Decision diagrams: Extensions and applications to reachability analysis

by

Junaid Babar

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University

Ames, Iowa

2019

DEDICATION

To Ammi.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

This PhD has been made possible by the influence of many. A “thank you” does not do justice to their contribution. I will be forever grateful and will strive to do for others what they have done for me.

First and foremost I would like to thank the members of my PhD committee—Dr. Andrew Miner, Dr. Gianfranco Ciardo, Dr. Samik Basu, Dr. Leslie Miller and Dr. Hridesh Rajan—for their guidance and patience throughout this process.

Dr. Samik Basu is an incredible combination of IQ and EQ. His ability to focus the conversation towards what is in the best interest of the student is unparalleled in my experience. He’s the person I sought out whenever I had an idea that was not yet fully formed and a bit *out-there*, because he had the ability to see through the fog. He was always encouraging of promising ideas and quick to point out a potential dead-end. He did all this with a warm smile and disarming humility.

Dr. Gianfranco Ciardo has been a tremendous influence on my PhD. His curiosity for the truth, no matter where it is hidden, is remarkable. He will not accept an imprecise statement, no matter the context, much to the chagrin of his students, including mine. He can be impossibly difficult to deal with—your conversation or presentation will come to a standstill unless you can satisfy all of his concerns. I will also gratefully, and with utmost admiration, state that had it not been for him I would likely not have unraveled the threads in existing works that revealed my most significant thesis topic.

Dr. Andrew Miner, my major professor. It is hard to acknowledge the breadth of Andy’s influence on my PhD. When I first met him, he was shy but friendly. I wanted him as my guide because he showed a passion for Model Checking and building software. He gave me enough freedom to make mistakes, and was patient enough to let me find my way again. He taught me to be responsible for myself. He has picked me up in many a desperate time. When I had taken a *break* from my PhD, he found a way to coax me back, helped me regain my confidence and to finish strong. He is a rare professor who treats his students as people who have lives outside of the lab. The quality of his doctoral thesis has been a driving factor in me aiming high for my doctoral work. I hope I have done him proud.

Dr. Gurpur Prabhu has been a mentor ever since I was his teaching assistant. His *dosas* will be fondly remembered, as will the numerous instances of timely advice I have received over the years.

Dr. Jack Lutz was my major advisor when I came to Iowa State University as a Master's student. I had a difficult first year in a new country but I got through it with his help and encouragement. I hope to someday be half as good a professor as Dr. Jack Lutz.

I would like to thank Dr. Robyn Lutz for introducing me to safety-critical systems; Dr. David Fernandez-Baca for his lectures on algorithms; and the faculty of the Department of Computer Science at Iowa State University for everything they have taught me. Mindy, Carla, Abby and Melanie, thank you for helping me sort through paperwork and for helping me find sources of funding. I also deeply appreciate the Department of Computer Science at Iowa State University for giving me the opportunity to pursue my dreams.

Thank you to Chuan, Shruti, Ben, Priyanka, Sai, Shiva, Mikaela, and to my colleagues at the Laboratory for Verification of Logics and Stochastics, for the wonderful discussions, Friday evening games and scrounging for free food.

Thank you Ranjit, Ruki, Mukkaram, Zareen, and Muiez for always looking out for me.

To my dear family: Abba, Ammi, Ambareen, Nausheen Aapa, Shahtaj Chacha, Chachijaan, Altaf Chacha, Phupu, Riaz Mama, Fahmida Khaljaan, Fazil, and many more who I cannot list, thank you for your boundless love and encouragement even when you did not quite understand what I was doing and why.

Abba and Ammi, thank you for sending me to my first Computer Science class in the summer of 1987. Riaz Mama, thank you for my first computer-programming book. Shahtaj Chacha and Chachijaan, thank you for my first computer. Thank you Mr. Amruthanathan, my math teacher at Don Bosco Matriculation Higher Secondary School, for instilling a love for all things Math. Thank you Dr. Ibrahim Shah, my mentor at Crescent Engineering College, and Javed, my project partner, for the research project that sparked my curiosity.

Thank you to my sisters, Nausheen Aapa and Ambareen, for your constant cheer-leading. Thank you Fazil for being there when no one else would.

Zara, my love, thank you for helping me dream again.

Ammi, this was all you. Everything. Your unconditional love. Your ever present smile. You've been there for me every step of the way. I dedicate this work to you.

ABSTRACT

Symbolic data structures and algorithms are increasingly popular tools for the analysis of complex systems. Given a high-level model of a system, such as a Petri Net, we can automatically verify certain properties about it. In this thesis, we develop data structures and techniques that can be used to improve such analyses.

First, we show how decision diagrams can be used efficiently in traditional *explicit* generation algorithms. Next, we show how symbolic reachability analysis can be used to detect deadlocks in Petri Nets. We also present a symbolic approach that can detect deadlocks in unbounded Petri Nets.

Finally, we introduce a new type of decision diagram, *ESRBDD*, that combines multiple reduction rules, is canonical, and produces a more compact representation than previous efforts. We show that operations on ESRBDDs are at least as efficient as those on the underlying decision diagrams and introduce extensions to ESRBDDs that improve on their compactness and operational efficiency.

CHAPTER 1. INTRODUCTION

Model Checking, introduced by Clarke and Emerson (1981) and Queille and Sifakis (1982), is the process of automatically and exhaustively checking if the model of a system satisfies a set of temporal properties. Over the years, Model Checking has helped identify errors in a variety of systems including digital circuits, communication protocols, distributed software, and safety-critical systems, to name a few [refer to Clarke et al. (1996) for a survey].

Model Checking was typically performed in two stages:

- *Construction of the State Graph:* A State Graph, $\mathcal{R}$, composed of the set of reachable states $\mathcal{S}$ and the transitions between these states, was constructed, and
- *Verification of the State Graph:* The State Graph was verified to satisfy a set of properties, φ, typically specified in Temporal Logic (Pnueli, 1977).

Early efforts in Model Checking were stymied by the so-called *State Explosion Problem.* Consider a system composed of n sub-systems where the behavior of each sub-system is represented by a finite-state automaton, $\mathcal{R}$. Combining these automata could result in an exponentially larger overall State Graph $O(\prod_{i=1}^{n} |R_i|)$. This State Explosion Problem limited the size of systems that could be verified to approximately 10^6 reachable states. The causes for this limitation were: a *very* large state space $\mathcal{S}$, stored using an *explicit* data structure whose space-complexity was proportional to $|\mathcal{S}|$; the algorithm for generating $\mathcal{S}$ had a time-complexity proportional to $|\mathcal{R}|$; and a verification procedure whose time- and space-complexity was $O(|\mathcal{R}|.|\varphi|)$.

Vardi and Wolper (1986) performed model verification by constructing a Büchi automaton representing the language $L(\mathcal{R} \cap \neg\varphi)$ and checking it for emptiness, and showed that this procedure was optimal. Future efforts, therefore, were directed towards the problems of reducing the state space, and improving its representation and generation. These efforts could be broadly classified as: (a) Reduce the size of the Reachability Set, and (b) Symbolic Reachability Set generation.

Reducing the Reachability Set: These efforts retained the explicit generation algorithms and data structures. Their focus was on removing, abstracting, or collapsing states from the Reachability Set that had no impact on the correctness of the verification. Partial Order Reduction (Peled, 1994), Abstraction (Clarke et al., 1994), Symmetry (Norris IP and Dill, 1996), and Compositional Reasoning (Clarke et al., 1999b) were able to reduce the Reachability Set and State Graphs to manageable sizes, and greatly increased the size of systems that could be verified by Explicit Model Checking methods.

Symbolic Reachability Set Generation: Symbolic Model Checking, introduced in McMillan (1993), attempted to use Reduced Ordered Binary Decision Diagrams (BDDs from here on), to compactly represent large sets of states. As described in Bryant (1986), BDDs in many instances do not grow linearly with the size of the set they represent—in fact, a set and its complement require the same amount of space when represented by BDDs—and, in practice, are able to represent large sets of states compactly. In addition, operations over BDDs have a time-complexity proportional to the complexity of the operand BDDs—as opposed to the size of the sets represented by the operands. As a result, Symbolic Model Checking increased the size of systems that could be handled to 10^{20} (Burch et al., 1992).

But there were drawbacks to using BDDs: BDDs could not guarantee a compact representation for all sets. The *order* of the BDD variables played a significant part in the size of the BDD, and finding an optimal variable-ordering was shown to be NP-Complete by Bollig and Wegener (1996). In addition, Bryant (1986) demonstrated that there are functions for which no compact BDD representation exists, and the BDD representation can require as much space as an explicit data structure. To alleviate this problem, heuristics have been proposed to compute *reasonably good* orderings (Fujita et al., 1991), and to dynamically change the variable order [*Sifting*, Rudell (1993)]. Since then, there has been plenty of empirical evidence to demonstrate the ability of BDDs to compactly represent very large reachable states spaces of complex real-world systems.

The Symbolic Generation algorithms have also evolved over the years: techniques such as *Event-Chaining* (Roig et al., 1996), and later *Saturation* (Ciardo et al., 2001) have reduced the time and space complexity for generating the Reachability Set by many orders of magnitude. The restrictions placed on the Saturation algorithm when first introduced, have been gradually removed, and the Saturation algorithm itself has been periodically improved: recent improvements include (a) *Fine-grained Saturation* introduced node-level chaining in Chung et al. (2006), and (b) *Extensible Decision Diagrams* were introduced in Wan and Ciardo (2009) to improve the reuse of nodes (and related cache entries) when the maximum bounds of system variables were not known a priori.

1.1 Contributions

Our contributions can be broadly divided into two categories: extending the application of symbolic data structures and algorithms, and improving on the symbolic data structures themselves.

Explicit State Space Generation

Decision diagrams, as discussed previously, have been successfully used for quite some time to generate the state space and reachability graph from models expressed in a high–level formalism. A variety of efficient, *symbolic* algorithms, which manipulate sets of states instead of individual ones, are known for this purpose. However, there are explicit generation algorithms that have not yet been replaced by any symbolic algorithm; notable examples include algorithms that utilize partial order reduction or otherwise exploit symmetry to reduce the size of the state space that must be generated. We show how explicit generation algorithms can efficiently use decision diagrams as the data structures to store the set of known reachable states, the list of reachable states to explore, and the reachability graph. We present the necessary decision diagram algorithms, and suggest small changes to the traditional explicit generation algorithm, to make the generation process more efficient. We illustrate the efficiency of our algorithms using several example models, and compare with the traditional algorithm using traditional data structures.

Deadlock Detection in Petri Nets

Deadlock detection in Petri Nets has tradtionally been performed by structural analysis of the Petri Net, and when the structural analysis is inconclusive, an *explicit* exploration of the reachable state space is performed. *Symbolic* techniques have grown in popularity for exploring the reachable states of Petri Nets, but are traditionally used to build the entire set of reachable states. We present symbolic data structures and algorithms that can be used for this purpose. Further, the state-of-the-art in symbolic reachability analysis, *Saturation*, explores states in a depth first order, which is not suited to Petri Nets that may be potentially unbounded. We present a modified Saturation-based strategy that is guaranteed to find a deadlock (if one exists) in an unbounded Petri Net. We provide experimental evidence of the effectiveness of these symbolic approaches.

Extensions to Decision Diagrams

Next, we present a generalization of decision diagrams that combines multiple reduction rules. Various versions of binary decision diagrams (BDDs) have been proposed in the past, differing in the reduction rule needed to give meaning to edges skipping levels. The most widely adopted, fully-reduced BDDs and zero-suppressed BDDs, excel at encoding different types of boolean functions (if the function has many *don't-care*, or it tends to have value zero when one of its arguments is nonzero, respectively). Lately, new classes of BDDs have been proposed that, at the cost of some additional complexity and larger memory requirements per node, exploit both cases. We introduce a new type of BDD, BDDs with Edge-Specified Reductions or ESRBDDs, that we believe is conceptually simpler, has small memory requirements in terms of node size, tends to result in fewer nodes, and can easily be further extended with additional reduction rules. We present a formal definition, prove canonicity and provide experimental results to support our claims. Finally, we present extensions to ESRBDDs by including additional reductions rules, and show that the extensions are canonical, more compact then BDDs, ZDDs and regular ESRBDDs, and also more efficient than BDDs and ZDDS for decision diagrams operations.

1.2 Organization

The remainder of the thesis is organized as follows. Chapter 2 describes our approach for integrating decision diagrams into explicit state space generation algorithms. In chapter 3 we present our novel technique on using symbolic algorithms for detecting deadlocks in Petri Nets. Chapter 4 introduces a new type of decision diagram *ESRBDDs* that can combine multiple reduction rules, details the proof of its canonicity, and provides comparisons with existing efforts. Chapter 5 details the manipulation algorithms on ESRBDDs, and provides formal and emperical evidence for their effciency. Chapter 6 extends ESRBDDs to include more reduction rules and backs our claim on their extensibility. Finally, Appendix describes specialized versions of the Apply operation for ESRBDDs, and discusses their time-complexity.

CHAPTER 2. EXPLICIT STATE SPACE AND REACHABILITY GRAPH GENERATION USING DECISION DIAGRAMS

Decision diagrams have been successfully used for quite some time to generate the state space and reachability graph from models expressed in a high–level formalism. A variety of efficient, “symbolic” algorithms, which manipulate sets of states instead of individual ones, are known for this purpose. However, there are explicit generation algorithms that have not yet been replaced by any symbolic algorithm; notable examples include algorithms that utilize partial order reduction or otherwise exploit symmetry to reduce the size of the state space that must be generated. In this chapter, we show how explicit generation algorithms can efficiently use decision diagrams as the data structures to store the set of known reachable states, the list of reachable states to explore, and the reachability graph. We present the necessary decision diagram algorithms, and suggest small changes to the traditional explicit generation algorithm, to make the generation process more efficient. We illustrate the efficiency of our algorithms using several example models, and compare with the traditional algorithm using traditional data structures.

2.1 Introduction

Generating the state space and reachability graph from a model described in a high–level formalism, such as a Petri net, is a necessary first step for many types of analysis. Important applications that utilize this information include model checking (Clarke et al., 1999a), where the reachability graph and propositions on states form a Kripke structure that can be checked against a formula expressed in an appropriate logic, and performance evaluation (e.g., Molloy (1982); Muppala et al. (1993)) or stochastic model checking (e.g., Baier et al. (2003)), where the reachability graph is converted to a Markov chain and analyzed. The well–known *state explosion problem*, in which a "small" high–level model may describe a huge state space and reachability graph, effectively limits the size and complexity of systems that may be analyzed in this manner.

A successful approach to help manage the state explosion problem has been the use of so–called "symbolic" or implicit algorithms, which utilize *decision diagrams* (e.g., Bryant (1986); Kam et al. (1998)), a structure that represents finite sets of integer vectors in a compact way for many (but not all) practical sets. As the symbolic algorithms work directly with decision diagrams, by building a transition relation from the high level model and constructing the state space by a sequence of operations on decision diagrams, they are limited by the sizes of the decision diagrams generated, which can remain small even for huge state spaces. Once this was realized (Burch et al., 1992), many other researchers adopted the idea and expanded upon it; for example, by applying it to Petri nets (Pastor et al., 1994), by developing generation algorithms (Ciardo et al., 2007; Derisavi et al., 2003), or by proposing new forms of decision diagrams (Couvreur and Thierry-Mieg, 2005; Lai et al., 1996; Miner, 2004; Yoneda et al., 1996). The literature on symbolic approaches is vast, and the above list is certainly not meant to be complete; a more thorough discussion may be found, for example, in (Clarke et al., 1999a) or (Miner and Parker, 2004).

While symbolic generation algorithms have enjoyed widespread success, they have not *completely* replaced explicit algorithms. For state space and reachability graph (or Markov chain) generation, the following are examples of cases where explicit algorithms are often or exclusively still used in practice. On–the–fly LTL model checking (cf. Clarke et al. (1999a)) explicitly generates states while performing an intersection with a *property automaton*; since this process terminates once a counter-example is found, often only a fraction of the complete state space is searched. Generation algorithms that utilize partial order reduction (Clarke et al., 1999a) to reduce the size of the state space, such as those implemented in SPIN (1990), tend to be explicit; there has been work, however, on integrating partial order reduction with symbolic generation (for example, Alur et al. (1997)). Other algorithms that exploit symmetry, such as the generation algorithms for Stochastic Well–Formed Nets (Chiola et al., 1993) that immediately build a lumped process, remain explicit, despite preliminary work to make them symbolic (Delamare et al., 2003).

In this work, we propose the use of decision diagrams as data structures to represent the state space and reachability graph (or Markov chain) while using *explicit* generation algorithms. This approach is not intended to compete with symbolic generation algorithms (although, for certain models, symbolic generation can be quite slow and explicit generation is a viable alternative); rather, the intention is to allow cases that cannot (or for various reasons, do not) use symbolic generation, such as those discussed above, to utilize decision diagrams. As decision diagrams are usually much more compact than traditional explicit data structures, the goal is to reduce the memory requirements during generation, and to do so without greatly increasing the generation times. Once the state space and reachability graph have been generated as decision diagrams, symbolic algorithms may be used as appropriate (for example, to perform CTL model checking).

Decision diagrams have been used with explicit algorithms for some time. For instance, to analyze a Markov chain represented with decision diagrams, a "hybrid" solution is usually employed (Miner and Parker, 2004), in which the solution vector is stored explicitly, and the numerical solver runs explicitly, with the decision diagram serving as a data structure for matrix storage. Algorithms for classifying the states of a Markov chain, again stored using decision diagrams, into recurrent and transient classes, are part explicit: the initial "seed" states are chosen, one at a time, at random; each seed state and potentially a large set of other states may then be classified using symbolic operations (Xie and Beerel, 1998). The "symblicit" approach for analyzing Markov Decision Processes (Wimmer et al., 2010) is a combination of explicit and symbolic approaches. Similarly, the lumping algorithm (Derisavi, 2007) is "mostly" symbolic, but does contain explicit loops over states.

However, for state space and reachability graph generation, most algorithms are either completely explicit or completely symbolic. We are only aware of a few exceptions to this rule, and these previous works are most relevant to our approach. In Miner (2001), a type of decision diagram called *matrix diagram* is used to store the Markov chain, and an explicit algorithm is used to generate the Markov chain. However, this work assumes that the state space is already known and is represented using a decision diagram. Recently, in Babar et al. (2010), an explicit state space generation algorithm was integrated into GreatSPN; however, this uses a fairly simple algorithm. The work presented here (Babar and Miner, 2014) builds upon these works, with extensions over Miner (2001) and algorithm improvements over Babar et al. (2010).

The remainder of this chapter is organized as follows. Section 2.2 briefly recalls the traditional, explicit generation algorithms and motivates the data structure requirements. Section 2.3 formally defines decision diagrams, describes how the data structure requirements are met, and introduces new decision diagram algorithms for this purpose. Section 2.4 describes how decision diagrams can be efficiently integrated into explicit generation algorithms, with some small modifications; some implementation details are discussed here as well. Section 2.5 gives experimental results for our new algorithms. Finally, section 2.6 concludes the work.

2.2 Background

Rather than use a specific high–level formalism, we use an abstract model definition. We assume a model consists of the following.

- A set of L state variables, $v_L, \ldots, v_1$. Each state variable may assume a finite number of possible values; for simplicity, we assume these are the first naturals, i.e., $v_i \in \mathcal{S}_i = \{0, 1, \ldots, b_i - 1\}$. The possible states of the model are therefore $\hat{\mathcal{S}} = \mathcal{S}_L \times \cdots \times \mathcal{S}_1$. While we assume the existence of bounds $b_L, \ldots, b_1$, we do not require that these are known *a priori*.

- A finite set of actions, $\mathcal{A}$, which change the state of the model. If a given action $a \in \mathcal{A}$ occurs when the model is in state $\mathbf{s}$, then we denote the set of possible new states as $\mathit{Occurs}(\mathbf{s}, a)$, with $\mathit{Occurs}(\mathbf{s}, a) = \emptyset$ when action a cannot occur in state $\mathbf{s}$.

- An initial set of states $\mathcal{I}$ with $\emptyset \subset \mathcal{I} \subseteq \hat{\mathcal{S}}$.

The state space of the model, $\mathcal{S}$, is the set of all states that can be reached from the initial states, when zero or more actions occur. Formally, $\mathcal{S}$ is the smallest superset of $\mathcal{I}$ that satisfies $\mathbf{s} \in \mathcal{S} \;\Rightarrow\; \forall a \in \mathcal{A}, \mathit{Occurs}(\mathbf{s}, a) \subseteq \mathcal{S}$. The reachability graph of the model, $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{S}$, holds the edges $(\mathbf{s}, \mathbf{s}')$ such that state $\mathbf{s}'$ can be reached via a single action from state $\mathbf{s}$; formally,

$$(\mathbf{s}, \mathbf{s}') \in \mathcal{E} \;\Leftrightarrow\; \mathbf{s} \in \mathcal{S}, \exists a \in \mathcal{A} : \mathbf{s}' \in \mathit{Occurs}(\mathbf{s}, a).$$

In practice, it may be useful to annotate the edges with additional information, such as the action name(s) or a rate.

Algorithm 2.1 One–pass algorithm to generate the state space and reachability graph.

```
$\mathcal{S} \leftarrow \mathcal{I}$;
$\mathcal{U} \leftarrow \mathcal{I}$;
while $\mathcal{U} \neq \emptyset$ do
    Choose and remove some $\mathbf{s}$ from $\mathcal{U}$;
    for all $a \in \mathcal{A}, \mathbf{s}' \in \mathit{Occurs}(\mathbf{s}, a)$ do
        if $\mathbf{s}' \notin \mathcal{S}$ then
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{s}'\}$;
            $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathbf{s}'\}$;
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{s}, \mathbf{s}')\}$;
```

A traditional algorithm to generate the state space and reachability graph is shown in Algorithm 2.1. In some cases, it may be beneficial to generate the reachability graph after the state space is known (for example, by collecting information while generating the state space, so that a more compact data structure may be used for the reachability graph); this may be done using the two–pass algorithm shown in Algorithm 2.2.

