M. Tech. Dissertation

Techniques In Symbolic Model Checking

Submitted By

Ashutosh Suresh Trivedi
Roll No: 01322006

Under the guidance of

Dr. Supratik Chakraborty

Interdisciplinary Program in Reliability Engineering
Affiliated to Department of Electrical Engineering
Indian Institute of Technology
Mumbai

January 2003

Dissertation Approval Sheet

This is to certify that the dissertation entitled **Techniques in Symboic Model Checking** by **Ashutosh Trivedi** is approved for the award of the degree of **Master of Technology**.

Dr. Supratik Chakraborty
(Guide)

Dr. S. Ramesh
(Internal Examiner)

Dr. Paritosh K. Pandya
(External Examiner)

Dr. G. Sivakumar
(Chairman)

Date : _____

CERTIFICATE

This is to certify that the M. Tech. dissertation entitled *Techniques in Symboic Model Checking* by *Ashutosh Trivedi* (01322006) is in the required format and the comments suggested during the presentation are incorporated in the report.

Dr. Supratik Chakraborty
(Guide)

Date : _____

Acknowledgment

I take this opportunity to express my profound gratitude and deep regards to my guide *Dr. Supratik Chakraborty* for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis work. Everything that I learnt from him shall carry me a long way in journey of life which I am about to embark.

Sincere appreciation is extended to *Dr. Alessandro Cimatti* for his immense help and critical remarks during the second stage and the implementation phase of this work.

I am grateful to *Dr. Paritosh Pandya* for providing me the timely help and encouragement during the second stage of the work.

I am the most fortunate on the earth to have these people as my family member. I dedicate this thesis for their enthusiasm and expectations from me. Thank you *papa, mummy, didi, dada* and *Richa.*

Special thanks to *Shweta Gehlot* for helping me selflessly whenever I asked for. Her creative ideas and technical skills helped me immensely through every phase of the work.

My several well-wishers helped me directly or indirectly. I am thankful to all of them and I am leaving this acknowledgement incomplete in their reminiscene...

Ashutosh Trivedi
IIT Bombay
April 23, 2003

To Papa, Mummy, Didi, Dada and Richa.

Efficiency is intelligent laziness.

-David Dunham

Abstract

Model Checking is a highly automatic verification technique for finite state concurrent systems. In this approach for verification, temporal specifications are exhaustively verified over the state-space of the concurrent system. The number of states grows exponentially with the concurrency of the system and that makes explicit state-space enumeration based techniques inefficient. This phenomenon is called *state space explosion*. One of the possible way to overcome this limitation is to avoid explicit enumeration of state space. These approaches, commonly known as *Symbolic Model Checking*[4], uses *Boolean formulas* to represent sets of states and transition relations. Traditionally, symbolic model checking has become identified with *Binary Decision Diagrams* (BDD)[3], a canonical form of representing Boolean formulas. But recently, some other representations like *Conjunctive Normal Form* (CNF) using *satisfiability solving* (SAT) and polynomial algebra have been demonstrated to be quite powerful in practice.

In this thesis, we propose an approach to symbolic model checking where model checking is performed by decomposing a finite state system into *components*. We first review the decomposition process proposed by Chakraborty & Soni [21] and enhance it by guiding the process of decomposition. We observed that computing the components of the system is an expensive operation. Therefore our approach computes a new component only if the information in existing components is not sufficient to prove (falsify) the safety property. We call this approach *Lazy decomposition*.

The ideas are evaluated on publicly available benchmarks from ISCAS-89 benchmark suite using BDD and SAT based implementations. We report the experimental results and compare it with earlier schemes.

Contents

List of Figures

Chapter 1

Introduction

> *The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe that this view is also accepted in other areas of Ariane 5 software design. The Board is in favor of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.*
>
> -Prof. J. L. LIONS, Chairman, Inquiry Board, ARIANE-5.

The above excerpt is directly taken from the *ARIANE-5, flight 501 crash report*[11], which exploded on June 4, 1996, less than 40 seconds after it was launched. The committee investigated the accident found that it was caused by a software error in the computer that was responsible for calculating the rocket's movement.

It is clear that the need for reliable hardware and software systems is critical. As the involvement of such systems in our life is increasing, criticality of ensuring their correct operation is also increasing. Traditionally, *testing* and *simulation* are used for establishing confidence in the design of software and hardware systems. These methods usually involve providing certain inputs and observing the corresponding outputs. Testing and simulation can be a cost efficient way of minimizing errors. However, covering all possible interactions and potential errors is rarely feasible. Hence there is an increasing interest in more *formal* approaches.

This thesis presents a technique for verification using formal methods. In this chapter we introduce formal verification methods and provide basic background of the problem addressed. Finally, there is a note on the main contribution of the thesis and the organization of subsequent chapters.

1.1 Formal Methods

Formal Verification methods aim at establishing that an *implementation* satisfies a *specification*. Here the term *implementation* refers to an abstracted model of the system to be verified and *specification* refers to some property of the system expressed in suitable form. Formal verification methods can be divided into two basic categories of *model-checking* and *theorem proving*.

1.1.1 Theorem Proving

Theorem Proving, also known as *deductive reasoning* is an approach of formal verification where the verification problem is represented as a theorem in a *formal theory*. A *formal theory* consists of *language* in which formulas are written, a set of *axioms* and a set of *inference rule*. These inference rules are for the syntactic transformation of the formulas. With these rules and axioms, a theorem can be proved.

An advantage of *Theorem Proving* methods is that it can be applied to infinite state systems. However, these approaches are often time-consuming and requires mathematician's intervention (highly skilled human intervention). This lack of automation makes its applicability more difficult and limited to safety critical systems.

1.1.2 Model Checking

Model Checking is a technique for verifying finite state concurrent systems. In this technique, *properties* are specified as *temporal logic formula* and implementation is represented using finite state systems. A major benefit of restricting the model to finite state systems is that verification can be performed automatically. Given sufficient resources, it always terminates with a yes/no answer.

The restriction to finite state systems may seems to be too restrictive, but in practice an important class of systems falls in this category. Hardware controllers are fortunately one of them and so is communication protocols. We can also verify the properties of other infinite state systems using their abstract finite model.

Since *model checking* is highly automatic, it is preferable to deductive verification, whenever it can be applied. However, a major limitation of model checking is the *state explosion* which can happen if the system to be verified has a large number of concurrent components.

This thesis will concentrate on model checking and present a different approach to it. In next section, we briefly discuss the history of model checking and the basic problem addressed in this thesis.

1.2 Motivation for the thesis

1.2.1 History

The basic principles of model checking were developed in the early 1980's independently by Quielle & Sifakis [20] and Clarke & Emerson [7, 6]. The basic idea is to model the system of interest so as to allow the generation of a graph that contains the reachable states of the system as nodes and the state transitions between them as edges. When a labeling of the nodes with atomic propositions which hold at each state is added, this graph is known as a *Kripke structure* of the system. The specification of the property we are interested in is given by a temporal logic formula. One can check with a model checking algorithm whether the system meets its specification, i.e., by checking if the Kripke structure of the system is a model of the specification. For small systems the approach is quite practical, but in systems with many concurrent parts, global state transition graph becomes too large to handle.

In 1987, McMillan proposed a new methodology for model checking called *Symbolic model checking* [4]. In symbolic model checking the main idea is to represent the behavior of the system in a symbolic form rather than explicitly constructing a Kripke structure as a graph. There are several variations to symbolic methods. Their common feature is the use of representations of sets of states of the system in implicit form rather than having each global state of a system explicitly represented, e.g., as a node of the Kripke structure. McMillan used *Ordered Binary Decision Diagrams* (henceforth BDD) [3], a canonical form for Boolean expression, to represent the characteristic functions. Model Checkers based on BDDs are usually able to handle billions of states. Although symbolic representation using BDDs has greatly increased the size of the systems that can be verified, many realistic systems are still too large to be handled. This is because the performance of BDD based approaches depends heavily on the *variable ordering* [see, Section 2.2]. Variable ordering is frequently hard to generate automatically and generally human intervention is needed.

Recently, several suggestions have been made to replace BDDs with methods based on propositional satisfiability (SAT) procedures [2, 17, 18] to further improve the scalability of symbolic model checking. SAT solver based algorithms usually require much less space and usually work well in default settings. In addition, a number of efficient implementations of SAT solvers are available, both proprietary (PROVER) and in public domain (GRASP, SATO, CHAFF etc), that can handle thousands of variables. These strengths of SAT solvers make these approaches a promising alternative to BDD for symbolic model checking.

In this thesis, we propose an algorithm for *symbolic model checking* as a possible solution to some of the problems with existing algorithms. In this thesis we will concentrate on SAT based implementation of the approach, but results shows that it can perform well irrespective of underlying decision procedure

(i.e., BDD or SAT). Before going into details, here we informally define the problem addressed by the thesis.

1.2.2 Outline of the problem

We are interested in checking *safety properties* of large finite state machine with a large number of inputs. *Safety properties* are the class of properties that state that "something bad does not happen". Given a finite state machine and a safety property, we wish to either validate that the machine respects the property, or find an execution path that shows how the model violates the property.

To perform safety checking of finite state machines, the issues we face include:

- **Satisfiability Checking:** *Checking whether a characteristic function*[1] *represents an empty set of states.*

  This step can be termed as "*satisfiability checking*". We need this operation at two places: (i) while deciding whether an initial set of states is a subset of another set of states; and (ii) while checking whether two sets of states are equivalent.
  Satisfiability checking is the area of expertise of SAT-based symbolic model checkers. A variety of efficient SAT-solvers are available which can find satisfying assignment faster because of the depth-first nature of SAT search procedures. In chapter 2, we discuss various SAT search procedures in detail.

- **Quantification Elimination:** *Given a set of states, computing the set of states that can be reached using one transition via the transition relation of the Kripke structure.*

  This step is termed as "*image computation*". In traditional symbolic model checking, *image computation* requires existential quantification of the characteristic function (Boolean expression) over some Boolean variables. *Quantification elimination* is an expensive operation in SAT based techniques. Existing SAT-based approaches eliminate the quantification over state variables using "*quantification-by-substitution*" (see, equation 3.3) rule and then naively resolve the quantification (using equation 3.4) over input variables by doubling the formula size. This makes these techniques inefficient when number of input variables are large.

This thesis presents a symbolic model checking technique which addresses the *quantification* problem by *decomposing* the finite state machine.

[1] We will use the term "characteristic function" to denote the function that represents a set of states symbolically.

1.3 Contributions

The main contributions of this thesis are summarized below:

1. A decomposition based *quantifier elimination* scheme which enhances the work given in [21] by guiding the decomposition process in the hope of searching counter-example faster.

2. A backward reachability based model checking algorithm called "*Lazy decomposition*", which computes the component sub-graph only if the validity of safety property is not answerable using available components.

1.4 Outline of the thesis

This thesis is organized as follows:

In the next chapter we discuss the basics of BDDs and SAT solvers. Chapter 3 briefly reviews the work done in the field of *symbolic model checking*. In particular, we discuss most popular algorithms based on BDDs and SAT, vis-a-vis our algorithm.

Chapter 4 describes our approach for quantifier removal. In that chapter we first discuss the basic framework set by Chakraborty & Soni [21] for Kripke structure decomposition and then explain our heuristic to guide the process to possibly converge towards counter-example faster.

In chapter 5, we propose an algorithm for safety property checking over the components of the Kripke structure. We then explain this technique, called as lazy decomposition, with the help of an example.

In chapter 6, we demonstrate the promise of our approach by providing the results of our experimental work. Experiments were carried out using BDD and SAT based implementations of the algorithm. We report the improvements by our technique over traditional symbolic model checking algorithms.

Finally In chapter 7, we draw some conclusions and discuss possible future work.

Chapter 2

BDDs and SAT

Boolean Algebra is the basic mathematical tool for *symbolic model checking*. This chapter introduces some basic definitions related to *Boolean functions*. *Satisfiability Checking* of Boolean functions is one of the most crucial operation of *symbolic model checking*. Most of the symbolic model checking algorithms either use Binary decision diagrams (BDDs) or satisfiability solvers (SAT) to determine the satisfiability of the Boolean functions. In this chapter, we define BDD and SAT formally and discuss underlying concepts in detail.

2.1 Preliminaries

Definition 2.1 (Boolean Expression): *The classical calculus for dealing with* truth values *consists of* Boolean variables x, y, ..., *the constants* true *1 and* false *0, the operators of conjunction* $\wedge$, *disjunction* $\vee$, *negation* $\neg$, *implication* $\Rightarrow$,*and bi-implication* $\Leftrightarrow$, *which together form the Boolean expressions. Sometimes the variables are called* propositional variables *or propositional letters and the Boolean expressions are then known as* Propositional Logic. *A* literal l *is either a proposition (say,* p*) or the complement of one (denoted by* $\bar{p}$*); in the first case, we say that* l *is a positive literal, and in the second, we say that* l *is a negative literal.*

Formally, Boolean expressions are generated from the following grammar:

$$t ::= x \mid 0 \mid 1 \mid \neg t \mid t \ \wedge \ t \mid t \ \vee \ t \mid t \ \Rightarrow \ t \mid t \ \Leftrightarrow \ t$$

where x *ranges over a set of Boolean variables. This is called the abstract syntax of Boolean expressions.*

Definition 2.2 (clause): *A clause is a finite disjunction of literals.*
A clause is called as "unit clause" if it contains only one literal.

