BDD MINIMIZATION USING DON'T CARES FOR FORMAL VERIFICATION AND LOGIC SYNTHESIS

by

Youpyo Hong

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

August 1998

Acknowledgments

I would like to wholeheartedly thank Professor Peter A. Beerel, my advisor, for everything he made possible to me throughout this work. I am grateful to him not only for his technical guidance but also for his unyielding encouragement and advice. I am also grateful to Professor Massoud Pedram for his inspiring courses that motivated me to become a CAD engineer. I would like to thank Professor Doug Ieradi for his encouragement and sincere care. I also would like to thank Professor Richard Nottenburg and Professor Sandeep Gupta for their insightful comments and guidance as my qualifying committee members. I also thank Dr. Barton Sano for teaching me so many basic and essential skills when I started my graduate study.

I would like to gratefully acknowledge that Dr. Jerry Burch and Dr. Kenneth McMillan significantly contributed to the work presented in Chapter 2 and Dr. Ellen Sentovich and Dr. Luciano Lavagno tremendously contributed to the work presented in Chapter 4, respectively.

I would like to thank Kangwoo Lee, Joonho Ha and Yongseon Koh for their constant help and advice. I also thank all Yonsei alumni members who made my time at USC enjoyable and memorable. It is a pleasure to offer my gratitude to USC Asynchronous CAD group members, Wei-chun Chou, Aiguo Xie, Vida Vakilatojar and Peter Yeh for the exciting discussions. I also would like to thank Mary Zittercob and Tim Boston for their administrative help.

I would like to express my sincere gratitude to my mother whose love and confidence on me made all things possible and to my wife Jayoung who always supported and endured me with her love. I dedicate the dissertation to them.

Contents

List of Tables

List of Figures

Abstract

Computer-aided design (CAD) tools have been an essential means to cope with the enormous complexity of designing, verifying, and testing very large scale integrated (VLSI) circuits. Efficient representation and manipulation of Boolean logic functions is crucial in most CAD applications and binary decision diagrams (BDDs) have proven to be an extremely useful representation because they are canonical and typically very compact.

For BDD-based tools, the size of the BDD representations often determine their run-time, the problem size that they can handle, and/or the quality of the circuits they synthesize. BDD size reduction, therefore, has been an intensive area of research. One of the approaches to reducing BDD size is to utilize the flexibility of the function to be represented from *don't care* conditions such as unused state codes or impossible input patterns.

It is the goal of this dissertation to explore techniques and applications of BDD minimization using don't cares. This dissertation addresses two basic questions about BDD minimization using don't cares: 1) how to minimize BDDs using don't cares and 2) how/where to use DC-based BDD minimization.

First, we develop new don't care-based BDD minimization heuristics which are significantly more powerful than traditional heuristics. Then we demonstrate that don't care-based BDD minimization can significantly improve the performance of symbolic reachability analysis tools. Finally, we describe how to derive don't cares and utilize the don't cares for BDD-based software synthesis.

Chapter 1

Introduction

Computer-aided design (CAD) tools have been an essential means to cope with the enormous complexity of automatically synthesizing and verifying integrated circuits (ICs).

Synthesis refers to the process of automatically transforming a given specification to an implementation. Depending on the levels of implementation, synthesis tasks can be classified into high-level synthesis, logic synthesis, and geometrical-level synthesis [27]. High-level synthesis is a process of producing a macroscopic, e.g. register-transfer level or block-level, structure from a higher level specification, e.g. high-level description languages (HDL). Logic synthesis generates a gate-level structure from a logical representation. Geometrical-level synthesis produces a physical structure, i.e. a layout.

Once the synthesis task is finished at each level, designers verify (or validate) the design to ensure that the design is correct. The correctness can be defined in functionality or timing. There are many techniques to functionally verify a circuit such as simulation, emulation and formal verification [42]. Simulation computes the response of a circuit to input test vectors. Although simulation is the most common form of functional verification, computing the response for all possible input vectors is often computationally intractable for complex circuits. Emulation obtains the response of a circuit from a hardware prototype implemented on reprogrammable devices, e.g. FPGAs (field programmable gate arrays). Emulation provides a very fast and flexible simulation environment. However, the hardware prototype is typically expensive and the setup-time can be large. *Formal verification* is the process of mathematically establishing that an implementation "satisfies" a specification [36, 53]. There are mainly two types of formal verification; design verification (or property checking) and implementation verification (or equivalence checking). Design verification checks if the design satisfies a given property, e.g, whether an asynchronous circuit is hazard-free. Implementation verification tests if the specification and the implementation have exactly the same behavior. Ideally, the synthesis process is designed to be error-free and the specification and implementation should be equivalent. In practice, however, synthesis tools may have bugs and the entire design flow is not fully automated, e.g. designers often participate in some synthesis process to further improve the quality of the designs or simply to fill in the gaps in the design flow.

The representation of logic functions is essential in most synthesis and verification tasks. Binary decision diagrams (BDDs), one of the representations for Boolean logic functions, are widely used in many CAD tools specially where efficient representation and manipulation of Boolean functions are extremely important. For BDD-based tools, the BDD size can determine their run-time, the problem size that they can handle, and/or the quality of the circuits they synthesize. Consequently, BDD size reduction has been an intensive area of research. One of the approaches to reducing BDD size is to utilize the flexibility of the function to represent, i.e. *don't care* conditions such as unused state codes or impossible input patterns.

This dissertation addresses the *efficient* use of BDDs in logic synthesis and formal verification. The key idea is to exploit don't care information to maximize the efficiency and/or capability of BDD-based logic synthesis and formal verification tools.

We now briefly summarize some relevant terminology and present the overview of our approach.

1.1 Terminology

1.1.1 Boolean Functions

A *completely specified* Boolean function is a mapping between Boolean spaces. An n-input single-output Boolean function f is a mapping $f : \mathrm{B}^n \rightarrow \mathrm{B}$. An *incompletely specified* Boolean function is defined over a subset of B^n.

A Boolean function f can also be described as the set of all input points, i.e. minterms, for which the function f evaluates to 1; this set is referred as the "on-set" of f. Similarly, the "off-set" of f is the set of all input points for which the function f evaluates to 0. If we do not care if the function evaluates to 0 or 1 for a set of input points, the function is said incompletely specified, and such input points are called the "don't care-set". Therefore, the domain of any single-output Boolean function ff can be partitioned into three subsets, ff_{on} (on-set), ff_{off} (off-set), and ff_{dc} (don't care-set). A completely specified function has ff_{dc} empty, while an incompletely specified function has a non-empty don't care set.

Any two of these sets uniquely describes an incompletely specified function.

An incompletely specified function ff can also be represented by a pair of completely specified functions [f, c] for which $f_{on} \supseteq ff_{on}$, $f_{off} \supseteq ff_{off}$, and $c = ff_{on} \cup ff_{off}$. For a given incompletely specified function ff, there are many such functions f, each referred to as a *cover* of ff [16, 73], representing different partitions of ff_{dc} into f_{on} and f_{off}.

1.1.2 Representations of Boolean Functions

There are various ways of representing Boolean functions and widely accepted representations can be summarized as follows [27].

1.1.2.1 Tabular Representations

Tabular representations basically use two dimensional tables. A truth table, the simplest example of a tabular representation, is a complete listing of function values for all input values. Because the size of a truth table is exponential in the number of inputs, various derivatives of a truth table (e.g. a multiple-output implications based table [27]) have been developed.

1.1.2.2 Expressions

Boolean functions can be represented by *expressions* consisting of literals and Boolean operators such as Boolean sum and product. For example, a sum-of-product form is a representation of literals combined by Boolean product operators and then by Boolean sum operators. As another example, a factored form is either a literal, a sum of factored forms, or a product of factored forms [27].

1.1.2.3 Binary Decision Diagrams (BDDs)

BDDs are graphical representation of Boolean functions [10]. A BDD can have two types of nodes; leaf nodes and non-leaf nodes. The leaf nodes are either 0 or 1, representing the Boolean functions 0 and 1, respectively. Each non-leaf node u has two outgoing edges; a *then-edge* and an *else-edge.* Each edge is connected to a *child* node of u and u is the *parent* of the child nodes. The two child nodes are *siblings* of each other. Each non-leaf node F is associated with a Boolean variable x. The child of F reached via the then-edge is called the *positive cofactor* of F with respect to x and is denoted by F_x; the other child is called the *negative cofactor* of F with respect to x and is denoted by $F_{\bar{x}}$. Each node F represents a Boolean function f. The size of F, denoted $|F|$, is the number of nodes in the BDD rooted at F.

An ordered binary decision diagram (OBDD) is a BDD with the constraint that the input variables are ordered to determine the relative positions of BDD nodes. A reduced ordered binary decision diagram (ROBDD) is an OBDD where there is only one node representing a distinct function. Figure 1.1 illustrates an example of the ROBDD representing the function $f = x \cdot y + \bar{x} \cdot \bar{y}$ with the variable order (x < y).

Bryant [10] prove that ROBDDs are canonical, i.e. the ROBDD representation of a completely specified function is unique under a fixed variable ordering. ROBDDs have practically proven most useful because of their canonicity and compactness. Therefore, we focus on ROBDDs and we refer to ROBDDs simply as BDDs in the dissertation.

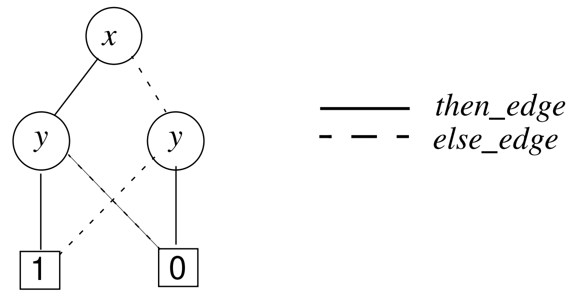


Figure 1.1: A ROBDD representing $f = x \cdot y + \overline{x} \cdot \overline{y}$ (with variable order x < y)

1.1.3 Finite State Machines

A synchronous circuit can be modeled by a *finite state machine* (FSM). A FSM is a 6-tuple $<I, O, S, \delta, \lambda, S^0>$, where I is the input space, O is the output space, S is the state space, δ is the next state function ($\delta : I \times S \rightarrow S$), λ is the output function ($\lambda : I \times S \rightarrow O$), and S^0 is the set of initial states.

1.2 Overview of the Thesis

It is the goal of this dissertation to explore techniques and applications of BDD minimization using don't cares. This proposal addresses two basic questions about BDD minimization using don't cares.

- How to minimize BDDs using don't cares?
- How/where to use DC-based BDD minimization?

First, we develop new don't care-based BDD minimization heuristics which are significantly more powerful than existing heuristics. Then, we present techniques to use don't care-based BDD minimization heuristics to improve the performance and/or capability of existing CAD tools.

1.2.1 BDD Minimization Using Don't Cares

We present new don't care-based BDD minimization heuristics. We identify potential problems of conventional BDD minimization heuristics and propose mechanisms to prevent them in order to achieve better BDD minimization quality. Experimental results show that new heuristics yield significantly smaller BDDs compared with existing heuristics yet still require manageable run-times.

1.2.2 Symbolic Reachability Analysis Using Don't Cares

Reachability analysis of finite state machines is the basic step in many formal verification techniques and the results, i.e. reachable states, are also useful for synthesis applications. We present novel techniques to improve both approximate and exact reachability analysis using don't cares. We first propose an iterative approximate reachability analysis technique in which don't care sets derived from previous iterations are used to improve the approximation quality in subsequent iterations. Then, we propose new techniques to use the final approximation results to enhance the capability and efficiency of *exact* reachability analysis. Experimental results show that our techniques can significantly improve existing reachability analysis techniques.

1.2.3 Don't Care Based BDD Minimization for Embedded Software

Hardware-software co-design is gaining much attention recently since the increasing complexity of systems requires an integral framework to design hardware and software together to guarantee the correctness of the entire system. Embedded systems is defined as a collection of programmable parts surrounded by ASICs and other standard components, and they typically have extremely tight real-time and code/data size constraints, which make even expensive optimizations desirable. We explore the use of don't cares in software synthesis for embedded systems. We propose applying BDD minimization techniques in the presence of don't care sets to synthesize codes for FSMs from the BDD-based representations of the FSM transition functions. The don't care sets are derived from local analysis as well as from external information, i.e. impossible input patterns. We show experimental results, discuss their implications, and propose directions for future work.

Chapter 2

BDD Minimization Using Don't Cares

This chapter presents new heuristics to minimize the size of the BDDs representing incompletely specified functions by intelligently assigning don't cares to binary values. These heuristics are particularly useful for synthesis application where the structure of the hardware/software is derived from the BDD representation of the function to implement because the minimization quality is more critical than the minimization speed in these applications. Experimental results show that the new heuristics yield significantly smaller BDDs compared with traditional heuristics at the expense of larger but manageable run-times.

2.1 Introduction

In logic synthesis, BDDs have been widely used as an excellent representation of the function to be manipulated. In addition, there are many synthesis tools in which circuits are directly derived from their BDD functional representation. For example, hazard-free multi-level logic based on multiplexor-based circuits can be directly derived from BDDs [49, 80]. In addition, T. Karoubalis et al. show that Differential Cascode Voltage Switch (DCVS) logic circuits, which have many potential advantages such as performance and high layout density, can be optimally synthesized from BDDs due the canonicity of BDDs [45]. Moreover, Lavagno et al. [46] present a BDD-based Timed Shannon Circuits synthesis tool in which reducing the BDD size can lead to lower power consumption.

In the technology mapping area, multiplexor-based FPGA mapping can be directly performed on the BDDs that represent the logic functions [54, 17]. Chang et al. [17] apply DC-based BDD minimization in their FPGA mapping framework to reduce the size of the BDDs representing subject graphs, yielding more area-efficient circuits.

The application of BDDs can also be found in software synthesis area. Chiodo et al. [18] use a BDD as an intermediate representation to generate a software program because of the close connection between the BDD representation of a function and the structure of the software program they synthesize. The size of the software is determined by the BDD size, which means that the size of the BDD is critical to reduce software size [3, 41].

A traditional approach to reducing BDD size is to find a good ordering of the variables of the function to represent because BDD size depends greatly on the ordering of the variables [10]. Many heuristics have been developed to find a good BDD variable ordering which is fixed once it is determined [52, 44, 32, 2]. Although these *static* variable orderings have been quite successful for some applications, there are still many applications where no single variable ordering is good enough. Therefore, techniques to dynamically reorder the variables (*e.g.* [65]) are used in many applications where the size of BDD is critical to complete complex tasks or to lead to small circuits [49, 80, 45, 54, 17, 46, 62, 7]. A drawback of dynamic variable ordering is that it has high time-complexity.

Another approach to reducing BDD size is to utilize *don't cares* (DCs) which exist if the functions to represent are incompletely specified, *i.e.*, the functions are not defined for some input values. For such incompletely specified functions, many BDDs can be used to represent the function, each associated with different assignment of binary values to don't cares (DCs). Finding the assignment that leads to the smallest BDD is known to be NP-complete [68] and exact techniques [56, 30] are typically too computationally expensive. Therefore, heuristic algorithms have been developed to address this BDD minimization problem [17, 24, 73, 29]. These heuristics try to maximize the instances of *node sharing* or *sibling-substitution* [73] during the minimization process. BDD nodes become shared if the re-assignment of DCs makes their associated functions identical. Sibling-substitution is a special case of node sharing where a child of a BDD node u is replaced by the other child of u. Sibling-substitution leads to fewer nodes because a parent and its two children are replaced by the child when the two children are made identical.

Restrict and *constrain* (also known as generalized-cofactor) [24, 25, 76] are well known BDD minimization algorithms based on sibling-substitution and often achieve significant size reduction. Interestingly, however, these algorithms may yield a BDD far from being optimal and in fact larger than the original BDD.

Chang et al. [17] propose a heuristic that makes multiple sub-BDDs shared by assigning DCs to binary values while traversing the BDD from top to bottom level by level. The reduction potential of their method is large, but its high computational complexity prohibits its application to large BDDs.

Shiple et al. [73] propose a framework to relate sibling-substitution-based heuristics and Chang's heuristic, and explored their variants. Their experimental results suggest that sibling-substitution-based heuristics, specifically restrict and its variants typically outperform others in terms of both run-time and resulting BDD size.

Recently, Drechsler et al. [29] propose evolutionary algorithm-based BDD minimization heuristics that can handle multiple-output functions. However, the minimization quality in this approach is very sensitive to some user-defined parameters [30].

Note that all the existing heuristics may produce larger results and a common way to avoid using a larger BDD is to compare the original BDD with the 'minimized' BDD and use the smaller one. We refer to this approach as thresholding. The potential for the size increase, however, suggests that these methods may not produce BDDs as small as those produced by algorithms that inherently guarantee that no sub-BDD becomes larger.

We describe *safe* BDD minimization heuristics, i.e., they guarantee the resulting BDD is not larger than the original BDD inherently under the assumption that the variable ordering is fixed. These algorithms are based on sibling-substitution because sibling-substitution itself is very powerful and efficient. The key idea of safe minimization heuristics is to perform sibling-substitution only on the nodes that we can guarantee will not cause an overall increase in BDD size. These techniques can lead to better minimization results by preventing sibling-substitutions that can cause overall size growth while allowing sibling-substitutions elsewhere. Our heuristics can also be applied to minimize multiple BDDs safely.

Our experimentations on ISCAS and MCNC benchmarks using various types of DCs demonstrate that our new heuristics outperform existing sibling-substitution based heuristics significantly in minimization quality. Another strength of our heuristics is their low computational complexity which allows them to be able to minimize large BDDs that cannot be handled by competitive existing heuristics.

The organization of the chapter is as follows. We present our two new heuristics in Section 2.3. We report our experimental results in Section 2.4 and present conclusions in Section 2.5.

2.2 BDD Minimization Using Don't Cares

An incompletely specified function can be represented by a BDD pair $[F, C]$ describing a pair of completely specified functions $[f, c]$, where f is a cover of the incompletely specified function and c denotes the care-function. Among all covers of ff, there must be at least one cover f' whose BDD, F', is smallest in size. Unfortunately, finding a smallest F' is NP-complete [68], so we consider heuristic approaches. Given $[F, C]$, finding an F' that is hopefully close to minimal in size is called *BDD minimization using don't cares*. We call F the original BDD and F' the *minimized* BDD.

2.3 Safe BDD Minimization Based on Sibling Substitution

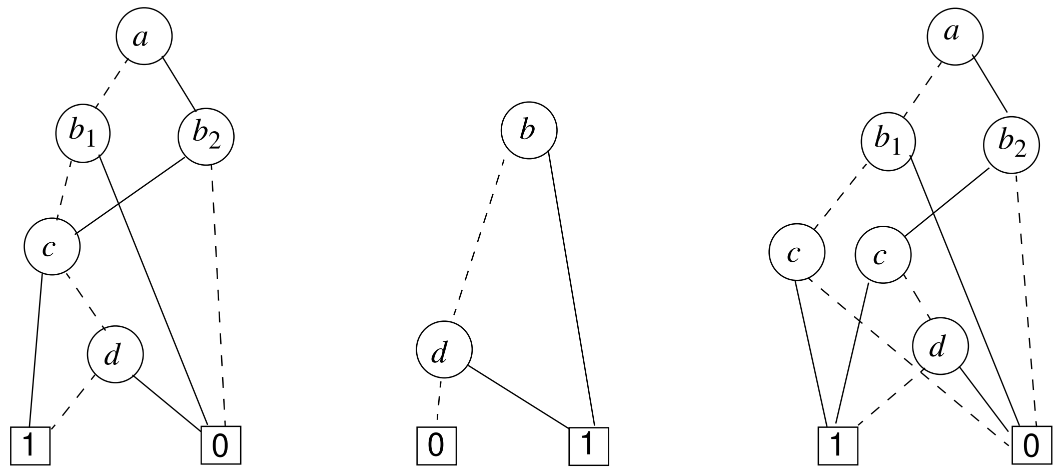
The main differences among sibling-substitution based BDD minimization techniques lie in the criteria on which they perform sibling-substitution. The simplest criterion is based on the observation that a function ff can be covered by any function if ff corresponds to a DC function. We refer to this condition as *DC-substitutability* which is formally defined as follows.