Algorithm 2.2 Two–pass algorithm to generate the state space and reachability graph.

```
$\mathcal{S} \leftarrow \mathcal{I}$;
$\mathcal{U} \leftarrow \mathcal{I}$;
while $\mathcal{U} \neq \emptyset$ do
    Choose and remove some $\mathbf{s}$ from $\mathcal{U}$;
    for all $a \in \mathcal{A}, \mathbf{s}' \in \mathit{Occurs}(\mathbf{s}, a)$ do
        if $\mathbf{s}' \notin \mathcal{S}$ then
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{s}'\}$;
            $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathbf{s}'\}$;
for all $\mathbf{s} \in \mathcal{S}$ do
    for all $a \in \mathcal{A}, \mathbf{s}' \in \mathit{Occurs}(\mathbf{s}, a)$ do
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{s}, \mathbf{s}')\}$;
```

In either case, the generation algorithm requires data structures for $\mathcal{S}$, the currently–known state space; $\mathcal{U}$, a set of states that still need to be explored; and $\mathcal{E}$, the edges in the reachability graph. The critical operations for $\mathcal{S}$ are adding states (see line 7), and determining if a given state is contained in the set (see line 6). The critical operations for $\mathcal{U}$ are checking if the set is empty (see line 3), adding states (see line 8), and removing *some* state (see line 4). Note that if $\mathcal{U}$ removes states in FIFO order (i.e., $\mathcal{U}$ is a queue), then the generation algorithm uses breadth–first search; however, this is not required. A critical operation applicable only to the Two–pass algorithm is enumeration (see line 9). The only critical operation for $\mathcal{E}$ is to add edges.

There are many possibilities for $\mathcal{S}$, $\mathcal{U}$, and $\mathcal{E}$ in implementation, and the design choice for one may affect the others. A traditional strategy is to store the states explicitly (but compacted) in such a way that each state has a unique index that coincides with the discovery order of the states; use any classical dictionary data structure (such as a splay tree or hash table) to determine if states are already contained in $\mathcal{S}$; use a single integer u for $\mathcal{U}$, which specifies that all states with index greater or equal to u are unexplored; and use a dynamic data structure for sparse graphs, such as adjacency lists, for $\mathcal{E}$. See for example Chiola (1989) for a more thorough discussion of efficient techniques for explicit generation.

2.3 Decision Diagrams

In this work we are exploring using Ordered Decision Diagrams (DDs) to store $\mathcal{S}$, $\mathcal{U}$, and $\mathcal{E}$. Decision Diagrams come in many forms, but for this work we propose to use Multi-Valued Decision Diagrams (MDDs) for $\mathcal{S}$ and $\mathcal{U}$, and Matrix Diagrams (MxDs) for $\mathcal{E}$.

A Decision Diagram is a directed acyclic graph used to represent function on a finite number L of variables, where each variable x_k can assume a finite number n_k of values. Nodes are either *terminal* (no outgoing edges), or *non-terminal* (labeled with a variable). Ordered Decision Diagrams require an *ordering* of nodes (there is a total ordering $\succ$ on the variables such that any outgoing edge from a node labeled x_k must go to either a terminal node or to a node labeled x_l with $x_k \succ x_l$). Also, the set of possible values for variable x_k is $\mathcal{D}_k = \{0, 1, \ldots, n_k - 1\}$. We refer to an ordered collection of variables with specified sizes as a *domain*, which we write as $\mathcal{D} = \mathcal{D}_L \times \cdots \times \mathcal{D}_1$.

Decision Diagrams employ *reduction rules* to eliminate duplication of nodes. Different reduction rules combined with different variable types and ranges, produce a variety of decision diagrams. Our work uses two types: MDDs and MxDs. Both of these decision diagrams (among others) are provided by Meddly (Babar and Miner, 2010) an open-source C/C++ library.

MDDs are used to represent functions of the form $f : \mathcal{D} \to \{0, 1\}$. MxDs on the other hand are used to represent functions of the form $f : \mathcal{D} \times \mathcal{D} \to \{0, 1\}$. In Meddly, MxDs use an "interleaved" ordering $x_L \succ x'_L \succ \cdots \succ x_1 \succ x'_1$, where $x'_L, ..., x'_1$ represent the function's *range*.

2.3.1 Representing $\mathcal{S}$, $\mathcal{U}$ and $\mathcal{E}$

Representing $\mathcal{S}$, $\mathcal{U}$ and $\mathcal{E}$ is quite straightforward using Meddly. First, we construct a domain of size L with each state variable v_i represented by a domain variable x_i. Now, each state $\mathbf{s} \in \mathcal{S}$ is represented using the *unprimed* variables of the domain (i.e. $x_L, ..., x_1$). $\mathcal{U}$ is similarly represented since it is also a set of states. For $\mathcal{E}$ we used both the *unprimed* and the *primed* variables, namely $x'_L, ..., x'_1$, of the domain. The $\mathbf{s}$ in edge $(\mathbf{s}, \mathbf{s}') \in \mathcal{E}$ is represented using the *unprimed* variables, while the $\mathbf{s}'$ is represented using the *primed* variables.

Conveniently, Meddly allows domain variables to be resized when needed. Therefore, we start domain variables at the smallest size, 1, and enlarge them as needed. This is quite useful as it allows us to start constructing the reachability graph without knowing the final bounds of each variable.

2.3.2 Efficiency of critical operation

Representing the decision diagrams is only one part of the equation. Just as important is the efficiency of the critical operations discussed in Section 2.2. To recall:

- Adding states to MDDs ($\mathcal{S}$ and $\mathcal{U}$).
- Adding edges to MxDs ($\mathcal{E}$).

- Checking if a state is contained in an MDD.
- Checking if an MDD is empty.
- Enumerating the states in a MDD.
- Removing a state from an MDD.

2.3.2.1 Adding elements to MDDs and MxDs

Adding states to MDDs or MxDs can be done in multiple ways using Meddly. The methods that interest us are:

- Using CreateEdge one can build an MDD or MxD representing a minterm, and then use the UNION operator to add it to an another MDD or MxD.
- Using CreateEdge one can also build an MDD or MxD to represent an array of minterms. These can also be added to other MDDs or MxDs using the UNION operator.
- Using *temporary edge* and Accumulate to "accumulate" a batch of minterms in to a temporary edge. A temporary edge is so named because it has not been processed using the reduction rules applicable to the decision diagram. Another version of Accumulate can add an MDD or MxD to an existing temporary edge. Once this a "accumulation" is complete, the temporary edge is "reduced" into a MDD or a MxD.

The first use of CreateEdge above suggests building an MDD for each minterm and then combining these MDDs in some fashion. This involves the heavy use of MDD Union (or equivalent for MxDs). The algorithm that CreateEdge uses to build a MDD from an array of minterms (given in Figure 2.3) aims to avoid using such relatively expensive operations. The central idea is to sort the minterms lexicographically (where a minterm is read as $x_L, ..., x_1$), but to do so efficiently. To achieve this, a Radix Sort is employed at each level (since the size of variable bounds are usually not very large). After sorting for x_L, all minterms whose x_L value is the same are grouped together. Note that in the final MDD, all such minterms will be accessed via the same child of the root node.

Algorithm 2.3 Algorithm to build an MDD from an array of minterms.

```
procedure SortBuild(var $k$, int** $\mathbf{M}$, int $n$)
    if $k = 0$ then
        return $true$;
    RadixSort($M$, $n$);
    /* Items are in order; now we recurse and build the node. */
    node $r \leftarrow$ new node at level $k$;
    int $i \leftarrow 0$;
    for all $i' \in \{1, \ldots, n\}$ do
        if $(i' = n) \vee (\mathbf{M}[i'][k] \neq \mathbf{M}[i][k])$ then
            node $r' \leftarrow$ SortBuild($k - 1$, $\mathbf{M} + i$, $i' - i$);
            $j \leftarrow \mathbf{M}[i][k]$;
            $r[j] \leftarrow r'$;
            $i \leftarrow i'$;
    return Reduce($r$);
```

The process then repeats via a recursive call for this group of minterms. The result of the above algorithm is that exactly one temporary node is created at level x_i for all minterms who values for variables x_L through x_{i+1} are identical. As seen in Figure 2.3 this algorithm can be implemented using a depth-first traversal to keep memory requirements low.

The Accumulate based approach uses both forms of accumulation: accumulating minterms and accumulating MDDs (or MxDs). Accumulating minterms leads to branching of nodes but this branching also makes accumulating MDDs into the temporary edges very fast. When using a limited buffer-size (in terms of the number of minterms added to the temporary edge before the temporary edge is reduced), this could be an effective balance between speed and memory usage. The algorithm that Meddly uses for accumulating minterms and decisions diagrams into a temporary edge before reducing it into a decision diagram was first described in Miner (2001), although the Canonical Matrix Diagrams (CMDs) used there had a different structure due to the use of different reduction rules. The algorithms discussed in Miner (2001), use CMDs to construct the reachability graph once the set of reachable states is computed (therefore the bounds of domain variables are also known before building the reachability graph). Although this gives CMDs an advantage it also limits the use of CMDs. For the sake of thoroughness, we have implemented and compared them to Meddly's MxDs where applicable, namely the Two–pass algorithm,

2.3.2.2 Checking if a state exists in an MDD

Checking if a state (represented by a minterm) is contained in an MDD is performed via the EVALUATE operation in Meddly. This operation's complexity is linear in the size of the minterm (the size of the minterm is equivalent to the number of variables in the domain).

2.3.2.3 Checking for an empty MDD

Checking if an MDD is empty is a trivial operation for MDDs. This is because an empty MDD reduces to single terminal node indicating "false".

2.3.2.4 Enumerating the elements in a MDD

The second pass of the Two–pass algorithm requires the ability to enumerate the elements in $\mathcal{S}$. Meddly provides iterators that are specifically built to enumerate the minterms stored in MDDs or MxDs. Meddly's iterators are based on the approach described in Miner and Parker (2004). These iterators can be initialized, incremented and checked for "any more minterms to enumerate?" They can also be copied so that it is possible to have multiple iterators to the same MDD, pointing at different minterms. One thing to keep in mind is that when an MDD is modified, its associated iterator (if any) is no longer valid, and will need to be reinitialized before being used. Also worth noting is that Meddly's MxD iterators can be used to enumerate minterms "by row" and "by column". Please refer to Miner and Parker (2004) for more details.

2.3.2.5 Removing *some* state from an MDD

A state can be removed from an MDD using multiple ways in Meddly. The most straightforward method is to use CreateEdge to build a MDD representation of a minterm. This MDD can then be "subtracted" from the original MDD using the MDD Difference operation. But before using CreateEdge and minterms needs to be "read" from the MDD. Meddly provides the FindFirstElement operation for this purpose. It returns the lexicographically-first element in the MDD in the form of a minterm. Multiple accesses to FindFirstElement will return the same minterm unless the MDD in question is modified.

Note that a combination of FindFirstElement, CreateEdge and MDD Difference can also be used to enumerate the minterms in an MDD as follows:

- Use FindFirstElement to get a minterm from an MDD.
- Build a MDD using CreateEdge and the minterm.
- Subtract the MDD representing the minterm from the original MDD.
- Repeat.

This approach has one advantage over using iterators—there are no iterators to be invalidated. But, this may potentially be expensive because of the cost of building an MDD and using MDD Difference.

2.4 Explicit Generation with Decision Diagrams

In this section we alter our original algorithms (One– and Two–pass) to take advantage of the features offered by MDDs and Meddly. Specifically, we will explore the effects of using buffered addition when dealing with $\mathcal{S}, \mathcal{U}$ and $\mathcal{E}$. We also explore the difference between using one-at-a-time removal using FindFirstElement and a buffered removal using Meddly's iterators. Although there is a vast difference in the implementation between the buffered versions of CreateEdge and Accumulate, the central idea and the interface is similar. For the purpose of this thesis, we abstract the details out and use a generic version of AddBatch as shown in Algorithm 2.4.

The steps involved with buffered CreateEdge are straightforward:

- Add minterms to the array buffer.

Algorithm 2.4 Generic version of AddBatch.

```
1: procedure AddBatch(DD $\mathcal{A}$, Buffer $\mathcal{A}'$, minterm $m$, bool $force$)
2:     $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{m\}$;
3:     if $\mathcal{A}'$ is full OR $force$ is true then        • Add buffer to $A$ and clear the buffer
4:         if using Accumulate then                           • Buffer is a temporary edge
5:             $\mathcal{A}' \leftarrow \mathcal{A}' \cup \mathcal{A}$;
6:             $\mathcal{A} \leftarrow$ Reduce($\mathcal{A}'$);
7:             $\mathcal{A}' \leftarrow \emptyset$;
8:         else                                               • Buffer is an array of minterms
9:             $\mathcal{A} \leftarrow \mathcal{A} \cup$ CreateEdge($\mathcal{A}'$);
10:            $\mathcal{A}' \leftarrow \emptyset$;
```

- When the array buffer is full, call CreateEdge to build an MDD from the minterms in the buffer.
- Clear the buffer and re-use it.

In the case of Accumulate a slightly different strategy is used to take advantage to the fast addition using AccumulateMxd between a temporary edge and an existing MDD:

- Add minterms to the temporary edge using AccumulateMinterms.
- When "enough" minterms have been added to the temporary edge, call AccumulateMxd to add an existing MDD to the temporary edge.
- Reduce the temporary edge to obtain a cumulative MDD.
- Temporary edge can be clear and re-used.

Note that AddBatch has a variable *force*. This is used to force the minterms to be transferred from the buffer into the primary Decision Diagram. This is useful when the primary Decision Diagram has run out of minterms but the buffer still has some in it. This will be illustrated in the algorithms below.

Now that we have a generic version of AddBatch, we provide modified versions of the One– and Two–pass algorithms that are differentiated mainly in the manner in which $\mathcal{U}$ is handled.

2.4.1 One-at-a-time Removal

The One-at-a-time Removal algorithm (Algorithm 2.5) accesses a minterm from $\mathcal{U}$, removes it from $\mathcal{U}$, and then processes it. Here, the minterms are removed from $\mathcal{U}$ one at a time. This strategy is applied using FindFirstElement, CreateEdge on a single minterm, and finally MDD Difference.

Note that line 14 forces the minterms in the buffer to be moved to $\mathcal{U}$ when $\mathcal{U}$ is empty. This is one of the things to be careful about when using buffers in updating the MDDs.

2.4.2 Batch Removal

The Batch Removal algorithm (Algorithm 2.6) uses a buffered approach to removing minterms from $\mathcal{U}$, using an MDD iterator and a buffered storage of minterms.

The algorithm looks more complex here because of the management of the iterator. As mentioned in Section 2.3.2.4, iterators are invalidated when the MDD associated with the iterator is modified. This situation can occur when a call to AddBatch for $\mathcal{U}$ results in the $\mathcal{U}$ getting updated with the minterms in $\mathcal{U}'$. Another point where the iterator will get invalidated, is when it has reached the end of the MDD to which it is associated. In this algorithm, once the iterator has reached the end of MDD $\mathcal{U}$, $\mathcal{U}$ must be updated with the contents of $\mathcal{U}'$ and the iterator reset to the beginning of $\mathcal{U}$.

We have provided the algorithms for One-at-a-time and Batch Removal for the One–pass algorithm. The equivalent Two–pass algorithms easily follow from this.

2.5 Experimental Results

The decision diagram algorithms discussed in Section 2.3 are implemented in Meddly (Babar and Miner, 2010), and our explicit generation algorithms based on decision diagrams are implemented in version 3 of SMART (Ciardo et al., 2009). Experiments were run on a 2.13 Ghz Intel Xeon processor running Linux, with sufficient RAM to avoid paging to disk.

Algorithm 2.5 One-At-A-Time Removal.

```
1: $\mathcal{S} \leftarrow \mathcal{I}$;
2: $\mathcal{U} \leftarrow \mathcal{I}$;
3: $\mathcal{S}' \leftarrow \emptyset$;
4: $\mathcal{U}' \leftarrow \emptyset$;
5: while $\mathcal{U} \neq \emptyset$ do
6:     $\mathbf{s} \leftarrow$ FindFirstElement($\mathcal{U}$);
7:     $\mathcal{U} \leftarrow \mathcal{U} \setminus$ CreateEdge($\mathbf{s}$);
8:     for all $a \in \mathcal{A}, \mathbf{s}' \in Occurs(\mathbf{s}, a)$ do
9:         if $\mathbf{s}' \notin \mathcal{S} \wedge \mathbf{s}' \notin \mathcal{S}'$ then
10:            AddBatch($\mathcal{S}$, $\mathcal{S}'$, $\mathbf{s}'$, $false$);
11:            AddBatch($\mathcal{U}$, $\mathcal{U}'$, $\mathbf{s}'$, $false$);
12:        AddBatch($\mathcal{E}$, $\mathcal{E}'$, $\{(\mathbf{s}, \mathbf{s}')\}$, $false$);
13:    if $\mathcal{U} = \emptyset$ then
14:        AddBatch($\mathcal{U}$, $\mathcal{U}'$, $\mathbf{s}'$, $true$);
```

Algorithm 2.6 Batch Removal.

```
1: $\mathcal{S} \leftarrow \mathcal{I}$;
2: $\mathcal{U} \leftarrow \mathcal{I}$;
3: $\mathcal{S}' \leftarrow \emptyset$;
4: $\mathcal{U}' \leftarrow \emptyset$;
5: while $\mathcal{U} \neq \emptyset$ do
6:     $i \leftarrow begin(\mathcal{U})$;
7:     while $i$ is not at the end do
8:         $\mathbf{s} = minterm(i)$;
9:         for all $a \in \mathcal{A}, \mathbf{s}' \in Occurs(\mathbf{s}, a)$ do
10:            if $\mathbf{s}' \notin \mathcal{S} \wedge \mathbf{s}' \notin \mathcal{S}'$ then
11:                AddBatch($\mathcal{S}$, $\mathcal{S}'$, $\mathbf{s}'$, $false$);
12:                AddBatch($\mathcal{U}$, $\mathcal{U}'$, $\mathbf{s}'$, $false$);
13:            AddBatch($\mathcal{E}$, $\mathcal{E}'$, $\{(\mathbf{s}, \mathbf{s}')\}$, $false$);
14:        if $\mathcal{U}$ was modified then
15:            $i \leftarrow begin(\mathcal{U})$;
16:        else
17:            Move iterator $i$ to the next minterm.
18:    $\mathcal{U} \leftarrow \emptyset$;
19:    AddBatch($\mathcal{U}$, $\mathcal{U}'$, $\mathbf{s}'$, $true$);
```

2.5.1 Models

We tested our algorithms on a variety of Petri net models taken from the literature.

The Kanban model describes a manufacturing system. The model parameter, N, determines the number of parts circulating in the system. The number of places is fixed at 16, and we use a decomposition where each MDD variable corresponds to a single place. As N increases, the number of variables remains fixed, but the set of possible values for each variable increases.

The dining philosophers model specifies the number of philosophers and number of forks, N, as a model parameter; our model decomposition uses MDD variables for the forks and for the philosophers. The MDD has 2N variables whose size remains fixed as N increases. Since each philosopher is modeled by several Petri net places, we index the submarkings for each philosopher and use the submarking index as the value in each minterm.

The Swaps model describes an array of N distinct integers, with operations to exchange values of neighboring array slots. Since all possible permutations are allowed, this model describes exactly N! reachable states. As N increases, the number of MDD variables increases, as does the size of each variable.

Finally, the FMS model describes a flexible manufacturing system where model parameter N specifies the number of pallets to move parts. We use a coarse partitioning of the model into 4 state variables, where the number of possible submarkings for each state variable (again, using the submarking index in each minterm) grows with N.

For a more thorough discussion of the Kanban, dining philosophers, and FMS models, see for example Ciardo et al. (2007). It must be pointed out that all the models we used in our experiments can be used with symbolic methods.

2.5.2 Observations

Figure 2.1 shows the time required to generate the reachability graph (assuming the state space is known) for the Kanban model with parameter $N = 4$, for different buffer sizes. Note that the x–axis is in $\log_2$ scale. In the figure, “CMDs” refers to the implementation based on Miner

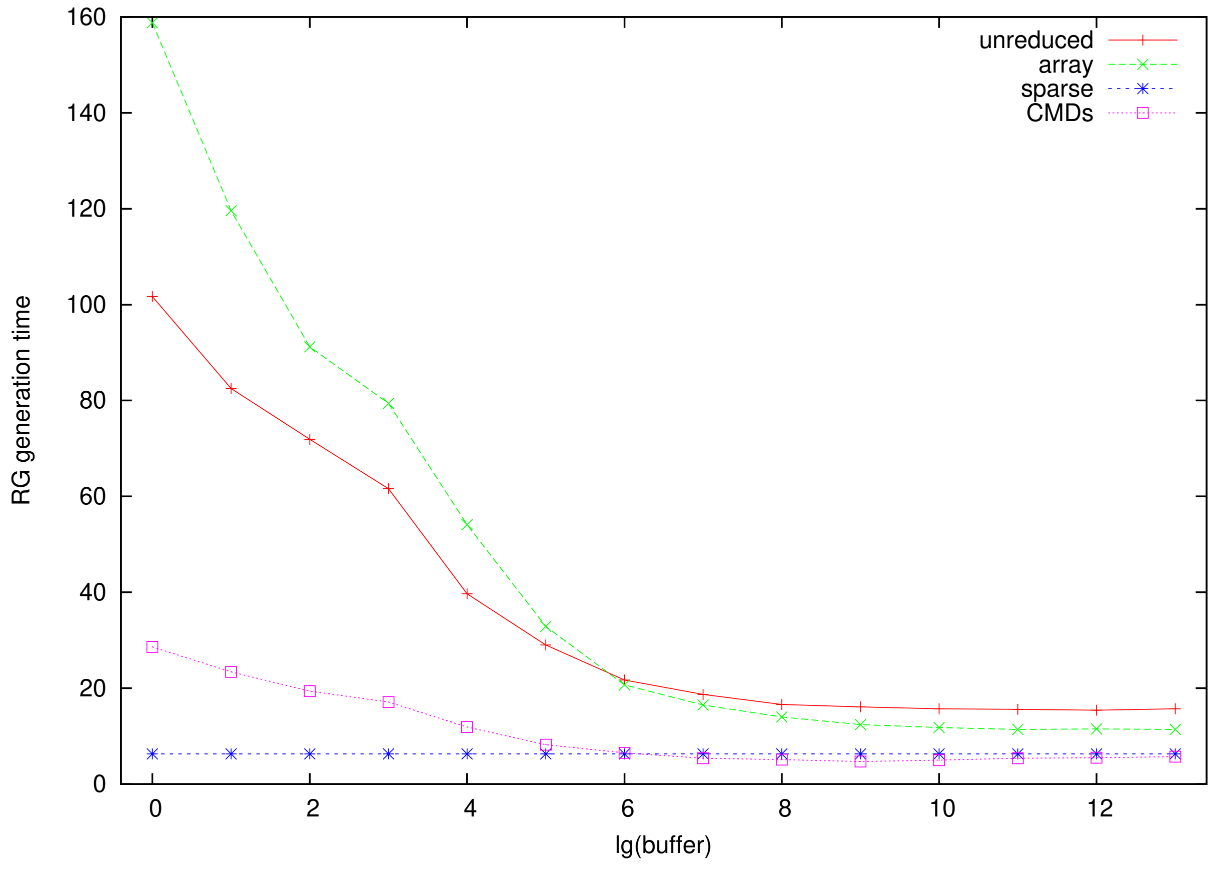


Figure 2.1 Reachability graph generation times for the Kanban model as a function of buffer size.