Definition 2.3 (Conjunctive Normal Form): *A propositional formula is in Conjunctive Normal Form (CNF) if it is a finite conjunction of clauses.*

Example 2.1 *Formula* $(w \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y}) \wedge (x \vee \bar{y} \vee \bar{z})$ *is in CNF, where* w, x, y *and* z *are propositional variables and* $(w \vee y \vee \bar{z}), (\bar{x} \vee \bar{y})$ *and* $(x \vee \bar{y} \vee \bar{z})$ *are the clauses of the formula.*

Definition 2.4 (Truth Assignment): *A truth assignment* v *is a partial function from the set of propositions to* $\{\mathbf{T}, \mathbf{F}\}$*, where* **T** *and* **F** *denotes logical truth and falsehood, respectively. We can extend the definition of* v *in a natural way so that it assigns truth values to literals, clauses and formulas. For a literal* l*, if* l *is a positive literal of* p *then* $v(l) = v(p)$ *else* $v(l) = \overline{v(p)}$[1]*. For a clause* $C = \vee_{i=1}^{n} l_i$*;* $v(C) = \vee_{i=1}^{n} v(l_i)$*. For a CNF formula* $F = \wedge_{j=1}^{n} C_j$*;* $v(F) = \wedge_{j=1}^{n} v(C_j)$*. Similarly this definition can be extended to any arbitrary formula.*
We say that a Truth assignment v *satisfies a (formula/clause/literal)* ϕ*, if* v *(*ϕ*)=***T***; and* v *falsifies* ϕ*, if* v *(*ϕ*)=***F***.*

Definition 2.5 (Satisfiability): *A formula* F *is called as satisfiable if there exists a truth assignment to its literals that satisfies* F*.*
A formula F *is called as unsatisfiable if if there exists no truth assignment to its literals that satisfies* F*.*
A formula F *is called as valid (tautology) if all possible truth assignment to its variables satisfies* F*.*

Example 2.2 *The formula discussed in Example 2.1, is satisfiable because the truth assignment* $\{1 \leftarrow w, 0 \leftarrow x, 0 \leftarrow y, 1 \leftarrow z\}$ *satisfies it.*

Definition 2.6 CNF Satisfiability problem : *The CNF satisfiability problem is simply this [12]:*
Given a propositional formula F *in CNF, is there any assignment to its literals that satisfies* F*.*
CNF satisfiability problem, commonly known as SAT, is a famous NP-complete problem. It means that it is not proven to be intractable but polynomial time algorithm for solving it is also not known.

Cofactor and quantification of Boolean formulas are important operations for symbolic model checking. They can be defined as follows:

Definition 2.7 (Cofactor and Quantification): *The positive cofactor of a Boolean function* F *with respect to a variable* a *is the function that is obtained by replacing every occurrence of* a *in* F *by constant* 1*, and it is denoted by* F_a*. Similarly* negative cofactor *of* F *with respect to* a *is the function obtained by replacing every occurrence of* a *in* F *by constant* 0*, and it is denoted by* $F_{\bar{a}}$*.*

[1] We use overline ($\overline{F}$) representation to show the negation of the formula ($\neg F$).

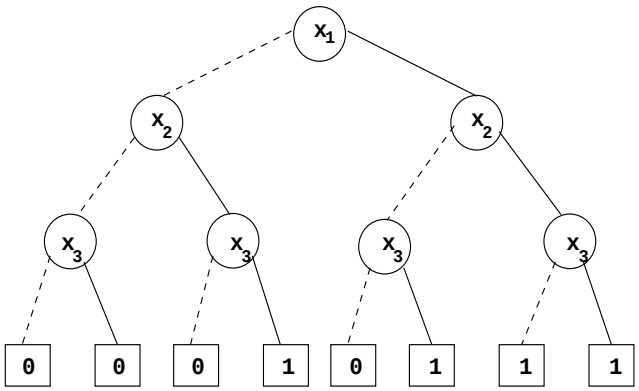

Figure 2.1: Representing Boolean function using BDDs

Existential quantification of a variable a from a function F is denoted by $\exists a \cdot F$ *and can be defined as :*

$$\exists a \cdot F = F_a \vee F_{\bar{a}} \tag{2.1}$$

Similarly universal quantification *can be defined as :*

$$\forall a \cdot F = F_a \wedge F_{\bar{a}} \tag{2.2}$$

2.2 Binary Decision Diagrams

Binary Decision Diagram (BDD) is a data structure suitable for representing binary function. Bryant [3] proposed this representation by imposing restriction on the representation first introduced by Lee [15] and Akers [1], such as the resulting form is canonical. BDDs[2] are substantially compact than other traditional representation like truth tables, conjunctive normal form and disjunctive normal form. In addition, they can be manipulated very efficiently. Hence they have been widely used for *symbolic model checking.*

In particular, BDDs represents a Boolean function as a rooted, directed acyclic graph. As an example, Figure 2.1 illustrates the representation of the Boolean function $f(x_1, x_2, x_3) = (x_1x_2 + x_2x_3 + x_3x_1)$, for the special case when the graph is a tree. Each *non-terminal* vertex v is labeled by a variable *var (v)* and has two children: *else (v)* (shown as a dashed line) corresponding to the case when variable is assigned to the value 0, and *then (v)* (shown as a solid line) corresponding to the case where the variable is assigned the value 1. Each terminal vertex is labeled 0 or 1. For a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the

[2]We use the term BDD to refer the restricted Binary decision diagram, introduced by Bryant et al.

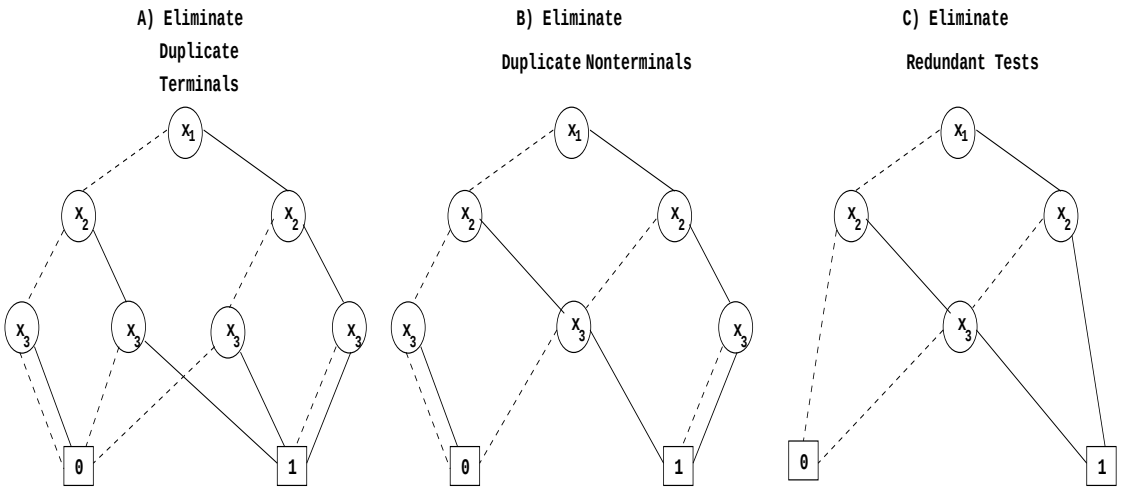

Figure 2.2: Transformation Rules for OBDDs

variables. Then the value of the function is then given by the terminal vertex.

2.2.1 Ordered Binary Decision Diagram

An Ordered BDD (OBDD), has a total ordering $<$ over the set of variables. For any vertex u, and either nonterminal child v of u, their respective variables must be ordered as *var (u)* $<$ *var (v)*. In the decision tree of Figure 2.1, for example, the variables are ordered x1 $<$ x2 $<$ x3. We further need three transformation rules over these graphs that do not alter the function represented, but result in more compact and canonical representations of the functions.

1. **Remove Duplicate Terminals:** Choose a representative terminal vertex for the constant 0 and one representative terminal vertex for the constant 1. All arcs going into a terminal 0 vertex are directed into the representative terminal 0 vertex, and similarly all arcs going into a terminal 1 vertex go to the representative terminal 1 vertex.

2. **Remove Duplicate Nonterminals:** If nonterminal vertices u and v have *var (u) = var (v), else (u) = else (v)*, and *then (u) = then (v)*, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex. This results in isomorphic subgraphs within the tree being shared. It is this sharing property that enables BDDs to be a compact representation for many Boolean functions.

3. **Remove redundant Tests:** If nonterminal vertex v has *else (v) = then (v)*, then eliminate v and direct all incoming arcs to *else (v)*.

Starting with any BDD satisfying the ordering property, we can reduce its size by repeatedly applying the transformation rules. We use the term "*Reduced Ordered Binary Decision Diagrams* (ROBDDs)" to refer to a maximal reduced

graph that obeys some ordering. Figure 2.2 illustrates the reduction of the decision diagram shown in Figure 2.1. Since we always use this data structure in its ordered and reduced form, we will use the term BDDs to mean ROBDDs.

2.2.2 Properties

Operations and Complexity

Bryant [3] gives algorithms for computing the BDD representation of $\neg f$ and $f \ \ op \ \ g$ (where *op* is a Boolean binary operator), given the BDDs for f and g. These functions have complexity linear in the size of the argument BDDs. Another useful operation over BDDs is quantification over Boolean variables. Bryant also gave an algorithm to compute the BDD for *Restrict* operator, using which existential and universal quantification can be computed by equation 2.1 and equation 2.2. *Satisfiability checking* is an important operation in *symbolic model checking*. Since BDDs are canonical, using BDDs satisfiability checking can be done in constant time.

Procedure	**Result**	**Time Complexity**
Reduce	G reduced to canonical form	$O(\|G\|.log\|G\|)$
Apply	$f_1 \ < op > \ f_2$	$O(\|G_1\|.\|G_2\|)$
Restrict	$f\|_{x_i=b}$	$O(\|G\| * log\|G\|)$
Compose	$f_1\|_{x_i=f_2}$	$O(\|G_1\|^2.\|G_2\|)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n.\|S_f\|)$
Satisfy-count	$\|S_f\|$	$O(\|G\|)$

Table 2.1: Summary of Basic Operations

The basic operation on Boolean functions represented as function graphs are summarized in Table 2.1. These few basic operations can be combined to perform a wide variety of operations on Boolean functions. As the table shows, most of the algorithms have time complexity proportional to the size of the graphs (represented by $|G|$) being manipulated. Hence, as long as the functions of interest can be represented by reasonably small graphs, BDD manipulation algorithms are quite efficient.

Ordering Dependency

The form and size of the BDD representing a function depends heavily on the ordering of the variable. In general, choice of variable can make a difference between linear and exponential (in terms of number of variables). For example, Figure 2.3 shows two BDD representation of the same formula $a_1b_1 + a_2b_2 + a_3b_3$ but with different variable ordering. The choice of variable ordering $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$ yield a BDD of 8 vertices, while the choice of the variable

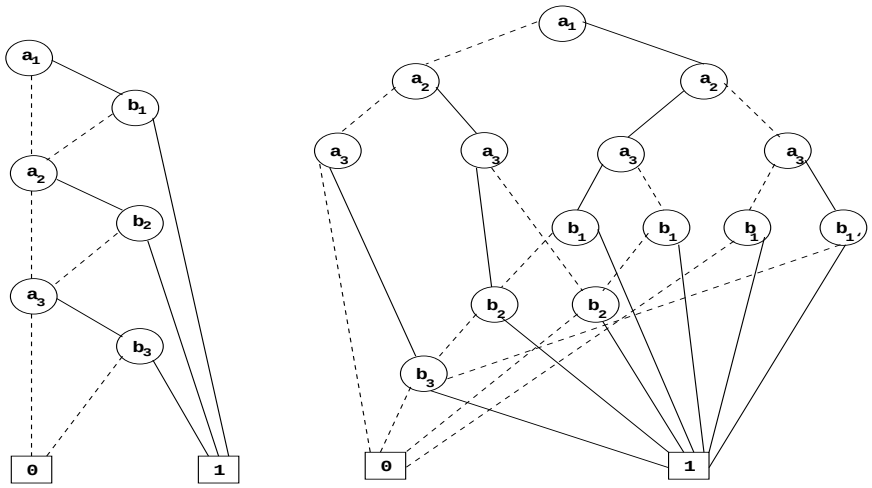

Figure 2.3: Effects of variable ordering

order $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$ yields a BDD with 16 nodes.

Most applications using BDDs choose some ordering at the beginning and construct graphs for all relevant functions according to this ordering. The generation of variable ordering is often time consuming or requires inputs from the designer. This limitation reduces the degree of automation of BDD based approaches. Furthermore, there exists some functions for which every variable ordering results in exponential number of nodes. Unfortunately, some functions of practical interest like *integer multiplication* [3] falls into this category.

2.3 Satisfiability Solving

Satisfiability checking is one of the basic operations in *symbolic model checking*. In SAT-based model checking, this step is performed using specialized tools for checking satisfiability, called as SAT-solvers. In this section we will discuss basics of SAT-solvers in detail.

As we know that checking the satisfiability of a Boolean formula (SAT) is NP-complete. Boolean satisfiability is probably the most researched combinatorial optimization/search problem. This research has culminated in the development of several SAT solving packages that can rapidly solve many SAT formulas of practical interest.

In this section, we first review two of the most popular algorithms for SAT solving, called as Davis and Putnam algorithm and Stalmarck's algorithm. Since these algorithms, and many other SAT-search algorithms works with Boolean formulas in CNF (clausal form), we need to convert a general formula to CNF before giving it to a SAT solvers. We discuss two common approaches to convert a general formula (non-clausal form) into CNF.

2.3.1 Popular Algorithms for SAT

Davis & Putnam Algorithm

The first SAT algorithm is traditionally attributed to Davis and Putnam [8] and referred to as *Davis Putnam procedure* (or DP in short). The original DP algorithm is based on *resolution*. In resolution, a variable v is selected and a resolvant (see the definition below) using v is added to the original formula. This process is repeated to exhaustion or until an empty clause is generated. The original formula is not satisfiable if and only if an empty clause is a resolvant. More formally, we can define *resolution* using following definition:

Definition 2.8 *(**Resolution**) Given two clauses $C_1 = (v \vee x_1 \vee x_2 \vee \ldots \vee x_m)$ and $C_1 = (\bar{v} \vee y_1 \vee y_2 \vee \ldots \vee y_n)$, where all x_i and y_i are distinct, the resolvant of C_1 and C_2 is the clause $(x_1 \vee \ldots \vee x_m \vee y_1 \vee \ldots \vee y_n)$ that is the disjunction of C_1 and C_2 without v or $\bar{v}$. The resolvant is the consequence of logical AND of the pair of clauses.*
Resolution is the process of repeatedly generating resolvant from the original clauses and previously generated resolvants, until either the null clause is derived or no more resolvants can be created. In former case, the formula is unsatisfiable and in the latter case, it is satisfiable.

A later version of DP, due to Davis, Logeman and Loveland [16], usually known as DPL, uses *splitting rule* which replaces the original problem into two smaller subproblems, whereas DP uses *resolution* which replaces original problem with one (usually larger problem).

Definition 2.9 *(**Splitting**) In splitting, a variable v is selected from a formula, and the formula is replaced by one sub-formula for each of two possible truth assignments to v. Each subformula has all the clauses of the original except those satisfied by the assignment to v and otherwise all the literals of the original formula except those falsified by the assignment. Neither sub-formula contains v, and the original formula has a satisfying truth assignment if and only if either sub-formula has one. Splitting insures that a search for a solution terminates with a result.*

DPL is implemented more often than DP because variable elimination (using resolution) has four disadvantages: (1). It is more difficult to implement than splitting rule; (2) it tends to repeatedly increase the length and number of clauses; (3) it tends to generate a lot of duplicate clauses and (4) it very rarely generates unit clauses. Furthermore, DPL's splitting rule also makes it easier to construct a certificate of satisfiability[3], whereas DP makes it easier to construct a certificate of unsatisfiability.