Definition 1. (DC-substitutability) ff is DC-substitutable if $ff_{on} \cup ff_{off} = \emptyset$.

The most widely used heuristic, restrict, is based on DC-substitutability. Restrict recursively traverses *F* and *C* concurrently in depth-first order. In each recursive call, a new pair of *F* and *C* nodes is visited and restrict checks if either of the positive or negative cofactor of the *F* node is DC-substitutable with respect to their corresponding *C* nodes. Whenever a cofactor is found DC-substitutable, sibling-substitution is performed to the parent (removing the parent and one child), and the result is built recursively by continuing traversal only to the cofactors that are not substituted.

Note that restrict, while always reducing the size of the target sub-BDD, can increase the size of a BDD that contains the target sub-BDD, as illustrated in Figure 2.1[1]. Consider the node *c* in Figure 2.1 (a) which can be reached from the root by two different *paths* and has two different associated care sub-sets, represented by node *d* and leaf-1 respectively in *C* depicted in Figure 2.1 (b). The different pairs of node *c* and associated care sub-sets will be analyzed in different recursive calls of restrict. Consider first the

1. In this dissertation, we label each node in a BDD by the variable that the node is associated with. When more than two nodes are associated with the same variable, numeric subscripts are used to distinguish the nodes. The then-edge (else-edge) is represented by a solid line (broken line).



(a) F (b) C (c) F′

Figure 2.1: Restrict example

recursive call in which the negative cofactor of node *c* (with respect to the variable *c*) in *F*, node *d*, is analyzed with the corresponding care node *d* in *C*. Because the care node is not leaf-0, the algorithm recurs to node *d* and analyzes its positive and negative cofactors. Since *d*'s negative cofactor corresponds to a DC (leaf-0 in *C*), sibling-substitution is applied to *d* (replacing *d* with its positive cofactor leaf-0). Notice that this results in a smaller sub-BDD rooted at *c*. However, when node *c* is analyzed with the care node leaf-1 in a subsequent recursive call, the sub-BDD rooted at *c* (including node *d*) cannot be reduced at all because the entire sub-BDD corresponds to a care sub-function. Consequently, node *c* becomes unshared or *split*, and this *node-splitting* leads to the overall size increase illustrated in Figure 2.1 (c).

We can formally describe node-splitting using a relation R from edges in F to nodes in C defined as follows.

Definition 2. Given two BDDs F and G, the then-edge (else-edge) e of a node u in F is *related* to a node v in C iff there exist cubes[2] p and $p' = p \cup \{x\}$ ($p' = p \cup \{\bar{x}\}$) such that u denotes f_p, u_x denotes $f_{p'}$, and v denotes $c_{p'}$, where x is the variable associated with u and f_p is the cofactor of f with respect to p. We denote a set of nodes in C that an edge e in F is related to by $R(e)$.

Intuitively, the set $R(e)$ describes the set of care nodes that are visited during the those recursive calls of restrict obtained by recurring through the edge e. More precisely, the set $R(e)$ consists of those care nodes analyzed in recursive calls 1) in which the sink of e is analyzed and 2) that are called by recursive calls in which the source of e is analyzed. For example, consider F and C shown in Figure 2.2. For the cubes $\bar{a}$ and $\overline{ab}$, $F_{\bar{a}}$ is denoted by the node b_1 in F, $F_{\overline{ab}}$ by c in F, and $C_{\overline{ab}}$ by d in C. According to Definition 2, the else-edge of b_1 in F is related to d in C. Intuitively, this means that in the only recursive call through the else-edge of b_1, the corresponding care node is d.

As another example, consider the cubes $\overline{ab}$ and ab. Both $F_{\overline{ab}}$ and F_{ab} are represented by c in F. $F_{\overline{ab}c}$ and F_{abc} are represented by leaf-1 in F, and $C_{\overline{ab}c}$ and C_{abc} are represented by d and leaf-1, respectively, in C. Consequently, the then-edge of c in F is related to both d and leaf-1 in C. This makes sense because the then-edge of c is recurred through twice, once with the corresponding care node d and a second time with the care node 1.

---

2. A cube is a set of literals and represents the function obtained by their product [8].

As another example, consider the cubes $\overline{ab}$ and ab. Both $F_{\overline{ab}}$ and F_{ab} are represented by c in F. $F_{\overline{ab}c}$ and F_{abc} are represented by leaf-1 in F, and $C_{\overline{ab}c}$ and C_{abc} are represented by d and leaf-1, respectively, in C. Consequently, the then-edge of c in F is related to both d and leaf-1 in C. This makes sense because the then-edge of c is recurred through twice, once with the corresponding care node d and a second time with the care node 1.

The related nodes of all edges of F are shown in Figure 2.2.

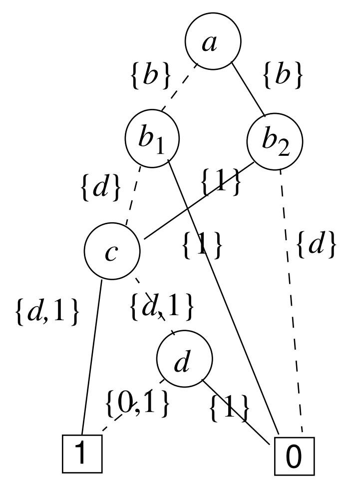


Figure 2.2: An example of the relation R (0 and 1 means leaf-0 and leaf-1, respectively)

Notice that restrict applies sibling-substitution to u if the related nodes of its outgoing edge e includes leaf-0, i.e., leaf-0 $\in R(e)$. Node-splitting may occur if $R(e)$ also includes a non-DC because the original node (or a modified version of it) is needed in the result in such case. Consequently, an originally shared node, such as u, can become unshared by the minimization process, leading to overall BDD size growth.

We develop safe minimization heuristics by performing only the sibling-substitution that we can guarantee will not cause an overall BDD size increase. In other words we further constrain the condition to perform sibling-substitution even if substitutability holds.

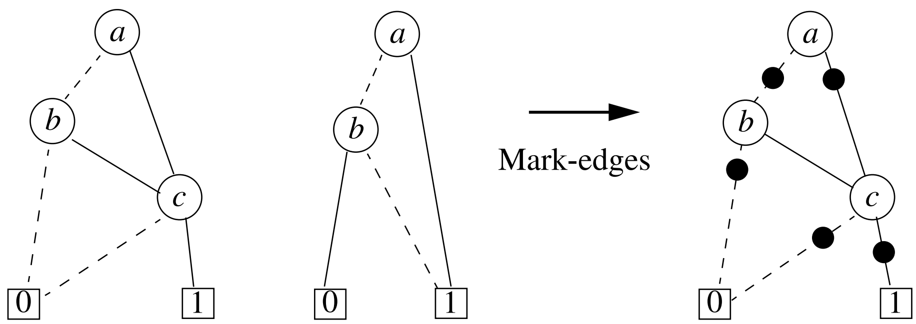
Our algorithms consist of two phases mainly. In the first phase, the original BDD is preprocessed to conservatively identify nodes for which applying sibling-substitution does not increase overall BDD size. In the second phase, sibling-substitution is selectively applied to the nodes identified in the first phase.

We present three algorithms. The first algorithm *basic compaction* performs a subset of the sibling-substitutions that we can conservatively guarantee do not lead to node-splitting. The next two algorithms *leaf-identifying compaction* and *essential-node-identifying compaction* allow special types of node-splitting. The last algorithm *generalized-substitutability-based compaction* uses a generalized sibling-substitution criterion to achieve further gain.

2.3.1 Basic Compaction

Basic compaction (*B-compaction*) is designed to conservatively avoid sibling-substitutions that may cause node-splitting. In particular, B-compaction applies sibling-substitution to a node only when an out-going edge e is related to the DC-leaf only, i.e., $R(e) = \{DC\}$. To do this, B-compaction *marks* edges that are related to a non-DC node in a preprocessing phase and then selectively performs sibling-substitution only on the source nodes of non-marked edges.

Consider an edge between nodes u and v in F that is related to multiple nodes in C. If any of these nodes is not leaf-0, we can conservatively assume that substituting v with its sibling may cause node-splitting (i.e., node v or a modified version of it is needed). Consequently, we mark an edge if it is related to anything other than leaf-0. For example, in Figure 2.3, the else-edge of c in F is related to both leaf-0 and leaf-1 in C and therefore it is marked. This preprocessing phase is called *mark-edges* and the pseudo-code is shown in Figure 2.4.



(a) F (b) C (c) Edge-marked F

Figure 2.3: An example of mark-edges

The second phase, we call *build-result*, creates minimized BDD solely based on the markings on edges in F. If an edge from a node v to any of its child nodes u is not marked, then v can be safely replaced by u's sibling via sibling-substitution. Otherwise, v is preserved and its children are recursively rebuilt. Figure 2.5 illustrates this *selective* sibling-substitution based rebuilding technique on an edge-marked BDD. Figure 2.5 illus-

```
void mark-edges (bdd f, bdd c) {
    if (c == 0) return;
    if (f == leaf) return;
    x = top variable(f, c);
    if (c_x != 0)
        if (f != f_x) f.then_mark = 1;
        mark-edges(f_x, c_x);
    if (c_x̄ != 0)
        if (f != f_x̄) f.else_mark  = 1;
        mark-edges(f_x̄, c_x̄);
}
```

Figure 2.4: Mark-edges pseudo-code

trates this selective sibling-substitution based rebuilding technique on an edge-marked BDD. Figure 2.6 shows the pseudo-code for the build-result routine.

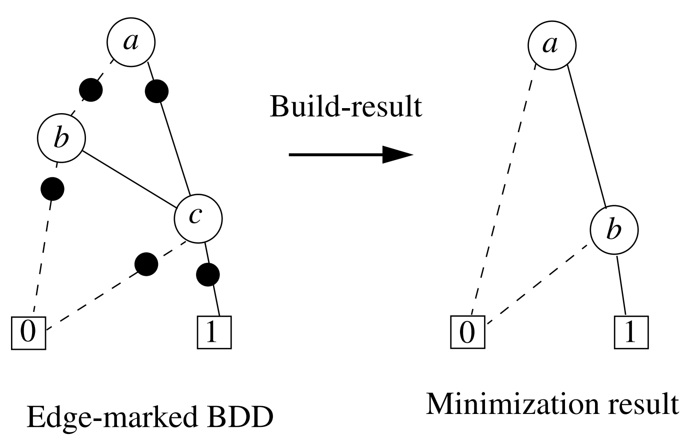


Figure 2.5: An example of build-result

```
bdd build-result(bdd f) {
    if (f == leaf) return(f);
    x = top variable(f);
    if (f.then_mark == 1 and f.else_mark == 0)
        return (build-result(f_x));
    else if (f.then_mark == 0 and f.else_mark == 1)
        return (build-result(f_x̄));
    else
        return (bdd-find(x, build-result(f_x), build-result(f_x̄)));
}
```

Figure 2.6: Build-result pseudo-code

Figure 2.7 presents the pseudo-code of B-compaction. The time complexity of *mark-edges* is $O(|F| \cdot |C|)$ because each pair of nodes from F and C is called only once by using an operation cache. Due to the application of a second operation cache, build-result processes each node only once, yielding a time complexity of $O(|F|)$. *Clear-edges* routine clears the edge-marking fields after building the result and has time complexity of $O(|F|)$. Consequently, the overall time complexity of B-compaction is $O(|F| \cdot |C|)$, the same complexity as restrict.

We show that B-compaction is *safe*. Recall that a BDD minimization using DCs is safe if the minimized BDD is guaranteed to be no larger than the original BDD, i.e. $|F| \geq |F'|$.

```
bdd B-compaction(bdd f, bdd c) {
    if (c == 0) return (0);
    mark-edges(f, c);
    result = build-result(f);
    clear-edges(f);
    return(result);
}
```

Figure 2.7: Basic compaction pseudocode

Theorem 1: B-compaction is safe.

Proof: The result of B-compaction is produced by build-result. Recall that build-result takes F as its only argument and that the nodes of F have two Boolean labels: a then-edge and an else-edge. For the purposes of this proof, it does not matter how these labels are set by mark-edges. Inspection of build-result shows it does not affect the status of any then-edge or else-edge. Thus, for a given sub-BDD G of F, a call to build-result with argument G always produces the same result. Let F' be the result of build-result applied to F. It is easy to show by induction on the depth of F that every sub-BDD of F' (including F' itself) is the result of calling build-result on some sub-BDD of F. Thus, since build-result always produces the same result when applied to a given sub-BDD of F, the number of sub-BDDs of F' is no larger than the number of sub-BDDs of F. Thus, the size of the F' is no larger than the size of the F. So, we can conclude that B-compaction is safe.

❑

Intuitively, B-compaction is safe because it ensures that no nodes will be split. This property can be deduced from the structure of build-result. It creates one node for each node it visits (which uniquely depends on the edge-marking) and visits each node at most once (because of the operation cache). Specifically, nodes that are not reachable from the root by a path of marked edges are not visited by build-result and thus not included in the minimized BDD.

2.3.2 Leaf-Identifying Compaction

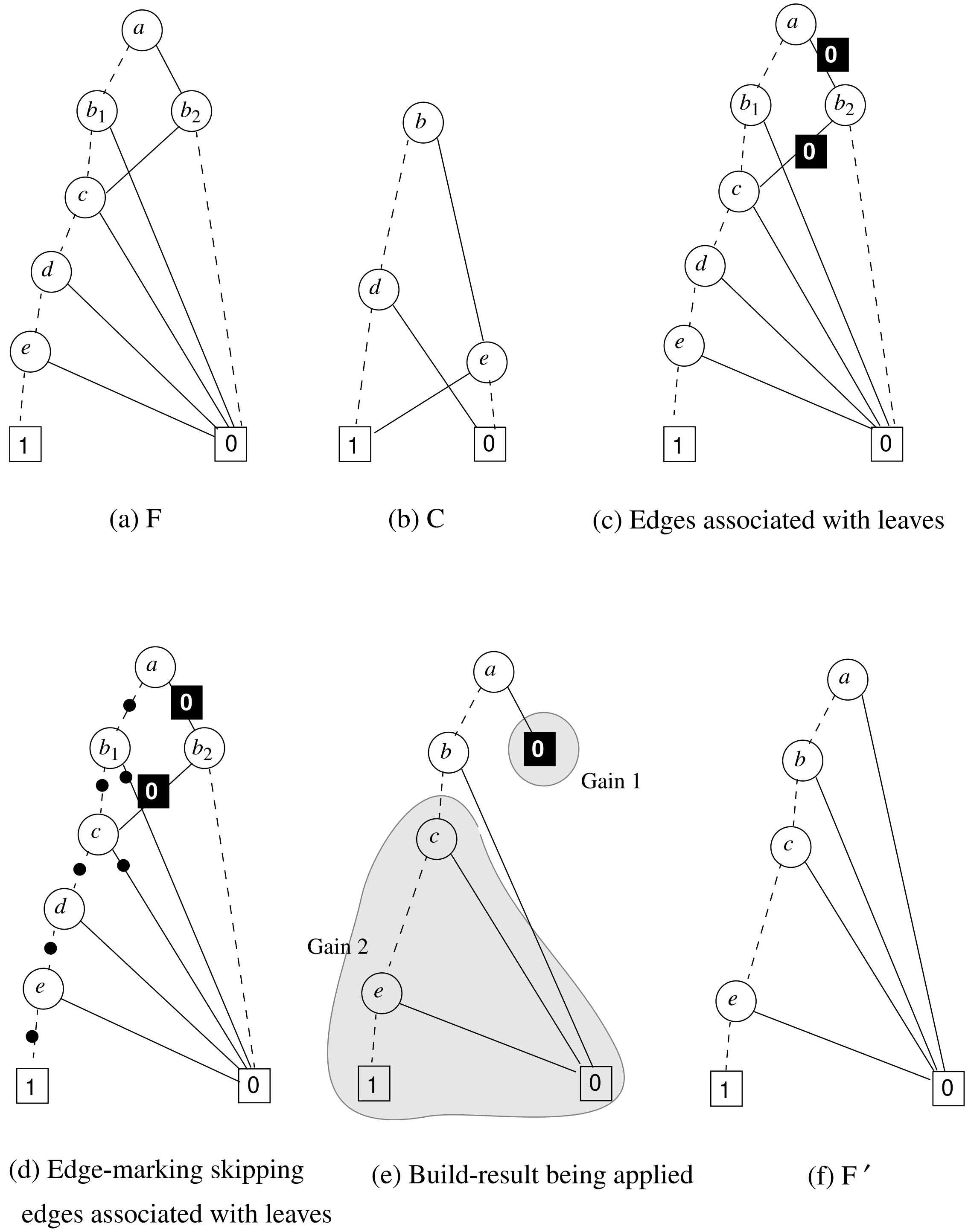
This subsection presents an enhanced safe minimization technique in which a special type of node splitting is allowed. Consider the set of sibling-substitutions applied to v to substitute its child u with another child of v. When the results of all the substitutions for u are unique, then the sibling-substitutions can increase the BDD size only by the size of the unique result. Leaf nodes are special in that they are *essential* for all non-trivial BDDs. So, the idea of new algorithm is to accept the result of sibling-substitution if the result is a unique leaf (i.e. replace the edge from v to u with an edge from v to the leaf). Note that, u may be preserved or replaced in the minimized BDD if it has multiple parents, depending on sibling-substitutions with respect to its other parents.

In effect, more sibling-substitution is allowed in LI-compaction compared to B-compaction. That is, LI-compaction allows sibling-substitution to a node u to replace a child v by its sibling if $R(e) = \{\text{leaf-0}\}$ (like B-compaction does) or to replace v by a leaf if the leaf is a cover of $[v, c]$ for all $c \in R(e)$, where e is the edge between u and v.

This approach will usually lead to better results for two reasons. First, a sub-BDD that is preserved in B-compaction can be replaced by a leaf in LI-compaction. We call this type of gain *Gain 1*. Second, the number of edges marked can be less than in B-compaction because the edge-marking routine needs not recur through edges to be redirected to leaves. This type of gain is called *Gain 2.* Typically, less edge-markings leads to smaller BDDs because build-result removes nodes connected by unmarked edges. Note, however, that this approach is not guaranteed to produce better results than B-compaction because the two algorithms can result in different unshared nodes becoming shared unpredictably.

This new approach can be implemented using a two-phase edge-marking routine and a modified build-result. The first phase of edge-marking computes the results of all possible sibling-substitutions from which it identifies the edges that can be redirected to leaves. The second phase is similar to basic mark-edges except that it does not recur through edges that can be redirected to leaves. After the edge-marking, the modified build-result routine redirects all identified edges to their annotated leaf and applies sibling substitution to all remaining unmarked edges.

Figure 2.8 shows an example where both gains contribute in minimizing the BDD. First, the then-edge from node a and the then-edge from node b_2 can be redirected to the leaf-0 (Gain 1). Consequently the then-edge of node d is unmarked (Gain 2). The modified build-result routine leads to a minimized BDD with two nodes less than the original BDD. In contrast, B-compaction leads to no minimization because the basic edge-marking routine must mark all edges.



