Cares*", in Proceedings of the 31st IEEE/ACM Design Automation Conference, pp. 225-331, San Diego, June 1994.

[Sho79] R. E. Shostak, "*A Practical Decision Procedure for Arithmetic With Function Symbols*", JACM Volume 26, No. 2, April 1979, pp. 351-360.

[SFC91] P. S. Sindhu, J. M. Frailong, M. Cekelov, "*Formal Specification of Memory Models*", Xerox Parc, Dec 1991.

[SVW87] A. P. Sistla, M. Y. Vardi, P. Wolper, "*The Complementation Problem for Buchi Automata with Applications to Temporal Logic*", Theoretical Computer Science 49, 1987, pp. 217-237.

[Str82] R. S. Streett, "*Propositional Dynamic Logic of Looping and Converse is Elementary Decidable*", Information and Control, volume 54, 1982, 121-141.

[Sun90] "*SPARC Architecture Reference Manual V8*", Sun Microsystems, Dec 1990.

[THB95] S. Tasiran, R. Hojati, R. K. Brayton, "*Language Containment of Non-Deterministic ω-Automata*", IFIP Conference on Correct Hardware Design and Verification Methods (CHARME), Frankfurt, October 1995.

[Tho90] W. Thomas, "*Automata on Infinite Objects*", in Formal Models and Semantics, Handbook of Theoretical Computer Science, volume B, 1990, 133-191, editor J. van Leeuwen, Elsevier Science.

[TSLBV90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. S. Vincentelli, "*Implicit State Enumeration of Finite State Machines Using BDDs*", Proc. of the IEEE International Conference on Computer-Aided Design, pp. 130-133, Nov. 1990.

[TKB91] H. Touati, R. Kurshan, R. Brayton. "*Testing Language Containment of ω-Automata Using BDDs*", International Workshop on Formal Methods in VLSI Design, 1991.

[VW86] M. Y. Vardi and P. L. Wolper, "*An Automata-Theoretic Approach to Program Verification*", Logic in Computer Science, 332-334, 1986.

[Wol86] P. Wolper, "*Expressing Interesting Properties of Programs*", 13th Annual ACM Symp. on Principles of Prog. Languages, 1986.

[Wol83] P. Wolper, "*Temporal Logic Can be More Expressive*", Information and Control, 56:72-99, 1983.