(2001); “array” refers to the Meddly–based implementation, using an array of states as a buffer, and “unreduced” refers to the Meddly–based implementation, using unreduced nodes as a buffer. For reference, “sparse” refers to a traditional implementation using sparse graphs, that does not use a buffer. The buffer size refers to the maximum number of additions, before converting the structure into reduced, canonical form. Figure 2.2 shows the time required to generate the reachability set for Kanban with $N = 4$ for different buffer sizes. In the figure, “single removal” and “batch removal” refer to the One-at-a-time Removal and Batch Removal algorithms respectively. In both figures, once the buffer size exceeds around 2^{10}, the generation times essentially remain constant. For the other models we tried, and for both the one–pass and two–pass algorithms, we found that the

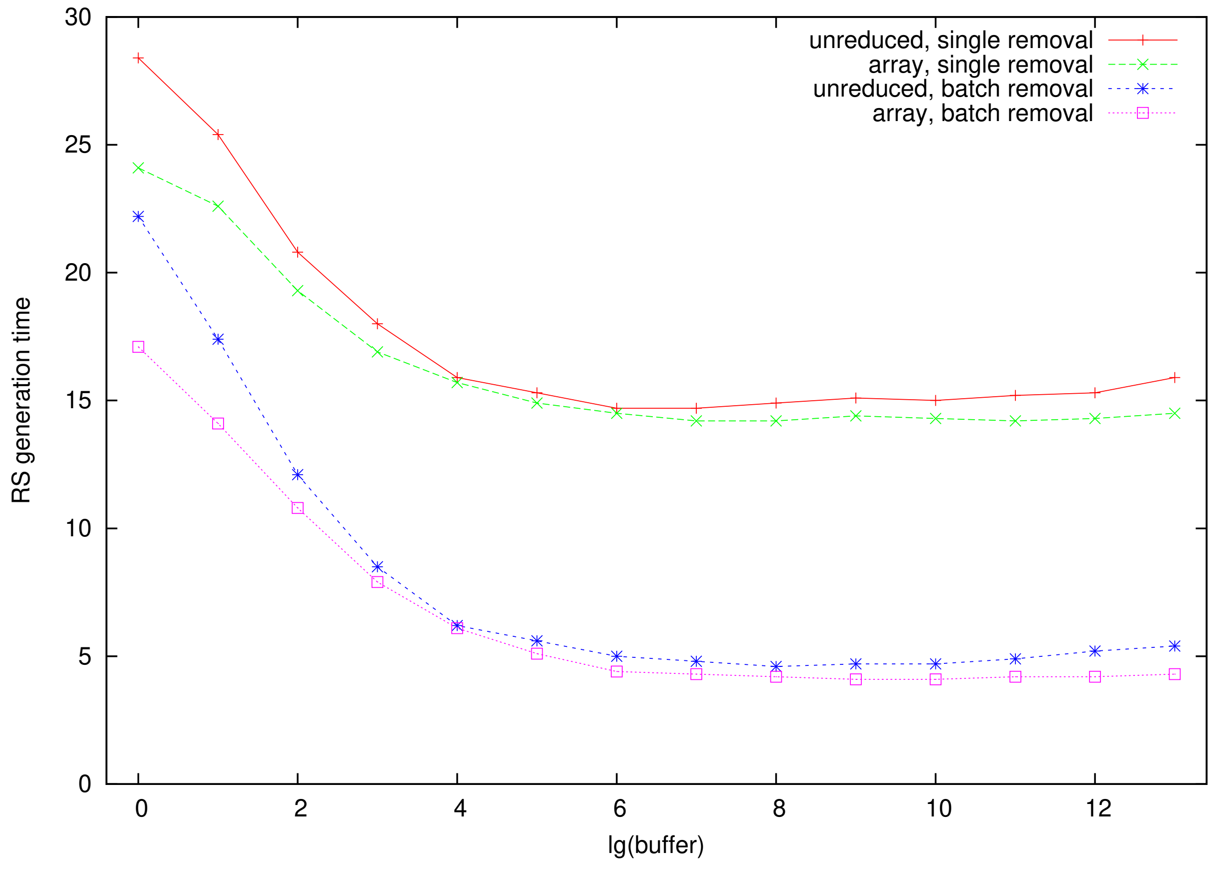


Figure 2.2 Reachability set generation times for the Kanban model as a function of buffer size.

results were consistent with those shown in Figure 2.1 and Figure 2.2: a buffer size of 2^{10} tends to be sufficient. We have, therefore, used a buffer size of 2^{10} for the experiments below.

The results of our experiments are tabulated in Table 2.1. Alg 1 uses the One-at-a-time Removal algorithm and Alg 2 uses the Batch Removal algorithm. CMDs are based on the 2001 implementation. As noted earlier, they only work for generating the reachability graph once the state space is known. Therefore, they are only employed in the Two–pass section of the tabulated results. Array buffer indicates the use of a batch of minterms stored as arrays (CreateEdge). Mdd buffer refers to the temporary edge based implementation using Accumulate.

Table 2.1 CPU and peak memory requirements for state space and reachability graph generation.

One–pass algorithm

			Generation times (seconds)					Peak memory required (k=1024 bytes, m=1024k)											
				Array buffer		Mdd buffer				Array buffer						Mdd buffer			
			BST	Alg1	Alg2	Alg1	Alg2	BST		Alg1			Alg2			Alg1		Alg2	
Model	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{E}\rvert$						$\mathcal{S}$	$\mathcal{E}$	$\mathcal{S}$	$\mathcal{E}$	buf	$\mathcal{S}$	$\mathcal{E}$	buf	$\mathcal{S}$	$\mathcal{E}$	$\mathcal{S}$	$\mathcal{E}$
Kanban 5	2,546,432	24,460,016	38	139	114	176	162	41m	206m	58k	156k	213k	68k	343k	212k	161k	373k	170k	569k
Kanban 6	11,261,376	115,708,992	196	642	532	835	796	187m	969m	68k	191k	207k	105k	561k	212k	186k	437k	207k	783k
Kanban 7	41,644,800	450,455,040	852	2490	2025	3292	3270	700m	3758m	79k	227k	214k	155k	874k	215k	257k	1034	203k	491k
Phils 10	1,860,498	17,391,050	21	147	126	171	156	31m	147m	14k	38k	227k	27k	114k	245k	125k	274k	140k	369k
Phils 11	7,881,196	81,036,637	107	727	624	861	785	141m	678m	16k	39k	261k	34k	139k	258k	122k	275k	150k	402k
Phils 12	33,385,282	374,483,676		3565	3067	4230	3906			17k	41k	279k	40k	162k	278k	117k	286k	182k	445k
Swaps 10	3,628,800	32,659,200	43	162	181	228	259	52m	277m	51k	1m	157k	489k	3.6m	157k	178k	1.2m	617k	3.8m
Swaps 11	39,916,800	399,168,000	609	2072	2328	2910	3340	571m	3.2G	108k	2.3m	165k	1.3m	11m	166k	192k	2.5m	1.4m	11m
FMS 7	1,639,440	14,998,792	44	337	306	623	615	32m	127m	17k	165k	108k	124k	933k	107k	87k	415k	875k	3.7m
FMS 8	4,459,455	42,302,007	129	1342	1213	2947	2920	89m	357m	23k	256k	108k	203k	1.6m	108k	118k	615k	1.4m	5.8m
FMS 9	11,058,190	108,084,295	361	4682	4567	12163	13337	221m	909m	33k	378k	108k	315k	2.7m	108k	157k	867k	2.2m	9m

Two–pass algorithm

	Generation times (seconds)									Peak memory required (k=1024 bytes, m=1024k)									
	BST		CMD	Array buffer			Mdd buffer			BST		CMD	Array buffer				Mdd buffer		
Model	$\mathcal{S}$	$\mathcal{E}$	$\mathcal{E}$	$\mathcal{S}$ Alg1	$\mathcal{S}$ Alg2	$\mathcal{E}$	$\mathcal{S}$ Alg1	$\mathcal{S}$ Alg2	$\mathcal{E}$	$\mathcal{S}$	$\mathcal{E}$	$\mathcal{E}$	$\mathcal{S}$ Alg1	$\mathcal{S}$ Alg2	$\mathcal{E}$	buf	$\mathcal{S}$ Alg1	$\mathcal{S}$ Alg2	$\mathcal{E}$
Kanban 5	27	55	31	80	23	66	83	26	90	41m	207m	279k	58k	68k	44k	125k	161k	170k	146k
Kanban 6	186	335	148	366	108	308	386	120	427	187m	969m	314k	68k	105k	57k	125k	186k	207k	159k
Kanban 7	782	1459	584	1435	420	1200	1484	469	1687	700m	3.67G	357k	79k	155k	72k	125k	203k	257k	175k
Phils 10	27	35	96	142	93	126	142	93	143	31m	147m	268k	14k	27k	29k	140k	125k	140k	177k
Phils 11	134	189	487	693	470	636	704	480	720	141m	678m	276k	16k	34k	33k	148k	122k	150k	182k
Phils 12	636	943	2445	3393	2351	3134	3445	2394	3542	605m	3G	284k	17k	40k	36k	156k	117k	182k	186k
Swaps 10	42	60	54	85	31	89	90	36	135	52m	277m	4.25m	51k	489k	1m	100k	178k	616k	1.2m
Swaps 11	565	945	710	1091	399	1145	1156	463	1748	571m	3.3G	10.3m	108k	1.3m	2.3m	104k	192k	1.4m	2.5m
FMS 7	43	55	159	291	155	199	546	412	481	32m	127m	1.7m	17k	124k	165k	76k	87k	875k	452k
FMS 8	126	165	631	1134	585	772	2620	1992	2354	89m	357m	2.8m	23k	203k	254k	76k	118k	1.4m	606k
FMS 9	350	507	2326	4107	2081	2781	12113	8746	10071	221m	910m	4.8m	33k	315k	378k	76k	157k	2.2m	916k

There are some interesting results observed. The first interesting observation is that in some circumstances it is possible for the Two–pass algorithm to take less time than the equivalent One–pass algorithm. But, this is only observed for the Decision Diagram based implementations. In almost every case, the array based buffer using CreateEdge is faster than the Accumulate based algorithm. Also, in almost every case, the Batch Removal algorithm seems to run faster than the equivalent One-at-a-time Removal Algorithm.

CMDs have done quite well in terms of speed when compared to the MxD versions, although they do not do as well when it comes to peak memory usage. MxDs use the least peak memory here for constructing the reachability graph. This is despite not having a priori knowledge of the bounds of the domain's variables. The BST based approach is consistently the fastest, but uses up about a 1000 times more memory than the decision diagram based implementations.

2.6 Conclusion

We have shown that decision diagrams can be used to efficiently store the set of known reachable states, the list of reachable states to explore, and the reachability graph. We have presented decision diagram algorithms that improve the performance of decision diagrams when used by traditional explicit generation algorithms. We have also presented small changes to the traditional explicit generation algorithms to take advantages of these decision diagram algorithms.

Our experimental results show that decision diagrams can greatly reduce the peak memory requirements of traditional explicit generation algorithms. There is a very obvious trade-off between the BST based implementation and the MDD based implementation. The BSTs are sometimes 20 times as fast as the fastest MDD implementation, but at the expense of using 1000 times as much memory. Models for which traditional explicit implementations run low on available memory could be prime candidates for the MDD based explicit generation.

We have been working to integrate Meddly's decision diagrams into GreatSPN Chiola et al. (1995). Our previous effort are mentioned in Babar et al. (2010), but our recent work should improve on that effort, as we our explicit generation algorithms have improved.

We have noticed that the CMDs structure lends themselves to explicit generation algorithms. Although we cannot be sure at this moment, we believe that it is a combination of CMDs reduction rule (which do not reduce *primed* level nodes), as well as the sparse storage of the prime level nodes that helps them accumulate minterms quickly.

While we do prefer the peak memory requirements of MxDs, we would like to explore if a CMD-like reduction rule, will improve MxD performance. In the future, we plan to incorporate this work into algorithms that exploit model symmetry.

Acknowledgments

This work was supported in part by the National Science Foundation under grant CNS-0546041.

CHAPTER 3. SYMBOLIC DEADLOCK DETECTION

Deadlock detection in Petri Nets has traditionally been performed by structural analysis of the Petri Net. If the structural analysis is inconclusive, an *explicit* exploration of the reachable state space is undertaken to either find a reachable deadlock state or to rule out the possibility of reaching such a state.

Explicit techniques for state space exploration have their limitations since their time- and space-complexity is proportional to the size of state graph. *Symbolic* techniques have grown in popularity for the reachability analysis of structured high-level models such Petri Nets. But symbolic reachability analysis is used to efficiently build the *entire* set of reachable states, and, *as is*, may not be suitable to the search for a single (or a set of) deadlock states. In this chapter, we present symbolic data structures and algorithms that can be used for deadlock detection in Petri Nets.

In addition, the state-of-the-art in symbolic reachability analysis, *Saturation*, explores reachable states in a depth first order, which is not suited to Petri Nets that may be potentially unbounded. We present a modified Saturation-based strategy that is guaranteed to find a deadlock (if one exists) in an unbounded Petri Net. We provide experimental evidence of the effectiveness of these symbolic approaches.

3.1 Introduction

Petri Nets, introduced in Petri (1962), are a popular high-level formalism for modeling dynamic systems. They are comprised of a *net*, composed of *places* (state variables) and *transitions* (events that modify places), and a *marking*, representing the overall state of the structure (a snapshot composed of the state of each place). A deadlock in a Petri Net is a reachable marking that disables all transitions in the Petri Net. Detecting deadlocks in Petri Nets has been well-studied over the years (Hack, 1972; Murata, 1989; Esparza, 1998, etc.). These techniques are typically a combination to two well-known kinds of Petri Net analysis: *structural* and *behavioral.* Structural analysis can be relatively fast since it is performed on the net structure without considering the *initial marking* (the starting state of the Petri Net), but in many cases (including for deadlock detection) this analysis is not complete. Behavioral analysis, on the other hand, includes the initial marking in the analysis and can therefore give conclusive answers, but it can be much slower than structural analysis.

Existing techniques for deadlock detection proceed as follows: perform structural analysis on the Petri Net to rule out the possibility of a deadlock, and if structural analysis is not conclusive perform behavioral analysis to arrive at a conclusion. Performing structural analysis, even when inconclusive, reduces the search space that the behavioral analysis must explore. Traditionally, behavioral analysis for deadlock detection involves exploring the reachable markings in the state space of the Petri Net until a deadlock state is found, and time- and space-complexity of this exploration using existing techniques is linear in terms of the reachability graph.

Symbolic data structures and algorithms have become popular for the reachability analysis of Petri Nets. Symbolic data structures, such as decision diagrams (Bryant, 1986), have been known to compactly represent the structured state space of Petri Nets (Miner and Ciardo, 1999; Ciardo et al., 2001). In addition, symbolic algorithms for manipulating decision diagrams have a time- and space- complexity proportional to the size of the symbolic data structures (instead of the number of elements stored in those data structures), making them more efficient than explicit algorithms for exhaustive state space exploration, especially when combined with state-of-the-art variable ordering heuristics (for example, Smith and Ciardo (2018) and Amparore et al. (2019)).

Our work focuses on using symbolic approaches for detecting deadlocks in Petri Nets, by replacing explicit exploration with symbolic exploration. Although symbolic reachability analysis has been shown to be efficient, its purpose is to build the *entire* set of reachable states. Detection of deadlocks, on the other hand, can terminate when a deadlock is found. We present symbolic data-structures and algorithms that integrate deadlock detection within the state space exploration.

The state-of-the-art in symbolic reachability analysis is the *Saturation* algorithm (introduced in Miner and Ciardo (1999)), which has been shown to be orders of magnitude faster than traditional symbolic state space exploration algorithms. But, Saturation explores the states in a *depth-first* order, which poses a problem when a Petri Net is potentially unbounded. We present a symbolic exploration algorithm, based on *Constrained* Saturation (Zhao and Ciardo, 2009), that is guaranteed to find a deadlock (if one exists) in a Petri Net, even when it is not known *a priori* if the Petri Net is bounded.

The rest of this chapter is organized as follows. Section 3.2 briefly describes Petri Nets and decision diagrams. Section 3.3 discusses our approach to detection deadlocks in Petri Nets using decision diagrams and the *saturation* heuristic. Section 3.4 details the experiments we have performed to study the effectiveness of our approach relative to existing works. Section 3.5 discusses the experimental results, and Section 3.6 concludes the work.

3.2 Background

This section describes the class of models that we consider in our work, and the symbolic data structures and algorithms to explore their state space.

3.2.1 Petri Nets

Petri Nets are one among many high-level formalisms that can represent the class of models that our work is focused on. Therefore, we first define the class of generic high-level discrete-state models that our work applies to. A discrete-state model $\mathcal{M}$ is defined by a tuple $(\mathcal{V}, \mathcal{E}, \mathbf{i}_0, \delta^-, \delta)$ where:

- $\mathcal{V} = \{v_1, v_2, \ldots, v_L\}$ is a finite set of *state variables* of the model. Each state variable v_k can assume a value from the set of natural numbers. A (*global*) state $\mathbf{i}$ of $\mathcal{M}$ is then an L-tuple $(i_1, i_2, \ldots, i_L) \in \mathbb{N}^L$.

- $\mathcal{E} = \{e_1, e_2, \ldots, e_{|\mathcal{E}|}\}$ is a finite set of *events* of the model.

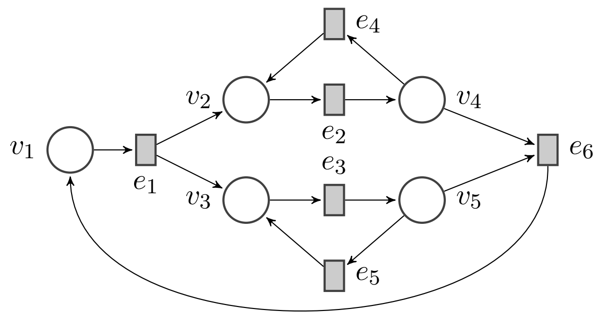
- $\mathbf{i}_0 \in \mathbb{N}^L$ is the initial state of the model.

- $\delta^-_{e_j,v_k} \in \mathbb{N}$ is the *enabling* condition for event e_j in terms of place v_k. We say that an event e_j is *enabled* in state i if $\forall k \in [1, L], i_k \geq \delta^-_{e_j,v_k}$.

- $\delta_{e_j,v_k} \in \mathbb{Z}$ is the change in variable v_k when event e_j occurs. If x is the current state of place v_k, then x', the state of v_k after event e_j occurs, is defined as $x' \leftarrow x + \delta_{e_j,v_k}$. We say that if an event e_j occurs then the model changes from state i to i', $i_k \xrightarrow{e_j} i'_k$, such that $\forall k \in [1, L], i'_k \leftarrow i_k + \delta_{e_j,v_k}$.

  We require that $\forall e \in \mathcal{E}, \forall v \in \mathcal{V}, \delta^-_{e,v} + \delta_{e,v} \geq 0$. This ensures that any reachable state i satisfies $i \in \mathbb{N}^L$.

For example, an *ordinary* Petri net (Murata, 1989) can be expressed using our model: the set $\mathcal{V}$ can correspond to the set of Petri net places, the set $\mathcal{E}$ can correspond to the set of Petri net transitions, the initial state $\mathbf{i}_0$ will correspond to the initial marking of the Petri net, and δ^- and δ will correspond to the Petri net firing rules. Specifically, for a place p_k and a transition t, we say that t is enabled with respect to p_k in state i, if $i_k \geq \delta^-_{t,p_k}$, i.e. i_k is greater than or equal to the number of edges from p_k to t, and we say that $j_k \leftarrow i_k + \delta_{t,p_k}$, where j is the state reached from state i after *firing* transition t, and δ_{t,p_k} equals the number of edges from t to p_k minus the number of edges from p_k to t. An example "fork-join" Petri net model is shown in Figure 3.1, where the circles correspond to places and the squares correspond to transitions. Transition e_1 performs a fork operation and transition e_6 performs a join operation.



$$
\begin{aligned}
\mathcal{V} &= \{v_1, v_2, v_3, v_4, v_5\} \\
\mathcal{E} &= \{e_1, e_2, e_3, e_4, e_5, e_6\} \\
\delta^- &= \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\
\begin{pmatrix}
1 & & & & & \\
 & 1 & & & & \\
 & & 1 & & & \\
 & & & 1 & & 1 \\
 & & & & 1 & 1
\end{pmatrix}
\end{array} \\
\delta &= \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\
\begin{pmatrix}
-1 & & & & & +1 \\
+1 & -1 & & +1 & & \\
+1 & & -1 & & +1 & \\
 & +1 & & -1 & & -1 \\
 & & +1 & & -1 & -1
\end{pmatrix}
\end{array} \\
\mathbf{i}_0 &= \{[10000]\} \\
\mathcal{S} &= \{[10000], [01100], [00110], [01001], [00011]\}
\end{aligned}
$$

Figure 3.1 A fork-join Petri net model, its description and reachability set.

For a given model $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathbf{i}_0, \delta^-, \delta)$, we can define the following.