(a) F (b) C (c) Edges associated with leaves

(d) Edge-marking skipping edges associated with leaves (e) Build-result being applied (f) F′

Figure 2.8: Improved result by leaf-identification

The run time complexity of this approach is almost twice as much as B-compaction because of the two-phase edge-marking routine since each edge-marking phase has $O(|F|\cdot|C|)$ complexity. If we do not pursue the gain from fewer marked edges (Gain 2), it is possible to merge the two phases of edge-marking into one. Our experiments suggest that degradation of quality is negligible. We believe this is because it is unlikely that all nodes on the paths leading to an excessively marked edge can be redirected to a unique leaf (so that no marking is required for the edge). Thus, this compromise represents a good performance/run-time trade-off.

We refer to this enhanced algorithm with the above compromise as *leaf-identifying compaction* (*LI-compaction*) and it is given in Figure 2.9. Finding and annotating nodes is performed in a preprocessing phase called LI-mark-edges. Like restrict, this phase recursively performs sibling-substitution. However, instead of returning the actual BDD result, it returns a classification of the result. This classification identifies whether the edge can be redirected to a 1 (encoded b01), 0 (encoded b10), DC (encoded b00), or non-leaf (encoded b11). The encoding facilitates a bitwise-OR scheme that implements the relative priority of non-leaves over leaves and leaves over DCs. Figure 2.10 illustrates an example of leaf-identifying compaction where one edge, the then-edge of d, is additionally marked compared to the example in Figure 2.10 (d).

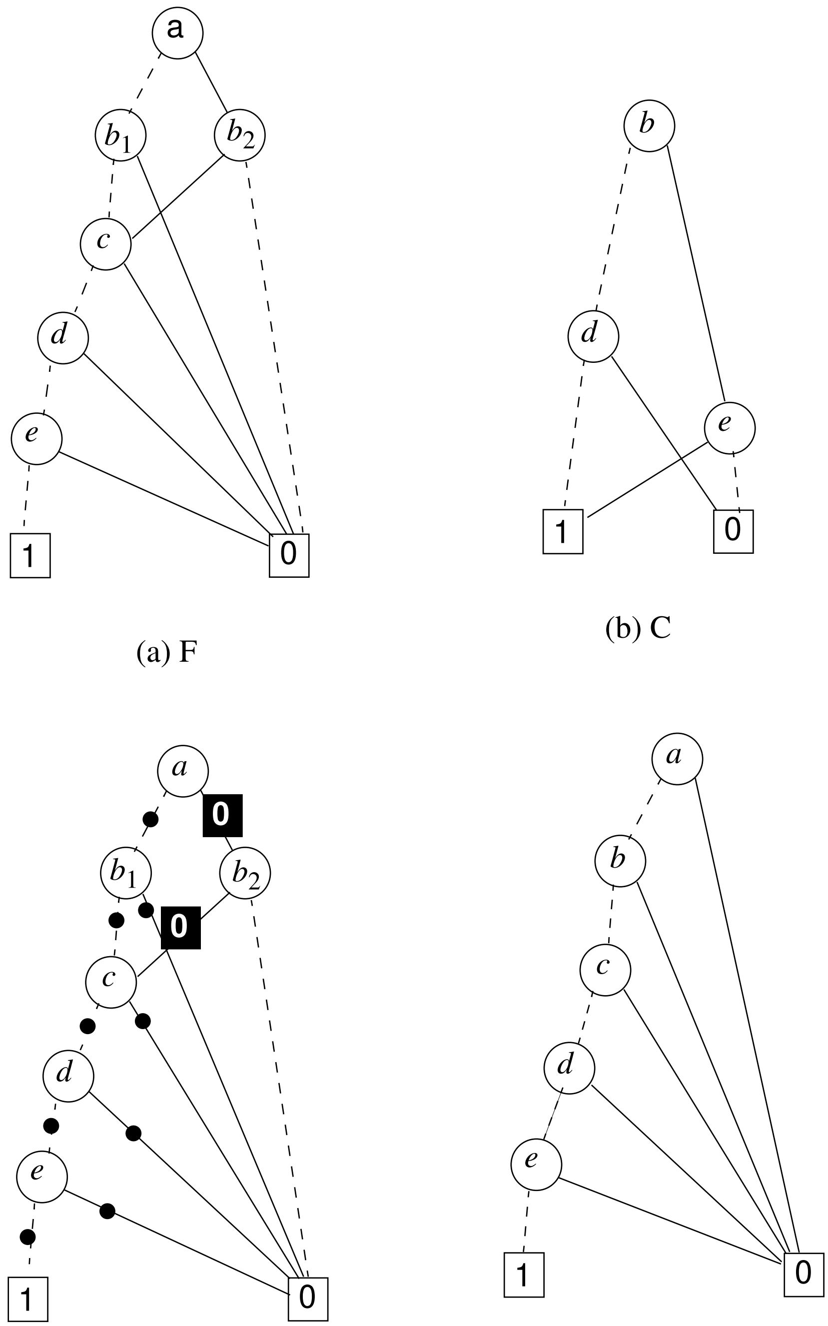
The overall time complexity of LI-compaction is the same as the complexity of B-compaction which is $O(|F|\cdot|C|)$.

```
bdd LI-compaction(bdd f, bdd c) {
        if (c == 0) return (0);
        (void) LI-mark-edges (f, c);
        result = LI-build-result (f);
        clear-edges(f);
        return (result);
}

int LI-mark-edges(bdd f, bdd c) {
        if (c == 0) return (00);
        if (f  == 0) return (01);
        if (f  == 0) return (10);
        x = top variable(f, c);
        temp1  = LI-mark-edges (f_x, c_x);
        temp2  = LI-mark-edges (f_x̄, c_x̄);
        if (f != f_x)
                f.then_mark =  f.then_mark |  temp1;  /* '|' is  bitwise-or */
                f.else_mark  = f.else_mark  |  temp2;
        return (temp1 | temp2);
}

bdd LI-build-result(bdd f) {
        if (f  == leaf) return (f);
        x = top variable (f);
        if (f.then_mark == 11) f_left = LI-build-result (f_x);
        else if (f.then_mark == 01) f_left = 0;
        else f_left = 0;
        if (f.else_mark == 11) f_right = LI-build-result (f_x̄);
        else if (f.else_mark == 01) f_right = 0;
        else f_right = 0;
        if (f.then_mark == 00 and f.else_mark != 00) return f_right;
        else if (f.then_mark != 00 and f.else_mark == 00) return f_left;
        else return (bdd-find (x, f_left, f_right));
}
```

Figure 2.9: LI-compaction pseudo-code



(a) F (b) C

(c) Edge-marking result (d) F′

Figure 2.10: LI-compaction example

2.3.3 Essential Nodes-Identifying Compaction

This subsection presents an improvement of LI-compaction. Recall that LI-compaction considers only leaf nodes as being essential, i.e., needed for all covers of a BDD. The key idea of the improvement is that we identify a larger subset of essential nodes. To be specific, we observe that a node is essential if it is connected by an edge e such that leaf-1 $\in$ $R(e)$. This condition means that there exists at least one input combination which requires the sub-function represented by the node u pointed to by e. Thus, the sub-BDD rooted at u must be in any cover of the BDD.

To identify essential nodes, we traverse F and C simultaneously. When an edge e corresponds to leaf-1 in C, we mark the node pointed by e as essential. We call this procedure find-essential-nodes and its pseudo code is shown in Figure 2.11.

Figure 2.12 shows an example of identifying essential nodes. Given F and C as shown in (a) and (b), the mapping nodes for each edge is illustrated in (c). We can see that d_1, d_2, leaf-1, and leaf-0 are essential, as shown in (d).

After identifying essential nodes, edge-marking applies sibling-substitution by DC substitutability for each F node and stores the results on the edge to which the child being substituted is connected. If the results associated with an edge are unique and essential, then we redirect the edge to the essential node. This redirection is ensured not to grow the size of the BDD because we know the essential node must be present in any cover (including the original). Figure 2.13 illustrates an example for this compaction called *essential node identifying compaction* (*EI-compaction*) and the complete pseudo code for EI-com-

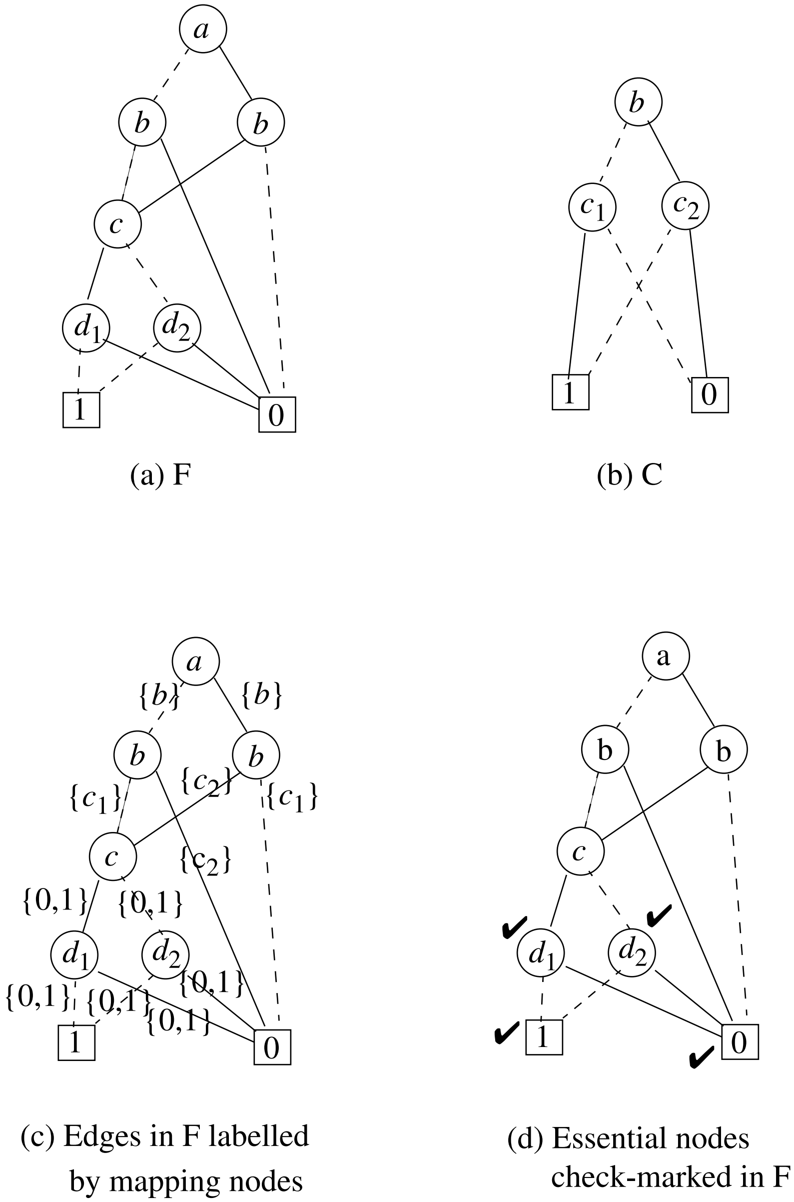
```
void find-essential-nodes (bdd f, bdd c, hash_table H) {
    if (f == leaf)
        insert f in H;
        return;
    insert 0 in H;
    insert 1 in H;
    find-essential-nodes-sub(f, c, H)
}

void find-essential-nodes -sub(bdd f, bdd c, hash_table H) {
    if (c == 0) return;
    if (f == leaf) return;
    if (f is in H) return;
    if (c == 1) insert f in H;
    x = top variable(f, c);
    find-essential-nodes-sub (f_x, c_x, H);
    find-essential-nodes-sub (f_x̄, c_x̄, H);
}
```

Figure 2.11: Find-essential-nodes pseudocode

paction is presented in Figure 2.14.

The time complexity of EI-compaction is $O(|F| \cdot |C|)$ which is the same as LI-compaction.



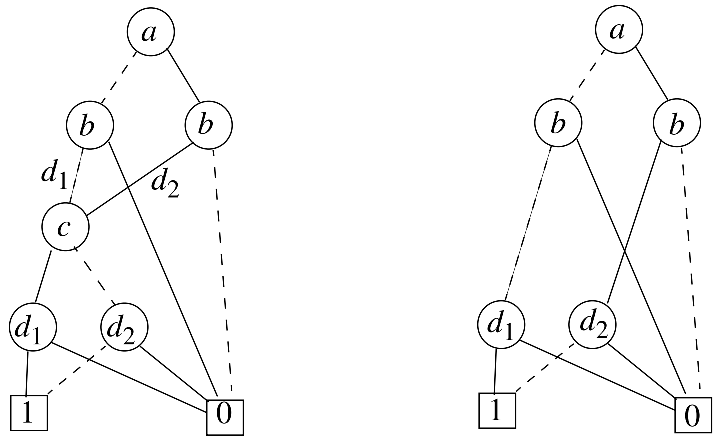
(a) F

(b) C

(c) Edges in F labelled by mapping nodes

(d) Essential nodes check-marked in F

Figure 2.12: Finding essential-nodes



(a) Essential node annotatation
(annotation omitted if the result is the child)

(b) Result

Figure 2.13: EI-compaction

bdd **EI-compaction** (bdd *f,* bdd *c*) {
    if (c == 0) return (0);
    H = create_hash();
    find-essential-nodes (*f*, *c*, *H*);
    (void) EI-mark-edges(*f*, *c*, *H*);
    result = EI-build-result(*f*);
    clear-edges(*f*);
    return(*result*);
}

bdd **EI-mark-edges** (bdd f, bdd c, hash_table H) {
if (c == 0) return (DC);
if (f == 0 || f == 1) return(f);
x = top variable(f, c);
temp1 = EI-mark-edges(f_x, c_x);
temp2 = EI-mark-edges($f_{\bar{x}}$, $c_{\bar{x}}$);
if (c_x != 0)
if (*temp1* is not in H) *temp1* = INVALID;
if ($c_{\bar{x}}$!= 0)
if (temp2 is not in H) *temp2* = INVALID;
else if (*f.else_mark* != DC && *f.else_mark* != *temp2*) *temp2* = INVALID;
if (f != f_x)
if (c_x != 0) *f.then_mark* = *temp1*;
if ($c_{\bar{x}}$!= 0) *f.else_mark* = *temp2*;
if (c_x == 0) return *temp2*;
else if ($c_{\bar{x}}$ == 0) return *temp1*;
else if (*temp1* == INVALID || *temp2* == INVALID) return INVALID;
else return bdd_find(x, *temp1*, *temp*2);
}

bdd **EI-build-result**(bdd f) {
if (f == leaf) return(f);
x = top variable(f);
if (*f.then_mark* == INVALID) *f_left* = EI-build-result(f_x);
else if (*f.then_mark* != DC) *f_left* = *f.then_mark*;
if (*f.else_mark* == INVALID) *f_right* = EI-build-result($f_{\bar{x}}$);
else if (*f.else_mark* != DC) *f_right* = *f.else_mark*;
if (*f.then_mark* == DC and *f.else_mark* != DC **)** return *f_right*;
else if (*f.then_mark* != DC and *f.else_mark* == DC) return *f_left*;
else return (bdd_find(x, *f_left*, *f_right*));
}

Figure 2.14: EI-compaction pseudo-code

2.3.4 General-Substitutability Based Compaction

Recall that restrict is based on DC-substitutability. Shiple et al. [73] pointed out that DC-substitutability is a sufficient but not necessary condition for a node to be able to substitute its sibling. They developed variants of restrict using relaxed criteria that allow more sibling substitutions. However, like restrict, many of these sibling substitutions cause node-splitting and consequently often cause BDD size growth.

We define general-substitutability as follows.

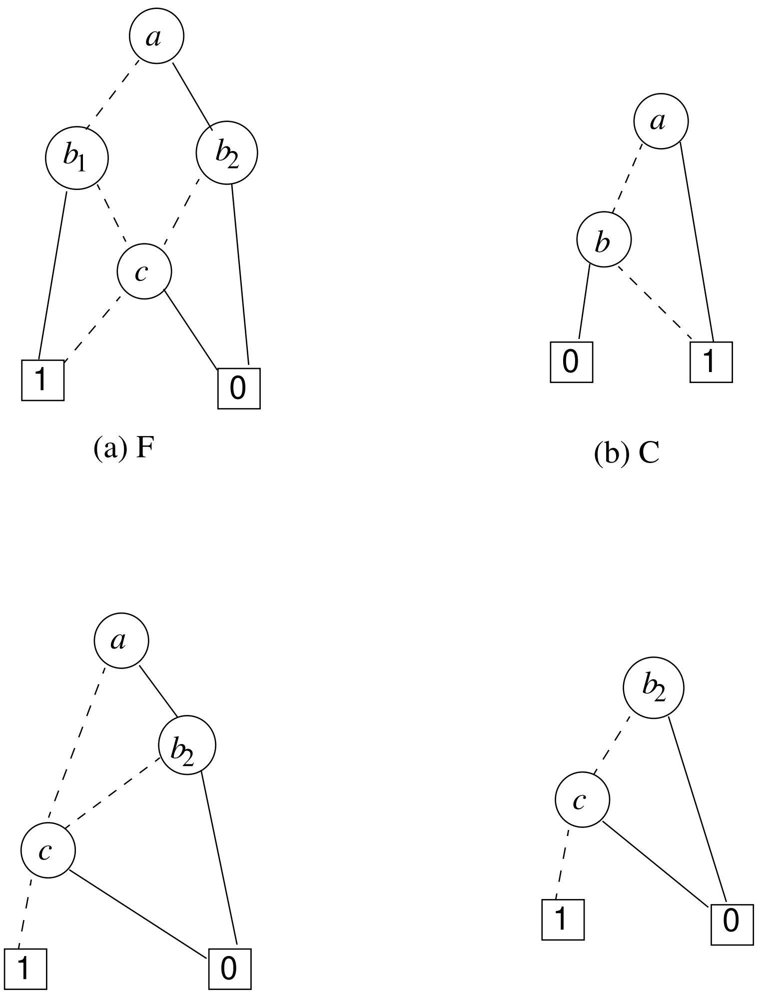
Definition 3. [General-substitutability[3]] *ff* is substitutable by *gg* if *gg* is a cover of *ff*.

To take advantage of a more generalized sibling-substitution criterion we propose to incorporate general-substitutability into the selective sibling-substitution framework in which we prevent sibling substitutions that can cause BDD size growth.

Figure 2.15 illustrates the effect of the different substitution criteria. Sibling-substitution can be applied to a or b_1 by general-substitutability or DC substitutability, respectively. The result produced by the sibling-substitution applied to a is smaller. Intuitively, this is because applying sibling-substitution to a higher node, i.e., larger sub-BDD, typically removes more nodes.

---

3. General-substitutability is more general than one-sided match proposed by Shiple et al. [73] because *one-sided match* requires $ff_{dc} \subseteq gg_{dc}$ to substitute *ff* by *gg* even if *ff* is a cover of *gg*.



(a) F

(b) C

(c) Sibling-substitution to b_1 by DC substitutability

(d) Sibling-substitution to a by general substitutability

Figure 2.15: DC-substitutability vs. general-substitutability

We present the pseudo code for the substitutability check in Figure 2.16. When two nodes can substitute each other, we give priority in such a way that we substitute the node at higher level because the BDD rooted at higher level is typically larger than the other and removing a larger BDD is likely to be more beneficial in terms of overall BDD size reduction. Alternatively, we can measure the size of each BDD to find larger one, but in our experience this does not lead to significant improvements.