There are many variants of Davis-Putnam procedure, each variant differing for the set of rules implemented for performance optimization. Two such rules are *unit clause rule* and *empty clause rule*.

[3]This can be used for counterexample generation in symbolic model checking.

Definition 2.10 *(**Unit Clause Rule**): If the formula contains some clause with only one literal, then select that variable and assign it a value that satisfies the clause containing it; otherwise select any other variable for splitting.*

Definition 2.11 *(**Empty Clause rule**): If the formula contains some empty clause (a clause which always has value false) then exit and report that the formula is unsatisfiable; otherwise select any other variable for splitting.*

Algorithm 2.3.1: DP(ϕ, μ)

$/* Schema for Davis and Putnam Algorithm */$

if $\phi = T$
  then return μ; $/* Base */$
if $\phi = F$
  then return $False$; $/* Backtrack */$
if *a unit clause*(l) *occurs in* ϕ $/* UnitClause */$
  then return DP$(assign(l, \phi), \mu \cup l)$;
$l \leftarrow$ CHOOSE-LITERAL(ϕ, μ); $/* Split */$
return DP$(assign(l, \phi), \mu \cup l)$; *or*
  DP$(assign(\neg l, \phi), \mu \cup \neg l)$;

The basic algorithm for DPL is given in Algorithm 2.3.1. To search the satisfiability of CNF formula ϕ, the procedure has to be invoked by the call DP $(\phi, \{\})$, where $\{\}$ is the empty assignment. DP $(\phi, \{\})$ returns as assignment μ if ϕ is satisfiable, and False otherwise. If N is the number of propositional variables in the formula, DP searches in a space of 2^N assignments. Notice that the number of the propositional variable is thus more critical than the size $|\phi|$ of the formula in determining the run-time of the basic DP procedure. There is a wide variety of tools that implement some flavor of DP algorithm. ZCHAFF, SATO, SIM and GRASP are few of them.

Stalmarck's procedure

Stalmarck's *Saturation method* [22] is a patented algorithm that can be used for satisfiability checking. The method has been successfully applied in an wide range of industrial applications. The algorithm takes the set of the formulas $\{x_1, x_2, \ldots, x_n\}$ as input, and produces an equivalence relation over the negated and unnegated subformulas of all x_i. Two subformulas are equivalent according to the resulting relation only when this is a logical consequence of assuming that all formulas x_i are true. The algorithm computes the relation by carefully propagating information according to the structure of the formulas.
The saturation algorithm is parameterized by a natural number k, the *saturation level*, which controls the complexity of the propagation procedure. The worst case time complexity of the algorithm is O (n^{2k+1}) in the size n of the

formulas, so that for a given k, the algorithm runs in polynomial time and space. A fortunate property is that this k is surprisingly low (usually 1 or 2) for many practical applications, even for extremely large formulas.
The advantage of having control over the saturation level is that the user can make a trade-off between the running time and the amount of information that is found. A disadvantage is that it is not always clear what k to choose in order to find enough information. In contrast, finding equivalences using BDDs results in discovering either all information, or no information at all due to excessive time and space usage. The SAT-solvers based on stalmarck's procedure is PROVER.

2.3.2 Dealing with non-CNF formulas

Traditionally, the satisfiability problem for propositional logic deals with formulas in CNF (also known as clausal form). A typical way to deal with non-CNF formulas requires (1) converting them into CNF, and (2)applying SAT solvers based on above discussed approaches. A formula can be converted to CNF using following methods:

1. **Classical Method:** Conversion of a formula ϕ into CNF can be done using this method as follows: first, ϕ is converted into negation normal form and then the rule

$$(\bigwedge_i \bigvee_j \phi_{ij}) \vee (\bigwedge_k \bigvee_m \phi_{km}) \Longrightarrow \bigwedge_{i,k} (\bigvee_j \phi_{ij} \vee \bigvee_m \phi_{km})$$

   is recursively applied to distribute $\wedge$'s over $\vee$'s. As result of this conversion, we get a formula ψ which is logically equivalent to ϕ. However, this transformation may lead to a considerable increase in the size of the formula (in the worst case, $|\psi|$ is $O(2^{|\phi|})$), which makes the method of no practical utility in many cases.

2. **Using Auxillary Variables:** A more convenient way to convert ϕ into CNF is based on the idea of renaming the sub-formulas of ϕ. In this method, a newly introduced variable a_{ϕ_i} is associated to each non-literal subformula ϕ_i of ϕ. Then each a_{ϕ_i} substitutes every occurrence of ϕ_i inside ϕ, and the expression $(a_{\phi_i} \Leftrightarrow \phi_i)$ is added to the result.
   In order to identify all subformulas of the expression ϕ, it is first represented as a *directed acyclic graph* (DAG) and then each nonterminal node of the DAG will represent a subformula ϕ_i of ϕ.

   Example 2.3 *To identify all subformulas of the Boolean function* $\phi = ((x_1 \cdot x_2 + (x_2 + x_3)) \Leftrightarrow (x_3 \cdot (x_3 + \overline{x_4})))$, *it can be represented in a DAG as Figure 2.4. Auxillary variables* a_1, a_2, a_3, a_4, a_5 *and* a_6 *are introduced corresponding to each nonterminal (subformula of* ϕ*). Using above defined*

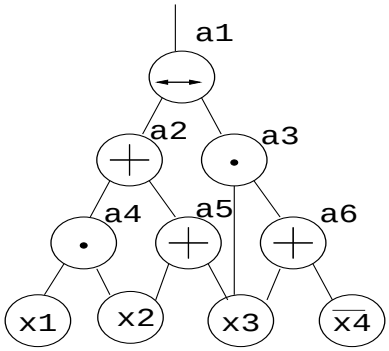

Figure 2.4: Representation of a Boolean formula as a DAG

approach, the CNF formula corresponding to ϕ would be:

$$\begin{aligned} a_1 \quad & \wedge \quad (a_1 \Leftrightarrow (a_2 \Leftrightarrow a_3)) \\ & \wedge \quad (a_2 \Leftrightarrow (a_4 + a_5)) \\ & \wedge \quad (a_4 \Leftrightarrow (x_1 \cdot x_2)) \\ & \wedge \quad (a_5 \Leftrightarrow (x_2 + x_3)) \\ & \wedge \quad (a_3 \Leftrightarrow (x_3 \cdot a_6)) \\ & \wedge \quad (a_6 \Leftrightarrow (x_3 + \overline{x_4})) \end{aligned}$$

The subformulas that are not in CNF form, can be converted to CNF using the definition of $\Leftrightarrow$ operator and the classical method discussed earlier. Here since the number of variables in each subformula is at most three, this conversion is practical.

Davis Putnam Algorithm for non-CNF formulas

An interesting improvement of D&P method is given by Giunchiglia et al. [13], when applied to non-CNF formulas. Suppose we want to check the satisfiability of a non-CNF formula ϕ. Before applying DP, we must convert ϕ to corresponding CNF formula ψ in the original variable and in a set of K newly added variables. Therefore, standard DP will search in a space of 2^{N+K} assignment, for it may backtrack on newly added *auxillary variables*. In the approach taken by [13], splitting is not performed on newly added variables, eliminating thus 2^K factor. The underlying idea of these variants is that splitting should occur only for the variables in original non-CNF formula. [13] gives two methods based on the above principle known as DP* and DP** algorithms. This method is implemented in a variety of SAT-solvers including SIM and ZCHAFF.

Chapter 3

Related Work

The main idea behind *symbolic model checking* is to represent sets of states and transition relations as *characteristic formulae* (Boolean expression). After the introduction of *Symbolic model checking* technique by McMillan, a lot of representations have been suggested for storing and manipulating Boolean expressions. As discussed in chapter 2, BDDs and SAT are forerunners among them. In this chapter, we will discuss the key points of various BDD and SAT based algorithms.

Definitions

Definition 3.1 *(**Kripke Structure**) A* Kripke structure M *is a five tuple* $M = \{S, S_0, N, L, AP\}$ *where*

1. S *is a finite set of states.*
2. $S_0 \subseteq S$ *is the set of initial states.*
3. $N \subseteq S \times S$ *is a transition relation that must be total, that is for every state* $s \in S$ *there is a state* $s' \in S$ *such that* $N(s, s')$*.*
4. AP *denotes the set of atomic propositions.*
5. $L : S \rightarrow 2^{AP}$ *is a function that labels each state with the set of atomic propositions true in that state.*

Kripke structure is one of the popular mathematical representation of finite state machines. In the remaining text we use Kripke structure to represent a finite state machine.

3.1 BDD-based symbolic model checking

Symbolic model checking has become identified with BDDs, a canonical form for Boolean formula representation that has proved to be quite efficient for this purpose in practice. As we discussed in the previous chapter, efficient

algorithms for manipulation of BDDs exist. Since BDDs are canonical representation, substantial subexpression sharing occurs and that results in compact representation of Boolean expressions. In addition, canonicity implies that satisfiability and validity can be checked in constant time.
Many of the ideas in BDD based model checking can be explained considering the problem of computing reachable states. In the following subsection, we will discuss basic algorithms for symbolic reachability in context of "safety properties".

3.1.1 Symbolic Reachability Algorithms

Given the BDD for the initial state $Init(\mathbf{x})$ and transition relation $N(\mathbf{x}, \mathbf{x}')$, one step successors (also termed as *Image*) and one step predecessor(also termed as *pre-image*) of any set of states can be computed using BDD based symbolic algorithms. This can be done repeatedly to explore all the reachable states. If the safety property to be checked is $Prop(\mathbf{x})$, we characterize set of bad states $Bad(\mathbf{x})$ as $\neg Prop(\mathbf{x})$. There are two standard algorithms for performing *reachability analysis*:

Forward Reachability

In forward reachability we compute a sequence of formulas $F_i(\mathbf{x})$ that characterize the set of states that initial states can reach in at most i steps:

$$\begin{aligned} F_0(\mathbf{x}) &= Init \\ F_{i+1}(\mathbf{x}') &= F_i(\mathbf{x}') \vee \exists_{\mathbf{x}}[F_i(\mathbf{x}) \wedge N(\mathbf{x}, \mathbf{x}')] \end{aligned} \tag{3.1}$$

We terminate the sequence generation if one of the following condition occurs:

1. $F_n(\mathbf{x}) \wedge Bad(\mathbf{x})$ is satisfiable; it means that a bad state is reachable. Hence we can conclude that safety property is violated.

2. $F_i(\mathbf{x}) = F_{i+1}(\mathbf{x})$; It means that no new state is reachable using the transitions in transition relation. This situation is termed as *fixed point*. If we reach a fixed point without encountering a bad state, we can conclude that the system is safe with respect to the property under consideration.

Backward Reachability

In backward reachability, we compute a sequence of formulas $B_i(\mathbf{x})$ that characterize a set of states that can reach a bad states within i steps:

$$\begin{aligned} B_0(\mathbf{x}) &= Bad \\ B_{i+1}(\mathbf{x}) &= B_i(\mathbf{x}) \vee \exists_{\mathbf{x}'}[B_i(\mathbf{x}') \wedge N(\mathbf{x}, \mathbf{x}')] \end{aligned} \tag{3.2}$$

In a similar manner to forward reachability, we can stop the sequence generation if:

1. $B_n(\mathbf{x}) \wedge Init(\mathbf{x})$ is satisfiable; or

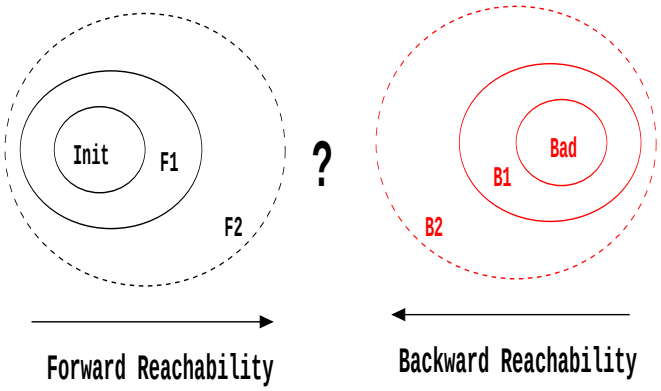

Figure 3.1: Forward and Backward reachability analysis

2. $B_i(\mathbf{x}) = B_{i+1}(\mathbf{x})$ holds.

The intuition behind *forward reachability* and *backward reachability* is shown in the Figure 3.1. These two reachability methods can be combined to perform reachability analysis by interleaving the steps of forward and backward reachability. The sequence generation can be terminated if we reach a fixpoint in any of the directions or F_n and B_n intersects.

3.1.2 Limitations of BDD based approaches

BDDs have proved to be a viable representation for doing symbolic reachability on large finite state machines. However, for many large systems, most sophisticated BDD based algorithms can not produce results. This is because the size of intermediate BDD, while computing the reachable state space, blows up beyond the memory capabilities of most machines. This is commonly known as *BDD blowup problem.*

As discussed in the previous chapter, the size of the BDD depends heavily on the variable ordering. The generation of a variable ordering that results in small BDDs is often time consuming or needs human intervention. Moreover, there exists some functions that can not be represented efficiently regardless of the input ordering. Unfortunately, some important functions like *integer multiplication* [3] falls within this category.

3.2 SAT-based symbolic model checking

Propositional decision procedures(SAT) also operate on Boolean expressions but do not use canonical forms. They also do not suffer from potential space blowup of BDDs and can handle propositional satisfiability problem with thousands of variables. These strengths of SAT procedures makes SAT-based model checking quite promising.

In this section, we review the work done in the field of SAT based model checking and will compare our solution with each of them.

3.2.1 Bounded Model Checking

Bounded model checking [2] procedure, proposed by Clarke et al., searches for counterexamples by "unrolling" the transition relation k steps, for increasing values of k. At each step k, the unrolling characterizes the set of paths of length k through the transition relation, and is described as a formula (without quantifiers). If no counterexample is found, the search is terminated when the value of k is equal to the diameter of the system. However, the value of the diameter is usually hard to compute, making BMC incomplete in practice. In other words, unless bound on the length of the counterexamples is not given, BMC can not actually verify the the given property, it can only produce counterexamples. The technique presented in this thesis is not "bounded" and hence can prove the correctness if the property is true.