- The next state function for event e, $\mathcal{N}_e : \mathbb{N}^L \to 2^{\mathbb{N}^L}$, is defined as

  $$\mathcal{N}_e(\mathbf{i}) = \{\mathbf{j} : \forall k \in [1, L], i_k \geq \delta^-_{e,v_k} \wedge j_k = i_k + \delta_{e,v_k}\}$$

  We then define the overall next state function as $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$, which gives the set of states reachable via the occurrence of one event from a single starting state, and further extend this to sets of starting states: $\mathcal{N}(\mathcal{I}) = \bigcup_{\mathbf{i} \in \mathcal{I}} \mathcal{N}(\mathbf{i})$.

- The *reachability set* $\mathcal{S} \subseteq \mathbb{N}^L$ is the set of states reachable via the occurrence of zero or more events from the initial state $\mathbf{i}_0$, and is the least fixed point satisfying $\mathcal{S} = \{\mathbf{i}_0\} \cup \mathcal{S} \cup \mathcal{N}(\mathcal{S})$.

As an example, the reachability set $\mathcal{S}$ is shown in Figure 3.1 for the fork-join Petri net model, where a state is shown as $(p_1, p_2, p_3, p_4, p_5)$.

3.2.2 Reachability Analysis using Decision Diagrams

The focus of this paper is to use symbolic reachability algorithms for the purpose of deadlock detection. In this section, we briefly describe decision diagrams and well-known symbolic algorithms that use decision diagrams to build the set $\mathcal{S}$ for Petri Nets.

3.2.2.1 Decision Diagrams

Decision diagrams come in many forms: we use Multi-valued Decision Diagrams (MDDs) for representing the set $\mathcal{S}$, and Extensible Decision Diagrams (XDDs) for representing the set of potential deadlock states $\hat{\mathcal{C}}$.

An MDD is a defined as follows:

- It is an directed acyclic graph over a finite number L of variables, $\mathcal{V} = \{v_1, v_2, \ldots, v_L\}$. Each variable v_k can assume a value in the range $[0, |v_k|)$ where $|v_k|$ is a finite natural number. We refer to $|v_k|$ as the *bound* of variable v_k.

- There is a total ordering $\succ$ on the variables: $v_L \succ v_{L-1} \ldots \succ v_1$.

- There are two *terminal* nodes **0** (representing *false*), and **1** (representing *true*). Terminal nodes have no children.
- Each *non-terminal* node p is labelled with some variable $v_k \in \mathcal{V}$ and can have up to $|v_k|$ children. Each child node, referred by $p[i]$, where $i \in [0, |v_k|)$, is either a terminal node or a non-terminal node labelled v_j such that $v_k \succ v_j$.
- No duplicate nodes: An MDD cannot have two non-terminal nodes p_i and p_j such that
  - p_i and p_j are labelled $v_k \in \mathcal{V}$, and
  - $\forall k \in [0, |v_k|), p_i[k] = p_j[k]$.
- No redundant nodes: An MDD cannot have a non-terminal node p_i such that $\forall k \in (0, |v_k|), p_i[0] = p_i[k]$.

MDDs represent functions of the form $\mathbb{N}^L \to \mathbb{B}$, with a finite set of variables L where each variable v_k has a finite bound $|v_k| \in \mathbb{N}$. We denote the size of a graph G as $|G|$ and the number of paths in it to terminal node **1** as $\Pi(G)$.

3.2.2.2 Extensible Decision Diagrams

Extensible Decision Diagram (XDDs) are a generalization of MDDs: like MDDs, XDDs are also defined over a finite set of variables L but the variables do not need to have finite bounds, i.e. $|v_k| \in \mathbb{N} \cup \infty$. XDDs represent functions of the form $(\mathbb{N} \cup \infty)^L \to \mathbb{B}$.

3.2.3 Overview of Symbolic Reachability Algorithms

We have previously described *explicit* reachability algorithms in Section 2.2, Therefore, in the rest of this section we give a brief overview of existing symbolic reachability algorithms.

3.2.3.1 Breadth-First Reachability

The first symbolic reachability algorithms explored the state space in *breadth-first* order, i.e. from the initial set of states $\mathcal{S}_0$, the states that are reachable in one step are found $\mathcal{S}_1$, followed by those that are reachable in two steps $\mathcal{S}_2$, and so on, until no more new states are discovered. This algorithm is shown in Algorithm 3.1. In this algorithm, $\mathcal{S}_{i+1}$ is built by finding the states reachable from $\mathcal{S}_i$ by firing each event $e \in \mathcal{E}$ on the set of unexplored states U and accumulating the result. U is known as the *frontier* set as it only contains the states that need to be explored.

Algorithm 3.1 Breadth-First Reachability.

```
procedure BFS($\mathbf{i}_0$)                                   • $\mathbf{i}_0$: initial states
    $S \leftarrow \emptyset$;                                           • Reachable states
    $U \leftarrow \mathbf{i}_0$;                                          • Unexplored states
    repeat
        $S \leftarrow S \cup U$;
        $U' \leftarrow \emptyset$;                       • $U'$: states reachable in one step from $U$
        for all $e \in \mathcal{E}$ do
            $U' \leftarrow U' \cup \mathcal{N}_e(U)$;
        $U \leftarrow U' \setminus S$;
    until $U = \emptyset$;
    return $S$;
```

3.2.3.2 Breadth-First with Event Chaining

A significant improvement over the BFS algorithm was *Chaining* (Roig et al., 1995) described in Algorithm 3.2. In BFS, every $\mathcal{N}_e(U)$ in iteration i is performed over the same set U. On the other hand, in Chaining, every $\mathcal{N}_e(S')$ is performed over the set of all states that have been discovered so far. This is likely to reduce the number of calls to $\mathcal{N}_e$, and has been shown to significantly reduce the time- and memory-requirements for reachability analysis compared to BFS.

Algorithm 3.2 Breadth-First Reachability with Chaining.

```
procedure CHAINING($\mathbf{i}_0$)                              • $\mathbf{i}_0$: initial states
    $S \leftarrow \mathbf{i}_0$;                                          • Reachable states
    $S' \leftarrow \mathbf{i}_0$;           • States reachable from $S$ in one additional iteration
    repeat
        $S \leftarrow S'$;
        for all $e \in \mathcal{E}$ do
            $S' \leftarrow S' \cup \mathcal{N}_e(S')$;
    until $S' = S$;
    return $S$;
```

If $S'_{i,0}$ and $S'_{i,|\mathcal{E}|}$ denote S' at the start and end respectively, of iteration i, then their relationship can be recursively expressed as

$$S'_{i,j} = S'_{i,j-1} \cup \mathcal{N}_{e_j}(S'_{i,j-1})$$

for all $j \in [1, |\mathcal{E}|]$.

While it may be obvious that Chaining reduces the number of calls to $\mathcal{N}_e(S')$, it may not be obvious how this reduces the time- and space-requirements compared to BFS. We note that $\mathcal{N}_e$ is implemented as a symbolic algorithm, and therefore its time- and space-requirements are dependent on the size of the MDDs representing event e and S', i.e. $|e|$ and $|S'|$ respectively, and not on the cardinality of e and S', i.e. $\Pi(e)$ and $\Pi(S')$ respectively. For many MDDs S_i and S_j, $|S_i \cup S_j| \ll \Pi(S_i \cup S_j)$. In addition, $|S_i \cup S_j| < |S_i| + |S_j|$. Therefore, $\mathcal{N}_e(S_i \cup S_j)$ will usually require less work than $\mathcal{N}_e(S_i) \cup \mathcal{N}_e(S_j)$, and this is borne out by the results in Roig et al. (1995).

3.2.3.3 Saturation

Ciardo et al. (2001) introduced the Saturation technique for generating the reachable states of Petri Nets. Saturation takes advantage of the *locality* of transitions to determine the order in which they are fired. To recall, each PN transition affects a subset of PN places. When representing $\mathcal{N}_e$ and S as MDDs, the order of variables (i.e. places) can be used to order the transitions by the *highest* variable they depend on or effect. An example is shown in Figure 3.1: assuming a variable order of $v_5 \succ v_4 \succ v_3 \succ v_2 \succ v_1$, the events in descending order of "top" variable are: e_1, e_2, e_3, e_4, e_6, e_5.

In BFS and Chaining, each iteration fires every transition once. In Saturation, the transitions fired depends on the MDD node being *saturated*. If the node p being saturated is labelled v_k, then all transitions whose highest variable is at or below v_k, $\mathcal{E}_{\leq v_k}$, are fired until a fixed point is reach. Node p is then considered to be saturated with respect to $\mathcal{E}_{\leq v_k}$. The saturation algorithm is described in Algorithm 3.3. Saturation, in many instances, was shown to orders of magnitude faster than Chaining and BFS for reachable set generation.

Algorithm 3.3 Saturation Part 1.

```
1: procedure $\textsc{Saturate}(s)$                                  • $s$: qrmdd node
2:     if $s$ is a terminal node then
3:         return $s$
4:     if Cache contains $\langle \mathsf{Saturate}, s, r \rangle$ then
5:         return $r$
6:     $r \leftarrow \emptyset$
7:     for $i \in [0, |s|)$ do
8:         $r_i \leftarrow \textsc{Saturate}(s_i)$
9:     repeat
10:        for all $e \in \mathcal{E}$ s.t. $\textsc{Top}(e) = l(s)$ do
11:            for $i \in [0, |r|)$ do
12:                for $j \in [0, |e_i|)$ do
13:                    $r'_j \leftarrow \textsc{RecFire}(r_i,\ e_{i,j})$        • See Algorithm 3.4
14:                    $r_j \leftarrow r_j \cup r'_j$
15:    until $r$ is a fix-point
16:    $r' \leftarrow \textsc{UniqueTableInsert}(r)$
17:    Store $\langle \mathsf{Saturate}, s, r' \rangle$ in Cache
18:    return $r'$
```

In addition, the memory requirement were also orders of magnitude smaller. One of the drawbacks of symbolic reachability algorithms is that the *peak* memory requirements are usually many times larger than the final memory requirements. This is because the the MDD representing the final set of reachable states can be much more compact than an intermediate set. Therefore, during the construction of the set of reachable states it is desirable to build nodes that are likely to be a part of the final MDD, and Saturation aims to do just that (saturated nodes), and consequently, tends to use significantly lower peak memory than BFS and Chaining.

3.2.3.4 Constrained Saturation

Constrained saturation, introduced in Zhao and Ciardo (2009), constrained state exploration to a set of states satisfying given properties, while still being able to exploit event locality and recursive local fixed point computations (*vis-à-vis* Saturation). The constraint is represented as an MDD and used in the recursive calls (of Saturation) to limit the paths to be explored. This

Algorithm 3.4 Saturation Part 2.

```
procedure RecFire($s$, $e$)                                   ▷ $s$: states, $e$: transition
    if $s$ is 0 or $e$ is 0 then
        return 0
    if $e$ is 1 then
        return $s$
    if Cache contains $\langle \mathsf{RecFire}, s, e, r \rangle$ then
        return $r$
    $r \leftarrow \emptyset$
    for $i \in [0, |s|)$ do
        for $j \in [0, |e_i|)$ do
            $r'_j \leftarrow$ RecFire($s_i$, $e_{i,j}$)
            $r_j \leftarrow r_j \cup r'_j$
    $r' \leftarrow$ UniqueTableInsert($r$)
    $r'' \leftarrow$ Saturate($r'$)
    Store $\langle \mathsf{RecFire}, s, e, r'' \rangle$ in Cache
    return $r''$
```

algorithm is shown to be effective at building the set of states satisfying EU and EG properties (EU and EG are temporal operators in Computation Tree Logic).

Now that we have seen various symbolic algorithms for reachability analysis, the next section will discuss using these algorithms for detecting deadlocks in Petri Nets.

3.3 Deadlock Detection in Petri Nets

Petri Net theory tells us that the deadlock detection problem is reducible to the general reachability problem. Mayr (1984) showed that reachability is decidable for (General) Petri Nets, and Lipton (1976) showed that reachability belongs to EXSPACE by describing a PN to generate 2^{2^n} tokens in a net with kn places, where k is a constant. Therefore even though deadlock detection for PNs is decidable in the general case, it is still a hard problem.

The language described by PNs, $\mathcal{L}(PN)$ is a superset of Regular languages. PNs can represent some (but not all) Context-free and Context-sensitive grammars, and $\mathcal{L}(PN)$ is a proper subset of Context-sensitive languages. Adding inhibitor arcs to Petri Nets makes them Turing-equivalent under certain cases, and therefore undecidable (Hack, 1976; Esparza, 1998).

In this section, we define what a deadlock means in a Petri Net, and describe symbolic algorithms for *bounded* Petri Nets. Later, we provide *semi*-algorithms that are guaranteed to find a deadlock in unbounded Petri Nets.

Definition 3.3.1 Bounded Petri Net

A k-bounded Petri Net is one in which every place can have at most k tokens, where k is a finite non-negative integer. Any k-bounded Petri Net is a bounded Petri Net.

Definition 3.3.2 Unbounded Petri Net

Any Petri Net that is not bounded is an unbounded Petri Net.

Definition 3.3.3 Petri Nets with Inhibitor Arcs

An inhibitor arc in a PN, is an edge between a place $v_k \in \mathcal{V}$ and a transition $e \in \mathcal{E}$. We define $I : |\mathcal{V}| \to |\mathcal{E}|$, to be a partial relation such that $(v_k, e_i) \in I$ if and only if there is an inhibitor arc from place v_k to transition e_i.

If a PN has inhibitor arcs, the enabling condition for any transition must also satisfy the following: if a place $v_k \in \mathcal{V}$ has one or more tokens in it, then any transition $e_i \in \mathcal{E}$ such that $(v_k, e_i) \in I$ is disabled. Formally, we say that an event e_j is *enabled* in state i iff

1. $\forall k \in [1, L], i_k \geq \delta^-_{e_j, v_k}$, and
2. $\forall k \in [1, L], (v_k, e_j) \in I \implies i_k = 0$.

We note that all bounded Petri Nets with inhibitor arcs are decidable under reachability. Indeed, even unbounded Petri Nets with inhibitor arcs are decidable under reachability as long as the places from which the inhibitor arcs originate are bounded.

Definition 3.3.4 Potential Deadlock

We define a *potential* deadlock in a Petri Net as any marking that disables all transitions.

Definition 3.3.5 Deadlock

We define a deadlock to be any potential deadlock that is also reachable.

3.3.1 Building the Set of Potential Deadlock States, C

The set of potential deadlock states, C can be built using δ^- and I. For each transition, e_i, the algorithm builds an XDD representing the disjunction of its *disabling* conditions (using $\delta^-_{e_i}$ and I_{e_i}). Recall that a transition e_i is disabled in marking m when

$$\exists v_k \in \mathcal{V}, m(v_k) < \delta^-_{e_i,v_k} \vee (m(v_k) > 0 \wedge (v_k, e_i) \in I).$$

The XDD representing the conjunction of the disabling conditions for each event represents the set of potential deadlock states, and this construction is described in Algorithm 3.5, and an example is illustrated in Figure 3.2.

Algorithm 3.5 Algorithm for building the set of potential deadlock states.

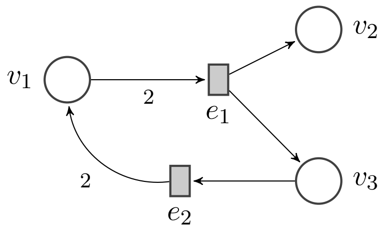
```
1:  procedure BuildDeadlockStates($\mathcal{E}$)                ● $\mathcal{E}$ is the set of transitions
2:      $C \leftarrow true$
3:      for all $e_i \in \mathcal{E}$ do
4:          $C_i \leftarrow false$
5:          for all $\delta^-_{e_i,v_k} > 0$ do
6:              $C_{i,k} \leftarrow$ XDD respresenting $v_k < \delta^-_{e_i,v_k}$
7:              $C_i \leftarrow C_i \vee C_{i,k}$
8:          for all $(v_k, e_i) \in I$ do
9:              $C_{i,k} \leftarrow$ XDD respresenting $v_k > 0$
10:             $C_i \leftarrow C_i \vee C_{i,k}$
11:         $C \leftarrow C \wedge C_i$
12:     return $C$
```

3.3.2 Deadlock detection using S and C

According to Definition 3.3, a deadlock is any potential deadlock that is also reachable. Therefore, a straightforward approach to detecting deadlocks in bounded Petri Nets is described in Algorithm 3.6:

1. Build the set of reachable states S using existing reachability algorithms,

2. Build the set of potential deadlock states C using the procedure described in Algorithm 3.5, and



Enabling conditions:

$$\begin{aligned} e_1 &: v_1 \geq 2 \\ e_2 &: v_3 \geq 1 \end{aligned}$$

Disabling conditions:

$$\begin{aligned} e_1 &: v_1 < 2 \\ e_2 &: v_3 < 1 \end{aligned}$$

Potential deadlock states:

$$(v_1 < 2) \wedge (v_3 < 1)$$

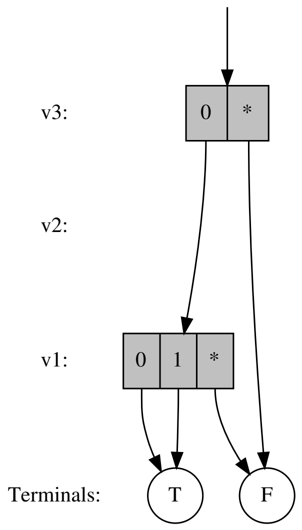


Figure 3.2 Building the XDD representing the constraint C for a Petri Net.

3. The intersection of S and C gives the deadlock states.

Any of the reachability algorithms described in Section 3.2.3 can be used for generating S.

Algorithm 3.6 Deadlock detection using S and C.

```
1: procedure FindDeadlock($S_0$, $\mathcal{E}$)                ● $S_0$: initial states, $\mathcal{E}$: events
2:     $S \leftarrow$ ReachableStates($\mathcal{E}$,$S_0$)
3:     $C \leftarrow$ BuildDeadlockStates($\mathcal{E}$)
4:     if $S \cap C$ is empty then
5:         return false
6:     else
7:         return true
```

3.3.3 Deadlock detection using Interrupted Symbolic Reachability Algorithms

The next set of deadlock detection algorithms are also for bounded Petri Nets. In these we employ the BFS, Chaining, Saturation and Constrained Saturation algorithms for generating S, but during every "iteration" we check if we have reached a potential deadlock state.

Checking if we have reached a potential deadlock state becomes an important operation. If S' is the MDD representing the intermediate set of reachable states for which we want to perform this check, we could perform the check as follows: "is $S' \cap C = \emptyset$?". If the answer is "false" then a deadlock state has been discovered in S'. But, computing $S' \cap C$ is likely to create many MDD nodes that may never be used again, and increase the peak memory requirements. We instead use the symbolic procedure IsIntersectionEmpty described in Algorithm 3.7 to answer this query without needing to build any nodes.

Algorithms 3.8, 3.9, and 3.10 describe these "interrupted" reachability based deadlock detection algorithms for BFS, Chaining and Saturation respectively.

3.3.3.1 Interrupted BFS

Algorithm 3.8 describes *Interrupted* BFS for deadlock detection. There are two choices for when to perform the deadlock check: every time U' is updated, or when U is updated. In Algorithm 3.8, we perform this check when U is updated.

Algorithm 3.7 Check if $A \cap B$ is empty without building an MDD for $A \cap B$.

```
1: procedure IsIntersectionEmpty($A$, $B$)                    • $A$, $B$: mdds
2:     if $A = \mathbf{0} \lor B = \mathbf{0}$ then
3:         return $true$
4:     else if $A = \mathbf{1} \lor B = \mathbf{1}$ then
5:         return $false$
6:     else if $\langle \mathsf{IsIntersectionEmpty}, A, B, r\rangle \in \mathsf{Cache}$ then
7:         return $r$
8:     $r \leftarrow true$
9:     for $i \in [0, \textsc{Min}(|A|, |B|))$ and $r = true$ do
10:        $r \leftarrow \textsc{IsIntersectionEmpty}(A_i, B_i)$
11:    Store $\langle \mathsf{IsIntersectionEmpty}, A, B, r\rangle$ in $\mathsf{Cache}$
12:    return $r$
```

Algorithm 3.8 Deadlock detection using Interrupted BFS.

```
1: procedure InterruptedBFS($S_0$, $C$, $\mathcal{E}$)        • $S_0$: initial states, $C$: constraint
2:     $S \leftarrow \emptyset$;                                • Reachable states
3:     $U \leftarrow S_0$;                                      • Unexplored states
4:     repeat
5:         $S \leftarrow S \cup U$;
6:         if IsIntersectionEmpty($S$, $C$) is $false$ then
7:             return $\langle true, S\rangle$
8:         $U' \leftarrow \emptyset$;                           • $U'$: states reachable in one step from $U$
9:         for all $e \in \mathcal{E}$ do
10:            $U' \leftarrow U' \cup \mathcal{N}_e(U)$;
11:        $U \leftarrow U' \setminus S$;
12:    until $U = \emptyset$;
13:    return $\langle false, S\rangle$;
```

3.3.3.2 Interrupted Chaining

Algorithm 3.9 describes Interrupted Chaining for deadlock detection. Again, there are two choices for when to perform the deadlock check: when S' is updated, or when S is updated. In Algorithm 3.9, we perform this check when S' is updated.

Algorithm 3.9 Deadlock detection using Interrupted Chaining.

```
procedure InterruptedChaining($S_0$, $C$, $\mathcal{E}$)            • $S_0$: initial states, $C$: constraint
    $S \leftarrow \emptyset$;                                          • Reachable states
    $S' \leftarrow S_0$;       • States reachable from $S$ in one additional iteration
    repeat
        $S \leftarrow S'$;
        if IsIntersectionEmpty($S$, $C$) is $false$ then
            return $\langle true, S \rangle$
        for all $e \in \mathcal{E}$ do
            $S' \leftarrow S' \cup \mathcal{N}_e(S')$;
    until $S' = S$;
    return $\langle false, S \rangle$;
```

3.3.3.3 Interrupted Saturation

Algorithm 3.10 describes Interrupted Saturation for deadlock detection. This algorithm is similar to the Constrained Saturation algorithm described in Section 3.2.3.4 except that it does not use the constraint to limit the paths that are explored, and instead uses it to check for deadlocks once a node has been saturated. There are some additional details in Algorithm 3.10, to ensure that any saturated node p retains the same semantics as before, namely, p is saturated with respect to $\mathcal{E}_{\leq v_k}$.