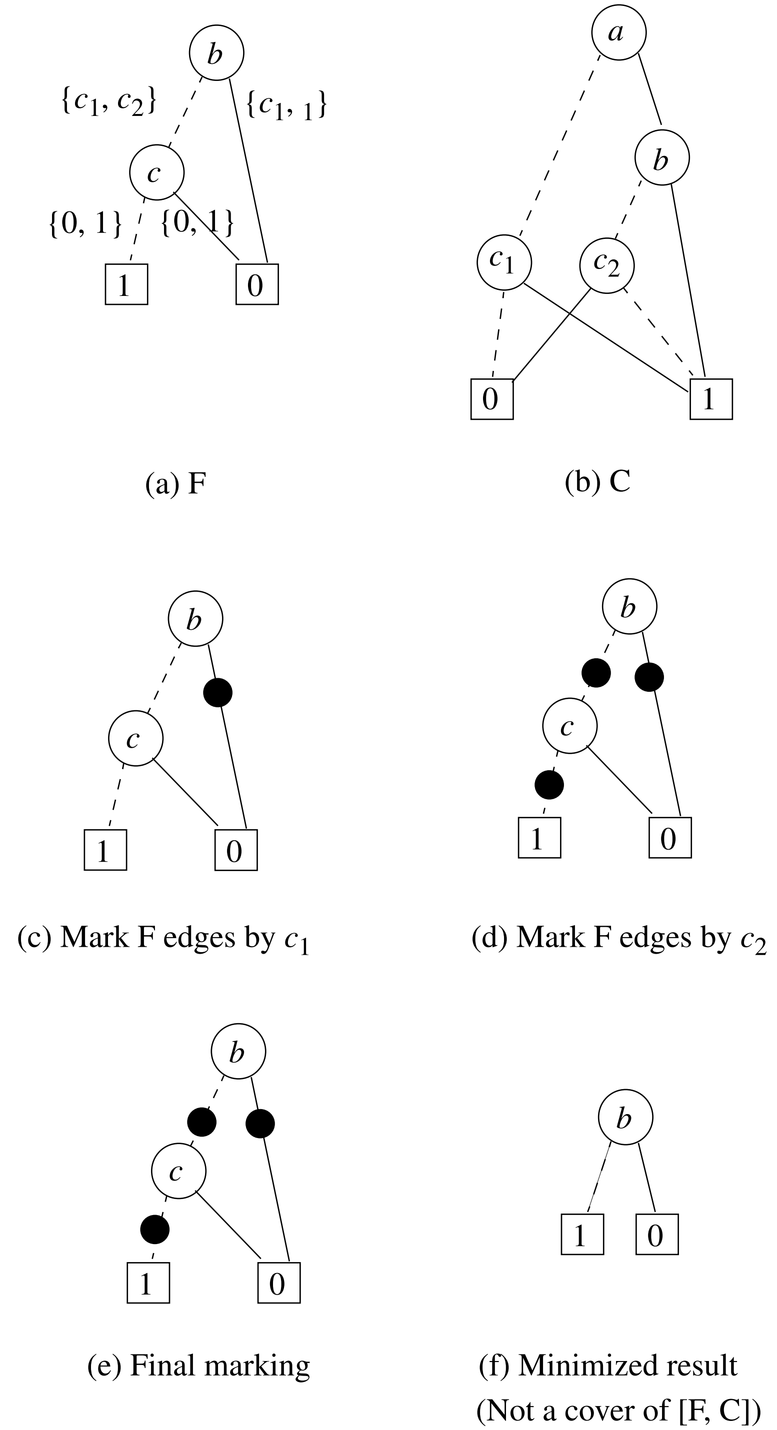
```
int substitutability (bdd f, bdd c) {
    x = top variable(f, c);
    if (c_x == 0) return (f_x to f_x̄);
    if (c_x̄ == 0) return (f_x̄ to f_x);
    fdiff = xor (f_x, f_x̄);
    temp1 = intersect (fdiff, c_x)
    temp2 = intersect (fdiff, c_x̄)
    if (temp1 == ∅ & temp2 == ∅)
        if (root level of f_x > root level of f_x̄) return (f_x to f_x̄);
        else return (f_x̄ to f_x);
    else if (temp1 == ∅) return (f_x to f_x̄);
    else if (temp2 == ∅) return (f_x̄ to f_x);
    else return (NONE);
}
```

Figure 2.16: Substitutability check pseudocode (A constant fxtofx, fxtofx, or NONE is returned according to substitutability direction.)

To incorporate the generalized criterion into B-compaction, we first consider a naive approach in which we simply change the sibling-substitution criteria from DC-substitutability to general substitutability. Consider the F and C BDDs illustrated in Figure 2.17 (a) and (b), respectively. The else-edge of node b (connected to node c) in F is related to care nodes $\{c_1, c_2\}$. Let's first apply edge-marking using the care node c_1 based on general substitutability. The node c is substitutable by its sibling in this case because they differ only when c is 0 which is in the don't care set specified by c_1. Consequently, the else-edge of b is not marked and using the naive edge-marking procedure we do not recur through b. Thus at this point all edges below c are not marked (see Figure 2.17 (c)). This makes sense because the entire sub-BDD rooted at node c at this point seems unnecessary since we are under the assumption that node c will be removed by a sibling substitution to node b. This assumption, however, breaks down when mark-edges processes the else-edge of node b with its other corresponding care node c_2 which prohibits sibling substitution to node b. The consequence of this assumption breakdown is that mark-edges does not mark some edges that should be marked. In our example, mark edges fails to mark the then-edge of c which should be marked because the sibling-substitution to node b cannot performed and this edge relates to leaf-1 (see Figure 2.17 (e)). This leads to an incorrect cover of F as shown in Figure 2.17 (f).

This example suggests that we may skip recurring mark-edges through an edge only when we are guaranteed that the intended sibling-substitution will be performed as assumed. In order to guarantee that the intended sibling-substitution on a node is performed, we must analyze all of its ancestors in conjunction with all of their associated care nodes. This motivates processing the nodes in F in a top-down level-by-level order.

We can process F edges while traversing F from top to bottom level by level. However, this requires cumbersome book-keeping of all related nodes for each edge. Alterna-



(a) F (b) C

(c) Mark F edges by c_1 (d) Mark F edges by c_2

(e) Final marking (f) Minimized result (Not a cover of [F, C])

Figure 2.17: Failure of using general-substitutability in B-compaction

tively, we can accomplish this top-down processing using a two-phase mark-edges routine without increasing run time complexity. The basic idea is that one phase performs edge-marking assuming that currently unmarked edges will stay unmarked. The other phase checks if the assumption associated with each sibling-substitution (that a particular edge remains unmarked) holds and re-invokes the first edge-marking phase as required if the assumption is invalidated.

The pseudo code for this new compaction algorithm called generalized substitutability based compaction (GS-compaction) is presented in Figure 2.18

.

```
void mark-essential-edges (bdd f, bdd c, list fcs_list) {
        if (c == 0  || f == leaf) return;
        x = top variable(f, c);
        if (f != f_x)
                s = substitutability (f, c);
                if (f.then_mark== 0)
                        if (s == f_x to f_x̄)  insert(fcs_list,{f, c, s});
                        else f.then-edge = 1;
                if (f.else_mark== 0)
                        if (s = f_x̄ to f_x) insert(fcs_list, {f, c, s});
                        else f.else-edge = 1;
        if (f.then_mark == 1 || f == f_x) mark-essential-edges(f_x, c_x, fcs_list);
        if (f.else_mark  == 1 || f == f_x) mark-essential-edges(f_x̄, c_x̄, fcs_list);
}
```

```
void mark-supplemental-edges (list fcs_list) {
    while (fcs_list not empty) {
        sort fcs_list by level of f in each triple;
        fcs = first(fcs_list);
        remove (fcs, fcs_list);
        f = fcs.f; c = fcs.c; s = fcs.s;
        x = top variable(f, c);
        if (s == f_x to f_x̄)
            if (f.then_mark == 0) mark-essential-edges(f_x̄, c_x, fcs_list);
            /* f_x̄ is guaranteed to substitute f_x. Include c_x to the care sets of f_x̄ */
            else mark-essential-edges(f_x, c_x, fcs_list);
            /* f_x̄ cannot substitute f_x. Do edge-marking that was skipped. */
        if (s == f_x̄ to f_x)
            if (f.else_mark == 0) mark-essential-edges(f_x, c_x̄, fcs_list);
            /* f_x is guaranteed to substitute f_x̄. Include c_x̄ to the care sets of f_x */
            else mark-essential-edges(f_x̄, c_x̄, fcs_list);
            /* f_x cannot substitute f_x̄. Do edge-marking that was skipped. */
    }
}

bdd GS-compaction (bdd f, bdd c) {
    if (c == 0) return(0);
    fcs_list = create-list();
    mark-essential_edges(f, c, fcs_list);
    mark-supplemental-edges(fcs_list);
    clear-edges(f);
    return(build-result(f));
}
```

Figure 2.18: GS-compaction pseudocode

The first edge-marking phase, called *mark-essential-edges*, tests the sibling-substitution condition only if the edge connected to the node considered has not been marked thus far. If the substitutability holds, it does not recur through the node to be substituted, marks the edge connected to its sibling, and continues this process to the node connected by the marked edge. The sibling-substitution is recorded in a list whose elements are triples of the node in F, the node in C, and the sibling-substitution direction. For the example presented in Figure 2.17, the marking-result shown in Figure 2.17 (e) is the result of *mark-essential-edges*. The second edge-marking phase, called *mark-supplemental-edges*, sorts the list by the level of F nodes in order to process the edges connected from higher F nodes first. Then, it removes the first sibling-substitution, e.g. substituting v in F with its sibling w, from the list and check if it is valid. If the sibling-substitution is still valid, the care-set of v needs to be added to the care-set of w. This care-set update is implicitly done by invoking mark-essential-edges with w and the node representing the care-set of v as parameters. If the sibling-substitution of v is found invalid, mark-essential-edges is invoked with v and the node representing the care-set of v as parameters. When *mark-essential-edges* finishes, *mark-supplemental-edges* resumes the validation process until the list is empty. Note that new sibling-substitutions might be recorded by *mark-essential-edges* which is invoked by *mark-supplemental-edges*. Build-result routine is the same as in B-compaction. In our example, the sibling-substitution to b in F is checked first. As both outgoing edges are found marked, the sibling-substitution is invalidated and then-edge of c in F becomes marked eventually.

The time complexity of mark-essential-edges is $O((|F| \cdot |C|)^2)$ because the substitutability routine requires $O(|F| \cdot |C|)$ and that routine is called at most $|F| \cdot |C|$ times by mark-essential-edges. The time complexity of mark-essential-edges is $O((|F| \cdot |C|)^2 + |F| \cdot |C| \cdot log(|F| \cdot |C|))$, where the first term describes the time complexity of mark-essential-edges and the second term describes the time complexity for quick sort. Note that mark-essential-edges does not repeat the same computation because of the operation cache. Build-result and clear-edges both have $O(|F| \cdot |C|)$ time complexity. Therefore, the overall time complexity of GS-compaction is $O((|F| \cdot |C|)^2)$.

2.3.5 Multiple BDDs Minimization

It is important to note that safe BDD minimization does not itself guarantee overall reduction in the size of multi-output circuits. This is because each output is represented by one BDD and existing safe BDD minimization does not consider the sharing among BDDs. Consequently, minimizing one BDD may reduce the sharing among BDDs, potentially leading to an overall increase in BDD nodes. This suggests that the synthesized circuit might be larger after BDD minimization using existing techniques.

Fortunately, we can extend the concept of safety to handle multiple BDDs. The basic idea is to first complete all edge-markings for each output function *f* and corresponding care set *c,* and only then apply build-result for each *f.* We can apply this separated edge-marking and build-result concept to all compaction heuristics. For circuits with lots of sharing among cones of logic this feature can have a significant impact.

2.4 Experimental Results

We conducted experiments in SIS-1.2 [70] to compare our heuristics to existing sibling-substitution based heuristics, specifically, restrict and one of its variants *osm-bt* [73]. Osm-bt was chosen among a variety of heuristics developed by Shiple et al. because it showed the best overall results in the examples they tested. Because the two existing heuristics can produce larger BDDs than the original BDDs, we applied thresholding to them which means that we return the original BDDs if the heuristics produce larger BDDs. All the experiments were conducted on a SUN SPARC 20/128Mbyte.

In our first experiment, we minimized the BDDs representing the combinational logics of sequential circuits from ISCAS-89 and ISCAS-Addendum-93 benchmark circuits using their unreachable states as DCs. For some of the largest circuits, exact reachable states could not be computed because the memory requirements were too high. For three of these circuits, however, we were able to compute a superset of reachable states using MBM traversal, an approximate FSM traversal technique proposed by Cho et al. [19], and used the complement of the result as DCs. The result of MBM traversal takes the form of implicit conjunction, i.e. a set of BDDs representing reachable states on partitioned state spaces, and we constructed a single care BDD by conjuncting all of them. We minimized each BDD representing a combinational logic cone using the single care BDD, individually.

The results are given in Table 2.1. The nodes shared by multiple BDDs are counted once for each BDD to better illustrate the minimization quality on individual BDDs. GS-compaction demonstrates the best performance except for one example s499 in which the difference is only 2 nodes. This suggests that the safety feature consistently improves minimization quality when combined with general substitutability.EI-compaction does not lead to noticeable improvements over LI-compaction and we believe that this is because the possibility of being able to replace a node with a non-leaf essential node is typically not high. We show the ratio of the smallest BDD sizes obtained using new heuristics and existing heuristics in the column denoted *improv*. New heuristics produce up to 25% (6.3% on average) smaller BDDs than the BDDs produced by existing heuristics.

An interesting observation is that there were many BDDs that no heuristics were able to reduce their size in our experiment, and if we consider only the BDDs whose size can be reduced by any heuristic, the improvement factor becomes almost two times larger as shown in the Table 2.3. One reason why there are many BDDs whose size cannot be reduced with so high DC fractions is that the effective DC fractions which are solely based on the supporting variables of the BDD to minimize can be very small.

Table 2.2 shows the run-time of each minimization in unit of CPU second. Because compaction heuristics require more phases than restrict, they are slower than restrict. Specially, GS-compaction requires a time-consuming substitutability check which requires significant run time overhead. However, GS-compaction is much faster than its counterpart osm-bt for large BDDs even though GS-compaction uses a more general substitution criterion than the one used by osm-bt. This is because GS-compaction skips many substitutability checks, i.e., GS-compaction does not check substitutability for the nodes connected by marked edges.

circuit	BDD_f size	BDD_c size	BDD_c on-set (%)	reduced BDD_f size: restrict	osm_bt	B	LI	EI	GS	improv. (%)
s298	183	57	1.3	137	129	149	140	140	118	9.3
s344	260	610	8.0	253	248	253	253	253	235	5.5
s382	262	304	0.4	245	231	265	245	245	200	15.5
s444	300	173	0.4	218	205	261	224	224	193	6.2
s499	1073	72	5.2e-5	340	340	340	340	340	272	25
s510	281	8	73.4	265	262	269	265	265	264	-1.0
s526	301	125	0.4	251	220	254	251	251	204	7.8
s641	884	83	0.3	607	577	599	599	599	548	5.3
s820	499	8	78.1	472	471	469	469	469	468	0.6
s953	867	562	1.6	739	739	751	739	739	712	3.8
s1196	3974	919	1.4	3974	3971	3974	3974	3974	3966	0.1
s1269	40487	199	1.1	29699	29727	28000	27995	27995	27971	6.2
s1423	35125	63	1.3	28469	28833	28470	28469	28469	28439	0.1
s3271	2917	417	3.6	2787	2733	2793	2781	2781	2629	4.0
average										6.3

Table 2.1: Minimization results on BDDs for sequential circuits using unreachable states as DCs. (All BDDs are included in the results.)

circuit	BDD_f size	BDD_c size	BDD_c on-set (%)	reduction time: restrict	osm_bt	B	LI	EI	GS
s298	183	57	1.3	0.01	0.06	0.38	0.37	0.52	1.08
s344	260	610	8.0	0.15	1.00	0.62	0.61	0.81	1.62
s382	262	304	0.4	0.04	0.35	0.54	0.53	0.87	1.54
s444	300	173	0.4	0.12	0.32	0.54	0.53	0.87	1.56
s499	1073	72	5.2e-5	0.05	0.11	0.85	0.54	1.24	2.42
s510	281	8	73.4	0.01	0.04	0.26	0.27	0.47	0.72
s526	301	125	0.4	0.03	0.13	0.53	0.52	0.62	1.51
s641	884	83	0.3	0.06	0.86	0.85	0.85	1.15	2.48
s820	499	8	78.1	0.02	0.07	0.47	0.45	0.85	1.39
s953	867	562	1.6	0.04	0.39	0.97	0.99	1.29	2.78
s1196	3974	919	1.4	0;49	10.5	0.93	0.95	1.25	3.05
s1269	40487	199	1.1	2.71	222	6.63	6.55	7.55	25.5
s1423	35125	63	1.3	1.50	1084	4.80	4.50	5.50	19.9
s3271	2917	417	3.6	0.74	29.5	10.6	11.0	13.0	21.0
average									

Table 2.2: Run-time results on BDDs minimization for sequential circuits using unreachable states as DCs. (All BDDs are included in the results.)

circuit	BDD_f size	BDD_c size	BDD_c on-set (%)	reduced BDD_f size: restrict	osm_bt	B	LI	EI	GS	improv. (%)
s298	155	57	1.3	109	101	121	112	112	90	12.2
s344	84	610	8.0	77	72	77	77	77	59	22.0
s382	262	154	0.4	203	190	223	203	203	158	20.3
s444	259	173	0.4	177	164	220	183	183	152	7.9
s499	1029	72	5.2e-5	296	296	296	296	296	228	29.8
s510	122	8	73.4	106	103	110	106	106	105	-1.9
s526	272	125	0.4	222	191	225	222	222	175	9.1
s641	805	83	0.3	528	498	520	520	520	469	6.2
s820	370	8	78.1	343	342	340	340	340	339	0.9
s953	833	562	1.6	705	705	717	705	705	678	4.0
s1196	19	919	1.4	19	16	19	19	19	11	45.5
s1269	40046	199	1.1	29258	29286	27559	27554	27554	27530	6.3
s1423	35125	63	1.3	28469	28833	28470	28469	28469	28439	0.1
s3271	948	417	3.6	636	618	644	637	637	552	12.0
average										12.4

Table 2.3: Minimization results on BDDs for sequential circuits using unreachable states as DCs. (BDDs whose size cannot be reduced by any heuristic are excluded from the results.)

In our second experiment, we minimized the BDDs representing the combinational circuits from ISCAS-89 and MCNC-91 benchmark. Following the approach used in [16, 77], we used randomly created DCs with 95% and 5% DC fractions for each BDD representing a combinational logic cone. The extreme DC fractions were chosen to demonstrate the impact of DC fractions on the improvement ratio. Each BDD representing a combinational logic cone is minimized using a respective care BDD. A modified version of GS-compaction, GSM-compaction, performs edge-markings for all BDDs before building result. The results from the modified version of other compaction heuristics are not shown because they show similar improvements compared to their original versions.

Table 2.4 and Table 2.6 show the results with 95% DCs and 5% DCs, respectively. We count the nodes shared by multiple BDDs only once for these results. An observation we can make is that the improvement factor is much higher when DC fractions are higher. This is because there is more flexibility of minimization with more DCs. We can also observe that the safety feature can make huge differences on multiple BDD minimizations. For example, the BDDs obtained using GSM-compaction for C499 are more than two times smaller than the BDDs obtained using GS-compaction. However, GSM-compaction might not produce better results than GS-compaction because sometimes node splitting leads to smaller BDDs whose total size is smaller than its originally shared BDD. (Recall that this is the same reason why safe BDD minimization might not be better than non-safe heuristics.)