3.2.2 Standard Reachability based Approaches

Abdulla et al. [19] have shown how to adapt standard algorithms for symbolic reachability analysis to work with SAT-solvers. They introduced *Reduced Boolean Circuits* (RBC) [10], a non canonical representation for Boolean formulas. The advantage of using a non canonical representation is that they are more succinct than canonical ones. On the other hand, satisfiability checking with non-canonical data structures is hard. They used SAT solvers to determine the satisfiability of RBC representation of the Boolean formula. The only operation of reachability analysis that does not straightforwardly carry over using RBCs is quantification over propositional variables.

For *quantifier elimination* they contributed some really effective heuristics to simplify *Quantified Boolean formula*. In particular *quantification-by-substitution rule* (or in-lining rule)

$$\exists x.(x \leftrightarrow \psi) \wedge \phi(x) \Longleftrightarrow \phi(\psi) \tag{3.3}$$

turned out to be very useful. We will also use this heuristic to remove the need of quantification over state variables. For quantification over input variables, they naively resolve it, using equation 3.4, yielding an exponential blowup in representation size.

$$\exists x.\phi(x) \Longleftrightarrow \phi(0) \vee \phi(1) \tag{3.4}$$

This makes their approach impractical, if the number of input variables are large. Our algorithm removes the requirement of quantification over input variables, making model checking of FSMs with large input variables practical.

Clarke et al. [24] adopted a similar approach for standard reachability analysis using SAT solvers. They introduced another non canonical data structure

Boolean Expression Diagrams (BED) [24] to represent the Boolean formula. In their approach, they use similar approaches for quantification removal. They have reported improvements over Abdulla et al.'s approach. But again, their method is also susceptible to possible blowup in representation size if number of inputs is large.

3.2.3 Induction based Approaches

This approach was initially suggested by Sheeran et al. [17] for verifying safety properties over finite state systems. It is based on unfolding the transition relation to the length of longest "shortest path" between two states. The fact that this length has been reached can be checked using a SAT solver. Thus, unlike BMC, this method can verify the correctness of a property.

The performance of SAT-solvers depends heavily on the size of the Boolean formula. The above scheme of "unfolding " the transition relation increases the size of the Boolean expression at each step. The approach taken by us does not involve unfolding the transition relation and hence can possibly produce small Boolean expressions.

3.2.4 SAT based Quantifier Elimination

In [18], McMillan used Satisfiability solving for quantification removal. In particular, he showed that with a slight modifications, modern SAT algorithms can be used to eliminate universal quantifiers from an arbitrary quantified Boolean formula, producing a result in CNF. This method requires modification in the satisfiability solvers whereas our method can work with existing SAT solvers.

3.2.5 Quantifier Elimination using decomposition

The work most closely related to ours is by Chakraborty & Soni [21]. They initially suggested the method of model checking using decomposition (defined formally in Chapter 4) of the Kripke structure. In their approach for decomposition, number of components can be large even if there exists a short counterexample. In this thesis we have used their approach for quantifier elimination and enhanced it by guiding the decomposition process such that the counterexample of length k can be detected using at most k components. We call this technique "counter-example focused" decomposition.

Chapter 4

Quantifier Elimination

From the discussion in previous chapters, it is clear that existential quantification plays a central role in symbolic backward (as well as forward) reachability analysis. The standard symbolic reachability algorithm [9] applies this operation repeatedly during a breadth-first traversal of the state space, until a fixed point is reached. If the diameter of the backward reachable part of the Kripke structure is large, backward reachability analysis entails a large number of applications of existential quantification, even with iterative squaring. Unfortunately, existential quantification is an expensive operation to implement in BDDs. In fact, BDD packages typically provide an optimized *AndAbstract* operator to aid this step of reachability analysis. While this works better than performing conjunction and quantification separately, it still leads to blowup of intermediate BDD sizes in most cases.

In this chapter, we address this bottleneck by presenting a technique that does not require quantification for symbolic reachability analysis. In this technique, the need of quantification over input variables is removed by decomposing the Kripke structure of the system into a set of Kripke structures (known as *components*). This technique was initially proposed by Chakraborty and Soni [21]. Their approach has the disadvantage that even if a short counterexample exists, decomposition may results in a large number of components. We further enhance this technique by guiding the decomposition process such that if the length of the shortest counterexample is k, counterexample can be detected using at most k components. In the remainder of this chapter, we present our ideas in the context of backward reachability analysis. Extensions to forward reachability analysis are discussed in Section *5.4*.

4.1 Motivation

For deterministic systems, Filkorn [23] proposed an alternative approach to traditional backward Image computation. Filkorn showed that *pre-image* of a set can be obtained by substituting the state variables with their corresponding *next state functions*. This approach removes the need of quantification over state variables. Similar ideas for quantifier simplification have been used by

Abdulla et al. [19] and Williams et al. [24] in the context of symbolic reachability analysis using SAT based techniques.

Consider the problem of backward image computation as discussed in Chapter 3. The most expensive step of backward image computation is computing the relational product $\exists_{\mathbf{x}'}[B_i(\mathbf{x}') \wedge N(\mathbf{x}, \mathbf{x}')]$ (equation 3.2), where $B_i(\mathbf{x}')$ is the current set of states and $N(\mathbf{x}, \mathbf{x}')$ is the next state transition relation. For deterministic finite state machines (like hardware sequential circuits), transition relation can also be written in the form of *transition functions*, where each of the $\mathbf{x}'$ can be written as $x'_k = f_k(\mathbf{x}, \mathbf{i})$.

Example 4.1 *Consider the finite state machine shown in Figure 4.2.3. Let the set of state variables is* $\mathbf{x} = \{x_1, x_2\}$ *and corresponding next state version is* $\mathbf{x}' = \{x'_1, x'_2\}$. i_1 *and* i_2 *are the input variables. Next state function of this finite state machine will be:*

$$\begin{aligned} x'_1 &= x_1 + i_1.x_2 \\ x'_2 &= x_1 \end{aligned}$$

The pre-image of a set $B_i(\mathbf{x}')$ can now be computed as $\exists_{\mathbf{x}',\mathbf{i}}[B_i(\mathbf{x}') \wedge (\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{i}))]$, which using Filkorn's approach can be rewritten as $\exists_{\mathbf{i}}[B_i(\mathbf{f}(\mathbf{x}, \mathbf{i}))]$. Now the resulting quantified Boolean formula is free from the quantification over state variables and we are left with the problem of quantifying the input variables. Thus, if there are fewer primary input variables than next state variables, it is advantageous to use this simplification.

4.1.1 Quantifying Input variables

While Filkorn's method allows us to quantify primary input variables instead of next state variables, it does not eliminate the need for quantification. If, however, the next state depends only on the present state, then we can express $N(\mathbf{x}, \mathbf{x}')$ as $\bigwedge_{j=1}^{k}(x'_j \Leftrightarrow g_j(\mathbf{x}))$. The pre-image of a set $B(\mathbf{x}')$ of states can then be obtained simply as $B(\mathbf{g}(\mathbf{x}))$, where $\mathbf{g}(\mathbf{x})$ denotes the vector of next state functions g_j. We will henceforth say that a Kripke structure has the *unique successor* property if the next state is uniquely determined by the present state. It follows that backward reachability analysis of unique-successor Kripke structures does not require existential quantification. This leads to the following observation: *Given an arbitrary Kripke structure, if we "decompose" it into a set of unique-successor Kripke structures, backward reachability analysis can be performed without existential quantification.*
This is the key intuition behind our work. Once the set of unique-successor Kripke structures are obtained, techniques analogous to MBM or FBF [14] can be applied to obtain the backward reachable states from a target set of states, without using existential quantification. Our problem therefore reduces to "decomposing" a Kripke structure into a set of unique-successor structures.

Basic Definitions

Definition 4.1 *(***Component***): A Component of a Kripke structure* $M = \{S, S_0, N, L, AP\}$ *is another Kripke structure* $C = \{S^c, S_0^c, N^c, L^c, AP^c\}$ *such that:*

1. $S^c = S$.
2. $S_0^c = S_0$.
3. $N^c \subseteq N$ *and* N^c *satisfies* unique successor property, *that is for* $s, s_1, s_2 \in S$; $(s, s1) \in N^c$ *and* $(s, s2) \in N^c$ *implies* $s_1 = s_2$.
4. $L^c = L$.
5. $AP^c = AP$.

Definition 4.2 *(***Notion of covering***): A component* C **covers a transition** (s_1, s_2) *of a Kripke structure* M, *if* $(s_1, s_2) \in N^c$.

Definition 4.3 *(***Decomposition***): A decomposition* D *of a Kripke structure* M *can be defined as a set of components of* M. *A decomposition is* complete *if every transition of* M *is covered by at least one component in* D. *Otherwise,* D *is a partial decomposition. A complete decomposition may be viewed as a collection of* transition functions, *the union of which gives the original transition relation of* M.

Since in each component every state has exactly one outgoing edge, transition function of the components will be of the form $\mathbf{x}' = \mathbf{f}(\mathbf{x})$, backward reachability computation over the component can be done without quantification. Since all the transitions of a Kripke structure M are covered in a complete decomposition, reachable states of the M are equal to the union of the reachable states of all the components of the complete decomposition. Hence if we can find a decomposition of a finite state machine, backward reachability analysis can be done without *existential quantification.*

Hence, the problem of existential quantification of inputs reduces to finding a decomposition of the finite state machine.

4.1.2 Minimality Criteria

For a particular finite state machine M, several complete decompositions are possible. A naive approach to obtain a complete decomposition of M is to substitute all possible values for the input variables in next state functions of M one-by-one to generate all components. For example in Figure 4.2.3, component(c_1) can be obtained by substituting (00) for inputs ($i_1 i_2$), $N_{c_1} = \{(x_1' = x_1), (x_2' = x_1)\}$. But the number of components generated using this

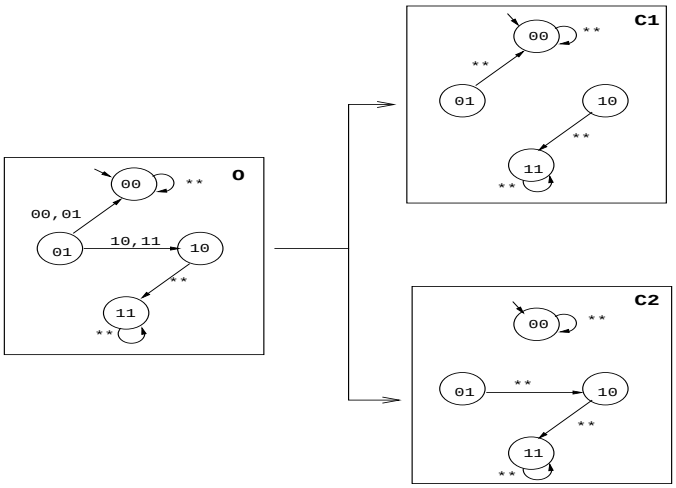

Figure 4.1: Our Approach

approach would be 2^m, where m is the number of input variables.

If a state in the transition graph goes to different states for all possible input values, that is its out-degree is (2^m), a minimum of 2^m components are needed to generate a complete decomposition. But fortunately in most practical circuits, there are lot of overlap of edges in the transition relation. If we exploit this feature we can expect the number of components to be significantly less than 2^m. We observed that any minimal decomposition of a Kripke structure should satisfy following properties:

Definition 4.4 *(***Minimal Decomposition***): A decomposition is* minimal *if:*

1. *It is complete.*
2. *There does not exist any other complete decomposition with fewer components.*

Note that an minimal decomposition may not be unique.

A minimal decomposition can be generated by following the two guidelines mentioned below:

1. Each of the components should have the maximum uncovered edges, that is repetition of edges should occur only when all the outgoing edges have been covered.
2. If a particular transition is covered for an input, the other input values which give the same transition, should also be considered as covered. We call such transitions *implicitly covered*. For example in Figure 4.2.3, the transition from state (01) to (00) occurs for input (01) and (00) so if we cover the transition on input (00) , transition from state (01) to (00) on input (01) will get covered implicitly.

In the next section, we describe a technique for obtaining a minimal decomposition of a Kripke structure arising out of a deterministic finite state machine. We will see later that our method generalizes to arbitrary Kripke structures as well.

Notations

In the following discussion, we will adhere to following notations until specified otherwise:

- Bold face alphabets are used to represent vectors.
- alphabet $\mathbf{x}$ will represent a vector of next state variables $\{x_1, x_2, x_3, \ldots, x_n\}$ and $\mathbf{i}$ will represent a vector of input variables $\{i_1, i_2, i_3, \ldots, i_m\}$.
- Similarly, $\mathbf{f}(\mathbf{x}, \mathbf{i})$ will represent a vector of functions having support set $\mathbf{x}$ and $\mathbf{i}$.
- $\overline{F}$ will represent the negation of the function F, that is, $\overline{F} = \neg F$.

4.2 Basic Technique for Symbolic Minimal Decomposition

As we discussed above, the next state of a deterministic finite state machine can be expressed as a function of the present state and primary inputs. Using notation introduced earlier, let $\mathbf{f}(\mathbf{x}, \mathbf{i})$ represent the vector of next state functions. Thus, $x'_j \Leftrightarrow f_j(\mathbf{x}, \mathbf{i})$ for all j in 1 through k, where k denotes the number of state variables. In order to obtain a component of a decomposition, however, we must remove the functional dependency of x'_j on the primary inputs $\mathbf{i}$. In other words, we must express x'_j as $h_j(\mathbf{x})$ for some suitable function h_j. We observe that this can be achieved if we express each primary input variable i_l as a function, $g_l(\mathbf{x})$, of the present state. Viewed in another way, this amounts to specifying a primary input combination for each state to help choose an outgoing transition from that state in the deterministic finite state machine. Let $\mathbf{G}(\mathbf{x})$ denote the vector of functions $g_l(\mathbf{x})$, where l ranges from 1 to m (number of primary inputs). We can then express each x'_j as $f_j(\mathbf{x}, \mathbf{G}(\mathbf{x}))$, thereby obtaining a component.

To formalize the above intuition, we define a few additional terms.

Definition 4.5 *(***Uncovered Edge function***): Let M be a Kripke structure arising out of a deterministic finite state machine, and let D be a decomposition (possibly partial) of M. We define a Boolean function $E^D(\mathbf{x}, \mathbf{i})$, called the* Uncovered Edge function *for D, such that:*
$E^D(\mathbf{x}, \mathbf{i}) = 1$ iff the transition $(\mathbf{x}, \mathbf{f}(\mathbf{x}, \mathbf{i}))$ is not covered by any component in D.
$E^D(\mathbf{x}, \mathbf{i}) = 0$ otherwise.