Algorithm 3.10 Deadlock detection through Interrupted Saturation.

```
procedure InterruptedSaturation($S_0$, $C$, $\mathcal{E}$)          • $S_0$: initial states, $C$: constraint
    if IsIntersectionEmpty($S_0$, $C$) is $true$ then                  • See Algorithm 3.7
        return IntSaturate($S_0$, $C$, $\mathcal{E}$)                  • See Algorithm 3.11
    else
        return $\langle true, S_0 \rangle$
```

Algorithm 3.11 Interrupted Saturation Part 1.

```
procedure IntSaturate($s$, $c$, $\mathcal{E}$)                    • $s$: states qrmdd, $c$: constraint qrxdd
    if $s$ is a terminal node then
        return $\langle \neg$IsIntersectionEmpty$(s, c), s\rangle$
    if $c$ is a terminal node then
        if $c = \mathbf{0}$ then
            return $\langle false,$ Saturate$(s)\rangle$
        else
            return $\langle true, s\rangle$                          • $\because$ $s$ is non-empty
    if Cache contains $\langle$IntSaturate$, s\rangle$ then
        $r \leftarrow$ Cache$\langle$IntSaturate$, s\rangle$
        return $\langle \neg$IsIntersectionEmpty$(r, c), r\rangle$
    $r \leftarrow \emptyset$
    for $i \in [0, |s|)$ do
        $\langle status, r_i\rangle \leftarrow$ IntSaturate$(s_i, c_i)$
        if $status$ is $true$ then
            return $\langle true, r\rangle$
    repeat
        for all $t \in \mathcal{E}$ s.t. Top(t) = Level(s) do
            for $i \in [0, |r|)$ do
                for $j \in [0, |t_i|)$ do
                    $\langle status, r'_j\rangle \leftarrow$ IntRecFire$(r_i, c_j, t_{i,j}, \mathcal{E})$      • See Algorithm 3.12
                    $r_j \leftarrow r_j \cup r'_j$
                    if $status$ is $true$ then
                        $r' \leftarrow$ UniqueTableInsert$(r)$
                        return $\langle true, r'\rangle$
    until $r$ is a fix-point
    $r' \leftarrow$ UniqueTableInsert$(r)$
    Store $\langle$IntSaturate$, s, r'\rangle$ in Cache
    return $\langle false, r'\rangle$
```

Algorithm 3.12 Interrupted Saturation Part 2.

```
procedure IntRecFire($s$, $c$, $t$, $\mathcal{E}$)                ● $s$: states, $c$: constraint, $t$: transition
    if $c$ is 0 then
        return $\langle false$, RecFire($s$, $t$, $\mathcal{E}$) $\rangle$
    if $s$ is 0 or $t$ is 0 then
        return $\langle false, 0\rangle$
    if $t$ is 1 then
        return $\langle \neg$IsIntersectionEmpty$(s, c), s\rangle$
    if Cache contains $\langle$IntRecFire$, s, t\rangle$ then
        $r \leftarrow$ Cache$\langle$IntRecFire$, s, t\rangle$
        return $\langle \neg$IsIntersectionEmpty$(r, c), r\rangle$
    $r \leftarrow \emptyset$
    for $i \in [0, |s|)$ do
        for $j \in [0, |t_i|)$ do
            $\langle status, r'_j\rangle \leftarrow$ IntRecFire($s_i$, $c_j$, $t_{i,j}$, $\mathcal{E}$)
            $r_j \leftarrow r_j \cup r'_j$
            if $status$ is $true$ then
                $r' \leftarrow$ UniqueTableInsert($r$)
                return $\langle true, r'\rangle$
    $r' \leftarrow$ UniqueTableInsert($r$)
    $\langle status, r''\rangle \leftarrow$ IntSaturate($r'$, $c$, $\mathcal{E}$)
    if $status$ is $false$ then
        Store $\langle$IntRecFire$, s, t, r''\rangle$ in Cache
    return $\langle status, r''\rangle$
```

All of these algorithms will construct the reachable set of states if no deadlock is found, and will construct a partial set of reachable states S' otherwise, containing at least one deadlock state.

3.3.4 Deadlock Detection in Unbounded Petri Nets

Next we discuss modifications to existing reachability algorithms that can be used to find deadlocks in unbounded Petri Nets.

Breadth-first exploration algorithms such as Interrupted BFS explore states at increasing *distances* from the set of initial states, where distance of state s_i from state s_j is the minimum number of events that must be fired to reach i from j.

Interrupted Chaining can also be placed in a similar category as Interrupted BFS since every iteration fires each transition once. Therefore any state discovered in iteration k, is (at most) at a distance of $k \times |\mathcal{E}|$ steps from the set of initial states.

If a deadlock exists, Interrupted BFS and Interrupted Chaining are guaranteed to find it even if the Petri Net is unbounded. If a deadlock does not exist in a unbounded Petri Net, neither algorithm will terminate, so we refer to them as semi-algorithms. But, as discussed previously, this is an undecidable problem, and building an efficient semi-algorithm is the best we can hope for.

Interrupted Saturation, on the other hand, pursues a depth-first exploration strategy, and makes no guarantees on the order in which states are discovered with respect to the distance from the set of initial states. Therefore, if a Petri Net is unbounded, we cannot guarantee that Interrupted Saturation will find a deadlock (if one exists).

3.3.4.1 Interrupted Saturation with Interrupted Chaining

We describe a algorithm that combines Interrupted Saturation with Interrupted Chaining that is guaranteed to find a deadlock in an unbounded Petri Net, if one exists.

Definition 3.3.6 *Net Effect* of a Transition

The Net Effect δ_e of a PN transition $e \in \mathcal{E}$, is defined as

$$\delta_e = \sum_{v_k \in \mathcal{V}} \delta_{e,v_k}.$$

Definition 3.3.7 Net Effect Transitions

- A *positive net effect* transition as any transition $e \in \mathcal{E}$ that increases the number of tokens in the Petri Net when fired. The set of all such transitions is denoted by $\mathcal{E}^{>0}$, and defined as

$$\mathcal{E}^{>0} = \{e | e \in \mathcal{E} \wedge \delta_e > 0\}.$$

- A *negative net effect* transition as any transition $e \in \mathcal{E}$ that decreases the number of tokens in the Petri Net when fired. The set of all such transitions is denoted by $\mathcal{E}^{<0}$, and defined as

$$\mathcal{E}^{<0} = \{e | e \in \mathcal{E} \wedge \delta_e < 0\}.$$

- A *no net effect* transition as any transition $e \in \mathcal{E}$ that does not change the number of tokens in the Petri Net when fired. The set of all such transitions is denoted by $\mathcal{E}^{=0}$, and defined as

$$\mathcal{E}^{=0} = \{e | e \in \mathcal{E} \wedge \delta_e = 0\}.$$

- We also define the set of transitions that do not increase the number of tokens in the Petri Net when fired, $\mathcal{E}^{\leq 0}$ as

$$\mathcal{E}^{\leq 0} = \{e | e \in \mathcal{E} \wedge \delta_e \leq 0\} = \mathcal{E}^{<0} \cup \mathcal{E}^{=0}.$$

- We say that a marking m is *covered* by marking m', $m \leq m'$ if

$$\forall k \in \mathcal{V}, m_k \leq m'_k.$$

- We say m is *strictly* covered by m', $m < m'$, if

  1. $m \leq m'$, and

2. $\exists k \in \mathcal{V}, m_k < m'_k$

We observe that, in a general Petri Net (no inhibitor arcs), if a marking m enables transition e, then any marking m' such that $m < m'$ also enables e. Also, by definition, m must have fewer tokens than m'. In any unbounded Petri Net, there must be markings m and m', such that $m < m'$ (Karp and Miller, 1969). Therefore, if e^* is a bag of transitions that transforms m to m', $m \xrightarrow{e^*} m'$, there must be some $e_i \in e^*$ such that $e_i \in \mathcal{E}^{>0}$.

We have previously stated that we cannot guarantee that a depth-first exploration strategy such as Interrupted Saturation will find a deadlock (if one exists) in an unbounded Petri Net. We know that Interrupted Saturation will terminate if a fix-point can be computed. We also know that Interrupted BFS and Interrupted Chaining are guaranteed to find a deadlock (if one exists) even if the fix-point cannot be computed. One way to improve Interrupted BFS and Interrupted Chaining is to hand-off exploration of a bounded subspace of markings to Interrupted Saturation.

If we restrict Interrupted Saturation to the set of events in $\mathcal{E}^{\leq 0}$, then, as shown below, it is guaranteed to terminate since the set of reachable markings with respect to any initial marking and $\mathcal{E}^{\leq 0}$, is finite.

Theorem 3.3.1

For any finite marking, and a set of events that do not increase the token count in the Petri Net, the set of reachable markings is finite.

Proof: Let n represent the token count in the given marking. By definition, n is a non-negative integer. The set of states whose token count is at most n, $S^{\leq n}$, is a subset of $\mathcal{V}^n = v_1 \times v_2 \times \cdots \times v_{|\mathcal{V}|}$ where each $v_i \in [0, n]$. $\mathcal{V}^n$ is a finite set, therefore $S^{\leq n}$ is also finite. Since no event can increase the token count in any marking reachable from the initial marking, it follows that the state space to be explored is a subset of $S^{\leq n}$. Since $S^{\leq n}$ is finite, the set of reachable markings is finite. $\square$

This leads to Algorithm 3.13 for the detection of deadlocks in unbounded Petri Nets. It classifies the events into two sets, $\mathcal{E}^{\leq 0}$ and $\mathcal{E}^{>0}$, and calls InterruptedSaturation on $\mathcal{E}^{\leq 0}$ and InterruptedChaining on $\mathcal{E}^{>0}$, until either a deadlock is found, or, if the reachable state space is finite, a fixed-point is reached.

Algorithm 3.13 Deadlock detection in unbounded Petri Nets with Saturation.

```
procedure InterruptedSaturationWithChaining($S_0$, $\mathcal{E}$)                • $S_0$: initial states
    $S \leftarrow S_0$;
    $C \leftarrow$ BuildDeadlockStates($\mathcal{E}$);
    $\mathcal{E}^{>0} \leftarrow \{e | e \in \mathcal{E} \wedge \delta_e > 0\}$;
    $\mathcal{E}^{\leq 0} \leftarrow \mathcal{E} \setminus \mathcal{E}^{>0}$;
    repeat
        $\langle r, S \rangle \leftarrow$ InterruptedSaturation($S$,$C$,$\mathcal{E}^{\leq 0}$);
        if $r = true$ then return $\langle true, S \rangle$;
        $\langle r, S \rangle \leftarrow$ InterruptedChaining($S$,$C$,$\mathcal{E}^{>0}$);
        if $r = true$ then return $\langle true, S \rangle$;
    until $S$ is a fixed point
    return $\langle false, S \rangle$;
```

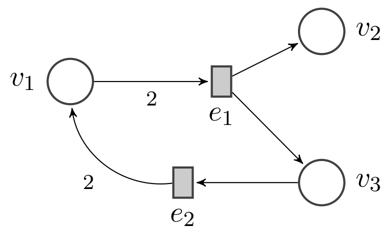
3.3.4.2 Interrupted Saturation with Interrupted Chaining and Invariant Analysis

We hypothesize that we can improve Algorithm 3.13 by increasing the cardinality of $\mathcal{E}^{\leq 0}$, i.e. the events that we can hand-off to Saturation. We make the following observations:

1. Any place $v_k \in \mathcal{V}$ can be marked *bounded* if $\forall e \in \mathcal{E}$, $\delta_{e,v_k} \leq 0$. In other words, if none of the transitions increase the number of tokens in v_k, the number of tokens in v_k in any reachable marking is bounded, and is at most $S_0(v_k)$, the number of tokens placed in v_k by the initial marking S_0.

2. Any place $v_k \in \mathcal{V}$ that is a part of a *p-semiflow* is bounded. In other words, if a Petri Net satisfies the invariant, $a.v_i + b.v_j = c$, where $a, b, c \in \mathbb{N}$, and $v_i, v_j \in \mathcal{V}$, then v_i and v_j are bounded by c. Also, since the invariant must be true of the initial marking, it follows that $c = a.S_0(v_i) + b.S_0(v_j)$.

Our idea for enlarging the number of transitions in $\mathcal{E}^{\leq 0}$ is to ignore bounded places when computing the net effect of transitions.



$$\mathcal{V} = \{v_1, v_2, v_3\}$$
$$\mathcal{E} = \{e_1, e_2\}$$
$$\delta^- = \begin{array}{c|cc} & e_1 & e_2 \\ v_1 & 2 & \\ v_2 & & \\ v_3 & & 1 \end{array}$$
$$\delta_{\mathcal{V}} = \begin{array}{c|cc} & e_1 & e_2 \\ v_1 & -2 & +2 \\ v_2 & +1 & \\ v_3 & +1 & -1 \\ -- & -- & -- \\ \delta_{\mathcal{V},e} & 0 & +1 \end{array}$$
$$\mathcal{E}^{>0} = \{e_2\}$$

$$\text{Invariant} \;:\; v_1 + 2 \times v_3 = c$$
$$\hat{\mathcal{V}} = \{v_1, v_3\}$$
$$\mathcal{V}' = \{v_2\}$$
$$\delta_{\mathcal{V}'} = \begin{array}{c|cc} & e_1 & e_2 \\ v_2 & +1 & \\ -- & -- & -- \\ \delta_{\mathcal{V}',e} & +1 & 0 \end{array}$$
$$\mathcal{E}^{>0}_{\mathcal{V}'} = \{e_1\}$$

Figure 3.3 A Petri net model showing the effect of invariant analysis on $\mathcal{E}^{>0}$.

Definition 3.3.8 Net Effect of a Transition with respect to a Set of Places

$\delta_{e,\mathcal{V}'}$, the Net Effect of a transition $e \in \mathcal{E}$ with respect to places $\mathcal{V}' \subseteq \mathcal{V}$, is defined as

$$\delta_{e,\mathcal{V}'} = \sum_{v_k \in \mathcal{V}'} \delta_{e,v_k}.$$

Definition 3.3.9 $\mathcal{E}^{>0}$ with respect to a Set of Places

We define the set of places as $\hat{\mathcal{V}} \subseteq \mathcal{V}$, and the set of positive effect transitions with respect to $\hat{\mathcal{V}}$ as

$$\mathcal{E}^{>0}_{\hat{\mathcal{V}}} = \{e | e \in \mathcal{E}, \delta_{e,\hat{\mathcal{V}}} > 0\}.$$

Algorithm 3.14 describes our refinement to Algorithm 3.13. The only difference between the two algorithms is in the construction of set $\mathcal{E}^{>0}$. Figure 3.3 illustrates the computation $\mathcal{E}^{>0}$ over $\mathcal{V}$ and $\mathcal{V}'$, where $\mathcal{V}' = \mathcal{V} \setminus \hat{\mathcal{V}}$, and $\hat{\mathcal{V}}$ is the set of places that are bounded based on invariant analysis.

Algorithm 3.14 Interrupted Saturation with Invariant Analysis.

```
procedure InterruptedSaturationWithChaining($S_0$, $\mathcal{E}$)        • $S_0$: initial states
    $S \leftarrow S_0$;
    $C \leftarrow$ BuildDeadlockStates($\mathcal{E}$);
    Build $\hat{\mathcal{V}} \subseteq \mathcal{V}$, the set of bounded places;
    $\mathcal{V}' \leftarrow \mathcal{V} \setminus \hat{\mathcal{V}}$;
    $\mathcal{E}^{>0} \leftarrow \{e | e \in \mathcal{E} \wedge \delta_{e,\mathcal{V}'} > 0\}$;
    $\mathcal{E}^{\leq 0} \leftarrow \mathcal{E} \setminus \mathcal{E}^{>0}$;
    repeat
        $\langle r, S\rangle \leftarrow$ InterruptedSaturation($S$,$C$,$\mathcal{E}^{\leq 0}$);
        if $r = true$ then return $\langle true, S\rangle$;
        $\langle r, S\rangle \leftarrow$ InterruptedChaining($S$,$C$,$\mathcal{E}^{>0}$);
        if $r = true$ then return $\langle true, S\rangle$;
    until $S$ is a fixed point
    return $\langle false, S\rangle$;
```

3.4 Experimental evaluation

The algorithms discussed in Section 3.3 were implemented using Meddly (Babar and Miner, 2010) and SMART (Ciardo et al., 2009). Experiments were run on a 2.13 Ghz Intel Xeon processor running Linux, with sufficient RAM to avoid paging to disk.

We tested our algorithms on a variety of Petri Net models taken from the 2018 Model Checking Contest https://mcc.lip6.fr/2018/. Most of these models have scaling parameters that affect their size and complexity, yielding $N = 767$ model instances.

3.4.1 Bounded Petri Nets

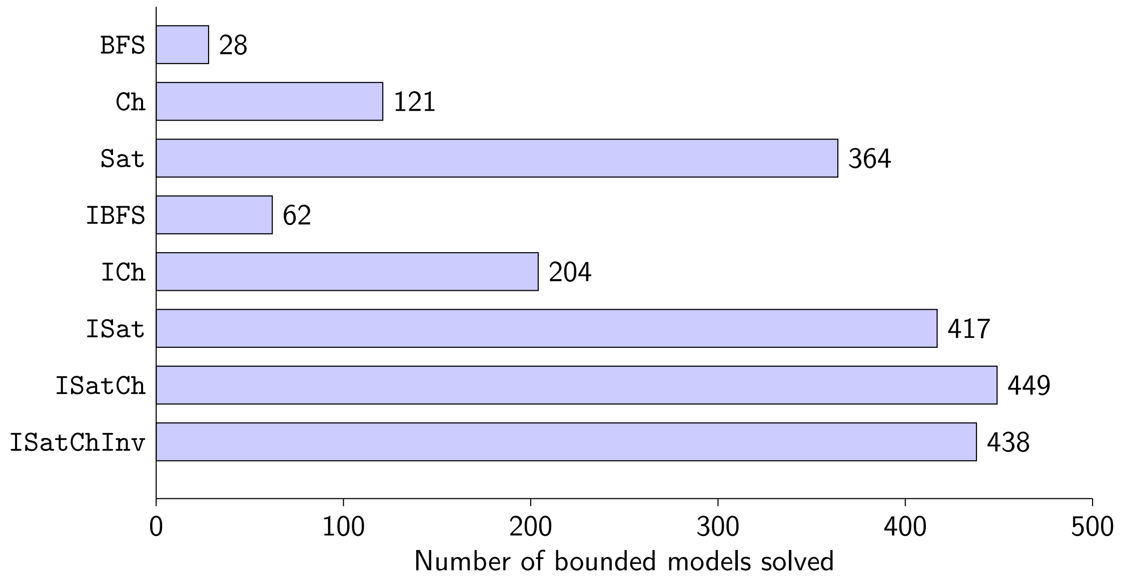


Figure 3.4 Results for bounded Petri Nets.

All but one of the 767 model instances are known to be bounded. Figure 3.4.1 gives the results for these bounded models. We used a time limit of 3600 seconds and a memory limit of 16 gigabytes for each model instance and counted the number of model instances each algorithm is able to complete. In the table, `BFS`, `Ch`, `Sat`, `IBFS`, `ICh`, `ISat`, `ISatCh` and `ISatChInv` refer to Breadth-first reachability described in Algorithm 3.1, Chaining in Algorithm 3.2, Saturation in Algorithm 3.3, Interrupted BFS in Algorithm 3.8, Interrupted Chaining in Algorithm 3.9, Interrupted Saturation in Algorithm

3.10, Interrupted Saturation with Chaining in Algorithm 3.13, and Interrupted Saturation with Chaining and Invariant Analysis in Algorithm 3.14, respectively.

The table clearly shows that `Sat` outperforms `Ch` which in turn outperforms `BFS`. Based on existing empirical evidence, this was the expected result. The corresponding interrupted counterparts `IBFS`, `ICh` and `ISat` were able to solve a larger number of models with `ISat` solving more than `ICh`, and `ICh` in turn solving more than `IBFS`, although the difference between `ISat` and `ICh` is proportionally smaller than the difference between `Sat` and `Ch`. This may indicate that in these models the distance between the initial state and the deadlock is relatively small.

Surprisingly `ISatCh` and `ISatChInv` did quite well on these bounded models. This may be partially due to the distance of the deadlocks from the initial state, as discussed above. In addition, `ISatCh` and `ISatChInv` are biased towards firing the transitions that remove tokens from the PN, which may have been beneficial for these set of models.

3.4.2 Potentially Unbounded Petri Nets

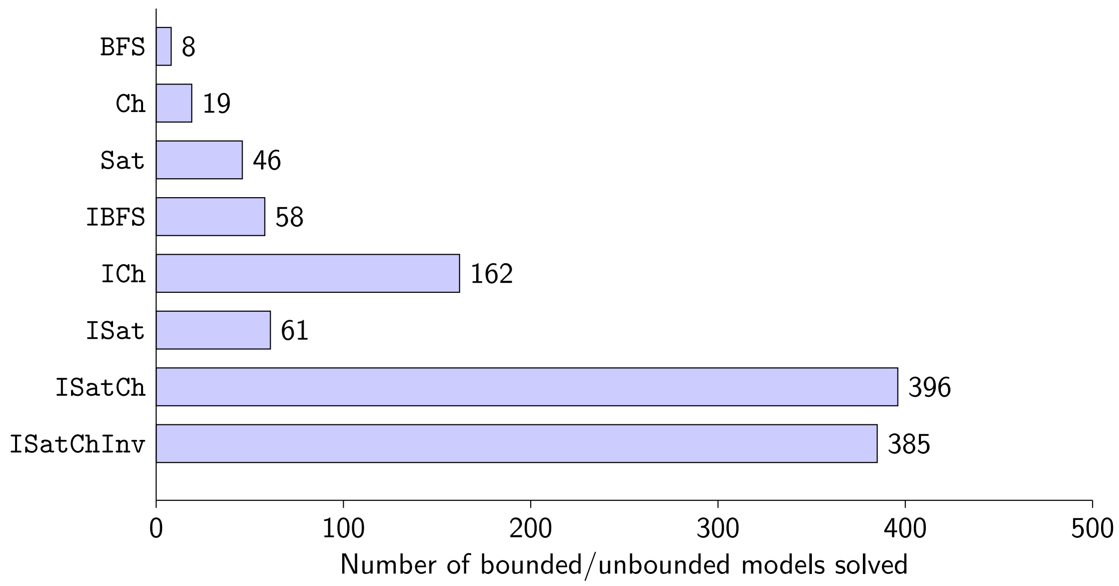


Figure 3.5 Results for bounded/unbounded Petri Nets.