Table 2.5 and Table 2.7 show the run-time results with 95% DCs and 5% DCs, respectively. The advantage of GSM-compaction over osm-bt in terms of run-time is clearer in this experiment because we observe two examples that osm_bt cannot complete the minimization in 10 hours.

circuit	BDD_f Size	BDD_c Size	reduced BDD_f size							improv. (%)
			restrict	osm_bt	B	LI	EI	GS	GSM	
bw	112	40	48	48	48	48	48	48	51	0.0
misex3c	797	359	249	220	304	281	281	213	226	3.29
duke2	973	1559	473	397	559	409	409	297	288	37.8
vg2	1044	5028	441	417	417	381	381	283	275	52.6
misex3	1301	359	471	427	583	504	504	414	410	4.15
C432	1733	202	1673	1673	1640	1634	1634	1611	1579	5.95
alupla	17266	21699	17483	15195	15609	15188	15188	6667	6563	131.5
C1908	36007	4490	31644	timeout	31264	32546	32546	31822	29494	7.30
C499	45922	726	75681	timeout	78809	96627	96627	90311	43490	74.0
seq	142252	7763	89892	82510	79330	77637	77637	67151	67949	22.9
average										34.0

Table 2.4: Minimization results on BDDs for combinational circuits with randomly generated 95% DCs.

circuit	BDD_f Size	BDD_c Size	reduction time						
			restrict	osm_bt	B	LI	EI	GS	GSM
bw	112	40	0.01	0.02	0.50	0.46	0.46	1.38	0.06
misex3c	797	359	0.05	0.38	0.34	0.32	0.32	0.92	0.27
duke2	973	1559	0.07	0.77	0.59	0.62	0.62	1.66	0.31
vg2	1044	5028	0.09	0.21	0.2	0.21	0.31	0.69	0.33
misex3	1301	359	0.05	0.38	0.34	0.32	0.43	0.92	0.27
C432	1733	202	0.01	0.01	0.14	0.09	0.19	0.42	0.06
alupla	17266	21699	3.51	78.8	2.87	2.85	4.42	14.1	14.7
C1908	36007	4490	20.0	timeout	16.3	16.4	26.4	77.1	84.8
C499	45922	726	44.5	timeout	36.7	37.9	57.9	244	391
seq	142252	7763	8.68	302)	14.0)	14.3	19.3	62.1	74.0
average									

Table 2.5: Run-time results on BDD minimizations for combinational circuits with randomly generated 95% DCs.

circuit	BDD_f size	BDD_c size	reduced BDD_f size							improv. (%)
			restrict	osm_bt	B	LI	EI	GS	GSM	
bw	112	40	92	87	92	92	92	87	91	0.0
misex3c	797	359	759	757	785	755	755	748	751	1.2
duke2	973	1559	956	950	966	953	953	932	931	0.02
vg2	1044	5028	1050	1050	1048	1048	1048	1048	1044	0.58
misex3	1301	359	1301	1301	1291	1287	1287	1300	1301	1.09
C432	1733	202	1734	1734	1734	1734	1734	1732	1731	0.17
alupla	17266	21699	17266	17266	17266	17266	17266	17257	17257	0.05
C1908	36007	4490	36001	timeout	36005	36053	36053	36031	35999	0.0
C499	45922	726	45922	timeout	45922	454922	454922	45922	45922	0.0
seq	142252	7763	142155	141590	138566	138384	138384	137991	137965	2.63
average										1.7

Table 2.6: Minimization results on BDDs for combinational circuits with randomly generated 5% DCs.

circuit	BDD_f size	BDD_c size	reduction time						
			restrict	osm_bt	B	LI	EI	GS	GSM
bw	112	40	0.01	0.01	0.51	0.46	0.56	1.30	0.04
misex3c	797	359	0.01	0.39	0.32	0.31	0.47	0.92	0.20
duke2	973	1559	0.07	1.01	0.61	0.62	0.72	1.78	0.37
vg2	1044	5028	0.07	2.12	0.24	0.35	0.39	0.65	0.34
misex3	1301	359	0.06)	0.77	0.35	0.39	0.49	1.00	0.78
C432	1733	202	0.16	44.0	0.32	0.32	0.52	1.56	1.25
alupla	17266	21699	2.94	119	3.00	3.10	5.10	12.9	12.7
C1908	36007	4490	22.3	timeout	17.6	18.7	28.7	67.5	69.2
C499	45922	726	50.8	timeout	47.1	48.0	68.0	435	429
seq	142252	7763	5.20	956	16.9	18.0	25.0	65.1	64.4
average									

Table 2.7: Run-time results on BDD minimizations for combinational circuits with randomly generated 5% DCs.

2.5 Conclusions

We presented new efficient heuristics to minimize the size of BDDs using DCs. The key idea of new heuristics is to selectively minimize sub-BDDs while traditional heuristics blindly minimize all sub-BDDs. By removing the source of size growth, we were able to achieve better overall minimization quality. We demonstrate that the new heuristics significantly outperform traditional heuristics on most examples from benchmark circuits. We also generalized our mechanism to prevent the size growth to handle multiple BDDs with sharing and present experimental results that demonstrate its effectiveness.

Chapter 3

Symbolic Reachability Analysis Using Don't Cares

This chapter presents novel techniques to improve both approximate and exact reachability analysis using don't cares. First, we propose an iterative approximate reachability analysis technique in which don't care sets derived from previous iterations are used to improve the approximation in subsequent iterations. Second, we propose new techniques to use the final approximation to enhance the capability and efficiency of exact reachability analysis. Experimental results show that our techniques can significantly improve existing reachability analysis.

3.1 Introduction

Reachable states of a finite state machine is the set of states a finite state machine can reach starting from a given set of initial states. Reachability analysis of a finite state machine refers to computing reachable states of the finite state machine give a set of initial states.

Reachability analysis is important for the following two reasons. First, the reachable states obtained from reachability analysis can be used as don't cares when optimizing circuits in logic synthesis application [27]. For these applications an overapproximated reachable state set can be also useful particularly when computing the exact reachable state set is not possible because it requires too much memory or time. For example, Sentovich et al. [72] use overapproximated reachable state sets to minimize the number of latches. The second reason why reachability analysis is important is because the core algorithm used for the reachability analysis can be applicable to many formal verification techniques, e.g. verification of equivalence of two machines [24, 76] and property checking [13].

The set of initial states is called *reset states*. In practice there are many sequential circuits for which no reset states are defined by designers because the extra circuitry required to guarantee the circuits to start with the reset states lead to inefficient circuits in terms of area and/or delay [74]. In such cases, an input sequence to guarantee the circuit to start from a known state may need to be identified. There are many approaches to determine if a reset sequence exists and compute shortest reset sequences efficiently. Pixley et al. [59] propose a method to determine if there exists a reset sequence without actually computing it. Cho et al. [21] and Pixley et al. [60] present methods to compute the reset sequences using implicit state enumeration techniques. Moreover, Rho et al. [64] proposed a technique to compute minimum length reset sequences. Retiming [47, 51, 31] is a widely used optimization technique to improve the quality of a circuit and this may lead to different reset state. Touati et al. [75] present an algorithm to compute the initial states of retimed circuits.

Reachability analysis computes reachable states from a *given* set of initial states of a finite state machine (FSM). The initial state set may be given by the user or previously described techniques may need to be used to compute the initial state set. There are two approaches to reachability analysis; explicit approach and implicit reachability analysis. Explicit reachability analysis processes one state at a time. A example of explicit state exploration in depth-first search order is shown in Figure 3.1. Note that it takes 4 steps to explore all reachable states.

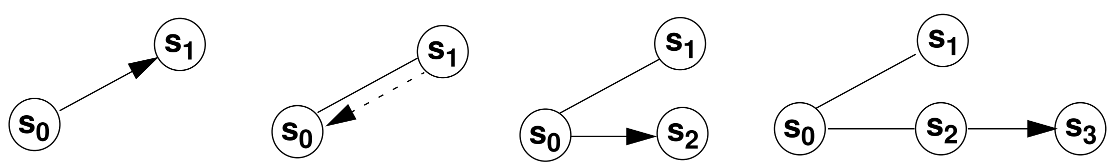


Figure 3.1: An example of explicit state exploration

Implicit approach identifies all next states from current states altogether in breadth-first order. Therefore, implicit reachability is typically much faster than explicit approach. Figure 3.2 illustrates how implicit state exploration works using the same example shown in Figure 3.1. It takes just 3 steps to explore all reachable states. Coudert et al. [24] have shown that BDDs are amenable to allow such breadth-first state exploration.

Symbolic reachability analysis techniques using BDDs [10] have been shown to be able to analyze much larger FSMs than was possible using explicit state techniques which

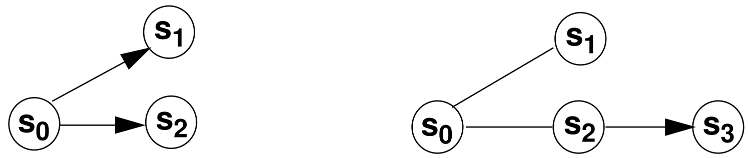


Figure 3.2: An example of implicit state exploration

process one state at a time [24, 76, 13]. Nevertheless, symbolic techniques cannot handle some large FSMs because they either require too much memory or are computationally too expensive.

Various techniques have been developed to enhance the capability and efficiency of symbolic reachability analysis. One class of the techniques is motivated by the observation that the size of a BDD depends greatly on the ordering of the BDD variables [10]. Many heuristics have been developed to find a good *initial* variable ordering for BDDs representing the transition relation of FSMs [2]. The initial variable ordering, however, is often not a good ordering for BDDs created during reachability analysis, so techniques to dynamically reorder the variables (e.g. [65]) are widely used [62, 61, 15].

Numerous other techniques to analyze large FSMs have been developed. For example, Ravi et al. [61] propose a mixed breadth-first and depth-first traversal to utilize subsets of states that are representable with small BDDs. This technique can reduce the memory requirement significantly but may produce an under-approximated reachable state sets. Cabodi et al. [15] and Narayan et al. [55] propose to *partition* state sets to guarantee the results to be exact when the traversal is complete. In addition, Cabodi et al. [14] present an approach that extends the application of disjunctive partitioned transition relations from asynchronous circuits to synchronous ones and utilizes iterative squaring.

The performance of reachability analysis may also be improved by applying BDD minimization using don't cares (DCs) [24, 7] to reduce the size of the BDDs. For example, Coudert et al. [24] propose to reduce the *frontier set* BDD which is used as a state space search frontier at each iteration. In addition, Brayton et al. [7] present techniques to reduce the size of the *transition relation BDDs* (*TR*) which represent the possible state transitions of FSMs. They propose to use transitions originating from non-frontier states and transitions to known reachable states as DC transitions. The major disadvantage of this approach is that the minimization of the transition relation BDD needs to be repeated at each iteration times because the frontier set and consequently the DC set is constantly changing.

Moreover, Ranjan et al. [62] discuss techniques to reduce the size of TRs using DCs derived from overapproximated reachable states [19]. The main advantage of this approach is that the minimization needs to be performed only once because the DC set is fixed for the entire reachability analysis. However, surprisingly, the BDD minimization led to larger BDDs in their experiments and performance improvements were not reported in their work.

In this chapter, we present new techniques to make use of the overapproximated reachable states to improve both approximate reachability analysis and exact reachability analysis. Novel aspects of our approach include:

- A *clustered DC BDD* constructed from partitioned DC BDDs to maintain high BDD minimization ratio with manageable DC BDD size.
- *Iterative* approximate reachability analysis in which DCs computed in previous iterations are used to improve the quality of the approximation computed in subsequent iterations.
- A heuristic to *minimize the support set size of TRs* that is useful to improve the early quantification schedule in exact reachability analysis.

Our experimental results demonstrate the effectiveness of the new techniques.

After describing background material in Section 3.2, we present our techniques to improve standard reachability analysis using DCs in Section 3.3. Section 3.5 reports our experimental results and Section 3.6 presents our conclusions.

3.2 Background

3.2.1 Symbolic Reachability Analysis

To describe the state transitions we use a set of input variables, denoted by i and two sets of state variables: present variables, denoted by a, and next state variables, denoted by y. The next state function δ for a FSM with n state bits consists of δ_i's for $1 \leq i \leq n$, referred to as a *transition function vector* [24], where δ_i represents the next state logic for i^{th} state bit.

A *transition relation* [24], denoted by *TR,* defines all possible current/next state pairs as follows.

$$TR(i, a, y) = \prod_{i=1}^{n} (y_i \equiv \delta_i\,(i, a)).$$

If we use x to represent the union of the input variable set and the present state variable set and $T_i(x, y)$ to denote $y_i \equiv \delta_i\,(x)$, then the transition relation can be described as

$$TR(x, y) = \prod_{i=1}^{n} T_i(x, y).$$

Both transition function vector [24, 25] and transition relation [24, 76, 13] can be used for reachability analysis. Coudert el al. [24] first used BDDs to represent both transition function and transition relation and dramatically extended the applicability of reachability analysis to large FSMs. Transition function has a limitation that it is not suitable to model nondeterministic systems [12]. Therefore we focus on transition relation based approach in this work.

Transition relation based reachability analysis has experienced significant improvements since it is first introduced. A major innovation is to represent the transition relation using an implicit conjunction of multiple BDDs [76, 11] because a single BDD representing the TR can be prohibitively large. Touati et al. [76] use T_i's to represent transition relation as an implicit conjunction. Burch et al. [11] proposed to using *partitioned transition relations*. They presented two types of partitioned transition relations; *conjunc-*

tive and *disjunctive* partitioned transition relations to describe synchronous circuits and asynchronous circuits, respectively. We focus on the conjunctive partitioned transition relation because our interest lie in synchronous circuits.

The basic idea of conjunctive partitioned transition relation is to partition T_i's into k groups, compute the conjunction of T_i's in each group producing a *cluster* TR_j for j^{th} group, and use the clusters to represent the transition relation implicitly. That is,

$$TR(x, y) = \prod_{j=1}^{k} TR_j(x, y).$$

Heuristics to automatically partition the T_i's are presented in [34, 62].

The states, *To,* reachable from a set of states, *From,* in at most one step can be obtained from the BDD representing the TR by performing existential quantification and conjunction as follows.

$$To(y) = From(y) \cup \exists_{x_i \in x} [From(x) \cdot TR(x, y)],$$

where $\exists$ represents existential quantification. When a partitioned transition relation is used,

$$To(y) = From(y) \cup \exists_{x_i \in x} [From(x) \cdot TR_1(x, y) \cdot \ldots \cdot TR_n(x, y)].$$

Here, the conjunctions are performed iteratively, forming many *intermediate results*.

If there exist a set of variables that some clusters do not depend on, the existential quantification can be distributed over the partial products. We can quantify out such variables from the intermediate results before the entire multiplication is finished, leading to smaller intermediate result BDDs [13]. This technique is called *early quantification*.

3.2.2 Approximate Reachability Analysis

Cho *et al.* [19] presented various approximate reachability analysis techniques that are guaranteed to produce a superset of the reachable states. It is suggested that the most robust technique for very large FSMs is their *machine-by-machine (MBM) traversal* technique in their work so we focus on the MBM traversal.

The MBM traversal technique starts by decomposing the state space by partitioning the set of next state variables. It then take the product of the transition function vectors associated with the next state variables belonging to the same partition to form a *cluster*.

The key idea to the MBM traversal technique is that it views a cluster as a sub-FSM by treating the current state variables that do not have corresponding next state variables in the cluster as external input variables. Consequently, each sub-FSM can be traversed separately to compute its set of reachable states in its corresponding state subspace by conjuncting the known reachable states from other sub-FSMs to identify the possible combinations of external inputs. More specifically, the set of possible combinations of external inputs are assumed to be identical in all sub-FSM states. For this reason, the approach may significantly overestimate the possible values of external inputs in each state of the machine.

The algorithm starts by assuming that all states of all sub-FSMs are reachable and all sub-FSMs are scheduled for traversal. Anytime the reached state of a sub-FSM changes, all the other sub-FSMs that depend on it are re-scheduled for traversal because their external inputs may now be further constrained. Each scheduled sub-FSM is processed until a fixed point in the computation of the reached set is obtained.

The approximate reachability analysis can handle much larger FSMs than feasible using the exact approach because the reachable states associated with each sub-FSM are maintained separately in the approximated reachability analysis while the exact approach uses a single BDD to represent the reachable states.

3.3 Symbolic traversal using don't cares

The basic premise of our work is that approximate reachability analysis can be used to quickly compute many states that are guaranteed to be unreachable from initial states. The complement of this set represents a subset of unreachable states. The key idea that any transition that originates from an unreachable state is a DC because it will never be explored during exact reachability analysis.

This section exploits these DCs to improve both approximate and exact reachability analysis. Figure 3.3 illustrates the overview of our approach. First, we start approximate reachability analysis with given T_i*'s*. The results of the approximation is a superset of reachable states, denoted R^+. We then derive a don't care set from R^+ and use the don't care set to simplify T_i's. We create new clusters using the new T_i's and perform new approximate traversal. We can iterate this process of simplifying T_i's and performing new approximate analysis until no improvement is made in terms of R^+.

Once the approximate reachability analysis is finished, we use the final approximation result to derive don't cares to simplify transition relation BDDs used during exact reachability analysis to improve the performance and the capability of it.

We first describe how to derive *clustered* DC BDDs from the results of approximate reachability analysis. We then describe how these clustered DCs can be used to improve approximate reachability analysis using an *iterative approach*, yielding larger DC sets. Finally, we describe how these DCs can be used to improve exact reachability analysis, introducing a new technique based on *support set minimization*.