Thus, $E^\phi(\mathbf{x}, \mathbf{i}) \equiv \mathbf{1}$, and $E^D(\mathbf{x}, \mathbf{i}) \equiv \mathbf{0}$ for a complete decomposition D.

Using *Uncovered Edge function* $E^D(\mathbf{x}, \mathbf{i})$, we have to determine the value of the *input vector of functions* $\mathbf{G}(\mathbf{x})$, which if substituted for the input variables in transition relation of M will results in a minimal component of M. We call this vector *Uncovered Input Function Vector.*

Definition 4.6 *(***Uncovered Input Function Vector** *): Let m denote the number of primary inputs of a deterministic finite state machine. Given a (possibly partial) decomposition D of M, we define a vector of functions $\mathbf{G}^D(\mathbf{x}) = \langle g_1^D(\mathbf{x}), \ldots, g_m^D(\mathbf{x})\rangle$, such that*

- *If one or more outgoing transitions from state $\mathbf{x}$ are uncovered by the components in D, then $(\mathbf{x}, \mathbf{f}(\mathbf{x}, \mathbf{i}))$ is an uncovered transition.*
- *If all outgoing transitions from state $\mathbf{x}$ are covered by components in D, then $(\mathbf{x}, \mathbf{f}(\mathbf{x}, \mathbf{i}))$ is a covered transition.*

Note that there could be multiple vectors $\mathbf{G}^D(\mathbf{x})$ satisfying the above condition, for a given finite state machine and decomposition D.
Intuitively, $\mathbf{G}^D(\mathbf{x})$ gives a combination of primary input values that takes state $\mathbf{x}$ to state $\mathbf{x}'$ in the finite state machine, where $(\mathbf{x}, \mathbf{x}')$ is not covered by any component in decomposition D. Every vector $\mathbf{G}^D(\mathbf{x})$ satisfying the above conditions is called an **uncovered input function vector** *for decomposition D.*

Definition 4.7 *(***Implicitly Covered Edge Function** *): Given an uncovered input function vector $\mathbf{G}(\mathbf{x})$, the* **implicitly covered edge function** *is a Boolean predicate, $S^{\mathbf{G}}(\mathbf{x}, \mathbf{i})$, that evaluates to 1 iff $\mathbf{f}(\mathbf{x}, \mathbf{i}) \Leftrightarrow \mathbf{f}(\mathbf{x}, \mathbf{G}(\mathbf{x}))$. In other words, $S^{\mathbf{G}}(\mathbf{x}, \mathbf{i})$ indicates whether application of primary input $\mathbf{i}$ in state $\mathbf{x}$ of the finite state machine takes us to the same state as the application of primary input $\mathbf{G}(\mathbf{x})$.*

4.2.1 Computing *Uncovered Input Function Vector*

Problem Statement Given an *Uncovered Edge function* $E^D(\mathbf{x}, \mathbf{i})$, find $G(\mathbf{x}) = \{(i_1 = g_1(\mathbf{x})), (i_2 = g_2(\mathbf{x})), ..., (i_m = g_m(\mathbf{x}))\}$ which takes a state $\mathbf{x}$ as input and gives value of input $i_1, .., i_m$ corresponding to an uncovered transition from $\mathbf{x}$.

For the sake of simplicity, assume that there is only one input variable. Now, given *uncovered edge function* $E^D(\mathbf{x}, \mathbf{i})$, we have to compute *uncovered input function vector* $G(\mathbf{x}) = \{(i = g(\mathbf{x}))\}$ to generate a component for the minimal decomposition. As clear from the Figure 4.2, for a given uncovered edge function $E^D(\mathbf{x}, \mathbf{i})$, for each state $\mathbf{x}$ following four cases are possible:

1. From the state $\mathbf{x}$ the edge labeled with input $i = 1$ is uncovered and $i = 0$ is covered (Figure 4.2.a). Hence to produce a component, we should choose the transition corresponding to $i = 1$. So in this case $g(\mathbf{x}) = 1$. Let us characterize this case by $f_1(\mathbf{x})$.

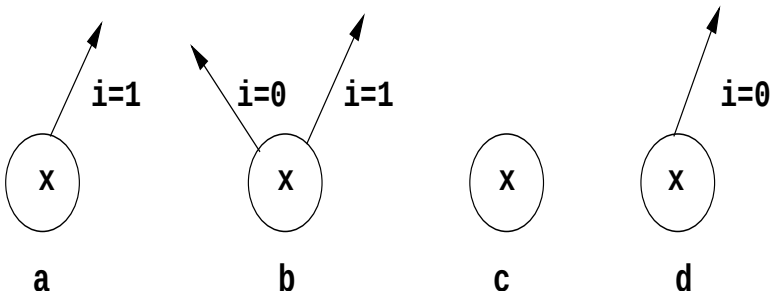

Figure 4.2: Uncovered Edge function: all possible cases

2. From the state $\mathbf{x}$ both the edges, that is, with $i = 0$ and $i = 1$ are uncovered (Figure 4.2.b). Hence we can choose any edge without violating the minimality property. In this case, $g(\mathbf{x})$ may be any function of $\mathbf{x}$. Let this function be $u(\mathbf{x})$. Let us characterize this case by $f_2(\mathbf{x})$.

3. From the state $\mathbf{x}$ both the edges, that is, with $i = 0$ and $i = 1$ are covered (Figure 4.2.c). Hence we can choose any edge without violating the minimality property. In this case also, $g(\mathbf{x})$ may be any function of $\mathbf{x}$. Let this function be $v(\mathbf{x})$. Let us characterize this case by $f_3(\mathbf{x})$.

4. From the state $\mathbf{x}$ the edge labeled with input $i = 0$ is uncovered and $i = 1$ is covered (Figure 4.2.d). Hence to produce a component we should choose edge corresponding to $i = 0$. So in this case $g(\mathbf{x}) = 0$. Let us characterize this case by $f_4(\mathbf{x})$

Based on the above observations, we can derive the value of *uncovered input function* $g(\mathbf{x})$ as

$$\begin{aligned} g(\mathbf{x}) &= f_1(\mathbf{x}).1 + f_2(\mathbf{x}).u(\mathbf{x}) + f_3(\mathbf{x}).v(\mathbf{x}) + f_4(\mathbf{x}).0 \\ &= f_1(\mathbf{x}) + f_2(\mathbf{x}).u(\mathbf{x}) + f_3(\mathbf{x}).v(\mathbf{x}) \end{aligned} \tag{4.1}$$

Given *uncovered edge function* $E^D(\mathbf{x}, \mathbf{i})$, f_1, f_2 and f_3 can be computed as:

$$f_1(\mathbf{x}) = E^D(\mathbf{x}, \mathbf{1}) \wedge \overline{E^D(\mathbf{x}, \mathbf{0})} \tag{4.2}$$

$$f_2(\mathbf{x}) = E^D(\mathbf{x}, \mathbf{1}) \wedge E^D(\mathbf{x}, \mathbf{0}) \tag{4.3}$$

$$f_3(\mathbf{x}) = \overline{E^D(\mathbf{x}, \mathbf{1})} \wedge \overline{E^D(\mathbf{x}, \mathbf{0})} \tag{4.4}$$

Hence given the *uncovered edge function* $E^D(\mathbf{x}, \mathbf{i})$, *uncovered input function* $g(\mathbf{x})$ can be computed using equations 4.1, 4.2, 4.3 and 4.4.

Generalization

The above procedure for computing $G(\mathbf{x})$ for one input variable can be generalized for any number of inputs in following manner. We consider each input one-by-one and consider other inputs as state variables and use the formula 4.1

to compute the value of *uncovered input function* for that input. For example, suppose our *uncovered edge function* is $E^D(\underbrace{\mathbf{x}, i_1, ...i_{m-1}}, i_m)$. We can compute *uncovered input function* corresponding to input i_m as $i_m = g_m(\mathbf{x}, i_1, ..., i_{m-1})$ using the equation 4.1. Now we substitute the uncovered input function g_m for the input i_m in the uncovered edge function as $E_m^D = E(\mathbf{x}, i_1, ...i_{m-1}, g_m(...))$ or $E_m^D(\underbrace{\mathbf{x}, i_1, ..., i_{m-2}}, i_{m-1})$. Now g_{m-1} can be computed in similar manner using E_m^D as the *uncovered edge function*. Similarly we can compute *uncovered input function* g_i for each input variables i to generate uncovered input function vector $\mathbf{G}(\mathbf{x})$. Now, to generate a component, we replace each input variable for corresponding *uncovered input function*, in the order they are computed, to get a component of M towards minimal decomposition.

Updating Uncovered Edge function

After generating a component, we need to update the *uncovered edge function* by eliminating all *implicitly covered edges* by that component. The edges covered by a component $S^{\mathbf{G}}(\mathbf{x}, \mathbf{i})$ can be computed using the Definition 4.7. Then to exclude these edges from the *uncovered edge graph* we compute the *uncovered edge graph* as $E^D(\mathbf{x}, \mathbf{i}) = E^D(\mathbf{x}, \mathbf{i}) \cdot \overline{S^{\mathbf{G}}(\mathbf{x}, \mathbf{i})}$. The algorithm for computing the covered edges is shown below:

Algorithm 4.2.1: COMPUTECOVEREDEDGES$(C, \mathbf{f})$

/ * *To compute the set of covered edges of* M *by a component* C
C : *A component of the finite state machine* M;
$\mathbf{G(x)}$: *Uncovered Input Function Vector of component* C;
$\mathbf{f(x, i)}$: *Transition Function Vector of* M;
n: number of state variables
return $\mathbf{f(x, i)} \Leftrightarrow \mathbf{f(x, G(x))}$

Example 4.2 *Let us consider the finite state machine described in example 4.1. The transition function of this finite state machine is as follows:*

$$\begin{aligned} x_1' &= x_1 + i_1.x_2 \\ x_2' &= x_1 \end{aligned}$$

To find a minimal decomposition, let us first define uncovered edge function $E^D(\mathbf{x}, \mathbf{i}) = 1$ *where* $D = \phi$.
Taking $u(\mathbf{x}) = 0$ *and* $v(\mathbf{x}) = 0$, *from equation 4.1 and equation 4.2:*
$g(\mathbf{x}) = f_1(\mathbf{x}) = E^D(\mathbf{x}, \mathbf{1}) \wedge \overline{E^D(\mathbf{x}, \mathbf{0})}$
Computation of the components is given below:
First Component

$i_1 = g_1(x_1, x_2) = 0$
after replacing i_1 *by* g_1 *in uncovered edge function,* $E^D(\mathbf{x}, \mathbf{i}) = 1$.
$i_2 = g_2(x_1, x_2) = 0$
So, the first component will be:

$$\begin{aligned} x_1' &= x_1 \\ x_2' &= x_1 \end{aligned}$$

This component is shown in Figure 4.2.3 as component c_1.
Covered transitions by this component can be computed using Definition 4.7 as:
$S^{\mathbf{G}}(x_1, x_2, i_1, i_2) = (((x_1 + i_1 \cdot x_2) \Leftrightarrow x_1) \cdot (x_1 \Leftrightarrow x_1))$
$S^{\mathbf{G}}(\mathbf{x}, \mathbf{i}) = x_1 + \bar{i}_1 + \bar{x}_2$.
New uncovered edge graph would be:
$E^D(\mathbf{x}, \mathbf{i}) = E^D(\mathbf{x}, \mathbf{i}) \cdot \overline{S^{\mathbf{G}}(\mathbf{x}, \mathbf{i})} = 1 \cdot \overline{(x_1 + \bar{i}_1 + \bar{x}_2)}$
$E^D(\mathbf{x}, \mathbf{i}) = \bar{x}_1 \cdot i_1 \cdot x_2$
Second Component
$i_1 = g_1(\mathbf{x}, i_2) = E^D(\mathbf{x}, \mathbf{i_2}, \mathbf{1}) \wedge \overline{E^D(\mathbf{x}, \mathbf{i_2}, \mathbf{0})} = \bar{x}_1 x_2$
after replacing i_1 *by* g_1 *in uncovered edge function,* $E^D(\mathbf{x}, \mathbf{i}) = \bar{x}_1 x_2$.
$i_2 = g_2(\mathbf{x}) = E^D(\mathbf{x}, \mathbf{1}) \wedge \overline{E^D(\mathbf{x}, \mathbf{0})} = 0$
So, the second component will be:

$$\begin{aligned} x_1' &= x_1 + x_2 \\ x_2' &= x_1 \end{aligned}$$

This component is shown in Figure 4.2.3 as component c_2.
Covered transitions by this component can be computed using Definition 4.7 as:
$S^{\mathbf{G}}(x_1, x_2, i_1, i_2) = (((x_1 + i_1 \cdot x_2) \Leftrightarrow x_1 + x_2) \cdot (x_1 \Leftrightarrow x_1))$
$S^{\mathbf{G}}(\mathbf{x}, \mathbf{i}) = 1$.
New uncovered edge graph would be:
$E^D(\mathbf{x}, \mathbf{i}) = E^D(\mathbf{x}, \mathbf{i}) \cdot \overline{S^{\mathbf{G}}(\mathbf{x}, \mathbf{i})} = (\bar{x}_1 \cdot i_1 \cdot x_2) \cdot \overline{(1)} = 0$
Since, uncovered edge function is 0, there exists no uncovered edges. So we can terminate the generation of the component. The minimal decomposition $D = \{c_1, c_2\}$.

4.2.2 Algorithm

In the computation of uncovered input function $g(\mathbf{x})$, $u(\mathbf{x})$ and $v(\mathbf{x})$ can take any value. As we know from previous discussion, computation of the functions f_1, f_2 and f_3 involves taking cofactor of uncovered edge function $E^D(\mathbf{x}, i)$ for both values of i. However, if we select the values of tunable functions $u(\mathbf{x})$ and $v(\mathbf{x})$ properly, we can save one cofactor computation. For example, if $u(\mathbf{x})$ is taken as 1 and $v(\mathbf{x})$ is taken as 0, the equation 4.1 reduces to:

$$g(\mathbf{x}) = f_1(\mathbf{x}) + f_2(\mathbf{x}) = E^D(\mathbf{x}, \mathbf{1})$$

The pseudocode for generating a minimal decomposition of the Kripke structure and a component is given as Algorithm 4.2.3 and Algorithm 4.2.2 respectively. In these algorithms we use $u(\mathbf{x}) = 1$ and $v(\mathbf{x}) = 0$ to minimize computation efforts.