Since only one of the MCC models is known to be unbounded, we modified these models to induce unboundedness. The procedure to induce unboundedness was relatively straightforward:

add an output arc from a randomly selected transition to a randomly selected place. We note that this is not guaranteed to make the model unbounded. Figure 3.4.2 gives the results for these models. Once again, we used a time limit of 3600 seconds and a memory limit of 16 gigabytes for each model instance and counted the number of model instances each algorithm is able to complete.

As was the case for bounded models, `Sat` performed better than `Ch` which in turn performed better than `BFS`. But they completed far fewer models when compared to the corresponding bounded model experiments. `ICh` performed much better than `ISat`, and `ISatCh` and `ISatInv` complete more than twice as many models as `ICh` and `ISat`.

3.5 Discussion

The experiments indicate that among the set of algorithms described in this chapter, `ISatCh` is best for bounded as well as for unbounded models. We think this may be a misleading conclusion as discussed below.

For the set of bounded models, the set of models solved by `ISat` is a superset of the models solved by `Sat`, but the set of models solved by `ISatCh` is not a superset of `ISat`. Compared to `ISatCh`, `ISat` is faster at computing the set of reachable states S, and detecting the absence of deadlocks. `ISatCh` on the other hand is better at detecting the presence of deadlocks since they are usually found at *relatively* short distances from initial state (at least in this set of models). We also note that almost 59% (432 out of 738) of the model instances are known to contain deadlocks, and may indicate why `ISatCh` performs better than `ISat` on this set of models.

For the set of unbounded models, `ISatCh` is clearly better than `ISat`. It is revealing how much worse `ISat` fares when the model is unbounded, and confirms our hypothesis that an *unguided* depth-first search in a potentially unbounded search space is likely to get *trapped* in an unbounded subspace. It may be possible to improve `ISat` by prioritizing the set of transitions based on their net effect.

`ISatChInv` did not do as well as we expected for these models, and the simpler `ISatCh` performed better. This may be due to the additional processing required for the invariant analysis, which may be significant since many of the models are rather large in terms of the number of places and transitions. It is also possible that removing the effect of bounded places in the calculation of the net effect was heavy-handed, and a more nuanced approach might yield better results.

We have discussed improvements to reachability analysis towards detecting deadlocks in Petri Nets. As stated in Section 3.1, structural analysis on Petri Nets can be extremely fast. State Equation analysis (Murata, 1989) and Siphon-Trap analysis (Hack, 1972), for example, can rule out the possibility of a deadlock in a Petri Net, or guarantee its presence without having to do any state space exploration. But, structural analysis techniques for deadlock detection in Petri Nets are known to be incomplete, i.e. the analysis may be inconclusive. Still, considering the cost of reachability analysis, it should only be performed when structural analysis is inconclusive. An additional advantage of performing structural analysis first, is that it may help reachability analysis by simplifying the Petri Net and by reducing the state space to be explored.

3.6 Conclusions

We have presented novel symbolic algorithms for detecting deadlocks in Petri Nets and given empirical evidence of their effectiveness in relation to existing symbolic algorithms. We have also demonstrated that these algorithms can be used to detect deadlocks in unbounded Petri Nets.

Interrupted Saturation was shown to outperform all existing symbolic algorithms for detecting deadlocks in bounded Petri Nets. Interrupted Saturation with Chaining performed very well even on unbounded Petri Nets and avoided the pitfalls of using a depth-first algorithm when exploring infinite state spaces.

We plan to integrate our algorithms into Petri Net analysis tools that perform structural analysis before reachability analysis, and compare them to explicit reachability techniques. Also, invariant analysis did not help Interrupted Saturation with Chaining as much as we expected, and we plan to make a more nuanced adjustment to the computation of net effect in the future.

Acknowledgment

This work was supported in part by the National Science Foundation under grant 420-71-69.

CHAPTER 4. BINARY DECISION DIAGRAMS WITH EDGE-SPECIFIED REDUCTIONS

Various versions of binary decision diagrams (BDDs) have been proposed in the past, differing in the reduction rule needed to give meaning to edges skipping levels. The most widely adopted, fully-reduced BDDs and zero-suppressed BDDs, excel at encoding different types of boolean functions (if the function has many "don't-care", or it tends to have value zero when one of its arguments is nonzero, respectively). Lately, new classes of BDDs have been proposed that, at the cost of some additional complexity and larger memory requirements per node, exploit both cases. In this chapter we introduce a new type of BDD, Binary Decision Diagrams with Edge-Specified Reductions or ESRBDDs, that we believe are conceptually simple, have a smaller memory requirements for each node, tend to need fewer nodes, and can easily be extended with additional reduction rules. We present a formal definition, prove canonicity, and provide experimental results to support our efficiency claims.

4.1 Introduction

Decision diagrams (DDs) have been widely adopted for a variety of applications. This is due to their often compact, graph-based representations of functions over boolean variables, along with operations to manipulate those boolean functions based on the sizes of the graph representations, rather than the size of the domain of the function. Most DD types are canonical for boolean functions: for a fixed ordering of the function variables, each function has a unique (modulo graph isomorphism) DD representation, or encoding.

Compactness, and canonicity, is achieved through careful rules for eliminating nodes. All canonical DDs eliminate nodes that duplicate information: if nodes p and q encode the same function, one of them is discarded. Additional compactness comes from a reduction rule (or rules) that specifies both how to interpret "long" edges that skip over function variables, and how to eliminate nodes and replace them with long edges. Two popular forms of decision diagrams, Binary Decision Diagrams (BDDs) (Bryant, 1986) and Zero-suppressed binary Decision Diagrams (ZDDs) (Minato, 2001), use different reduction rules. Some applications are more suitable for BDDs while others are more suitable for ZDDs, depending on which of the two reductions can be applied to a greater number of nodes. Unfortunately, it is not always easy to know, *a priori*, which reduction rule is best for a particular application. Worse, there are applications where *both* rules are useful.

Tagged BDDs (TBDDs) (van Dijk et al., 2017) and Chain-reduced BDDs (CBDDs) or ZDDs (CZDDs) (Bryant, 2018) have been introduced to combine the reduction rules of BDDs and ZDDs. We introduce a new type of BDD, called *Edge Specified Reduction* BDDs (ESRBDDs), that we believe is conceptually simpler and has smaller node storage requirements than TBDDs, CBDDs, and CZDDs, while still exploiting the BDD and ZDD reduction rules. Additionally, ESRBDDs are flexible in that additional reduction rules may be added with low cost. Finally, unlike TBDDs, CBDDs, and CZDDs, ESRBDDs treat the BDD and ZDD reduction rules equally: there is no need to prioritize one rule over another.

The chapter is organized as follows. Section 4.2 recalls definitions for BDDs and ZDDs and describes related work. Section 4.3 formally defines ESRBDDs, gives their reduction algorithm, proves that they are a canonical form, and compares them to related DDs. Section 4.4 provides experimental results to show how the various DDs compare in practice. Section 4.5 provides conclusions.

4.2 Related Decision Diagrams

We focus on various types of DDs that have been proposed to efficiently encode boolean functions of boolean variables, and briefly recall DDs relevant to our work. For consistency in notation, all DD types we present encode functions of the form $f : \mathbb{B}^L \to \mathbb{B}$ and have L levels, with level L at the top.

The first and most widely-known type is the *reduced-ordered binary decision diagrams* (BDDs) (Bryant, 1986). A BDD is a directed acyclic graph where the two terminal nodes $\mathbf{0}$ and $\mathbf{1}$ are at level 0, we write $l(\mathbf{0}) = l(\mathbf{1}) = 0$, while each nonterminal node p belongs to a level $l(p) \in \{1, ..., L\}$ and has two outgoing edges, $p[0]$ and $p[1]$, pointing to nodes at lower levels (this is the "ordered" property). The "reduced" property instead forbids both *duplicate* nodes (p and q are duplicates if $l(p) = l(q)$, $p[0] = q[0]$, and $p[1] = q[1]$), and *redundant* nodes (p is redundant if $p[0] = p[1]$). The function F_p encoded by BDD node p is defined as

$$F_p(x_{1:L}) \quad = \quad \begin{cases} F_{p[x_{l(p)}]}(x_{1:L}) & l(p) > 0 \\ p & l(p) = 0, \end{cases}$$

where $(x_{1:L})$ is a shorthand for the boolean tuple $(x_1, ..., x_L)$.

Another widely-used type is the *zero-suppressed binary decision diagrams* (ZDDs) (Minato, 2001), which differ from BDDs only in that they forbid *high-zero* nodes (node p is high-zero if $p[1] = \mathbf{0}$) instead of redundant nodes. The function encoded by ZDD node p is defined with respect to a level $n \geq m = l(p)$, as

$$F_p^n(x_{1:n}) \quad = \quad \begin{cases} 0 & n > m \wedge \exists i, m < i \leq n, x_i = 1 \\ F_p^m(x_{1:m}) & n > m \wedge \forall i, m < i \leq n, x_i = 0 \\ F_{p[x_m]}^{m-1}(x_{1:m-1}) & n = m > 0 \\ p & n = m = 0. \end{cases}$$

Both BDDs and ZDDs are *canonical*: any function $f : \mathbb{B}^L \to \mathbb{B}$ has a unique node p encoding it, an essential property guaranteeing *time* efficiency. Just as important is their *memory* efficiency, i.e., the number of nodes required to encode a given function. In this respect, BDDs and ZDDs are particularly suited to different situations. BDDs require fewer nodes if there are many "don't cares", i.e., it often happens that $F_p(x_{1:L}) = F_p(y_{1:L})$ when $x_{1:L}$ and $y_{1:L}$ differ in one position, as this corresponds to redundant nodes, not stored in BDDs. ZDDs require fewer nodes if the function tends to have value 0 when many arguments have value 1 as this corresponds to high-zero nodes, not stored in ZDDs.

Quasi-reduced BDDs (QBDDs) (Kimura and Clarke, 1990) are also canonical: they are just like BDDs (or ZDDs) except they only forbid duplicate nodes. QBDD edges connect nodes on adjacent levels, therefore nodes do not need to store level information. This also means that redundant and high-zero nodes are not eliminated. A useful variation is to eliminate only redundant (or high-zero) nodes whose children are **0**, and thus allow long edges directly to **0**. In either case, QBDDs require at least as many nodes as BDDs and ZDDs to encode a given function, so they provide an upper bound on both the BDD and the ZDD size.

Various decision diagrams have been proposed to combine the characteristics of BDDs and ZDDs and exploit their reduction potential. *Tagged binary decision diagrams* (TBDDs) (van Dijk et al., 2017) associate a level tag to each edge. BDD reductions are implied along the edge from the level of the node until the level of the tag, and ZDD reductions are implied from the level of the tag to the level of the node pointed to by the edge. Alternatively, TBDDs can apply reductions in the reverse order along an edge: ZDD reductions first and BDD reductions second. Either reduction order can be used in TBDDs, but a TBDD can only use one of them, i.e., they cannot both be used in the same TBDD.

Chain-reduced BDDs (CBDDs) and *chain-reduced ZDDs* (CZDDs) (Bryant, 2018) augment BDDs and ZDDs by using nodes to encode chains of high-zero nodes and redundant nodes, respectively. Each node specifies two levels, the first level indicating where the chain starts (similar to the level of an ordinary BDD or ZDD node), and the second, additional, level indicating where the chain ends.

Finally, *ordered Kronecker functional decision diagrams* (Drechsler and Becker, 2006) allow multiple decomposition types (Shannon, positive Davio, and negative Davio), enabling both BDD and ZDD reductions. However, each level has a fixed decomposition type, thus this approach is less flexible, potentially less efficient, and hindered by the need to know *a priori* which decomposition will perform best for each level.

4.3 ESRBDDs

Definition 4.3.1

An L-level *(ordered) edge-specified reduction* binary decision diagram (ESRBDD) is a directed acyclic graph where the two *terminal* nodes $\mathbf{0}$ and $\mathbf{1}$ are at level 0, $l(\mathbf{0}) = l(\mathbf{1}) = 0$, while each *nonterminal* node p belongs to a level $l(p) \in \{1, ..., L\}$ and has two outgoing edges, $p[0]$ and $p[1]$, pointing to nodes at lower levels. An edge is a pair $e = \langle e.r, e.p\rangle$, where $e.r$ is a *reduction rule* in $\{\mathtt{S}, \mathtt{L_0}, \mathtt{H_0}, \mathtt{X}\}$ and $e.p$ is the node to which edge e points. For $i \in \{0, 1\}$, if $l(p[i].p) = l(p) - 1$, we say that $p[i]$ is a *short* edge and require that $p[i].r = \mathtt{S}$. If instead $l(p[i].p) < l(p) - 1$, the only other possibility, we say that $p[i]$ is a *long* edge, since it "skips over" one or more levels, and require that $p[i].r \in \{\mathtt{H_0}, \mathtt{L_0}, \mathtt{X}\}$. □

The reduction rule on an edge specifies its meaning when skipping levels, thus it is just $\mathtt{S}$ for short edges while, for long edges, the rules $\mathtt{H_0}$, $\mathtt{L_0}$, and $\mathtt{X}$ correspond to the "zero-suppressed" rule of Minato (2001), the "one-suppressed" rule (a new rule analogous to the zero-suppressed, as we shall see), and the "fully-reduced" rule of Bryant (1986), respectively. To make this more precise, we recursively define the boolean function $F^n_{\langle \kappa, p\rangle} : \mathbb{B}^n \to \mathbb{B}$ encoded by an ESRBDD edge $\langle \kappa, p\rangle$ with respect to a level $n \in \{0, ..., L\}$, subject to $l(p) \le n$, as

$$F^n_{\langle \kappa,p\rangle}(x_{1:n}) = \begin{cases} \text{if } l(p) = n = 0 & p \\ \text{if } l(p) = n > 0 & (x_n) \;?\; F^{n-1}_{p[1]}(x_{1:n-1}) \;:\; F^{n-1}_{p[0]}(x_{1:n-1}) \\ \text{if } l(p) < n, \kappa = \mathtt{X}, & (x_n) \;?\; F^{n-1}_{\langle \kappa,p\rangle}(x_{1:n-1}) \;:\; F^{n-1}_{\langle \kappa,p\rangle}(x_{1:n-1}) \\ \text{if } l(p) < n, \kappa = \mathtt{H_0}, & (x_n) \;?\; \mathbf{0} \;:\; F^{n-1}_{\langle \kappa,p\rangle}(x_{1:n-1}) \\ \text{if } l(p) < n, \kappa = \mathtt{L_0}, & (x_n) \;?\; F^{n-1}_{\langle \kappa,p\rangle}(x_{1:n-1}) \;:\; \mathbf{0}, \end{cases}$$

where the if-then-else operator $(x_n)?f_1{:}f_0$ is a shorthand for $(\neg x_n \wedge f_0) \vee (x_n \wedge f_1)$.

We defined an ESRBDD as a directed acyclic graph, so it can potentially have multiple *roots* (nodes with no incoming edges). However, since our focus is on the size of the DD encoding a given function, we assume from now on that our ESRBDDs have a single root node $p^\star$, pointed to by a *dangling edge* with rule $\kappa^\star$. We denote the set of all nodes reachable from $p^\star$ (and therefore all nodes in the ESRBDD) as $Nodes(p^\star)$. The dangling edge $\langle\kappa^\star,p^\star\rangle$ encodes the function $F^L\langle\kappa^\star,p^\star\rangle$, which is independent of $\kappa^\star$ only if $l(p^\star) = L$, in which case we require $\kappa^\star = \texttt{S}$, while we require $\kappa^\star \in \{\texttt{L}_0, \texttt{H}_0, \texttt{X}\}$ if $l(p^\star) < L$. Finally, we will informally say "ESRBDD $\langle\kappa^\star,p^\star\rangle$" to refer to the entire graph below (and including) dangling edge $\langle\kappa^\star,p^\star\rangle$.

Before introducing reduced ESRBDDs and showing they are canonical, we need some terminology. We say that an ESRBDD nonterminal node q:

- *duplicates* node p if $l(p) = l(q)$, $p[0] = q[0]$, and $p[1] = q[1]$ (Figure 4.1),

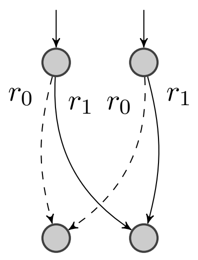


Figure 4.1 Duplicate nodes.

- is *redundant* if $q[0] = q[1] = \langle\kappa,p\rangle$, with $\kappa \in \{\texttt{S}, \texttt{X}\}$ (Figure 4.2),

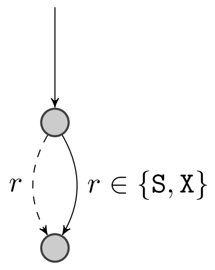


Figure 4.2 A redundant node.

- is *high-zero* if $q[0].r \in \{\texttt{S}, \texttt{H}_0\}$, and $q[1].p = \mathbf{0}$ (Figure 4.3),

- is *low-zero* if $q[0].p = \mathbf{0}$, and $q[1].r \in \{\texttt{S}, \texttt{L}_0\}$ (Figure 4.4).

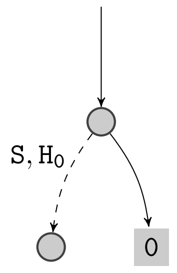


Figure 4.3 A high-zero node.

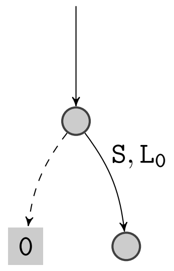


Figure 4.4 A low-zero node.

Note that BDDs (Bryant, 1986) can be viewed as ESRBDDs where the edge labels are restricted to $\{\texttt{S}, \texttt{X}\}$, and a reduced BDD corresponds to an ESRBDD with no duplicate nodes and no redundant nodes. Similarly, ZDDs (Minato, 2001) can be viewed as ESRBDDs where edge labels are restricted to $\{\texttt{S}, \texttt{H}_0\}$, and a reduced ZDD corresponds to an ESRBDD with no duplicate nodes and no high-zero nodes. Also, we note that there is no corresponding definition in the existing literature for the version of ESRBDDs where the edge labels are restricted to $\{\texttt{S}, \texttt{L}_0\}$.

Definition 4.3.2

An ESRBDD is *reduced* if the following restrictions hold:

R1. There are no duplicate nodes.

R2. There are no redundant nodes.

R3. There are no high-zero nodes.

R4. There are no low-zero nodes.

R5. For any edge $e = \langle \kappa, \mathbf{0} \rangle$, $\kappa \in \{\texttt{S}, \texttt{X}\}$. The last restriction disallows edges $\langle \texttt{H}_0, \mathbf{0} \rangle$ and $\langle \texttt{L}_0, \mathbf{0} \rangle$ in the reduced ESRBDD. This is because $F^n_{\langle \texttt{H}_0, \mathbf{0} \rangle} \equiv F^n_{\langle \texttt{L}_0, \mathbf{0} \rangle} \equiv F^n_{\langle \texttt{X}, \mathbf{0} \rangle} \equiv \mathbf{0}$, and since we want to enforce canonicity in the reduced ESRBDD, we have *arbitrarily* chosen $\langle \mathtt{X},\mathbf{0}\rangle$ as the unique representation for such long edges.