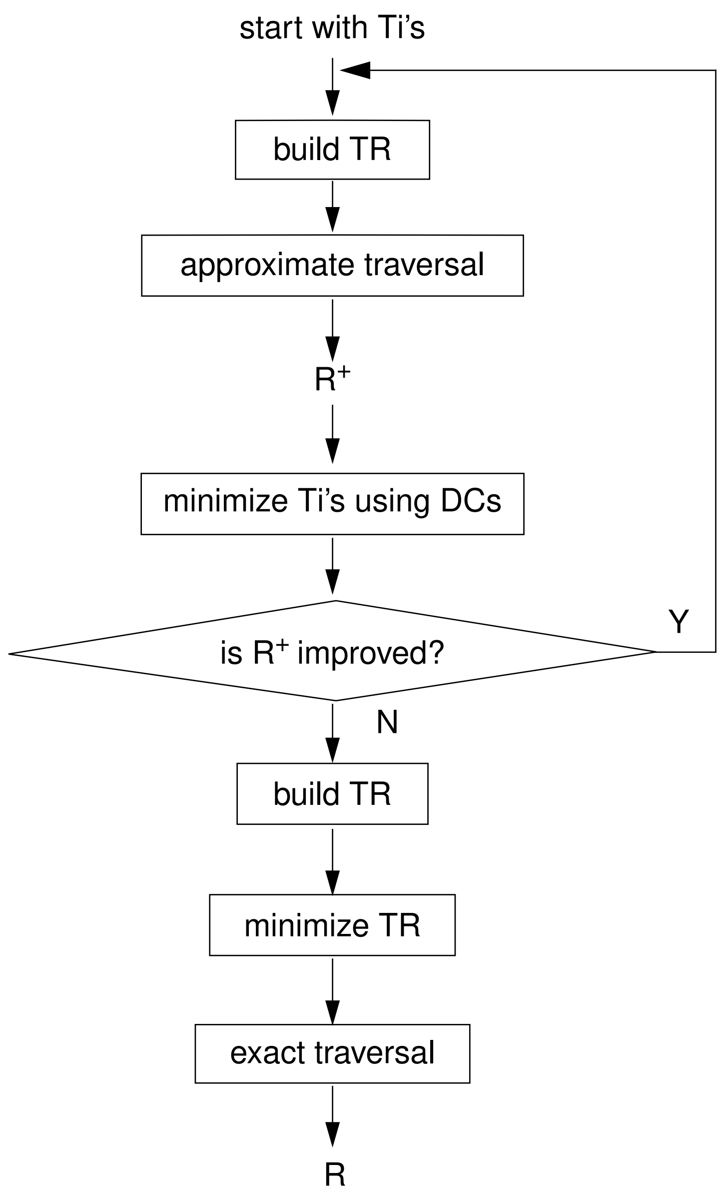


Figure 3.3: Overiew of our approach

3.3.1 Clustered Don't Care

The approximate reachability techniques known to us are based on state space decomposition [19, 48, 35]. The results of these analysis are typically in the form of a list of BDDs, ${R^+}_1, \ldots, {R^+}_n$, each of which corresponds to a superset of reachable states associated with the state space of each cluster. The conjunction of all ${R^+}_i$'s yields the smallest superset of the reachable states associated with the entire state space (assuming that ${R^+}_i$'s cannot be further refined). Thus, its complement yields the largest DC set available. In practice, however, the conjuncted BDD can be prohibitively large. Alternatively, it is possible to individually apply BDD minimization to each TR_i (or each T_i if the T_i is minimized) using their respective ${R^+}_i$ as the care set. However, this approach may lead to poor minimization because the DC fraction obtained from individual ${R^+}_i$ may be very small. This suggests that we need to derive DC sets with reasonably high DC fractions and affordable BDD size.[1]

We propose a c*lustered DC* approach, in which we construct a BDD C_i representing the care set associated with TR_i by conjuncting ${R^+}_j$'s which are likely to improve the minimization quality for TR_i. We identify ${R^+}_j$*'s* for each TR_i such that the support set of ${R^+}_j$ includes at least one variable that is also contained in the support set of TR_i. Then we existentially quantify out the nonsupporting variables of TR_i from such ${R^+}_j$'s to further reduce their size. This means that the support set of C_i is a subset of the support set of TR_i

1. An analogy can be found in logic synthesis area where the satisfiability DC set of the whole network is too large for two-level minimizer [69] so a subset of the DC set is used instead.

and consequently that C_i is typically much smaller than TR_i. Figure 3.4 shows the algorithm to build clustered care BDDs whose complements are clustered-DCs.

Build-clustered-Cs ($\{TR_i\}$, $\{R^+{}_j\}$)
  for $i = 1$ to m /* m : number of TR_i */
    C_i = bdd_one
    for $j = 1$ to n
      if support(TR_i) $\cap$ support(TR_j) $\neq \emptyset$ **then** $C_i = C_i \cdot (\exists_{x \notin TR_i} R^+{}_j)$
  return ($\{C_i\}$)

Figure 3.4: Algorithm to build clustered-Cs

We note that it is also possible to use the transitions *to* unreachable states as DC transitions during a traversal iteration. The problem here is that the incorrect states may be added during the traversal which must be filtered out at the end of each iteration, causing significant computational overhead. We refer the reader to [38] for more details.

3.3.2 Iterative Approximate FSM Traversal

The main reason for the computational efficiency of the approximate reachability analysis is that each sub-FSM represented by a cluster is traversed separately. The separated traversal, however, introduces some loss of information concerning the interaction among sub-FSMs. If we decompose an FSM into too many sub-FSMs, the information loss is increased leading to a coarse approximation. Consequently, there is a trade-off between the computational efficiency and the quality of the approximation and existing clustering techniques ensure that each cluster size is smaller than a predetermined threshold [19, 35].

We propose to improve the quality of the approximation using BDD minimization. In particular, we propose to derive DCs from one approximate reachability analysis, minimize the original TR_i's using these DCs, re-cluster these minimized TR_i's, and repeat approximate reachability analysis using the new clusters. Because the TR_i's are smaller, the re-clustering algorithm can lead to fewer clusters and consequently the approximation can be improved, leading to more DCs. The larger set of DCs often leads to better minimization of the TR_i's. Therefore, we propose to repeat this process until no significant improvement is made.

To quantify improvement we use the number of clusters as our cost function rather than the exact DC fraction, because the exact DC fraction (that requires the conjunction of all ${R^+}_i$'s when the state space of each ${R^+}_i$' is not mutually disjoint) is computationally expensive or infeasible to compute. Consequently, the process is completed when the number of clusters obtained is not reduced.

Another important factor determining the quality of the approximation is the effectiveness of the state decomposition. The basic idea of existing state decomposition techniques [20, 35] is to combine *closely related* T_i's into one cluster where the closeness of the relationship between two T_i's is estimated by examining the number of common variables in their support sets. BDD minimization can indirectly lead to better state decomposition because it sometimes removes some variables from the support sets of BDDs leading to a more accurate analysis of the relationships among T_i's.

It is obvious that the choice of a BDD minimization heuristic is important. We use *restrict*[2] [24] because it is fast and is competitive in terms of both size minimization and support set minimization. For example, *constrain* [24] has the problem of introducing variables to the BDD to minimize from its care BDD as a side effect. Compaction algorithms [39] are typically better than *restrict* in terms of minimization power but worse in terms of support set reduction. This is because compaction algorithms apply a node minimization operation, called *sibling-substitution*, less aggressively than *restrict* does to prevent overall BDD size growth. There exists even more aggressive minimization heuristics presented by Shiple et al. [73] but they are prohibitively slow for large BDDs such as the TRs and clustered DC BDDs.

Finally, we note that reduced support sets of the sub-FSMs can also lead to faster run-times in the approximate FSM traversal. This is because if fewer sub-FSMs have support sets that intersect with the support set of a sub-FSM, fewer sub-FSMs need to be re-traversed after the sub-FSM is traversed [19]. This typically means that the total number of sub-FSM traversals needed is reduced, reducing overall run-time.

3.3.3 Exact FSM Traversal Using Support Set Minimization (SSM)

Using the DCs obtained from the above approximate reachability analysis, we propose to minimize the clusters used in the exact traversal. This has two effects. First, the smaller TR size can directly lead to reduced run-times because the TR*s* are heavily used. Second, BDD minimization can minimize the support set of each cluster which can lead to a better

---

2. To be precise, we use *thresholded restrict* [39] which returns the smaller BDD from the result of restrict and the original BDD.

Iterative-approximate-traversal $(\{TR_i\},\ S^0$: initial states) {

$\{R^+_0\} = \{$bdd_one$\}$, $\{C_0\} = \{$bdd_one$\}$, *n_clusters* = infinity

do {

old_n_clusters = *n_clusters*

$\{C_i\}$ = build-clustered-Cs $(\{T_i\}, \{R^+_j\})$

$\{T_i\}$ = minimize-T_i-using-DC $(\{T_i\}, \{C_i\})$

$\{TR_k\}$ = build-clusters $(\{T_i\})$

$\{R^+_l\} = \{R^+_l\} \cup$ standard-approximate-traversal $(\{TR_k\}, S^0)$

n_clusters = *number_of_clusters*$(\{TR_k\})$

} **while** (*n_clusters* < *old_n_clusters*)

return $\{R^+_l\}$

}

Figure 3.5: Algorithm for iterative approximate traversal

early quantification schedule. This reduces the maximum number of variables that an intermediate result can have, leading to reduced memory requirements. We first consider using *restrict* as the TR minimization algorithm and then explore a new heuristic specifically targeting support set minimization.

The choice of minimization algorithms is important. Ideally we want an algorithm that minimizes the support set while still reducing BDD size. Unfortunately, there is typically a trade-off between BDD size and support set size, both cannot be minimum simultaneously. Our experiences suggest that the more important factor is the support set size because the quality of the early quantification schedule often dictates the size of the systems that can be analyzed.

We first apply *restrict* because it heuristically minimizes the support set without increasing the BDD size and it is fast. Then, we apply a more expensive new heuristic that specifically tries to reduce the support set size.

Our heuristic is based on the notion of a *nonessential* variable. Given an incompletely specified function *ff* represented by f and c, where f is a cover of *ff* and c is the care-function of *ff*, we call a variable x nonessential if $f_x \oplus f_{\bar{x}} \subseteq \bar{c}_x + \bar{c}_{\bar{x}}$. A nonessential variable can be removed from the support set of the function by reassigning some function values on don't care points, e.g. $f' = f_x c_x + f_{\bar{x}} c_{\bar{x}}$ is a cover f whose support set does not include the nonessential variable x. Note that f' has the same function values of f on all care points [73]. Notice however that it may not be possible to simultaneously remove all nonessential variables from a cover because different nonessential variables may require conflicting DC assignments.

Our heuristic removes nonessential nodes greedily in each cluster in order from last cluster to first cluster. The first step is to compute the *benefit* of removing each supporting variable in a cluster. Let's consider a TR consisting of n clusters, TR_1 to TR_n, and a non-essential variable x of a cluster TR_i. If x is in the support set of any cluster whose index is larger than i, that means removing x from TR_i will not improve early quantification schedule under current cluster ordering. In that case, the benefit of removing x is described as 0. However, if v is not in the support sets of all clusters with index larger than i, removing x from TR_i will allow x to be quantified out right after the last cluster TR_k with

x in its support set is multiplied to intermediate results. The benefit of removing x is i - k + 1 in this case.

The second step is to actually remove non-essential variables following the order determined by the benefit computed in the previous step. We first check if the variable with the highest benefit is essential for the cluster. If it is nonessential, we remove the variable and move on to the next variable with highest benefit.

Note that removing a nonessential variable from a cluster implies that some DCs must be fixed to boolean values. Consequently, the care set of the cluster is expanded once a cover replaces the original function. In particular, when we remove a nonessential variable x from a cluster f, we produce a new cover $f' = f_x c_x + f_{\bar{x}} c_{\bar{x}}$ with the new care set $c' = c_x + c_{\bar{x}}$. With the updated cover and care set, we repeat the process of checking and removing the next best nonessential variable.

Note also that the size of the cover with reduced support set can be larger than the original cover. For a nonessential variable with benefit larger than 0, if the size of the cover is smaller than the maximum cluster size, we remove the variable because it leads to direct early quantification schedule improvement. If the size of the cover is larger than the maximum cluster size, we choose not to remove this variable and instead consider removing the next best nonessential variable. For a nonessential variable with benefit 0, we remove the variable only if the cover size is smaller than the original cluster because the benefit 0 means we cannot achieve direct early quantification schedule improvement. If the cover size is smaller, we remove the variable because it leads to smaller cluster as well as potential improvement on early quantification schedule under reordered clustering.

After all clusters are processed, we re-order the clusters to further improve the early quantification schedule. Thus, in this case the best choice of which nonessential variable to remove is not obvious and we are currently exploring alternative heuristics.

Figure 3.6 shows the complete flow of our approach which we call support set minimization (*SSM*) based exact reachability analysis.

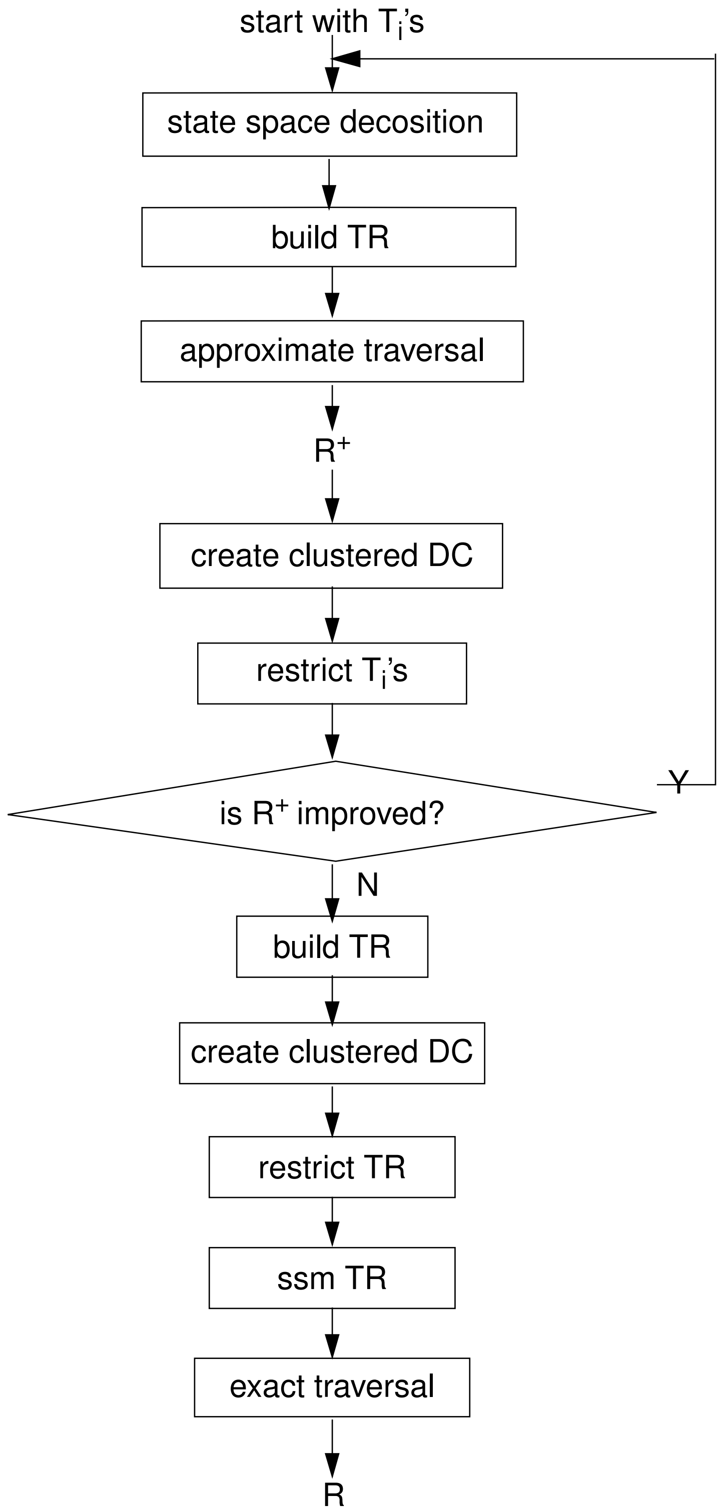


Figure 3.6: Complete flow of support set minimization (SSM) based exact reachability analysis.

3.4 Applicability of Our Techniques to Existing Traversal Algorithms

A salient feature of our *approximate reachable states based TR minimization technique* is that it is orthogonal to most existing TR based reachability analysis algorithms. We explain how to apply our techniques to minimize various types of transition relations, i.e. monolithic TR, conjunctive partitioned TR, and disjunctive partitioned TR, and discuss their impacts.

The application of DC-based minimization to monolithic TR is straightforward. The support set minimization is not helpful in this case because no early quantification is used in monolithic TR based traversal. issue. Yet we can still expect performance improvement due to the smaller TR size.

The conjunctive partitioned TR based traversal algorithms form a major category in reachability analysis and many state-of-the-art traversal algorithms [61, 15, 55] have been developed. A common idea of these algorithms is to extract subsets of the state sets that are representable with small BDDs and use the subsets as frontier sets. We may store the clusters on hard disk to better utilize main memory but the functionality of TR is not modified. Therefore our DC-based BDD minimization techniques can be applied to conjunctive partitioned TR without any modification.

We also note that the method proposed by Touati et al. [76] using implicit conjunction of T_i's to represent TR can also be improved using our techniques. They first multiply frontier set with each T_i and conjunct the results to form the final results in a balanced binary tree fashion. During this process, they existentially quantify out all variables from a partial product when all the BDDs that depend on the variables are combined into the partial product. We can derive the clustered-DC for each partial product and use it to minimize the support set size of the partial product in order to maximize the number of that we can existentially quantify out.

The disjunctive TR originally introduced to model asynchronous circuits [12] is later extended to model synchronous circuits [14]. The basic idea is to represent TR by the list of $TR \cdot p_1, \ldots, TR \cdot p_k$ implicitly disjuncted, i.e. $TR = TR \cdot p_1 + \ldots + TR \cdot p_k$, with $\Sigma\, p_i = 1$. During reachable state computation, the existential quantification is performed independently for each partitioned TR. That is support set minimization of each partitioned TR is not useful in this case. However the size of each partition will be still a major concern and the DC based minimization technique can still improve the performance.

3.5 Experimental Results

We incorporated the new techniques into VIS-1.2 [7] using Long's BDD package [50]. We conducted experiments on the reachability analysis of large examples taken from ISCAS 89 and ISCAS-Addendum 93 benchmark circuits. All the experiments were conducted on a SUN UltraSPARC 3000 / 168MHz. The data-size limit was set to 128Mb. We used all default parameter setting provided by VIS including image_cluster_size of 5000 and the clustering heuristic described in [62] for both approximate and exact reachability analysis. The sifting algorithm [65] was used for dynamic variable ordering. Dynamic variable ordering was enabled at all times except during approximate reachability analysis. Because different variable ordering typically leads to different clusterings, dynamic variable ordering during iterative approximate traversal can yield better approximation results. For these experiments, however, we disabled it so that we can accurately estimate the contribution of BDD minimization.

The results from approximate reachability analysis are given in Table 3.1. We used Cho et al.'s MBM traversal [19] as the standard approximate traversal algorithm. The column $\boldsymbol{R^+}$ represents the percentage of approximate reachable states over the entire state space, ***Time*** denotes the run-time in CPU seconds, and ***# Iter*** reports the number of standard approximate traversals our approach performed. In one case, we could not compute the R^+ fraction exactly because R^+ was too large to build. In this case, denoted with a *, we used an upper bound of the approximate reachable states computed using the method described in [35]. The last column describes the improvement factor by showing the ratio of the R^+ obtained by the standard approximate traversal over the R^+ obtained by our iterative approximate traversal.

The results show that our iterative approach produces significantly better results in terms of approximation quality with minor run-time overhead (two times slower than standard traversal in the worst case). Note that the run-times for the approximate traversals are still very small fraction of typical run-time needed for an exact traversal of a large FSM.

Before we show the results from exact reachability analysis, we report the size of TR and the number of current state variables from the TR used in each traversal in Table

Circuit	MBM		Iterative			MBM / Iterative
	R$^+$ (%)	Time	R$^+$ (%)	Time	# Iter	
s1423	0.608	202	0.424	225	3	1.434
s3271	5.32	115	4.520	142	3	1.178
s3330	7.147	119	1.98e-3	140	4	3.61e3
s4863	1.16e-3	490	2.25e-4	548	4	5.167
s5378	4.33e-9	261	4.31e-8*	519	4	> 10.00

Table 3.1: Comparison between standard and iterative approximate traversals. (* indicates an upper bound)

2.2. The column *VIS* represents the standard exact traversal and SSM represents our new traversal using both *restrict* and our support set minimization heuristic. The parenthesized number shows the data from the traversal using *restrict only.*

The results show that the TRs used in SSM are significantly smaller in size and have fewer current state variables than the TRs used in the standard traversal. The reduction of current state variables is largest for the three biggest examples whose DC fractions are much higher than those for the smaller examples.