Algorithm 4.2.2: GENERATENEXTCOMPONENT1($E^D, \mathbf{f}$)

/ * *To Generate A Component Using Basic Graph Decomposition* * /
$E^D(\mathbf{x},\mathbf{i})$: *Uncovered Edge Function*
$\mathbf{f}(\mathbf{x},\mathbf{i})$: *Transition Function Vector of Original Kripke structure*
/ * $\mathbf{x}$ *is* $n-$*dimensional state vector and* $\mathbf{i}$ *is* $m-$*dimensional input vector* * /

$E^D_{m+1} \leftarrow E^D(\mathbf{x},\mathbf{i})$
$\mathcal{C}_{m+1} \leftarrow \bigwedge_{j=1}^{n}(x'_j \Leftrightarrow f_j(\mathbf{x},\mathbf{i}))$
for $k \leftarrow m$ **downto** 1
  do
    *Taking u(***x***)=1 and v(***x***)=0*
    compute $g_k(\mathbf{x}, i_1, \ldots, i_{k-1})$ *from* $(E^D_{k+1}(\mathbf{x},\mathbf{i}))_{1/i_k}$
    $E^D_k \leftarrow (E^D_{k+1})_{g_k/i_k}$
    $\mathcal{C}_k \leftarrow (\mathcal{C}_{k+1})_{g_k/i_k}$
return $\mathcal{C}_1$

Algorithm 4.2.3: GENERATEMINIMALDECOMPOSITION($\mathbf{f}$)

/ * *To Generate A Minimal Decomposition of the Kripke structure* * /
$\mathbf{f}(\mathbf{x},\mathbf{i})$: *Transition Function Vector of Original Kripke structure*
/ * n : *number of state variables*, m: *number of input variables* * /

local $E^D(\mathbf{x},\mathbf{i})$: *Uncovered Edge Function*
local $\mathcal{D}$: *Decomposition*
local $\mathcal{C}$: *Component*
$\mathcal{D} \leftarrow \phi$
$E^D(\mathbf{x},\mathbf{i}) \leftarrow 1$
repeat
  $\mathcal{C} \leftarrow$ GENERATENEXTCOMPONENT1($E^D, \mathbf{f}$)
  $COV \leftarrow$ COMPUTECOVEREDEDGES($C, \mathbf{f}$)
  $E^D \leftarrow E^D \wedge \overline{COV}$
  $\mathcal{D} \leftarrow \mathcal{D} \bigcup \{\mathcal{C}\}$
until $E^D = \phi$
return $\mathcal{D}$

4.3 Counter-example focused Graph Decomposition

In the decomposition scheme discussed in last section, to generate a component we select the edges to be covered in that component randomly. So while computing the backward image, a large number of components may require before termination of the algorithm; even if the counter-example is of small length. To overcome this disadvantage, we can guide the decomposition process in such a way that while selecting the transitions for a component, it prefer

those edges which can possibly lead to a counter-example. We call this scheme *Counter-example focused* Graph Decomposition.

4.3.1 Basic Technique

The main idea behind this scheme is to search the counterexample using as few components as possible. This is achieved by preferring those edges, which can possibly be the part of the counterexample in the generated component, over other ordinary edges. One way to select "preferred edges" is to choose those edges which are (1) uncovered and (2) coming directly from an state outside the set of failed states to any state in the set of failed states. *Failed states* are those states which are known to have transitions into *Bad states*. Preferring such edges guarantees that if there exists any edge which causes an initial state to make transition to a failed state in one step, new component will cover that edge. To implement it, we need a function which takes a state assignment as input and can give us the "preferred edges". We call this function "*preferred edge function*" $P^D(\mathbf{x}, \mathbf{i})$.

Definition 4.8 Preferred edge function *is a Boolean function* $P^D(\mathbf{x}, \mathbf{i})$, *such that:*
$P^D(\mathbf{x}, \mathbf{i}) = 1$ *iff (1) state* $\mathbf{x}$ *is not a failed state and from state* $\mathbf{x}$ *on input* $\mathbf{i}$ *there is a transition to any failed state and (2) edge* $(\mathbf{x}, \mathbf{i})$ *is not covered in any of the previously generated components.*
$P^D(\mathbf{x}, \mathbf{i}) = 0$ *otherwise.*

If $Fail(\mathbf{x})$ *represent the set of fail states, then* Preferred edge function *can be computed using following equation:*

$$P^D(\mathbf{x}, \mathbf{i}) = E^D(\mathbf{x}, \mathbf{i}) \wedge (\overline{Fail(\mathbf{x})} \wedge Fail(\mathbf{f}(\mathbf{x}, \mathbf{i})))$$

where $E^D(\mathbf{x}, \mathbf{i})$ *is uncovered edge function.*

Computing Uncovered Input Vector

Problem Statement Given a *Preferred Edge function* $P^D(\mathbf{x}, \mathbf{i})$ and an *Uncovered Edge function* $E^D(\mathbf{x}, \mathbf{i})$, find $G(\mathbf{x}) = \{(i_1 = g_1(\mathbf{x})), (i_2 = g_2(\mathbf{x})), ..., (i_m = g_m(\mathbf{x}))\}$ which takes a state $\mathbf{x}$ as input and gives value of input $i_1, .., i_m$ corresponding to preferable transition from $\mathbf{x}$.

While deciding the transitions to be covered in a component, our decomposition algorithm always tries to choose an edge (transition) from *Preferred edge function*. If there exists no *preferred edge*, it can select an edge from *Uncovered edge function*. Using a similar reasoning to basic decomposition algorithm, uncovered input function can be computed as:

$$g(\mathbf{x}) = f_1^P(\mathbf{x}) + f_2^P(\mathbf{x}).u^P(\mathbf{x}) + f_3^P(\mathbf{x}).v^P(\mathbf{x}) \tag{4.5}$$

Where f_1^P, f_2^P and f_3^P can be defined as

1.
$$f_1^P(\mathbf{x}) = P^D(\mathbf{x}, \mathbf{1}) \wedge \overline{P^D(\mathbf{x}, \mathbf{0})}$$
$f_1^P(\mathbf{x})$ will be true if from the state $\mathbf{x}$ the edge labeled with input $i = 1$ is a "preferred edge" and $i = 0$ is a non-preferred edge. Hence to generate the new component we choose edge corresponding to $i = 1$.

2.
$$f_2^P(\mathbf{x}) = P^D(\mathbf{x}, \mathbf{1}) \wedge P^D(\mathbf{x}, \mathbf{0})$$
$f_2^P(\mathbf{x})$ will be true if from the state $\mathbf{x}$ both the edges, that is, with $i = 0$ and $i = 1$ are preferred. Hence we can choose any of them. So $u^P(\mathbf{x})$ can be any function of $\mathbf{x}$.

3.
$$f_3^P(\mathbf{x}) = \overline{P^D(\mathbf{x}, \mathbf{1})} \wedge \overline{P^D(\mathbf{x}, \mathbf{0})}$$
$f_3^P(\mathbf{x})$ will be true if from the state $\mathbf{x}$ both the edges, that is, with $i = 0$ and $i = 1$ are not there in "preferred edge function". Now we can choose any edge from *uncovered edge function*. So from equation 4.1, $v^P(\mathbf{x})$ can be computed as:
$$v^P(\mathbf{x}) = f_1(\mathbf{x}) + f_2(\mathbf{x}).u(\mathbf{x}) + f_3(\mathbf{x}).v(\mathbf{x}) \tag{4.6}$$
where f_1, f_2 and f_3 are same as defined in 4.2, 4.3 and 4.4.

4.3.2 Exploiting Tunable Parameters

In the computation of $g(\mathbf{x})$, the functions $u^P(\mathbf{x}), u(\mathbf{x})$ and $v(\mathbf{x})$ can take any value. As we know from previous discussion, computation of the functions $f_1^P, f_2^P, f_3^P, f_1, f_2$ and f_3 involves taking cofactor of $P^D(\mathbf{x}, i)$ and $E^D(\mathbf{x}, i)$ with respect to both assignments of i. Since computation of cofactor with respect to both values of i has the complexity similar to that of *quantification* (equation 2.1), we would like to choose the values of the tunable parameters such that we can save at least one cofactor operation. Below we are giving one possible choice of $u^P(\mathbf{x}), u(\mathbf{x})$ and $v(\mathbf{x})$ that allows us to compute $g(\mathbf{x})$ taking only one cofactor of $P^D(\mathbf{x}, i)$ and $E^D(\mathbf{x}, i)$.

$g(\mathbf{x}) = f_1^P(\mathbf{x}) + f_2^P(\mathbf{x}).u^P(\mathbf{x}) + f_3^P(\mathbf{x}).v^P(\mathbf{x})$
and $v^P(\mathbf{x}) = f_1(\mathbf{x}) + f_2(\mathbf{x}).u(\mathbf{x}) + f_3(\mathbf{x}).v(\mathbf{x})$
So $g(\mathbf{x}) = f_1^P(\mathbf{x}) + f_2^P(\mathbf{x}).u^P(\mathbf{x}) + f_3^P(\mathbf{x}).(f_1(\mathbf{x}) + f_2(\mathbf{x}).u(\mathbf{x}) + f_3(\mathbf{x}).v(\mathbf{x}))$
Putting $u^P = 0, u = 1$ and $v = 0$:
$g(\mathbf{x}) = f_1^P(\mathbf{x}) + f_3^P(\mathbf{x}).(f_1(\mathbf{x}) + f_2(\mathbf{x}))$
since $f_1^P(\mathbf{x}) \subseteq (f_1(\mathbf{x}) + f_2(\mathbf{x}))./ *$ From the definition of Preferred Edge function $* /$
$g(\mathbf{x}) = f_1^P(\mathbf{x}).(f_1(\mathbf{x}) + f_2(\mathbf{x})) + f_3^P(\mathbf{x}).(f_1(\mathbf{x}) + f_2(\mathbf{x}))$
$g(\mathbf{x}) = (f_1^P(\mathbf{x}) + f_3^P(\mathbf{x})).(f_1(\mathbf{x}) + f_2(\mathbf{x}))$
$g(\mathbf{x}) = \overline{P^D(\mathbf{x}, \mathbf{0})}.E^D(\mathbf{x}, \mathbf{1})$ (4.7)

4.3.3 Algorithm

The algorithm for generating a new component is given below. It takes preferred edge function, uncovered edge function and transition function of the Kripke structure as input and generates a *component*.

Algorithm 4.3.1: GENERATENEXTCOMPONENT2($E^D, P^D, \mathbf{f}$)

$/* To\ Generate\ A\ Component\ Using\ Counterexample\ Focused\ *$
$*\ Decomposition\ */$
$E^D(\mathbf{x},\mathbf{i})$: *Uncovered Edge Function*
$P^D(\mathbf{x},\mathbf{i})$: *Preferred Edge Function*
$\mathbf{f}(\mathbf{x},\mathbf{i})$: *Transition Function Vector of Original Kripke structure*
/*n : *number of state variables*, m : *number of input variables* */

$E^D_{m+1} \leftarrow E^D(\mathbf{x},\mathbf{i})$
$P^D_{m+1} \leftarrow P^D(\mathbf{x},\mathbf{i})$
$\mathcal{C}_{m+1} \leftarrow \bigwedge_{j=1}^{n}(x'_j \Leftrightarrow f_j(\mathbf{x},\mathbf{i}))$
for $k \leftarrow m$ **downto** 1
do
  compute $g_k(\mathbf{x}, i_1, \ldots, i_{k-1})$ *from* $((E^D_{k+1}(\mathbf{x},\mathbf{i}))_{1/i_k} \cdot \overline{(P^D_{k+1}(\mathbf{x},\mathbf{i}))_{0/i_k}})$
  $E^D_k \leftarrow (E^D_{k+1})_{g_k/i_k}$
  $P^D_k \leftarrow (P^D_{k+1})_{g_k/i_k}$
  $\mathcal{C}_k \leftarrow (\mathcal{C}_{k+1})_{g_k/i_k}$
return $\mathcal{C}_1$;

After computing the component, set of covered edges by this component can be computed using Definition 4.7. In next chapter, we discuss an approach for checking safety properties over finite state machines using the *decomposition* approach discussed here.

Chapter 5

Lazy Decomposition

We are interested in checking safety properties over a large Kripke structure with large number of input variables. In the previous chapter, we explained that the need of quantification over input variables can be removed by performing reachability over the *decomposition* of the finite state machine. In this chapter, we present an efficient method for checking "safety properties" over the decomposition of the finite state machine. We call this algorithm *Lazy decomposition*. In the following text we are considering only *backward reachability*. At the end of the chapter, there is a note on extending this technique to perform *forward reachability*.

5.1 Basic Idea

Given a finite state machine M with next state transition function $\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{i})$, a set of initial states $Init(\mathbf{x})$, and a set of Bad states, characterize by $Fail(\mathbf{x})$; we wish to explore whether a state in $Init(\mathbf{x})$ can reach a state in $Fail(\mathbf{x})$ via the transitions of M. This is also known as *safety problem*.

A naive approach for checking a safety property over a finite state machine M would be computing a complete decomposition ($D = \{C_1, C_2, \ldots C_k\}$) of the Kripke structure using *basic decomposition approach* and traverse each component *serially* and *iteratively* until one of the following condition occurs: (1) fixed point in computation of the *backward reached set* $Bi(\mathbf{x})$ is obtained, or (2) backward reached set intersects with set of initial states. In the first case, we can conclude that M respects the property and in the second case, we can produce a counterexample showing the unsafe behavior of the system. This way of traversing the components is called as *machine-by-machine* (MBM) [14] traversal. Alternatively, we can traverse the components using an approach called *frame-by-frame*(FBF) [14]traversal. In FBF, instead of traversing C_i^{th} component before processing component C_{i+1}, as the MBM procedure does, we handle all components in *parallel*, and the traversal is a one-sweep process. We start traversing each component starting from set of *fail* states, and pre-image of the set is computed; then all component moves one time frame back and

another pre-image is computed(one per component) of the backward reached set. The traversal is terminated on either of two conditions discussed above.

Irrespective of the traversal method used, the above approach has a serious drawback. In this approach, complete decomposition is computed before starting the image computation process. Since generating a component is an expensive operation, we would like to avoid it as far as possible. We observed that it is not always necessary to compute complete decomposition in order to check the safety property over the finite state machine. It may so happen that using the partial decomposition we may detect a path from an initial state to a bad state. This is the main motivation behind *Lazy Decomposition*.