4.3.1 Reducing an ESRBDD

Algorithm 4.1 Reduce an ESRBDD.

```
1: procedure Reduce(ESRBDD $\langle \kappa^\star,p^\star\rangle$)
2:     $V \leftarrow Nodes(p^\star)$;
3:     $\forall \kappa \in \{\mathtt{H_0},\mathtt{L_0}\}$, replace all $\langle \kappa,\mathbf{0}\rangle$ edges with $\langle \mathtt{X},\mathbf{0}\rangle$;
4:     while $V$ contains a high-zero, low-zero, redundant, or duplicate node do
5:         Choose a high-zero, low-zero, redundant, or duplicate node $q \in V$;
6:         if $q$ duplicates $p$ then
7:             $\forall \kappa \in \{\mathtt{S},\mathtt{X},\mathtt{H_0},\mathtt{L_0}\}$, replace all $\langle \kappa,q\rangle$ edges with $\langle \kappa,p\rangle$;
8:         else
9:             if $q$ is a redundant node then
10:                $\kappa' \leftarrow \mathtt{X}$;    $d' \leftarrow q[1].p$;
11:            else if $q$ is a high-zero node then
12:                $\kappa' \leftarrow \mathtt{H_0}$;    $d' \leftarrow q[0].p$;
13:            else if $q$ is a low-zero node then
14:                $\kappa' \leftarrow \mathtt{L_0}$;    $d' \leftarrow q[1].p$;
15:            if $d' = \mathbf{0}$ then
16:                $\forall \kappa \in \{\mathtt{S},\mathtt{X},\mathtt{H_0},\mathtt{L_0}\}$, replace all $\langle \kappa,q\rangle$ edges with $\langle \mathtt{X},\mathbf{0}\rangle$;
17:            else
18:                Replace all $\langle \mathtt{S},q\rangle$ edges with $\langle \kappa',d'\rangle$;
19:                Replace all $\langle \kappa',q\rangle$ edges with $\langle \kappa',d'\rangle$;
20:                for all rules $\kappa \in \{\mathtt{X},\mathtt{H_0},\mathtt{L_0}\} \setminus \{\kappa'\}$, such that an edge $\langle \kappa,q\rangle$ exists do
21:                    Create node $q'$ at level $l(q)+1$ and add $q'$ to $V$;
22:                    if $\kappa = \mathtt{X}$ then
23:                        $q'[0] \leftarrow \langle \kappa',d'\rangle$;    $q'[1] \leftarrow \langle \kappa',d'\rangle$;
24:                    else if $\kappa = \mathtt{H_0}$ then
25:                        $q'[0] \leftarrow \langle \kappa',d'\rangle$;    $q'[1] \leftarrow \langle \mathtt{X},\mathbf{0}\rangle$;
26:                    else if $\kappa = \mathtt{L_0}$ then
27:                        $q'[0] \leftarrow \langle \mathtt{X},\mathbf{0}\rangle$;    $q'[1] \leftarrow \langle \kappa',d'\rangle$;
28:                    Replace all $\langle \kappa,q\rangle$ edges with $\langle \kappa,q'\rangle$ or $\langle \mathtt{S},q'\rangle$;
29:        Remove $q$ from $V$;
```

An ESRBDD can be converted into a reduced ESRBDD using Algorithm 4.1. The algorithm first replaces any edges $\langle \mathtt{H_0},\mathbf{0}\rangle$ or $\langle \mathtt{L_0},\mathbf{0}\rangle$ with $\langle \mathtt{X},\mathbf{0}\rangle$, to satisfy restriction R5. Then, it repeatedly chooses a high-zero, low-zero, redundant, or duplicate node q and eliminates it. If node q duplicates

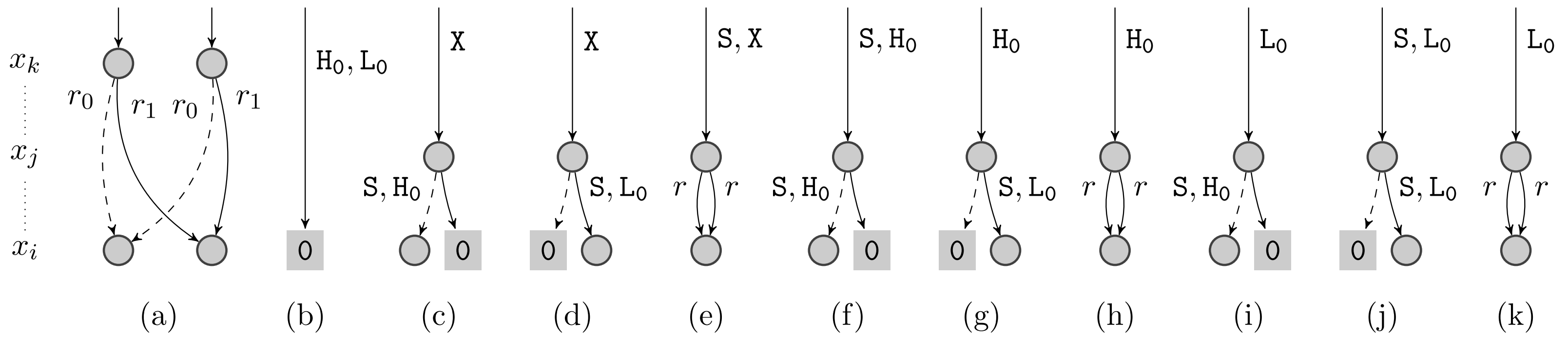


Figure 4.5 Patterns not allowed in RESRBDDs. $r \in \{\texttt{S}, \texttt{X}\}$.

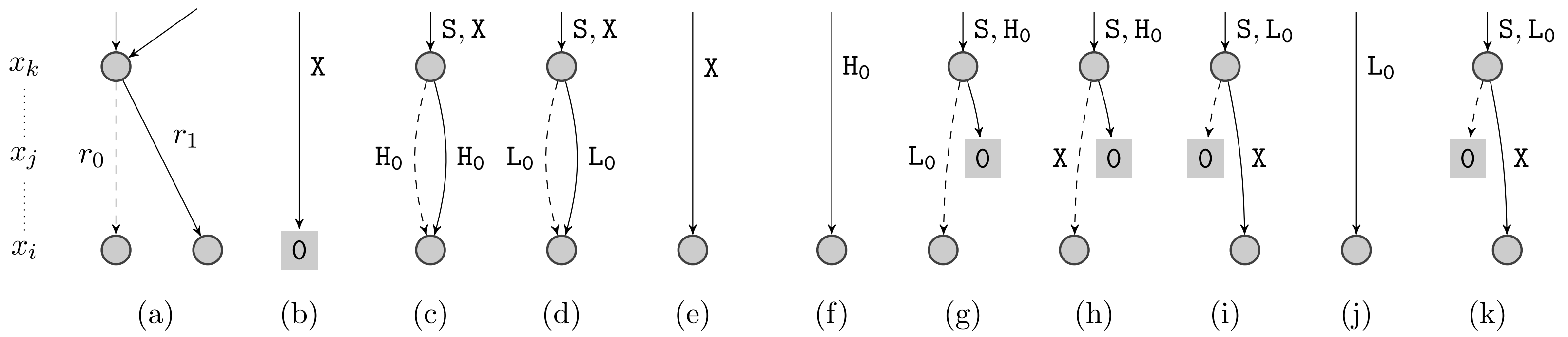


Figure 4.6 Replacement rules for patterns in Figure 4.5.

node p, then it redirects all incoming edges from q to p (line 7). Otherwise, q is a high-zero, low-zero, or redundant node, and lines 9–14 find a node d' with $l(d') < l(q) = n-1$, and a rule $\kappa' \in \{\mathtt{X}, \mathtt{H_0}, \mathtt{L_0}\}$ such that $F^n_{\langle \mathtt{S},q\rangle}(x_{1:n}) = F^n_{\langle \kappa',d'\rangle}(x_{1:n})$. Note that a short edge to node q becomes a long edge to node d' because $l(d') < l(q)$. For the special case of $d' = \mathbf{0}$, *any* edge to q is equivalent to edge $\langle \mathtt{X},\mathbf{0}\rangle$, so the algorithm replaces those edges (line 16).

When $d' \neq \mathbf{0}$, we have $F^n_{\langle \mathtt{S},q\rangle}(x_{1:n}) = F^n_{\langle \kappa',d'\rangle}(x_{1:n})$ for $n = l(q)+1$, and these edges are replaced in line 18. It follows that $F^n_{\langle \kappa',q\rangle}(x_{1:n}) = F^n_{\langle \kappa',d'\rangle}(x_{1:n})$ for $n > l(q)+1$; these replacements are made in line 19. For rules $\kappa \in \{\mathtt{X}, \mathtt{H_0}, \mathtt{L_0}\}$ with $\kappa \neq \kappa'$, we cannot replace $\langle \kappa,q\rangle$ with a single long edge to node d', because the edge needs different reduction rules: the κ rule is needed above level $l(q)$, and the κ' rule is needed from level $l(q)$ down. So lines 21–27 of the algorithm create a new node q' at level $l(q)+1$, of the appropriate shape such that $F^n_{\langle \kappa,q\rangle}(x_{1:n}) = F^n_{\langle \mathtt{S},q'\rangle}(x_{1:n})$ for $n = l(q')+1$. It then follows that $F^n_{\langle \kappa,q\rangle}(x_{1:n}) = F^n_{\langle \kappa,q'\rangle}(x_{1:n})$ for $n > l(q')+1$. These replacements are made in line 28, where the replacement $\langle \kappa,q'\rangle$ is used for long edges, and $\langle \mathtt{S},q'\rangle$ is used for short edges.

In the above discussion, any edge that is replaced by the algorithm encodes the same function as its replacement, giving us the following lemma.

Lemma 4.3.1

In Algorithm 4.1, each edge replacement preserves the function encoded by the ESRBDD $\langle \kappa^\star,p^\star\rangle$.

It remains to show that the algorithm always terminates.

Lemma 4.3.2

Algorithm 4.1 terminates in $\mathcal{O}(|\mathit{Nodes}(p^\star)|)$ steps.

Proof: The proof is based on the observation that, at every iteration of the algorithm, a node q is chosen to be processed (line 5), at most two nodes are created at level $l(q)+1$ (line 21), and node q is removed (line 29). These new nodes (q' on line 21), by construction, satisfy one of the following patterns:

- $q'[0] = q'[1] = \langle \kappa',d'\rangle$, where $d' \neq \mathbf{0}$, and $\kappa' \in \{\mathtt{H_0}, \mathtt{L_0}\}$,

- $q'[0] = \langle \mathtt{X},\mathbf{0}\rangle$, and $q'[1] = \langle \kappa',d'\rangle$, where $d' \neq \mathbf{0}$, and $\kappa' \in \{\mathtt{X}, \mathtt{H_0}\}$,

- $q'[0] = \langle \kappa',d'\rangle$, and $q'[1] = \langle \mathtt{X},\mathbf{0}\rangle$, where $d' \neq \mathbf{0}$, and $\kappa' \in \{\mathtt{X}, \mathtt{L_0}\}$.

These nodes are neither redundant, high-zero, nor low-zero, but they could be duplicates. Since the elimination of duplicate nodes (line 7) does not create new nodes, the two nodes created at level $l(q)+1$ generate at most two additional iterations of the algorithm. Therefore, for every node in the original ESRBDD, the algorithm iterates at most three times. $\square$

Theorem 4.3.1

Algorithm 4.1 converts ESRBDD $\langle \kappa^\star, p^\star \rangle$ to an equivalent reduced ESRBDD in $\mathcal{O}(|Nodes(p^\star)|)$ steps.

Proof: Lemma 4.3.1 establishes that Algorithm 4.1 terminates in $\mathcal{O}(|Nodes(p^\star)|)$ steps. Based on the condition of the while loop, when the loop terminates, we know that the ESRBDD contains no high-zero, low-zero, redundant, or duplicate nodes. From line 3 and the fact that the algorithm never adds an edge of the form $\langle \mathtt{H}_0, \mathbf{0} \rangle$ or $\langle \mathtt{L}_0, \mathbf{0} \rangle$, we conclude that when Algorithm 4.1 terminates, any edge to terminal node $\mathbf{0}$ must have edge rule $\mathtt{S}$ or $\mathtt{X}$. Therefore, when the Algorithm terminates, the ESRBDD is reduced. Lemma 4.3.1 establishes that Algorithm 4.1 produces an equivalent (in terms of encoded function) ESRBDD. □

While we have established that Algorithm 4.1 always terminates and produces a reduced ESRBDD, we have not yet established that the Algorithm produces the *same* reduced ESRBDD, regardless of the order in which nodes are chosen in line 5. This is guaranteed by the canonicity property, discussed next. Additionally, we note here that, unlike most other decision diagrams (including BDDs, ZDDs, CBDDs, CZDDs, and TDDs), a reduced ESRBDD is not necessarily a minimum size ESRBDD encoding of a function, even for a fixed variable order, as elimination of some node q during the reduction could trigger the creation of two new nodes. An example of this is shown in Figure 4.7, where redundant node q is eliminated. Edges $\langle \mathtt{S}, q \rangle$ and $\langle \mathtt{X}, q \rangle$ can be simply redirected as $\langle \mathtt{X}, p \rangle$, but the $\langle \mathtt{H}_0, q \rangle$ and $\langle \mathtt{L}_0, q \rangle$ edges require the creation of two new nodes $q_{\mathtt{H}_0}$ and $q_{\mathtt{L}_0}$.

While the "chaotic" non-deterministic reduction procedure in Algorithm 4.1 is handy in proving termination under the most general conditions, in practice we utilize a deterministic depth-first version of this algorithm that reduces a node only after having reduced its children.

4.3.2 Canonicity of reduced ESRBDDs

We are now ready to discuss the *canonicity* of reduced ESRBDDs, i.e., to show that a function has a unique encoding as a reduced ESRBDD. In the following, we say that functions $F^n_{\langle \kappa, p \rangle}$ and

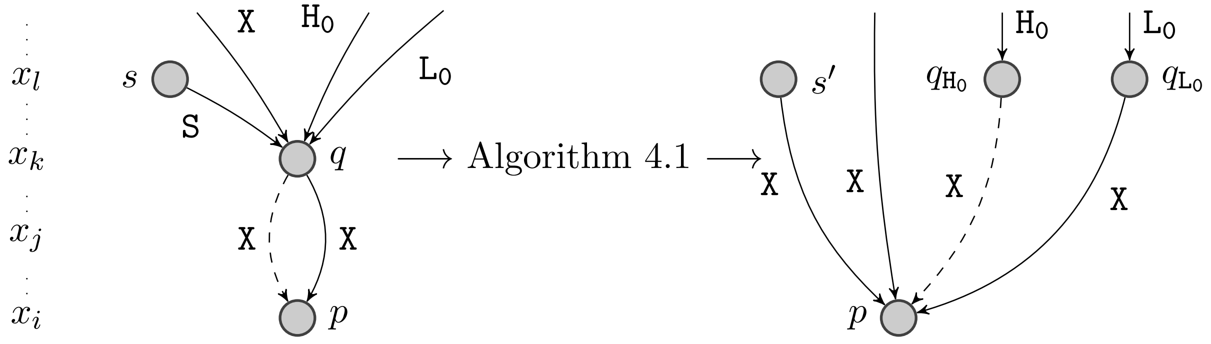


Figure 4.7 A worst-case example where elimination of node q creates two nodes.

$F^n_{\langle\kappa',p'\rangle}$ are *equivalent*, written $F^n_{\langle\kappa,p\rangle} \equiv F^n_{\langle\kappa',p'\rangle}$, if $F^n_{\langle\kappa,p\rangle}(x_{1:n}) = F^n_{\langle\kappa',p'\rangle}(x_{1:n})$ for all possible inputs $(x_{1:n}) \in \mathbb{B}^n$.

Theorem 4.3.2

In a reduced ESRBDD, for any $n \in \mathbb{N}$, for any two edges $e = \langle\kappa,p\rangle$, $e' = \langle\kappa',p'\rangle$ with $l(p) \leq n$, $l(p') \leq n$, if $F^n_e \equiv F^n_{e'}$ then (1) $p = p'$, and (2) if $l(p) < n$ then $\kappa = \kappa'$.

Proof: The proof is by induction on n. For the base case, we use $n = 0$ and from the definition of F we have $F^0_e \equiv F^0_{e'} \ \rightarrow \ p = p'$.

Now, suppose the theorem holds for $n = m$, where $m \geq 0$, we will prove it holds for $n = m + 1$. Regardless of $\langle\kappa,p\rangle$, we have

$$F^n_{\langle\kappa,p\rangle}(x_{1:n}) = (x_n)?f_1(x_{1:n-1}){:}f_0(x_{1:n-1})$$

for some functions f_0 and f_1. Similarly, we have

$$F^n_{\langle\kappa',p'\rangle}(x_{1:n}) = (x_n)?f'_1(x_{1:n-1}){:}f'_0(x_{1:n-1}).$$

It follows that $F^n_{\langle\kappa,p\rangle} \equiv F^n_{\langle\kappa',p'\rangle}$ if and only if $f_0 \equiv f'_0$ and $f_1 \equiv f'_1$.

First, suppose $l(p) = n$ and $l(p') = n$. From the definition of F, it follows that $F^{n-1}_{p[0]} \equiv F^{n-1}_{p'[0]}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{p'[1]}$. By inductive hypothesis, $p[0].p = p'[0].p$ and $p[1].p = p'[1].p$. If $l(p[0].p) < n - 1$, then by inductive hypothesis, $p[0] = p'[0]$; otherwise, $l(p[0].p) = n - 1$ and we must have $p[0].r = \mathtt{S}$

and $p'[0].r = \mathtt{S}$, thus $p[0] = p'[0]$. By a similar argument, it follows that $p[1] = p'[1]$. We therefore have either that $p = p'$ and the theorem holds, or that p duplicates p', which is impossible because of restriction R1.

Next, suppose $l(p) < n$ and $l(p') < n$. If $\kappa = \kappa'$, then in all cases for F we conclude that $F^{n-1}_{\langle\kappa,p\rangle} \equiv F^{n-1}_{\langle\kappa',p'\rangle}$ and by inductive hypothesis we have that $p = p'$, so the theorem holds. We now show that $\kappa \neq \kappa'$ is impossible, by contradiction. Consider the possible cases for $\kappa \neq \kappa'$:

1. $\kappa = \mathtt{X}$: If $\kappa' = \mathtt{L_0}$ or $\kappa' = \mathtt{H_0}$, from the definition of F we conclude that $F^{n-1}_{\langle\kappa,p\rangle} \equiv F^{n-1}_{\langle\kappa',p'\rangle}$ and that $F^{n-1}_{\langle\kappa,p\rangle} \equiv \mathbf{0}$.

2. $\kappa = \mathtt{L_0}$: If $\kappa' = \mathtt{H_0}$, from the definition of F we conclude that $F^{n-1}_{\langle\kappa,p\rangle} \equiv \mathbf{0}$ and $F^{n-1}_{\langle\kappa',p'\rangle} \equiv \mathbf{0}$.

3. The remaining cases are symmetric.

In all cases, we conclude that $F^{n-1}_{\langle\kappa,p\rangle} \equiv \mathbf{0}$ and $F^{n-1}_{\langle\kappa',p'\rangle} \equiv \mathbf{0}$. By the inductive hypothesis, we have that $p = \mathbf{0}$ and $p' = \mathbf{0}$. According to R5, if $p = \mathbf{0}$ then κ cannot be $\mathtt{L_0}$ or $\mathtt{H_0}$. But this implies $\kappa = \mathtt{X}$ and $\kappa' = \mathtt{X}$, contradicting our assumption that $\kappa \neq \kappa'$.

Finally, suppose $l(p) = n$ and $l(p') < n$ (the case $l(p) < n$ and $l(p') = n$ is symmetric). We show that this is impossible, by contradiction. Consider the possible cases for κ':

1. $\kappa' = \mathtt{X}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv F^{n-1}_{\langle\kappa',p'\rangle}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{\langle\kappa',p'\rangle}$. By the inductive hypothesis, we conclude that $p[0].p = p'$ and $p[1].p = p'$. If $l(p') = n-1$, then we have $p[0] = p[1] = \langle\mathtt{S},p'\rangle$; otherwise, we have $l(p') < n-1$ and by inductive hypothesis, $p[0] = p[1] = \langle\kappa',p'\rangle = \langle\mathtt{X},p'\rangle$. Either way, node p is redundant, and from R2 we have a contradiction.

2. $\kappa' = \mathtt{H_0}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv F^{n-1}_{\langle\kappa',p'\rangle}$ and $F^{n-1}_{p[1]} \equiv \mathbf{0}$. By the inductive hypothesis, we conclude that $p[0].p = p'$ and $p[1].p = \mathbf{0}$. If $l(p') = n-1$, then we have $p[0] = \langle\mathtt{S},p'\rangle$; otherwise, we have $l(p') < n-1$ and by inductive hypothesis, $p[0] = \langle\kappa',p'\rangle = \langle\mathtt{H_0},p'\rangle$. Either way, node p is high-zero, and from R3 we have a contradiction.

3. $\kappa' = \mathtt{L_0}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv \mathbf{0}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{\langle\kappa',p'\rangle}$. By the inductive hypothesis, we conclude that $p[0].p = \mathbf{0}$ and $p[1].p = p'$. If $l(p') = n-1$, then we have $p[1] = \langle \mathtt{S},p'\rangle$; otherwise, we have $l(p') < n-1$ and by inductive hypothesis, $p[1] = \langle\kappa',p'\rangle = \langle \mathtt{L_0},p'\rangle$. Either way, node p is low-zero, and from R4 we have a contradiction. $\square$

The canonicity result establishes that, regardless of how a ESRBDD is constructed for a given function, the resulting reduced ESRBDD is guaranteed to be unique (assuming a given variable order). Thus, we can determine in constant time whether two functions encoded as reduced ESRBDDs are equivalent (as is already the case for reduced ordered BDDs and ZDDs). From now on, unless otherwise specified, we assume that all ESRBDDs are reduced.

4.3.3 Comparing ESRBDDs to other types of decision diagrams

For the remainder of this section, we consider the relative size of the different types of DD based on the interpretation of long edges, namely, BDDs, ZDDs, CBDDs, CZDDs, TBDDs, and ESRBDDs. We also consider ESRBDDs without the $\mathtt{L_0}$ edge label, denoted ESRBDD$-\mathtt{L_0}$. These are summarized in Table 4.1, some entries (comparisons between BDDs, ZDDs, CBDDs, and CZDDs) are known from prior work (Bryant, 2018; Knuth, 2011), some entries are discussed below, and some entries are unknown. Entry $[T_1, T_2]$ describes the worst-case increase in the number of nodes, as a multiplicative factor, More formally, it is the bound for "number of nodes required to encode f using T_2" divided by "number of nodes required to encode f using T_1" for all functions f over L boolean variables. Note that the node counts always include both terminal nodes. A factor of 1 indicates that type T_1 cannot require fewer nodes than type T_2.

First, we discuss how an arbitrary BDD can be converted into a TBDD or ESRBDD, and fill in the BDD row in Table 4.1. To build a TBDD from a BDD, every edge to a non-terminal node p in the BDD is annotated with the level tag $l(p)$. By definition, any such annotated edge in a TBDD implies BDD reductions for the skipped levels. A TBDD thus constructed is no larger than the BDD, and may be further reduced (since it could contain high-zero nodes) by applying the TBDD

Table 4.1 Worst-case relative increase when converting one DD type into another.

†: Knuth (2011); ‡: Bryant (2018)

	BDD	**ZDD**	**CBDD**	**CZDD**	**TBDD**	**ESR**$-\mathtt{L_0}$	**ESR**
BDD $\rightarrow$	—	$L/2$ †	1 ‡	2 ‡	1	1	1
ZDD $\rightarrow$	$L/2$ †	—	3 ‡	1 ‡	1	1	1
CBDD $\rightarrow$	?	?	—	2 ‡	?	2	2
CZDD $\rightarrow$	?	?	3 ‡	—	?	2	2
TBDD $\rightarrow$	?	?	?	?	—	3	3
ESRBDD$-\mathtt{L_0}$ $\rightarrow$	$L/2$	$L/2$	3	2	1	—	3/2
ESRBDD $\rightarrow$	$2L/3$	$2L/3$	$L/2$	$L/2$	$L/2$	$L/2$	—

reduction described in van Dijk et al. (2017). Similarly, we can annotate long edges in the BDD with $\mathtt{X}$ (Figure 4.10(a)), and short edges with $\mathtt{S}$, to obtain an unreduced ESRBDD. We then apply Algorithm 4.1. We now show that this will not increase the ESRBDD size, and thus the resulting ESRBDD cannot be larger than the original BDD.