Circuit	TR Size		# Current State Vars	
	VIS	SSM (restrict only)	VIS	SSM (restrict only)
s1423	20,039	12,277 (12,277)	237	235 (235)
s3271	14,305	11,654 (11,654)	252	247 (247)
s3330	18,099	19,353 (17,797)	167	162 (167)
s4863	125,206	35,001 (35,344)	508	395 (417)
s5378	43,987	19,141 (18,131)	397	347 (377)

Table 2.2: Comparison of TRs used in each traversal method.

The results from exact reachability analysis are given in Table 2.3. The column denoted ***# Level*** reports the number of iterations and * symbol indicates the entire traversal successfully completed. The last three columns report the run-times taken only for exact traversals, i.e., excluding clustering and approximate traversal.

The results show that SSM outperforms standard traversal in all aspects. SSM traversal successfully completes the entire traversal for s3271 while the standard traversal stops after 7 of 17 iterations. The SSM traversal runs out of memory after 7 iterations (6 iterations using *restrict* only) of s5378 while the standard traversal runs out of memory after 4. For all examples, SSM takes significantly less time than the standard traversal to complete the same number of iterations.

Notice that for the examples s3330 and s5378 SSM with support set minimization yields significantly better results than SSM using the "restrict only" method even though the support set minimization increased the BDD sizes significantly. This suggests that the impact of support set minimization on the early quantification schedule may be more critical than the BDD sizes in determining the overall run-time and memory requirements.

Circuit	#Level	# States	Traversal Time		
			VIS	SSM	(restrict only)
s1423	11	7.991e11	27,020	22,424	(22,420)
s3271	7	4.129e22	5,873	5,269	(5,273)
	17*	1.318e31	space out	13,070	(13,065)
s3330	9*	7.278e17	18,002	10,134	(16,811)
s4863	5*	2.191e19	50,201	7,120	(8,387)
s5378	4	2.393e13	17,802	7,900	(8,371)
	6	2.470e20	space out	10,040	(11,020)
	7	1.165e21	space out	19,040	(space out)

Table 2.3: Comparison on traversal results. (* indicates complete traversal.)

3.6 Conclusions

We have presented techniques to improve approximate and exact reachability analysis using DCs. The basis of the techniques is that approximated reachable states can be used as DCs to simplify the state transition representation of the FSM to be analyzed.

We have demonstrated the effectiveness of the techniques on a standard reachability analysis framework. The experimental results show that, compared to the traditional approach, our techniques can significantly reduce the reachability analysis run-time for large FSMs and explore more states.

Chapter 4

Don't Care-based BDD Minimization for Embedded Software

We explore the use of don't cares in software synthesis for embedded systems in this chapter. Embedded systems have extremely tight real-time and code/data size constraints, that make expensive optimizations desirable. We propose applying BDD minimization techniques in the presence of a don't care set to synthesize code for extended Finite State Machines from a BDD-based representation of the FSM transition function. The don't care set can be derived from local analysis (such as unused state codes or don't care inputs) as well as from external information (such as impossible input patterns). We show experimental results, discuss their implications, the interaction between BDD-based minimization and dynamic variable reordering, and propose directions for future work.

4.1 Introduction

Embedded systems are informally defined as a collection of programmable parts surrounded by ASICs and other standard components, that interact continuously with an environment through sensors and actuators. The programmable parts includes micro-controllers and Digital Signal Processors (DSPs).

Real-time and cost constraints imply that manual optimization (often at the assembly code level) of software executed on the programmable parts has been considered imperative for this class of systems. However, the growing complexity of such systems and the ever-shrinking time-to-market requirements now mandate a higher level of abstraction for specification and design. These constraints and trends explain the resurgence of interest in *domain-specific software synthesis techniques*. Such techniques exploit knowledge of the target domain, as well as the limitations of embedded systems (e.g., absence of virtual memory and tight memory space severely limit programming techniques such as liberal use of recursion, “wild” pointers and so on) to achieve a quality that is comparable with expert designers, at a fraction of the programming time cost.

Software synthesis is especially popular in the Digital Signal Processing domain, where “code stitching” of highly optimized library function kernels allows the designer to quickly simulate, prototype and implement in hardware and software sophisticated data processing applications from Data Flow network specifications (see, e.g., [4, 63]). Code synthesis for control-dominated applications, in which rapid reaction to external events and complex decision mechanisms dominate over pure data computations, require software or hardware to be efficiently synthesized from Finite State machine-like specifications [5, 37, 22, 23].

The aim of this chapter is to exploit don’t care information (coming, e.g., from impossible or irrelevant conditions -- limited controllability) in software code synthesis.

The use of DCs, to the best of our knowledge, has been limited so far only to hardware synthesis [28, 54].

Note that some compiler optimizations, such as variable lifetime analysis, constant value propagation, and so on [1] can be considered a form of DC exploitation. For example, avoiding assigning a variable that is not read before being assigned again, is exploiting a form of "observability DC", just as the elimination of an **if** statement with a constant condition is exploiting a form of "controllability DC". However, we use DCs in a much more global and systematic sense than standard compilers that often take the *control flow* as given, and perform only global *data flow* analysis (if any).

The software synthesis technique that we use in order to exploit DCs is based on the use of Binary Decision Diagrams (BDDs) [10] to synthesize software (in particular C code) from a specification in the form of Finite State Machines extended with integer arithmetic capabilities [18, 3]. That technique uses a direct mapping between BDD nodes and low-level C statements in order to derive a highly optimized implementation of the FSM transition relation. We use this synthesis path because it is publicly available as part of an embedded system design tool called POLIS [3].

The classes of DCs that we consider in this chapter are as follows:

1. Internal DCs, arising from the FSM itself. In particular, we consider DCs that arise from setting FSM inputs that are irrelevant in a given state to a fixed and known value (all other combinations automatically become DCs).

2. External DCs, arising from the FSM environment. Such DCs generally belong to two categories:

(a) input (also called “controllability”) DCs, due to impossible input combinations. In particular, we consider input combinations that are specified as impossible by the designer, or input combinations that are forbidden by the specification language semantics.

(b) output (also called “observability”) DCs, due to output combinations that are ignored by the following FSMs.

In this work we apply several different BDD minimization algorithms from the literature to DCs arising from various sources among those listed above, for the purpose of *minimizing code size* while preserving the execution speed.

The chapter is organized as follows: relevant background material is presented in Section 4.2, our application of BDD minimization in Section 4.3, results in Section 4.5, and conclusions and future work in Section 4.5.

4.2 Background

This section summarizes some previous results that are used in the rest of the chapter. The reader is referred to [3] for a complete explanation of notions and algorithms.

4.2.1 Multioutput Functions

Multioutput functions (or, equivalently, sets of single-output functions on the same domain) can be represented by their *characteristic functions*. A single-output binary-valued function $\chi_f : (X \times Y) \to \{0,1\}$, where $X = X_1 \times \ldots X_m$ and $Y = Y_1 \times \ldots Y_l$, represents the multioutput multivalued (MV) function $f : X \to Y$ if $\chi_f (x, y) \Leftrightarrow (y = f(x))$.

4.2.2 Co-design Finite State Machines

Co-design Finite State Machines (CFSMs) are Extended Finite State Machines interconnected asynchronously by means of one-place buffers. We consider only the function of a *single CFSM* although more powerful DCs could be extracted from a *network* of CFSMs.

A CFSM is a reactive Extended Finite State Machine with a set of input and output events. States of the CFSM are events that are fed back. Events are entities that may occur at determinate instants of time and may or may not carry a value. (cfr. VHDL events, that however always carry a value, and Esterel signals).

For example, consider a temperature event carrying the value every time a new sample arrives, and a reset event that has no value. For the purpose of this discussion, each event is associated with

- a binary-valued variable (the event buffer) which is true in the time interval between the emission (by the environment or a CFSM) and the time the CFSM makes a transition, and
- an optional discrete-valued variable carrying its value.

The function of a CFSM is specified by its *transition function*, that for the purpose of this work will be considered as a set (a disjunction) of transitions. Each CFSM *transi-*

tion has a *pre-condition* for determinism, pre-conditions must be disjoint) and a *post-condition*.

The pre-condition is the conjunction of a set of:

- input event presence or absence conditions, and

- Boolean-valued expressions over the values of its input signals (including present state conditions).

The post-condition is the conjunction of a set of

- output event presence or absence conditions (presence implies emission, absence implies no action), and

- expressions assigned to output data signals (including next state assignments).

Given a BDD representation of the characteristic function of the set of transitions of a CFSM, the technique of [18] derives a fragment of C code, as follows:

- A BDD node associated with an input variable becomes an **if-then-else** statement, branching to the statements corresponding to the BDD children nodes.

- A BDD node associated with an output variable becomes an assignment of the value corresponding to the *single child* with a path to the BDD leaf labeled with 1, followed by a branch to the statement corresponding to that child.

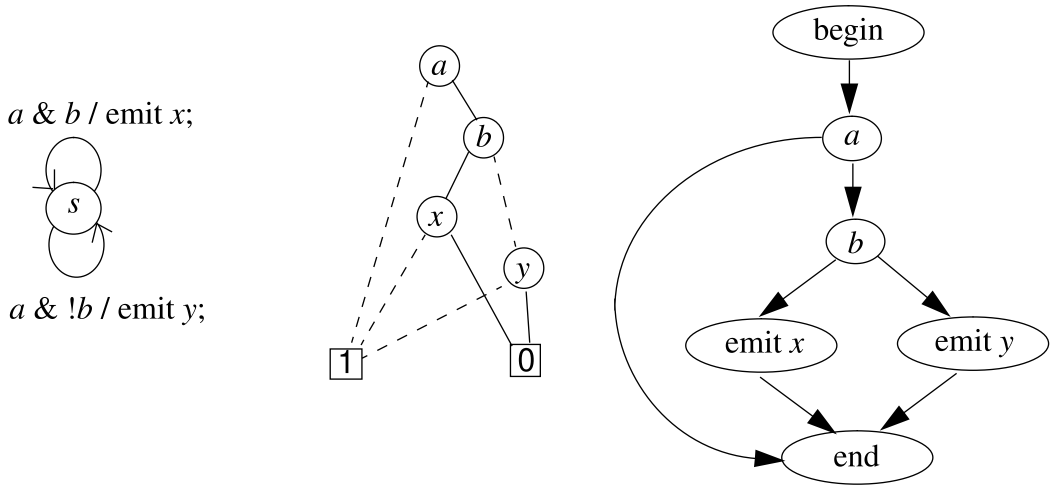
The fragment of C dode is generated from an intermediate representation called software-graph (S-graph) [3] and an example of generating s-graph from a given CFSM is shown in Figure 4.1. This technique produces *extremely fast* code (each input variable is tested at most once in an execution of a transition of a CFSM) with a reasonable size (comparable with good quality hand-written code [3]). For example, it is faster and more compact than code generated in a more standard way from a CFSM, by producing two levels of **switch** statements (one based on the state, one on the inputs) followed by a sequence of assignments.

Another very important characteristic of this technique is that, since BDD nodes and code fragments are almost in 1-1 correspondence, the BDD size becomes an excellent *predictor* of the code size.

4.3 Don't Care-Based BDD Minimization for Embedded Software

4.3.1 Internal Don't Cares

The first class of DCs that we consider comes from the structure of the transition function of the CFSM itself.



(a) A CFSM *f* (b) A BDD for *f* (c) S-graph for *f*

Figure 4.1: An example of s-graph generation from a CFSM description

Before giving a more precise definition of this transformation, we offer an intuitive explanation. The transition function can be described as a table of transitions, each with an input and an output part. A '-' in the input part is shorthand for specifying several input minterms for the same transition; a '-' in the output part is a DC. Our first minimization procedure effectively transfers a '-' in the input part to a '-' in the output part. This works as follows. Given a transition with a '-' for a particular input, for example *ab-d*, if we can guarantee that the value for that input is always fixed for this cube, for example *abcd*, we know that none of the other values ever occur, in this case $ab\bar{c}d$, and this is a DC. In order for this optimization to be correct, all input minterms contained by this cube must be mapped to this distinct identifier *before* calling the CFSM.

In our case, in order to preserve the specified CFSM behavior, this optimization strategy assumes that we make a simple modification to the RTOS. That is, we force the RTOS under certain conditions to feed only some input combinations to the CFSM; the remaining input combinations become DC conditions.

Specifically, we assume the RTOS uses a state-based mask which sets certain MV-input signals to a user-definable fixed value in a given state. Once a mask is defined, the BDD representation of the characteristic function of a CFSM can be minimized by taking into account that for that state, other values of the MV-input signal are not possible. Thus, the behavior of the CFSM for such an impossible input assignment becomes a DC.

Consider an MV-input variable i with possible values whose range is $I_i = \{i_1, i_2, \ldots i_l\}$. The transition relation f is *independent* of i in state s if $(f \cdot s)_{i_k} = f \cdot s$ for all $1 \leq k \leq l$ (where $(f \cdot s)_{i_k}$ denotes $f \cdot s$ evaluated for i_k). Thus, in this state the behavior of f does not depend on the value of i. We can utilize this independence by making the RTOS fix the value of i in this state. Then, the other state-value combinations become DCs with which f can be minimized.

Formally, the mask M is a function from states and inputs variables to the union of MV-values and a special *do not change value symbol* '-', as follows: $states \times i \rightarrow I_i \cup$ '-'. For example, for a given state s and input variable i, $M[s, i]$ can be set to a user-specifiable constant $a_{s,i}$, where $a_{s,i}$ is a possible value of i, $a_{s,i} \in I_i$. For those combinations of input variables and states where the MV-input signal should not be altered by the RTOS, the mask evaluates to '-'.

Our algorithm to compute the mask and the corresponding DC set is shown in Figure 4.2.

Find_DC_and_Mask (T: characteristic function)

/* D: resulting DC function, d: a DC cube */
$D = 0$
foreach state s
  foreach input signal i
    $d = 0$
    if (T is independent of i in s) **then**
      $M[s,i] = a_{s,i}$ /* $a_{s,i}$: the user-defined value to assign to i in s */
      $d = \text{complement}(a_{s,i})$
      $D = D + d$

Figure 4.2: Computing the DC set and the Mask

Example 1. Consider the following example transition table with the BDD representation for its characteristic function. s is a current state variable, s' is a next state variable, and i is an input variable.

s	i	s′
0	–	1
1	0	1
1	1	0

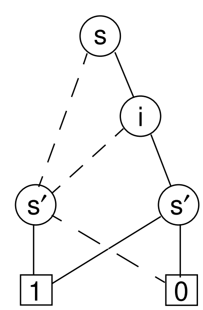


Figure 4.3: An example transition table and its BDD representation

The next state assignment for state s = 0 is independent of the binary-valued input signal *i*. Thus, we can set *M*[0, *i*] to an arbitrary value, for example 0. This means This means the operating system will set ‘i’ to 0 in state *s* = 0, so the CFSM will never see the input combination *s* = 0, *i* = 1 and thus this combination becomes a don’t care condition for all outputs. Notice also that the next state assignment for state *s* = 1 is not independent of *i*. The care set is expressed in the following table. The BDD for its characteristic function is given to the right.

s	i
0	–
1	0
1	1

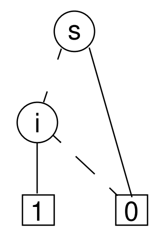


Figure 4.4: Care-set table and its BDD representation

Using this care function we can perform BDD minimization using *osm_bt* [26] or GS-compaction [29] and obtain the reduced BDD for the transition relation. Notice that this BDD has one less node than the original, which leads to a reduction in estimated software cost. The reader can verify that this reduced BDD is smaller than any BDD obtainable by dynamic variable ordering alone. ❏

The cost of this optimization is the overhead in the RTOS required to implement the mask *M*. Fortunately, in POLIS the variables are bit-packed. Thus, we can typically create the mask with an array of integers, one integer per state, and perform the input presence test by using that mask at the beginning of every CFSM transition, at an almost insignificant cost.

4.3.2 External Don't Cares

Another class of DCs that we explored is external *input* DCs. These DCs indicate that some input event combinations are impossible, either due to environmental constraints, or due to the functionality of other CFSMs. While it would be possible in principle to automatically extract such DCs due to other CFSMs from the CFSM network specification, in this work we consider only designer-specified input DCs embedded in the specification language. In particular, we focus on two input languages: Esterel [6] and SDL [67].

Esterel is a textual specification language for reactive systems, whose semantics is defined in terms of Extended Finite State Machines, and that is used as an input language for POLIS. An Esterel specification may contain relations that indicate limited controllability on the inputs. For example, the Esterel statement

relation a # b # c

indicates that none of inputs a, b, or c occur simultaneously. For any function of these variables, a valid DC set is $DC = ab + ac + bc$. The use of the '#' operator for specifying limited controllability does not allow the specification of *any* conceivable DC set; however, it is convenient for the reactive control-type designs that are designed using Esterel.

In the current Esterel compiler, which is used as a front-end by POLIS and hence by our BDD minimization procedure, these relations are taken into account during simulation (an error is signaled to the user if one of these conditions is given as an input), and during causality analysis. However, the compilation path does not by default use these relations to optimize the implementation.[1] Furthermore, in the Esterel compiler there is no exploration of and optimization using both sifting and relation-based minimization.

SDL is a graphical and textual language for telecommunication systems, especially suited for specification of network protocols. SDL also has an EFSM semantics (but it cannot currently be used as an input to POLIS). The semantics, however, requires that input signals are delivered to each EFSM *one at a time*, through a priority queue. This means that for a CFSM hypothetically specified in SDL, any condition with no input events or more than one input event present is DC (the former due to the reactivity of CFSMs, that are not activated unless they have at least one event, the latter due to the SDL single-event constraint). Since POLIS currently does not allow specification of CFSMs in SDL, we have analyzed the effect of BDD minimization on this kind of specification by using an Esterel relation that states that all CFSM inputs are mutually exclusive.

4.4 Experimental Results

In this section we report the results from the application of various minimization algorithms to the classes of DCs discussed in the previous section. We report minimization results as a single column (with the best result among all the algorithms), since there was no significant difference in performance or quality of results (it was at most 2%). We also report only examples for which we could find a significant effect of minimization. Such examples are generally characterized as having a *high degree of con-*

1. One can use these relations in optimizing the control portion of the design after separating it from the data portion, but this optimization is not yet fully automatic.

currency and symmetry, so that the external DCs discussed in Section 4.3.2 can be especially effective.

In the following sections we describe the results of exploiting the external DCs from Esterel described in Section 4.3.2. In all cases we specified that at most one input event could be delivered to a CFSM at any given time. This, of course, caused a big reduction in size of the 12-element Distributed Mutual Access element modeled in **arbiter12**.

The main point, however, is to show that BDD minimization using *external DCs* is orthogonal to other minimization opportunities, e.g., by BDD variable reordering. Hence, they should be used whenever an appropriate DC set is available from the designer or from the language semantics.

On the other hand, the class of *internal DCs* described in Section 4.3.1 interacted heavily with minimization by dynamic variable ordering, specifically *sifting* [65] in our experiments. This is the reason why we do not report any results about that class of internal DCs in this work. In particular, DC minimization *before* sifting (i.e., with the initial variable ordering in which all inputs appear before all outputs) reduces the BDD size by 20% on average and the code size by 10% on average. However, it also prevents sifting from obtaining the 50%-200% improvement it can achieve alone. This may be due to the effect of sifting on a CFSM in which some states are independent of some inputs. In that case, a good ordering will most likely put the present state variable at the top of the characteristic function BDD, and then the BDD will have one distinct branch for each state. Of course, DC variables will not occur in those branches, and the minimization effect due to them will be achieved.

It is not clear, though, if the absence of positive effect in the case of internal DCs is only due to the particular type of specifications that we considered in our benchmark set.