In *lazy decomposition* algorithm, we generate a new *component* only when the safety problem is not answerable using the current partial decomposition. In next section we explain the intuition behind the approach with the help of an example. Later we give a formal pseudocode of the algorithm and explain it briefly. In the following discussion, we use the term *Isolated Set of states* quite often. We have defined it as follows:

Definition 5.1 (Isolated Set of states): *A set of states S of a Kripke structure M is called as isolated set, if there exists no state outside of the set S, that can reach a state inside S via the transition relation of M.*

Check for isolated set of states can be performed using Algorithm 5.1.1.

Algorithm 5.1.1: ISOLATE$(\mathbf{f}(\mathbf{x},\mathbf{i}), S(\mathbf{x}))$

$/* \ Checks\ If\ a\ Set\ of\ states\ is\ an\ Isolate\ set\ on\ states\ */$
$S(\mathbf{x})$: *some set of states* ;
$\mathbf{f}(\mathbf{x},\mathbf{i})$: *Transition Function Vector of a FSM* M;
if SAT$(\overline{S(\mathbf{x})} \wedge S(\mathbf{f}(\mathbf{x},\mathbf{i}))) = true$
then return $false$;
else return $true$;

5.2 Example

Let us consider the finite state machine shown in Figure 5.1. Let the set of state variables is $\mathbf{x} = \{x_1, x_2\}$ and corresponding next state version is $\mathbf{x}' = \{x_1', x_2'\}$. i_1 and i_2 are the input variables. Next state function of this finite state machine will be:

$$x_1' = x_1 + i_1.x_2$$
$$x_2' = x_1$$

Here set of initial states is $Init(\mathbf{x}) = \overline{x_1} \cdot \overline{x_2}$ (00) and set of bad states is $Bad(\mathbf{x}) = x_1 \cdot x_2$ (11).

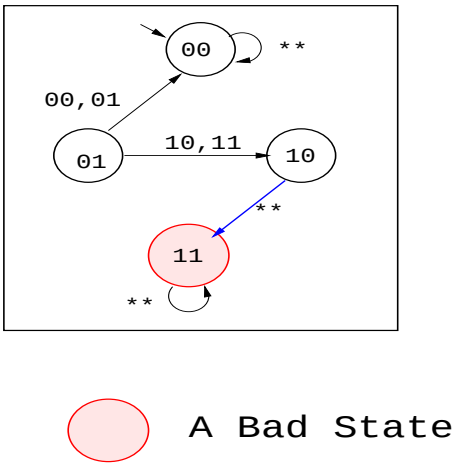

Figure 5.1: Original State Transition Graph

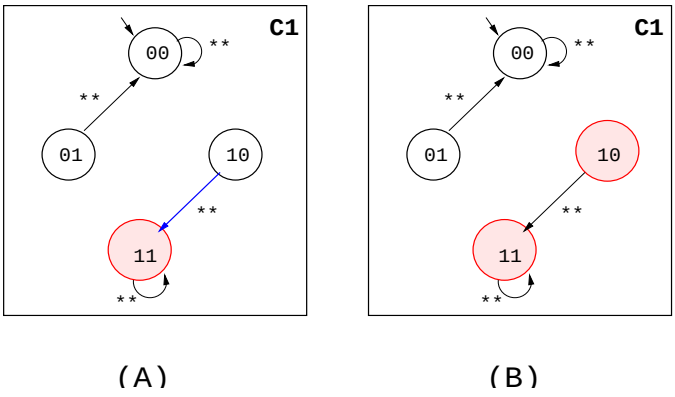

Figure 5.2: First Component's Fixpoint Iterations

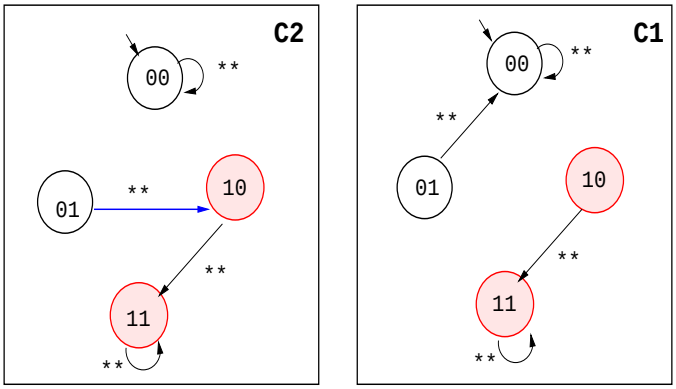

Figure 5.3: Second Component decomposed

- In this example, set of initial states $Init(\mathbf{x})$ (00) is not intersecting with Bad states $Bad(\mathbf{x})$ (11).
- Here set of bad states $Bad(\mathbf{x})$ (11) is also not an *isolated set* as there is a transition from the state (10) to the state (11). It can be checked using Algorithm 5.1.1.
- Let us initialize the partial decomposition as null set, i.e. $D_p = \{\}$.
- Since no edge of the Kripke structure M is covered by any component of the partial decomposition D_p, D_p is not a complete decomposition. Now we generate a component of the Kripke structure using *counter example focused decomposition*.
- Component C_1 is generated (Figure 5.2.A) using the Algorithm 4.3.1. Now $D_p = \{C_1\}$.
- After performing backward reachability over component C_1, fixed point is detected (Figure 5.2.B). At this point set of backward reached states is (11, 10) or $F(\mathbf{x}) = x_1$. This set is also not an isolated set of states as there is a transition from the state (01) to state (10). It can be checked using Algorithm 5.1.1.
- It is clear from the Figure 5.1 and Figure 5.2 that transition from state (01) to state (10) is not covered by the component C_1. Since all the edges of M are not covered in component C_1, D_p is not a complete decomposition. Now we have to generate another component.
- Component C_2 is generated using the decomposition Algorithm 4.3.1. Now new partial decomposition would be $D_p = \{C_1, C_2\}$ (Figure 5.3).
- Using MBM traversal of partial decomposition D_p, we traverse C_2 first. After reaching fixed point in C_2, state (01) will also get included in set of Failed states. Now we consider component C_1 for traversal. In this component we also reach fixed point in first iteration only. After reaching

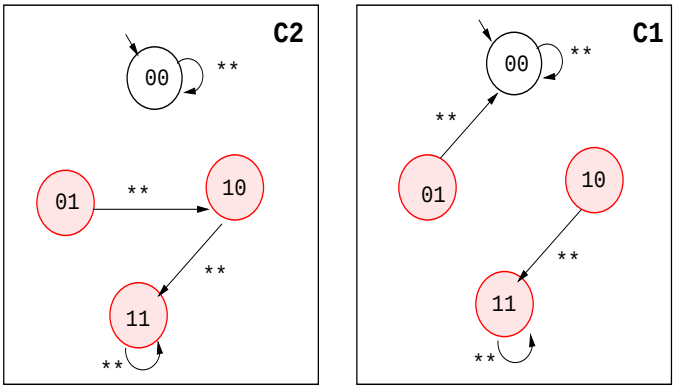

Figure 5.4: Fixpoint in both components

fixed point in both components (Figure 5.4), we check whether set of failed states constitute an isolated set of states. In this case it is true. Now the algorithm gets terminated by declaring the system safe.

The algorithm for *machine by machine* traversal (MBM) of the partial decomposition is given in Algorithm 5.2.1. The pseudocode for checking the safety property of the Kripke structure using lazy decomposition is given in Algorithm 5.2.2. In the next section we briefly explain the steps of the *Lazy decomposition algorithm.*

Algorithm 5.2.1: TRAVERSEPARTIALDECOMPOSITION(D_p, **var**$Fail(\mathbf{x})$)

```
/* MBM traversal of Partial Decomposition D_p */
Fail(x) : set of backward reached states/* passed by reference */
global Init(x) : set of initial states
D_p : partial decomposition containing n components of M.
f_i^c(x) : transition function vector of i^th component .
local fixpoint : Boolean
local traverse : Boolean
fixpoint ← false
while fixpoint = false
  do { fixpoint ← true
       for i ← 1 to n
         do                    /* For each component do */
       { traverse ← true
         while traverse = true
           do { Fail_old(x) ← Fail(x)
                Fail(x) ← Fail(f_i^c(x))
                if SAT(Init(x) ∧ Fail(x)) = true
                  then return false
                if SAT(Fail(x) ∧ ¬Fail_old(x)) = false
                  then traverse ← false
                  else { Fail ← Fail_old ∧ Fail
                         fixpoint ← false
```

Algorithm 5.2.2: CHECKSAFETYPROPERTY($\mathbf{f}(\mathbf{x},\mathbf{i})$, $Init(\mathbf{x})$, $Bad(\mathbf{x})$)

comment: Checking the Safety Property Using Lazy Decomposition

$Init(\mathbf{x})$: *set of initial states*
$Bad(\mathbf{x})$: *set of bad states*
$\mathbf{f}(\mathbf{x},\mathbf{i})$: *Transition Function Vector of the FSM* M

local $Fail(\mathbf{x})$: *set of backward reached states*
local D_p : *Partial Decomposition of* M
local $E^D(\mathbf{x},\mathbf{i})$: *Uncovered Edge Function*
local $P^D(\mathbf{x},\mathbf{i})$: *Preferred Edge Function*

if SAT($Init(\mathbf{x}) \wedge Bad(\mathbf{x})) = true$
  then return *System* M *is not safe* /* *step* − 1 */

if ISOLATE($\mathbf{f}(\mathbf{x},\mathbf{i}), Bad(\mathbf{x})) = true$
  then return *System* M *is safe* /* *step* − 2 */

$D_p \leftarrow \{\}$ /* *step* − 3 */
$Fail(\mathbf{x}) \leftarrow Bad(\mathbf{x})$
$E^D(\mathbf{x},\mathbf{i}) \leftarrow 1$
$P^D(\mathbf{x},\mathbf{i}) \leftarrow E^D(\mathbf{x},\mathbf{i}) \wedge (\overline{Fail(\mathbf{x})} \wedge Fail(\mathbf{f}(\mathbf{x},\mathbf{i})))$

while *true*
do
  local $\mathcal{C}$: *component*
  local *result* : *Boolean*
  local *covered*$(\mathbf{x},\mathbf{i})$: *set of covered edges by a component*

  $\mathcal{C} \leftarrow$ GENERATENEWCOMPONENT_2$(E^D(\mathbf{x},\mathbf{i}), P^D(\mathbf{x},\mathbf{i}), \mathbf{f}(\mathbf{x},\mathbf{i}))$ /* *step* − 5 */
  $D_p \leftarrow D_p \bigcup \{\mathcal{C}\}$
  result $\leftarrow$ TRAVERSEPARTIALDECOMPOASITION$(D_p, Fail(\mathbf{x}))$ /* *step* − 6 */
  if *result* = *false*
    then return "*system is not safe*" /* *step* − 7 */

  $covered(\mathbf{x},\mathbf{i}) \leftarrow$ COMPUTECOVEREDEDGES$(\mathbf{f}(\mathbf{x},\mathbf{i}), \mathcal{C})$
  $covered(\mathbf{x},\mathbf{i}) \leftarrow covered(\mathbf{x},\mathbf{i}) \vee Fail(\mathbf{x},\mathbf{i})$
  $E^D(\mathbf{x},\mathbf{i}) \leftarrow E^D(\mathbf{x},\mathbf{i}) \wedge \overline{covered(\mathbf{x},\mathbf{i})}$
  $P^D(\mathbf{x},\mathbf{i}) \leftarrow E^D(\mathbf{x},\mathbf{i}) \wedge (Fail(\mathbf{f}(\mathbf{x},\mathbf{i})))$
  if SAT($P^D(\mathbf{x},\mathbf{i})) = false$
    then return "*System is safe*" /* *step* − 4 */

5.3 Algorithm

In this section we explain our approach for checking the safety property $prop(\mathbf{x})$ over a finite state machine $M(\mathbf{f}(\mathbf{x},\mathbf{i}), Init(\mathbf{x}))$, where $\mathbf{f}(\mathbf{x},\mathbf{i})$ represents the next state transition function vector of the finite state machine and $Init(\mathbf{x})$) characterize the initial set of states. We characterize the set of bad states $Bad(\mathbf{x}) = \overline{prop(\mathbf{x})}$. The pseudocode of the approach is given as Algorithm 5.2.2. The basic steps of the algorithm are explained below:

1. The algorithm start the *reachability analysis* by initializing the set of backward reached states(also called as failed states $Fail(\mathbf{x})$) to set of bad state $Bad(\mathbf{x})$. At this point in time it checks whether set of initial states $Init(\mathbf{x})$ intersects with set of failed states. If it is the case, then the search is terminated by declaring the system unsafe and producing the counter-example, otherwise it will perform step 2.

2. Now algorithm checks whether set of bad states $Bad(\mathbf{x})$, is an *isolated set of states*. If yes, then it halts by declaring the system safe, as there exists no state out of set of bad states that can make transition into $Bad(\mathbf{x})$. Otherwise it will continue with step 3.

3. Next, it initializes the partial decomposition D_p as a null set.

4. Here algorithm checks if the decomposition is a complete decomposition. This can be checked by looking at the satisfiability of the *preferred edge function* $P^D(\mathbf{x},\mathbf{i})$. If $P^D(\mathbf{x},\mathbf{i})$ is unsatisfiable then it means that current set of reached states is an isolated set of states. So this implies that the fixed point of the finite state machine M is reached; now algorithm terminates by declaring the system safe; otherwise it performs next step.

5. Now the partial decomposition D_p is modified by generating a new component of M using *counter-example focused decomposition* (see Algorithm 4.3.1) approach and adding it to partial decomposition.

6. Now algorithm traverse the components of partial decomposition D_p. Any approach, that is MBM or FBF, for the traversal of *partial decomposition* D_p can be used. According to Cho et al. [14], MBM traversal results in faster exploration of counterexample. We have implemented MBM algorithm for the traversal of *partial decomposition*. The pseudocode of this algorithm is given in Algorithm 5.2.1.

7. If traversal of partial decomposition finds a counterexample, the algorithm terminates by declaring the system unsafe. If a fixed point is reached using the *partial decomposition*, then it checks whether set of backward reached states constitutes an isolated set of states. If yes, then the algorithm terminates by declaring the finite state machine safe; otherwise we can not conclude that the system is safe, as a *partial decomposition* does not cover all the transitions of M and it may possible that any transition that is not covered in any of the components of the *partial decomposition* may

cause the system to fail. So in this case, algorithm tries to generate a new component and continues from step 4.