4.4.1 Optimization within Esterel

The next set of experiments was run to see how much improvement in the size of the C code could be obtained using external DCs with the current Esterel technology alone. Here, we can compare the percentage gains with the percentages we obtain using a combination of reordering and BDD minimization.

For each example under test, three different C files were produced: one for the original Esterel example, one for the original after optimization, and one for the original with a relation added and with subsequent optimization. To use optimization in Esterel, one must first extract the control portion using the Esterel compiler, run a logic optimization tool on it, and then reinsert this optimized control part and continue the Esterel compilation to C code. The optimization that is available through the Esterel compiler is a well-tested set of scripts that uses the SIS sequential logic synthesis program [70] in conjunction with the **rem_latch** program [71, 72] which contains more sophisticated algorithms for latch removal. The relations that were added were always of the nature of severely restricting the inputs to arrive one at a time -- that is, all inputs are assumed to be mutually exclusive.

The results of these experiments are shown in Table 4.1. The C code sizes are given in bytes. Modest reductions in the size of the C code when optimization and optimization with a relation present are observed. In the case of the arbiter, clearly, the reduction is considerable.

Name	Original	Opt	Rel & Opt
abcd	16673	15903	14767
abcdef	24450	23329	22593
arbiter12	55439	18438	18442

Table 4.1: Effects of DCs on Esterel C-code generation

4.4.2 Optimization within POLIS

The next set of experiments shows the results of applying DC-based BDD minimization *after* performing minimization by sifting. In all cases, we report the number of basic C statements (**if-then-else** and assignments), as it tracks the final code size very well especially in cases, such as the examples we used, in which there is no data computation (note: *this table does not use the same units as Table* [4.1]).

The meaning of the columns is as follows:

No Esterel Opt does not include the effect of Esterel optimization, as described in the previous section.

Esterel Opt includes the effect of Esterel optimization, as a first step *before* BDD construction. Note that, even though Esterel optimization reduces the code size produced by the Esterel compiler, this may not change the function at all, but only modify the Boolean network structure. Hence the BDD size may not change at all.

No DC Opt shows the result of POLIS software synthesis without exploiting DCs.

DC Opt shows the result of POLIS software synthesis after the best DC-based BDD minimization.

Examples whose name ends in **_rel** had external DCs already specified in Esterel. A simple conclusion that can be drawn from this table is that in general it is advisable *not* to use the DCs in the Esterel optimization stage, but only in the "standard" Esterel compilation procedure, since the former may reduce further possibilities of optimization. This is probably due to the same reasons as in the case of DC minimization *before* sifting, as discussed above.

	No Esterel Opt		**Esterel Opt**	
Name	**No DC Opt**	**DC Opt**	**No DC Opt**	**DC Opt**
abcd	496	469	496	473
abcd_rel	93	63	94	85
abcdef	4023	3959	4023	3959
abcdef_rel	171	105	162	149
arbiter12	406	172	406	172
arbiter12_rel	406	172	197	195

Table 4.2: Effect of DCs on POLIS C-code generation

4.5 Conclusions

It is clear that there is an important relationship between DC-based BDD minimization and BDD variable ordering. Minimization is potentially more powerful from the standpoint that it explores a *set* of possible functions; reordering changes only the structure of the representation of a single function. Our experiments reveal that while minimization is potentially more powerful, variable ordering has such a strong influence on BDD size that fixed-order minimization must be considered a secondary optimization technique. We expect the observed results to vary widely with the application domain and the nature of the derived DCs; this is an analogy to the observation that BDD size varies widely with the type of BDD employed and the type of function represented.

Chapter 5

Conclusions and Future Work

We have presented BDD minimization heuristics using don't cares and the use of don't cares in reachability analysis and embedded software synthesis applications. This chapter summarizes our contributions and discuss directions for future research.

5.1 Contributions

We first presented safe DC-based BDD minimization heuristics that guarantee the resulting BDD is not larger than the original BDD. The safe minimization heuristics are based on sibling-substitution which is also the core operation used by most traditional heuristics. The distinct feature of the safe minimization heuristics is that they perform sibling-substitution selectively while traditional heuristics apply all possible sibling-substitution blindly. To be specific, the safe minimization heuristics are designed to skip sibling-substitutions that can cause overall size growth while allowing sibling-substitutions elsewhere.

The safe minimization heuristics yield significantly smaller BDDs compared with traditional heuristics yet still require manageable run times. These heuristics are particularly useful for the BDD-based synthesis application where the structure of the hardware/ software is derived from the BDD representation of the function to implement.

Secondly, we have presented techniques to improve approximate and exact reachability analysis using DCs. The basis of the techniques is that approximated reachable states can be used to derive DCs to simplify the state transition representation of the FSM to be analyzed.

We have demonstrated the effectiveness of the techniques on a standard reachability analysis framework. The experimental results show that, compared to the traditional approach, our techniques can explore more states than traditional approaches can and significantly reduce the run time.

Another important advantage of our techniques is that they are orthogonal to most traditional reachability analysis algorithms. Our techniques are expected to easily improve the capability and efficiency of most verification tools based on symbolic reachability analysis.

Finally, we explored the use of don't cares in software synthesis for embedded systems. We proposed applying BDD minimization techniques using the don't care set can be derived from local analysis as well as from external information to minimize the size of the code synthesized. Our experimental results demonstrate the effectiveness of using DCs. To the best knowledge of ours, this was the first attempt to utilize DCs in software synthesis application.

5.2 Future Research

There are several very interesting directions to explore further. Here, we describe two that are particularly intriguing.

The study of the relationship between variable reordering and DC-based minimization is an area of completely open research. From our experience, we observed that variable ordering is more critical factor in determining the BDD size than DC assignment unless the DC fraction is extremely high. This motivates assigning DCs to binary values in such a way that the subsequent dynamic variable ordering can better perform. For example, increasing the symmetric variables using don't care information would be helpful because grouping symmetrical variables during dynamic variable ordering typically leads to smaller BDD size [57].

In the formal verification area, many approaches have been newly proposed recently (e.g. combining formal and informal verification [79] and automatic abstraction [58]). Yet, the problem of reachable state computation will still remain as a challenging and rewarding task.

We learned that the approximation quality is crucial to maximize the capability of exact analysis. Currently, all existing approximate reachability analysis methods use static state decomposition by analyzing the connectivity of each component of the FSM [19, 20, 35]. Once the state space is decomposed no attempt is made to refine the decomposition in those approaches. An interesting direction is to dynamically refine the state space decomposition based on already obtained approximation results during approximate analysis.

This approach is expected to be more powerful and robust in the sense that it is less sensitive to the initial state space decomposition compared to the traditional approach.

Bibliography

[1] A. V. Aho and R. Sethi and J. D. Ullman, Compilers: Principles, Techniques and Tools. Addison-Wesley, 1988.

[2] A. Aziz, S. Tasiran, and R. K. Brayton, "BDD variable ordering for interacting finite state machines," in *Proc. Design Automation Conference*, pp. 283 - 288, 1994.

[3] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, "Hardware-software co-design of embedded systems - The POLIS approach," Kluwer Academic Publishers, 1997.

[4] S. S. Bhattacharyya, P. Murthy, and E. A. Lee, "Software Synthesis from Dataflow Graph," Kluwer Academic Publishers, 1996.

[5] G. Berry. The Constructive Semantics of Pure Esterel. ftp: //www.inria.fr/meije/esterel/papers/constructiveness.ps.gz, 1996.

[6] G. Berry, P. Couronne, and G. Gonthier. "The synchronous approach to reactive and real-time systems," *IEEE proceedings*, 79, September 1991.

[7] R. K. Brayton and *et al.*, "VIS: A system for verification and synthesis," Technical Report UCB/ERL M95, University of California, Berkeley, December 1995.

[8] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*. vol. 78, no. 2, pp. 264-300, Februrary 1990.

[9] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. on Computer-Aided Design*, vol. CAD-6, pp. 1062-1081, November 1987.

[10] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677 - 691, August 1986.

[11] J. R. Burch, E. M. Clarke and D. E. Long, "Representing circuits more efficiently in symbolic model checking," in *Proc. Design Automation Conference*, pp. 403 - 407, 1991.

[12] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. on Computers*, vol. 13, pp. 401 - 424, April 1994.

[13] J. R. Burch, E. M. Clarke, D. Long, K. L. McMillan and D. L. Dill, "Sequential circuit verification using symbolic model checking," in *Proc. Design Automation Conference*, pp. 46 - 51, 1990.

[14] G. Cabodi, P. Camurati, L. Lavagno and S. Quer, "Disjunctive partitioning and partial iterative squaring," in *Proc. Design Automation Conference*, pp. 728 - 733, 1997.

[15] G. Cabodi, P. Camurati and S. Quer, “Improved reachability analysis of large finite state machines,” in *Proc. International Conference on Computer-Aided Design*, pp. 354 - 360, 1996.

[16] S. Chang, D. I. Cheng and M. Marek-Sadowska, “Minimizing ROBDD size of incompletely specified multiple output functions,” in *Proc. European Design and Test Conference*, pp. 620 - 624, 1994.

[17] S. Chang, M. Marek-Sadowaska and T. Hwang, “Technology mapping for TLU FPGA’s based on decomposition of binary decision diagrams,” *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1226 - 1236, October 1997.

[18] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich, “Synthesis of software programs for embedded Control Applications,” in *Proc. Design Automation Conference*, pp. 587 - 592, 1995.

[19] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, F. Somenzi, “Algorithms for approximate FSM traversal,” in *Proc. Design Automation Conference*, pp. 25 - 30, 1993.

[20] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi, “A structural approach to state decomposition for approximate reachability analysis,” in *Proc. International Conference on Computer Design*, pp. 236 - 239, 1994.

[21] H. Cho, S. Jeong, F. Somenzi, and C. Pixley, “Synchronizing sequences and symbolic traversal techniques in test generation,” Journal of Electron. Testing, Theory andn Applications, 4, 1, pp. 19 - 31, 1993.

[22] C. N. Coelho and G. De Micheli. "Analysis and synthesis of concurrent digital circuits using control-flow expressions," *IEEE Trans. Computer-Aided Design*, vol. 15, no. 8, pp. 854 - 876, August 1996.

[23] P. Chou and G. Borriello. Software Scheduling in the Co-synthesis of Reactive Real-time Systems. In *Proc. Design Automation Conference*, pp. 1 - 4, 1994.

[24] O. Coudert, C. Berthet and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Automatic Verification Methods for Finite State systems*, Springer-Verlag, pp. 365 - 373, 1989.

[25] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *Proc. International Conference on Computer-Aided Design*, pp.126 - 129, 1990.

[26] M. Damiani and G. De Micheli, "Observability don't care sets and Boolean relations," in *Proc. International Conference on Computer-Aided Design*, pp. 502 - 505, 1990.

[27] G. De Micheli, Synthesis and optmization of digital circuits, McGraw-Hill, 1994.

[28] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[29] R. Drechsler and N. Göckel, "Minimization of BDDs by evolutionary algorithms" in *Proc. International Workshop on Logic Synthesis*, 1997.

[30] R. Drechsler, N. Drechsler and W. Gunther, "Fast exact minimization of BDDs," in *Proc. Design Automation Conference*, pp. 200 - 205, 1998.

[31] G. Even, I. Y. Spilinger, and L. Stok, “Retiming revisited and reversed,” *IEEE Trans. Computer-Aided Design*, vol. 15, no. 3, pp. 348-357, March 1996.

[32] H. Fujii, G. Ootomo, and C. Hori, “Interleaving based variable ordering methods for ordered binary decision diagrams,” in *Proc. International Conference on Computer-Aided Design*, pp. 38 - 41, 1993.

[33] M. Fujita, Y. Matsunaga, and T. Kakuda, “On variable ordering of binary decision diagrams for the application of multi-level logic synthesis,” in *Proc. European Design Automation Conference*, pp. 50 - 54, 1991.

[34] D. Geist and I. Beer, “Efficient model checking by automated ordering of transition relation partitions,” in *Proc. Computer-Aided Verification*, pp. 299 - 310, 1994.

[35] S. G. Govindaraju, D. L. Dill, A. J. Hu, and M. A. Horowitz, “Approximate Reachability with BDDs Using Overlapping Projections”, to appear in *Proc. ACM/IEEE Design Automation Conference*, 1998.

[36] A. Gupta, “Formal hardware verification methods: A survey,” Formal Methods in System Design, pp. 151 - 238, Klwer Academic Publishers, October 1992.

[37] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Program Implementation Schemes for Hardware-software Systems. *IEEE Computer*, pp. 48 - 55, January 1994.

[38] Y.. Hong, P. A. Beerel, “Symbolic Reachability Analysis of Large Finite State Machines Using Don’t Cares,” in *International Workshop on Logic Synthesis*, pp. 284 - 289, 1998.

[39] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, “Safe BDD minimization using don’t cares,” in *Proc. Design Automation Conference*, pp. 208 - 213, 1997.

[40] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, “Sibling-substitution based BDD minimization using don’t cares,” submitted to *IEEE Trans. on Computer-Aided Design*, 1998.

[41] Y. Hong, P. A. Beerel. L. Lavagno, and E. M. Sentovich, “Don’t care-based BDD minimization for embedded software,” in *Proc. Design Automation Conference*, pp. 506 - 509, 1998.

[42] S. Huang and K. Cheng, “Formal equivalence checking and design debugging,” Kluwer academic publishers, 1998.

[43] N. Ishiura, H. Sawada, and S. Yajima, “Minimization of binary decision diagrams based on exchanges of variables,” in *Proc. International Conference on Computer-Aided Design*, pp. 472 - 475, 1991.

[44] S-W. Jeong, B. Plessier, G. D. Hactel, and F. Somenzi, “Variable ordering for FSM traversal,” in *Proc. International Conference on Computer-Aided Design*, pp. 476 - 479, 1991.

[45] T. Karoubalis, G. P. Alexiou and N. Kanopoulos, “Optimal synthesis of differential cascode switch logic circuits using ordered binary decision diagrams,” in *Proc. European Design Automation Conference*, pp. 282 - 287, 1995.

[46] L. Lavagno, P. McGeer, A. Saldanha and A. L. Sangiovanni-Vincentelli, “Timed Shannon Circuits: A powerful Design Style and Synthesis Tool,” Proc. Design Automation Conference, pp. 254 - 260, 1995.

[47] C. E. Leiserson and J. B. Saxe, “Optimizing synchronous systems,” in Proc. Twenty-Second Annual Symposium on Foundations of Computer Science, pp. 23 - 26, 1981.

[48] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi, “Tearing Based Automatic Abstraction for CTL Model Checking,” in *Proc. IEEE International Conference on Computer-Aided Design*, pp.76 - 81, 1996.

[49] B. Lin and S. Devadas, “Synthesis of hazard-free multi-level Logic under multiple-input changes from binary decision diagrams,” in *Proc. International Conference on Computer-Aided Design*, pp. 542 - 549, 1994.

[50] D. E. Long, A binary decision diagram (BDD) package, June 1993, Manual Page.

[51] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Retiming and resynthesis: optimizing sequential networks with combinational techniques,” *IEEE Trans. Computer-Aided Design*, vol. 13, no. 8. pp. 74 - 84, January 1991.

[52] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *Proc. International Conference on Computer-Aided Design*, pp.6-9, 1988.

[53] M. C. McFarland, "Formal verification of sequential hardware: A tutorial," *IEEE Trans. Computer-Aided Design*, vol. 12, no. 5. pp. 633 - 654, May 1993.

[54] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *Proc. Design Automation Conference*, pp. 620-625, 1990.

[55] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Reachability Analysis Using Partitioned-ROBDDs", in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 388 - 393, 1997.

[56] A. L. Oliveira, L. Carloni, T. Villa and A. Sangiovanni-Vincentelli, "Exact minimization of boolean decision diagrams using implicit techniques," Technical Report UCB ERL M96/16, University of California, 1996.

[57] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic ordering of decision diagrams," in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 628 - 631, 1994.

[58] A. Pardo and G. D. Hachtel, "Automatic abstraction techniques for propositional u-calculus model checking," in *Proc. International Conference on Computer-Aided Verification*, pp.12 - 23, 1997.

[59] C. Pixley and G. Beihl, “Calculating resetability and reset sequences”, in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 376 - 379, 1991.

[60] C. Pixly, S. Jeong, and G. D. Hachtel, “Exact calculation of synchronizing sequences based on binary decision diagrams,” *IEEE Trans. Computer-Aided Design*, vol. 13, no. 8. pp. 1024 - 1034, August 1994.

[61] K. Ravi and F. Somenzi, “High-density reachability analysis,” in *Proc. International Conference on Computer-Aided Design*, pp. 154 - 158, 1995.

[62] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier and C. Pixely, “Efficient formal design verification: data structure + algorithms”, Technical Report UCB/ERL M94, University of California, Berkeley, October, 1994.

[63] S. Ritz, M. Pankert, J. Waltenberger, V. Zivojnovik, and H. Meyr, “Code generation techniques in the block diagram oriented design tool COSSAP/DESCARTES,” in *Proc. of International Conference on Signal Processing Applications and Technology*, pp. 709 - 714, 1994.

[64] J. Rho and F. Somenzi, “Minimum length synchronizing sequences of finite state machines,” in *Proc. Design Automation Conference*, pp. 463 - 468, 1991.

[65] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” in *Proc. International Conference on Computer-Aided Design*, pp. 42-47, 1993.

[66] A. Saldanha, A. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Multi-level, logic simplification using don't cares and filters," in *Proc. Design Automation Conference*, pp. 277-282, 1989.

[67] R. Saracco, J. R. W. Smith, and R. Reed., "Telecommunications systems engineering using SDL," North Holland - Elsevier, 1989.

[68] M. Sauerhoff and I. Wegener, "On the complexity of minimizing the OBDD size for incompletely specified functions," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 1435-1437, November 1996.

[69] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting Local Don't Cares for Network Optimization," in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 514-517, 1991.

[70] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, and A. Sangiovanni-Vincentelli, "SIS: Sequential Circuit Design Using Synthesis and Optimization," in *Proc. International Conference on Computer Design*, pp. 328-333, 1992.

[71] E. M. Sentovich, H. Toma, and G. Berry, Latch optimization in circuits generated from high-level description,in *Proc. International Conference on Computer-Aided Design*, pp. 428 - 435, 1996.

[72] E. M. Sentovich, H. Toma, and G. Berry, " Efficient latch optimization using exclusive sets," in *Proc. Design Automation Conference*, pp. 8-11, 1997.

[73] T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. K. Brayton, "Heuristic minimization of BDDs using don't cares," in *Proc. Design Automation Conference*, pp. 225-231, 1994.

[74] V. Singhal, "Design replacements for wequential circuits," Ph. D. Thesis, Univeristy of California, Berkeley, 1996.

[75] H. J. Touati and R. K. Brayton, "Computing the initial states of retimed circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, no. 1, pp. 157-162, Februrary 1993.

[76] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton and A. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDD's," in *Proc. International Conference on Computer-Aided Design*, pp.130 - 133, 1990.

[77] K. Wang and T. Hwang, "Boolean matching for incompletely specified functions," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 160-168, Februrary 1997.

[78] J. A. Wehbeh and D. G. Saab, "On the initialization of sequential circuits," in *Proc. Internaltional Test Conference*, pp. 233-239, 1994.

[79] J. Yuan, J. Shen, J. Abraham, and A. Aziz, "On combining formal and informal verification," in *Proc. International Conference on Computer-Aided Verification*, pp.376 - 387, 1997.

[80] K. Y. Yun, B. Lin, D. Dill and S. Devadas, "Performance-driven synthesis of asynchronous controllers," in *Proc. International Conference on Computer-Aided Design*, pp.550-557, 1994.