5.4 Forward Reachability

The Lazy decomposition algorithm introduced in this chapter deals with backward reachability. The same algorithm can be used to work with *forward reachability* also. To compute the states reachable in one step using forward reachability (equation 3.1) we need to do existential quantification over current state variables. This need of quantification over state variables can also be removed in the same way as we removed for input variables.

Chapter 6

Implementation and Results

The technique proposed in this thesis is implemented to work with BDDs and SAT solvers. In next section, we discuss some of the important features of our implementations. In section 6.2, we describe the experiments with these implementations and discuss the results.

6.1 Implementation

The proposed algorithm is implemented using NuSMV2.0, an open-source model checking framework developed by IRST. NuSMV2.0 provides extensive set of functions needed for both BDD and SAT based model checking. NuSMV uses the state of the art BDD (CuDD) package developed at Colorado University, and provides a general interface for linking with state-of-the-art SAT solvers.

NuSMV takes the *model* of finite state machine written in SMV [5] language as input. In our implementation, we have written a parser to parse the language of ISCAS89 benchmark (a restricted subset of verilog). We also modified the syntax to read *specifications* (*initial states* and *bad states*) from a separate file. We have implemented the *lazy decomposition* algorithm using both SAT and BDD based engines. Each implementation is discussed below:

6.1.1 SAT-based Implementation

Representation of the formulas: NuSMV2.0 supports *Reduced Boolean Circuits* (RBC) [19, 10] to represent the Boolean formulas for SAT based model checking. RBCs are non-canonical representation of the Boolean formulas. Non-canonicity has both advantages and disadvantages: Non-canonical data structures can be more succinct than canonical ones – sometimes exponentially more. On the other hand, determining satisfiability of non-canonical data structures is hard, whereas with canonical data structures (like BDDs) it is constant time operation.

The fundamental operations for symbolic model checking are (1) quantification, for computing the image of a set; (2) satisfiability checking, for fixpoint checking; and (3) Simplification, for space efficient representation. The state-of-the-art of our RBC-based implementation with respect to these operations is given below:

Quantification: In our algorithm this step is not required, as we are eliminating the need of quantification by using partial decomposition for model checking instead of the monolithic Kripke structure.

Satisfiability Checking: To check the satisfiability of a Boolean expression in RBC, we convert it to CNF before passing it to SAT solvers. RBCs represents a Boolean formula using only two binary connectives, that is conjunction ($\wedge$) and bi-implication ($\Leftrightarrow$). Conversion of the RBC (a DAG) to CNF can be done by introducing an auxiliary variable at each non-terminal. Thus:

$$\phi(x \wedge y) \text{ is converted to } (a_i \Leftrightarrow (x \wedge y)) \wedge \phi(a_i)$$

$$\phi(x \Leftrightarrow y) \text{ is converted to } (a_i \Leftrightarrow (x \Leftrightarrow y)) \wedge \phi(a_i)$$

where a_i is the new auxiliary variable for each non-terminal node of the RBC. For the intermediate subformulas, the following clauses are generated:

$$a_i \Leftrightarrow (x \wedge y)) \text{ produces} \{a_i, \bar{x}, \bar{y}\}, \{\bar{a}_i, x\}, \{\bar{a}_i, y\}$$

$$a_i \Leftrightarrow (x \Leftrightarrow y)) \text{ produces} \{a_i, x, y\}, \{a_i, \bar{x}, \bar{y}\}, \{\bar{a}_i, x, \bar{y}\}, \{\bar{a}_i, \bar{x}, y\}$$

We are currently using SIM sat solver that is based on DPL algorithm (DP*). In SIM, *splitting* is not performed on auxiliary variables.

Simplification: The goal of simplification is to avoid doing repetition of calculation . By drawing conclusions from a formula and simplify it accordingly, we save the unnecessary overhead for symbolic model checking. Unfortunately, RBC do not provide sophisticated algorithms for Boolean formula simplification.

Function composition is the most critical operation in our algorithm. In our algorithm, we essentially reduced the problem of existentially quantifying over propositional variables to function composition. *Function composition* operation in RBCs is very inefficient as they do not offer global simplification of the resulting RBC after *function composition*. This results in space blowup after each function composition operation.

Currently we are handling the problem in an ad-hoc way by converting the RBC to BDD. Since BDD is a canonical representation, the conversion simplifies the formula and then we convert it back to RBC. The complexity of the algorithms for this conversion is polynomial in the size of the graphs.

6.1.2 BDD-based Implementation

In BDD based implementation, we used CuDD package for BDD manipulation. For all three fundamental operation, BDD performs better than RBC. Since BDD is a canonical data structure, no simplification is required for BDD based representation and satisfiability can also be checked in constant time. However, the restriction imposed by canonicity can in some cases results in space blowup, making memory the bottleneck in the application of the BDD based algorithms. The size of the representation can be reduced to some extent by providing efficient variable ordering, but that requires manual intervention and reduces the degree of automation.

6.2 Experimental Results

We compared traditional monolithic Kripke structure based algorithms with our lazy decomposition scheme. In BDD based implementation, we compared it with backward reachability algorithm of original NuSMV. In the case of RBC based implementation comparison is done with state of the art SAT based model checking that uses *quantification-by-substitution* rule for quantifier elimination over state variables. In the discussion that follows, we refer to each traditional model checking implementation as *monolithic* implementation.
We used ISCAS89 benchmark suite is used for the experiments. We used eight set of initial state and bad state combinations with each circuit and average of these eight combinations is reported. These combinations are listed below. We will refer each combination as c_1, c_2, c_3 and c_4:

1. (c1) Initial states: All 0's and bad states: All 1's.
2. (c2) Initial states: All 1's and bad states: All 0's.
3. (c3) Initial states: All 0's and bad states: Alternating 1's and 0's.
4. (c4) Initial states: Alternating 1's and 0's and bad states: All 0's.

All the tests were carried out on a Linux i686 PC with two 997.533 MHz pentium-III processors with 2GB memory. An upper limit of 30 minutes was set for each experiment. Figure 6.2 and Figure 6.3 shows the results of the tests performed on six circuits from the ISCAS89 benchmark suite. The experimental results reported in the thesis correspond to circuits of significant complexity

In BDD based implementation (figure 6.2) column "CPU Time" shows the average of the CPU time (system time + user time) for all the combinations for which computation terminated within 30 minutes.The column "not terminated" lists all those combinations for which algorithm did not terminated within 30 minutes. Time for the computation of the transition function and transition relation is also included in the reported time. For lazy decomposition based implementation, *maximum* number of components reported in all the combinations. Average number of BDD nodes used is also reported with each result.

Circuits	No. of Input Variables	No. of State Variables
s510.v	19	6
s526.v	3	21
s820.v	18	6
s1488.v	8	6
s444.v	3	21
s420.v	18	16

Figure 6.1: ISCAS89 Benchmark suite

	BDD MONOLITHIC			BDD LAZY DECOMPOSITION			
Circuits	CPU Time(sec)	BDD Nodes	Not Terminated	CPU Time(sec)	BDD Nodes	Not Terminated	No. of Components
s510.v	2.655	365014		1.0375	199474		2
s526.v	49.9225	4295681		0.5225	49987		2
s820.v	0.300	18847		0.335	17278		5
s1488.v	0.2875	4316		0.2825	3666		3
s444.v	1.16	512126		0.2675	1382		0
s420.v	1.16	393512	c1, c2, c3	3.428	686224		2

Figure 6.2: Experimental results for BDD based implementation

	RBC MONOLITHIC				RBC LAZY DECOMPOSITION				
Circuits	CPU Time(sec)	RBC/ BDD Nodes	Not Terminated	No of Compose	CPU Time(sec)	RBC/ BDD Nodes	Not Terminated	No. of Components	No of Compose
s510.v	4.2050	1010 / 97826		560	16.993	30874 / 111754		2	608
s526.v	1.5775	636/ 4674		60	2.9025	1557/ 5217		3	160
s820.v	1.6125	906/ 9581		171	9.002	8051/ 22115		6	819
s1488.v	1.5775	636/ 4675		60	2.9025	1557/ 5217		2	160
s444.v	1.6225	229/ 872		6	1.5725	269/ 872		0	0
s420.v			c1. c2, c3				c1, c2, c3, c4		

Figure 6.3: Experimental results for RBC based implementation

In the current implementation, there are some problems with the accounting of the BDD nodes. We report the exact number of BDD nodes in the final version of the thesis.

For RBC based implementation, "CPU time" column lists the average time (user time + system time) taken by the algorithm for all the combinations for which computation terminated within 30 minutes. Column RBC/BDD nodes lists average RBC nodes taken by the algorithm and total BDD nodes required for the simplification of the generated RBCs. Here also number of RBC and BDD are erroneous. The figure presented can be treated as upper bound for the number of nodes. Column labeled "Not terminated" lists all those combinations for which particular algorithm did not terminate within 30 minutes. In addition, column "No of compose" shows total number of function composition operations performed by the algorithm. For lazy decomposition the maximum number of components generated is also reported.

6.3 Conclusions

6.3.1 BDD based Implementation

Experimental results (Figure 6.2) shows that our algorithm performed better than traditional model checking algorithm both in terms of BDD nodes and in terms of CPU time. One important thing to note is that number of components required to verify the property are also very less. This proves the efficiency of

our lazy decomposition based technique. However, there exists some examples (s820.v) for which the performance of our circuits degrades. This can happen when our choice of successive components does not lead us to the early detection of the counterexamples. Nevertheless, our approach guarantees that counterexample of length k will get detected in less than k components. Another interesting thing to note is that in some combinations we got the counterexample without decomposing the Kripke structure (0 components). This can happen when bad set of states constitutes an isolated set of states.

6.3.2 RBC based implementation

Experimental results of Figure 6.3 shows that lazy decomposition technique performed worse than traditional model checking in the terms of both memory and CPU time. The reason behind that becomes more clear after looking at number of function composition operations. In almost every case, number of composition operation with our technique is greater than that in monolithic model checking and as we discussed earlier, RBC is not very efficient for function composition. More number of composition operation implies more bigger RBCs and that implies more calls to RBC to BDD conversion. This in turns results in more CPU and memory resource consumption. This suggests that RBC is not a very good choice for implementing "lazy decomposition" algorithm unless good techniques for function composition and RBC simplification exists.

Chapter 7

Conclusion

It is a widely recognized fact that the complexity of the systems containing hardware and software components is growing. The technology race combined with small inexpensive microprocessors has made it such that our society is giving computer control to everything possible. From washing machine to medical equipments, from air-crafts to spacecrafts and from kids toys to nuclear weapon systems. Therefore computer scientists are faced with the problem of designing safety critical systems of large complexity.

Symbolic model checking is an approach to ensure the correctness of such systems. However state of the art is such that it is not possible to deal with complexity of all classes of the systems using a single approach. In this thesis we reviewed various *symbolic model checking* approaches that works well with certain class of problems. The approach proposed by us provides one more alternative which can handle certain systems with ease on which previous approaches takes more time.

We showed that quantification elimination from the Boolean expressions can be done by decomposing the Kripke structure into components. Experimental results demonstrate that the technique is practical and promising if the representation is efficient for function composition.

7.1 Future Work

In particular, there are several ways in which the current implementation might be improved. In the first place, our SAT implementation is highly inefficient, since it uses a data structure (RBC) which do not provide simplification of the Boolean formula. In addition the current algorithms for function composition are not very sophisticated. We believe that the performance of our SAT based implementation can be improved by using a suitable data structure.

Secondly, currently we are handling only safety properties of the systems. Extending this technique to do full CTL model checking would be worth considering.

Recently, there has been increased interest in using multiprocessor systems or workstation clusters to deal with state explosion problem. These systems often have a very large (distributed) main memory. Furthermore, the large computational power of such systems also helps in effectively reducing model checking time. Our decomposition based algorithms are inherently amenable to concurrent reachability analysis. Our work can also be extended to run in distributed fashion to take advantage of multiprocessor systems.

References

[1] S. B. Akers. Binary decision diagrams. In *IEEE Transaction on Computers*, June 1978.

[2] Edmund Clarke Armin Biere, Alessandro Cimatti and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS*, 1999.

[3] R. E. Bryant. Graph based algorithm for boolean function manipulation. In *IEEE Transaction on Computers*, pages C–35(8), 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. G. Hwang. Symbolic model checking 10exp20 states and beyond. In *Logic in Computer Science*, pages 428–439, 1990.

[5] E. M. Clarke, J. R. Burch, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, July 1993.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. In *ACM Transactions on Programming languages and systems*, pages 8(2:244–263), 1986.

[7] E.M. Clarke and E.A Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. Logic of Programs: Workshop*, page LNCS 131, 1981.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of ACM*, volume 7, page 201.

[9] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. Cambridge, MA, 1999.

[10] Niklas Een. Symbolic reachability analysis based on sat-solvers. In *Master's Thesis, Department of Computer Science, Uppsala University*, 1999.

[11] J. L. Lions et al. ARIANE 5; Flight 501 Failure. 1996. Copy.

[12] M.R. Gary and D.S. Johnson. Computers and intractability. In *A Guide to the Theory of NP-Completeness*, San Francisco, 1979. Freeman.

[13] Enrico Giunchiglia and Roberto Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. *Lecture Notes in Computer Science*, 1792:84–88, 2000.

[14] Enrico Macii Massimo Poncino Fabio Somenzi Hyunwoo Cho, Gary D. Hachtel. A state space decomposition algorithm for approximate fsm traversal. In *The European Conference on Design Automation*, 1994.

[15] C. Y. Lee. Representations of the switching circuits by binary decision diagrams. In *Bell Systems Technical Journal*, July 1959.

[16] G. Logemann M. Davis and D. Loveland. A machine program for theorem proving. In *Communications of the ACM*, volume 5, page 394.

[17] Satnam Singh Mary Sheeran and G Stalmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, 2000.

[18] Kenneth L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. Computer Aided Verification (CAV)*, Lecture Notes in Computer Science.

[19] P. Bjesse P. A. Abdulla and N. Een. Symbolic reachability analysis based on sat-solver. In *TACAS*, 2000.

[20] J-P. Quielle and J. Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. of Fifth ISP*, 1981.

[21] Arvind Soni. *Approximate Symbolic Model Checking.*, Bachelor's Thesis, Department of Computer Science, IITB, Mumbai, May 2002.

[22] G. Stalmarck. Stalmarck's method with extensions to quantified boolean formulas. In *Proc. 11th International Computer Aided Verification Conference*, pages 23–43, 1999.

[23] Filkorn T. Functional extension of symbolic model checking. In *Proc. Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 225–232.

[24] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, U.S.A., July 2000. Springer-Verlag.