Lemma 4.3.3

Suppose we have an unreduced ESRBDD where, for every node q, there exists a rule $\kappa \in \{\mathtt{X}, \mathtt{H_0}, \mathtt{L_0}\}$ such that every edge to q is either $\langle \mathtt{S}, q\rangle$ or $\langle \kappa, q\rangle$. Then reducing the ESRBDD will not increase the number of nodes.

Proof: Apply Algorithm 4.1 and in line 5, always choose a node at the lowest level. Then, when a node q is chosen, all incoming edges to q will be labeled either with $\mathtt{S}$ or with κ. The $\langle \mathtt{S}, q\rangle$ edges will not cause any node to be created. The $\langle \kappa, q\rangle$ edges will cause at most one node to be created. But then node q is removed. Thus, the overall number of nodes cannot increase. □

It is also easy to convert a ZDD into a TBDD or ESRBDD. To obtain a TBDD, annotate every edge from non-terminal node p with the level tag $l(p)$, so that ZDD reductions are used for all the edges; then reduce the TBDD. To obtain an ESRBDD, annotate long edges in the ZDD with $\mathtt{H_0}$, see Figure 4.10(b), and short edges with $\mathtt{S}$, and apply Algorithm 4.1.

$$\begin{tikzcd}
 & & \text{TBDD} \arrow[dd, bend right, "3"'] & & \\
\text{BDD} \arrow[drr, bend right, "1"'] & & & & \text{ZDD} \arrow[dll, bend right, "1"'] \\
 & & \text{ESRBDD}{-}\mathtt{L_0} \arrow[uu, bend right, "1"'] \arrow[ull, bend right, "L/2"'] \arrow[urr, bend right, "L/2"'] \arrow[dl, bend right, "3"'] \arrow[dr, bend right, "2"'] & & \\
 & \text{CBDD} \arrow[ur, bend right, "2"'] & & \text{CZDD} \arrow[ul, bend right, "2"'] &
\end{tikzcd}$$

An edge from A to B with weight n says, "Converting from A to B could increase the size of the representation by a factor of n".

Figure 4.8 ESRBDD$-\mathtt{L_0}$ size comparison.

$$\begin{tikzcd}
 & & \text{TBDD} \arrow[dd, bend right, "3"'] & & \\
\text{BDD} \arrow[drr, bend right, "1"'] & & & & \text{ZDD} \arrow[dll, bend right, "1"'] \\
 & & \text{ESRBDD} \arrow[uu, bend right, "L/2"'] \arrow[ull, bend right, "2L/3"'] \arrow[urr, bend right, "2L/3"'] \arrow[dl, bend right, "L/2"'] \arrow[dr, bend right, "L/2"'] & & \\
 & \text{CBDD} \arrow[ur, bend right, "2"'] & & \text{CZDD} \arrow[ul, bend right, "2"'] &
\end{tikzcd}$$

An edge from A to B with weight n says, "Converting from A to B could increase the size of the representation by a factor of n".

Figure 4.9 ESRBDD size comparison.

The conversion from a chained DD to an unreduced ESRBDD is illustrated in Figure 4.10(c) and (d). For each chain node $x_k{:}x_i$ with $x_k > x_i$, create a "top node" with variable x_k, and a "bottom node" with variable x_i, that is only pointed to by its corresponding top node. In a CBDD, the top node will be a high-zero node, and all top nodes and non-chained nodes will have incoming edges labeled with X or S. In a CZDD, the top node will be a redundant node, and all top nodes and non-chained nodes will have incoming edges labeled with $\mathtt{H_0}$ or S. At worst, the unreduced ESRBDD has twice the nodes of the original CBDD or CZDD and, from Lemma 4.3.3, reducing this ESRBDD does not increase its size.

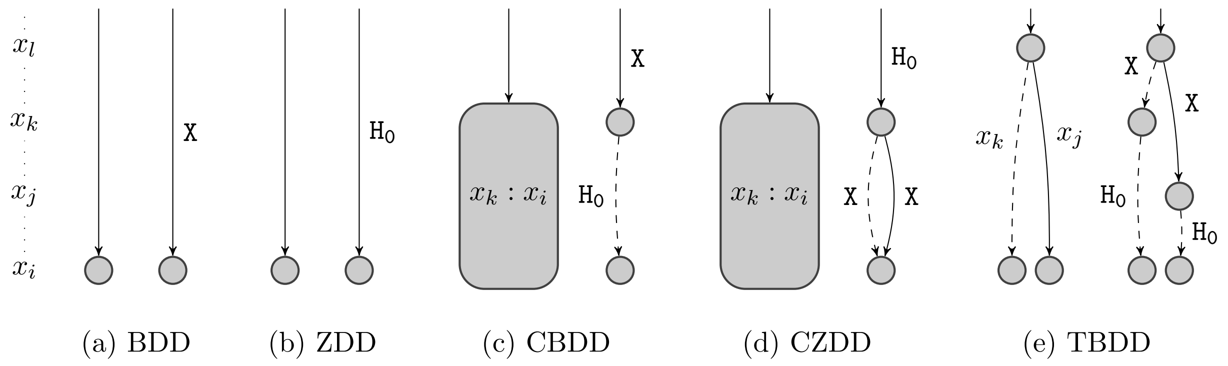


Figure 4.10 Converting to ESRBDDs.

In a TBDD, each edge can be characterized as short, purely X, purely $\mathtt{H_0}$, or partly X and partly $\mathtt{H_0}$. To convert into an ESRBDD, the short edges are labeled with S, the purely X edges are labeled with X, the purely $\mathtt{H_0}$ edges are labeled with $\mathtt{H_0}$. Edges that are partly X and partly $\mathtt{H_0}$ require the addition of a node at the level where the reduction rule changes, as shown in Figure 4.10(e). The worst case occurs when *every* edge requires such a node. Then, since every TBDD node has two outgoing edges, the resulting unreduced ESRBDD will have triple the number of nodes. Since all of the introduced nodes have incoming X edges, and all other nodes have incoming S or $\mathtt{H_0}$ edges, from Lemma 4.3.3 this ESRBDD will not increase in size when it is reduced. We note here that, if there are some purely X edges in the TBDD, then Lemma 4.3.3 no longer applies; however, the number of nodes that would be added during reduction is no more than the number of nodes saved by not having to introduce a node on the purely X edges.

We now consider converting from ESRBDDs into the other DD types. In the case where $\mathtt{L_0}$ edges are not allowed (row ESRBDD$-\mathtt{L_0}$ in Table 4.1), the worst case BDD is from ESRBDD $\langle \mathtt{H_0},\mathbf{1}\rangle$ and the worst case ZDD is from ESRBDD $\langle \mathtt{X},\mathbf{1}\rangle$. In both cases, the ESRBDD has 2 nodes, while the resulting BDD/ZDD has $L + 2$ nodes, giving ratios of $L/2 + o(L)$, similar to the discussion in (Knuth, 2011, p. 250). The example ZDD in Bryant (2018), which produces a CBDD with three times as many nodes, can be converted into an ESRBDD of the same size. Similarly, the example BDD in Bryant (2018), which produces a CZDD with twice as many nodes, can be converted into an ESRBDD of the same size. Any ESRBDD without $\mathtt{L_0}$ edges can be converted into a TBDD by labeling $\mathtt{X}$ edges with a level tag such that the $\mathtt{X}$ rule is always applied, and labelling $\mathtt{H_0}$ edges with a level tag such that the $\mathtt{H_0}$ rule is always applied. Therefore, the TBDD cannot be larger than the ESRBDD. An ESRBDD$-\mathtt{L_0}$ can be converted into an ESRBDD by running Algorithm 4.1 to eliminate any low-zero nodes. For each low-zero node that is eliminated, we could have an incoming $\mathtt{X}$ and $\mathtt{H_0}$ edge, causing the creation of two nodes. Suppose we eliminate n low-zero nodes that cause creation of two nodes. Then, because each low-zero node must have 2 incoming edges, we must have $2n$ incoming edges to these nodes. Above, we must have at least $2n - 1$ nodes to produce these edges. We could then "stack" such a pattern m times. This gives an ESRBDD with $m(n + 2n - 1) + 2 = m(3n - 1) + 2$ nodes, and a reduced ESRBDD with $m(2n + 2n - 1) + 2 = m(4n - 1) + 2$ nodes. The upper bound of this ratio is $3/2$, which occurs when $n = 1$ and m goes to infinity.

For the case of ESRBDDs with all types of edges (row ESRBDD in Table 4.1), the $\mathtt{L_0}$ edge allows us to build different worst cases. Consider an ESRBDD $\langle \mathtt{S},p\rangle$ where $l(p) = L$, $p[0] = \langle \mathtt{H_0},\mathbf{1}\rangle$, and $p[1] = \langle \mathtt{L_0},\mathbf{1}\rangle$. This ESRBDD has 3 nodes. Because BDDs cannot exploit $\mathtt{H_0}$ or $\mathtt{L_0}$ edges, this will produce a BDD with $2(L - 1) + 3 = 2L + 1$ nodes, giving a worst-case ratio of $2L/3$. The ZDD worst-case is similar, using instead $p[0] = \langle \mathtt{X},\mathbf{1}\rangle$. Finally, for DD types that can exploit both $\mathtt{X}$ and $\mathtt{H_0}$ edges, the ESRBDD $\langle \mathtt{L_0},\mathbf{1}\rangle$ corresponds to the worst case: the CBDD, CZDD, TBDD, and ESRBDD$-\mathtt{L_0}$ will all require $L + 2$ nodes.

4.4 Experimental results

We compare the performance of QBDDs (with long edges to **0**), BDDs, ZDDs, CBDDs, CZDDs, TBDDs, and ESRBDDs on three sets of benchmarks. The first two benchmarks are similar to those used in Bryant (2018), and representative of general textual information and digital logic functions, respectively. The third benchmark is typical in state space analysis of concurrent systems.

4.4.1 Dictionaries

A dictionary can be encoded as an indicator function over the set of strings of a given length from either the `compact` alphabet consisting of the distinct symbols found in the dictionary plus `NULL`, or the `full` alphabet of all 128 ASCII characters (to ensure that all encoded strings have the same length, shorter ones are padded with the ASCII symbol `NULL`). We use the encoding schemes described in Bryant (2018): *one-hot* and *binary*. Therefore, each dictionary generates four benchmarks, one for each choice of encoding and alphabet.

We compare the different DD types on two dictionaries. The first one is the English words in file `/usr/share/dict/words` under MacOS, containing 235,886 words with lengths ranging from 1 to 24. Its compact alphabet contains lower and upper case letters plus hyphen and `NULL` (54 in total). The second one is a set of passwords from SecLists (Miessler and Haddix) (non-ASCII characters are replaced with `NULL`), containing 999,999 passwords with lengths ranging from 1 to 39. Its compact alphabet consists of 91 symbols including `NULL`.

Table 4.2 reports the number of nodes required to store each dictionary, according to different encodings and alphabets (the best result on each row is in boldface). Except for QBDDs and BDDs, the one-hot encoding results in fewer nodes, demonstrating the effectiveness of the zero-suppressed idea when encoding large, sparse data. Among the DD types we consider, ESRBDDs have the fewest nodes, regardless of encoding and alphabet. For binary encodings, ESRBDDs use 19% $\sim$ 39% fewer nodes than TBDDs, the second best choice. With one-hot encodings, ZDDs, CZDDs, TBDDs, and ESRBDDs tie for best because (a) there are no redundant nodes and (b) any low-zero nodes that are eliminated do not cause an overall decrease in the number nodes in the ESRBDDs. Indeed, redundant nodes are rare even with binary encodings, as they arise when two words w_1 and w_2 not only have bit patterns that differ in a position, but they also share all their possible continuations, i.e., w_1w' is a word if and only if w_2w' is also a word, for all w'. In the English word list, "Hlidhskjalf" and its alternate spelling "Hlithskjalf" is one such rare instance (note that no w' can continue either of them to form an additional word).

4.4.2 Combinational Circuits

BDDs are commonly used to synthesize and verify digital circuits. We select a set of combinational circuits from the LGSynth'91 benchmarks (Yang, 1991) and, for each circuit, we build a DD encoding all its output logic functions. For each circuit, the variable order is determined using Sifting (Rudell, 1993) while building the BDD.

Table 4.2 reports the number of nodes needed to encode all outputs of each circuit. In contrast to the dictionaries, these benchmarks place importance on the ability to eliminate redundant nodes. Thus, QBDDs and ZDDs have the worst performance. TBDDs and ESRBDDs are always the two best representations, and the difference between them is less than 0.7%.

4.4.3 Safe Petri Nets

Decision diagrams are frequently used in symbolic model checking to represent sets of states. We have selected a set of 37 *safe* Petri nets from the 2018 Model Checking Contest `https://mcc.lip6.fr/2018/`. A *safe* Petri net is one where every place can have at most one token—each place can, therefore, be mapped directly to a boolean variable. Most of these models have scaling parameters that affect their size and complexity, yielding $N = 103$ model instances.

Table 4.2 Experimental results for dictionary and combinational circuit benchmarks.

Dictionary benchmarks

Word List		**QBDD**	**BDD**	**CBDD**	**ZDD**	**CZDD**	**TBDD**	**ESR**
Binary	Compact	1,120,437	1,120,250	971,387	657,969	657,969	657,902	**484,765**
	Full	1,285,501	1,285,285	1,153,438	851,555	851,554	851,479	**520,576**
One-hot	Compact	9,739,638	9,739,638	656,649	**311,227**	**311,227**	**311,227**	**311,227**
	Full	22,775,492	22,775,492	656,712	**311,227**	**311,227**	**311,227**	**311,227**

Password List		**QBDD**	**BDD**	**CBDD**	**ZDD**	**CZDD**	**TBDD**	**ESR**
Binary	Compact	5,705,516	5,704,777	4,542,925	2,960,478	2,960,465	2,960,209	**2,399,272**
	Full	5,649,626	5,648,670	4,960,446	3,532,847	3,532,816	3,532,467	**2,410,589**
One-hot	Compact	72,858,088	72,858,088	3,055,784	**1,486,430**	**1,486,430**	**1,486,430**	**1,486,430**
	Full	101,737,047	101,737,047	3,056,067	**1,486,430**	**1,486,430**	**1,486,430**	**1,486,430**

Combinational circuit benchmarks

Circuit	**QBDD**	**BDD**	**CBDD**	**ZDD**	**CZDD**	**TBDD**	**ESR**
C432	2,675	1,506	1,506	2,658	2,189	**1,494**	1,498
C499	29,483	28,769	28,769	29,316	28,749	28,610	**28,428**
C880	15,048	6,496	6,496	15,044	9,640	6,496	**6,491**
C1355	85,694	75,498	75,498	85,439	77,976	75,243	**74,757**
C1908	18,456	16,210	16,174	17,859	16,047	15,687	**15,685**
C2670	74,940	15,662	15,658	74,468	21,012	**15,539**	15,601
C3540	152,523	51,878	51,778	150,539	64,563	**50,871**	51,146
C5315	26,011	3,793	3,784	25,785	4,749	**3,716**	3,742

Table 4.3 Number of nodes for a subset of the safe Petri net benchmarks.

Model	**QBDD**	**BDD**	**CBDD**	**ZDD**	**CZDD**	**TBDD**	**ESR**
AutoFlight-PT-06a	520,755	520,755	356,729	207,409	207,409	207,409	**178,855**
Dekker-PT-015	1,191,942	1,191,942	844,466	504,726	504,726	504,726	**403,801**
DES-PT-01b	709,303	709,303	442,610	246,647	246,647	246,647	**217,325**
DiscoveryGPU-PT-14a	80,865	75,682	75,571	43,016	43,016	**39,689**	40,953
DLCround-PT-04a	19,865	19,865	19,633	15,140	15,140	15,140	**6,885**
FlexibleBarrier-PT-12a	19,614	19,614	19,548	10,081	10,081	10,081	**9,763**
IBM319-PT-none	16,235	16,166	1,913	826	826	826	**816**
IBM5964-PT-none	31,291	31,291	4,624	2,013	2,013	2,013	**2,013**
LamportFastMutEx-PT-4	507,897	507,897	252,361	122,487	122,487	122,487	**119,111**
MAPKbis-PT-5320	79,037	79,037	69,057	43,568	43,568	43,568	**34,813**
NeoElection-PT-3	414,962	414,962	34,860	15,519	15,519	15,519	**15,507**
NQueens-PT-08	3,698,534	3,698,534	2,295,689	1,443,628	1,443,628	1,443,628	**1,069,242**
ParamProductionCell-PT-4	94,752	94,752	50,588	27,158	27,158	27,158	**26,631**
Peterson-PT-2	11,433	11,433	6,660	3,250	3,250	3,250	**3,136**
Philosophers-PT-000010	83,216	83,216	57,349	28,257	28,257	28,257	**26,974**
Raft-PT-03	24,502	24,472	23,040	12,280	12,280	12,260	**11,635**
Railroad-PT-010	2,109,610	2,109,610	1,096,122	554,541	554,541	554,541	**516,121**
Referendum-PT-0020	343,676	343,676	339,552	194,607	194,607	194,607	**184,789**
Allocation-PT-R020C002	5,532,167	5,532,167	4,554,792	2,826,856	2,826,856	2,826,856	**2,167,111**
utex-PT-r0020w0010	553,073	553,073	502,831	358,580	358,580	358,580	**195,377**
SafeBus-PT-03	12,056	12,056	6,967	3,534	3,534	3,534	**3,334**
SharedMemory-PT-000010	40,225	40,225	21,980	12,229	12,229	12,229	**10,032**
SimpleLoadBal-PT-10	503,777	503,777	376,896	191,460	191,460	191,460	**182,403**

Providing detailed results for all the model instances would require excessive space, so to summarize over all model instances, Table 4.4 shows a score for each DD type i. The score is the geometric mean (Fleming and Wallace, 1986):

$$score(i) = \sqrt[N]{\prod_{n=1}^{N} \frac{T_i(n)}{T_{min}(n)}}$$

where N is the total number of model instances, $T_i(n)$ is the number of nodes needed to represent the state space of instance n using DD type i, and $T_{min}(n)$ is the smallest number of nodes needed to represent the state space of instance n by any DD type. ESRBDDs have by far the smallest overall score, barely larger than 1, indicating that they are either the smallest or slightly larger than the smallest for each model instance.

Table 4.3 shows $T_i(n)$ for each DD type i, for the largest instance n of every model, but only if it required more than 250,000 nodes in the QBDD representation. We also include results for *DiscoveryGPU* — the only model where ESRBDDs were not the best (they were a close second).

4.4.4 Memory considerations: the size of nodes

So far, we have compared DD types based on how many nodes they require. However, the actual memory consumption also depends on the size of the respective nodes. All of these DDs store two child pointers. In addition, BDDs and ZDDs store a level, CBDDs and CZDDs store two levels, TBDDs store three levels, while ESRBDDs store a level and two edge rules. Since all short edges must be labeled by `S`, it is only necessary to label the long edges, and this requires $\log_2 n$ bits per edge if there are n non-`S` reduction rules. Without $\mathtt{L_0}$ edges, a single bit distinguishes $\mathtt{H_0}$ from `X`; otherwise, two bits are required for rules $\{\mathtt{H_0}, \mathtt{L_0}, \mathtt{X}\}$. QBDD nodes are therefore the smallest (typically requiring 64 or 128 bits, when 32–bit or 64–bit pointers are used, respectively) and Table 4.5 indicates the *additional* cost required for each node type, when the level integers

Table 4.4 Final scores for the safe Petri net benchmarks.

QBDD	**BDD**	**CBDD**	**ZDD**	**CZDD**	**TBDD**	**ESR**
3.108	2.971	2.038	1.215	1.167	1.160	**1.001**

Table 4.5 Overhead of node sizes (bits per node) as compared to QBDD nodes.

Level bits	**BDD**	**ZDD**	**CBDD**	**CZDD**	**TBDD**	**ESR−$\mathtt{L}_0$**	**ESR**
16	+16	+16	+32	+32	+48	+18	+20
20	+20	+20	+40	+40	+60	+22	+24
32	+32	+32	+64	+64	+96	+34	+36

are stored using 16 bits (as suggested by Bryant (2018)), 20 bits (as suggested by van Dijk et al. (2017)), and 32 bits.

ESRBDDs are clearly more memory efficient than CBDDs, CZDDs and TBDDs. There are a few instances in our experiments where TBDDs use marginally fewer nodes than ESRBDDs (less than 3.2% fewer nodes in every such instance), but not enough to overcome their per-node memory overhead.

4.5 Conclusions

We have shown that ESRBDDs are a simple, yet efficient, generalization of previous attempts at combining reduction rules. Unlike previous efforts, they are not biased towards any particular reduction rule and therefore eliminate the need for the user to prioritize the reduction rules. They also provide a framework for further generalizations through additional reduction rules—for example, “high-one” and “low-one”, the duals of “low-zero” and “high-zero” respectively.

ESRBDDs allow users to select a subset of reduction rules that suit their needs, and make it possible to integrate domain-specific reduction rules (a common phenomenon) with a subset of existing ones. ESRBDD nodes are also more compact than all previous such efforts, and new reduction rules can be added at a small cost—$\log_2 n$ bits per edge, where n is the number of reduction rules.

In the following chapters, we will demonstrate that ESRBDDs are easily extendable by including complement edges and other reduction rules, such as “high-one” or “low-one” reductions, while maintaining canonicity. We shall also demonstrate the compounding benefits of the compactness of ESRBDDs when combined with symbolic manipulation algorithms.

Acknowledgments

This work was supported in part by National Science Foundation grant ACI-1642397.

CHAPTER 5. ESRBDD OPERATIONS

In this chapter, we take a closer look at operations over ESRBDDs. First we give a deterministic depth-first reduction algorithm for ESRBDDs in Section 5.1. Next, in Section 5.2, we describe the *Apply* operation over ESRBDDs. We then give theoretical results that attest to the efficiency of Apply operations on ESRBDDs compared to BDDs and ZDDs.

5.1 Depth-First Reduction of ESRBDDs

We have previously seen a *chaotic* reduction algorithm in Algorithm 4.1, and proved that it terminates in time linear to the number of reducible nodes in the unreduced ESRBDD. In practice, we utilize a deterministic depth-first version of this algorithm, described in Algorithm 5.1. The depth-first reduction of any edge $\langle\kappa{,}p\rangle$ that starts at level n proceeds as follows:

1 *Reduce node p to $\langle\kappa'{,}p'\rangle$:*

- i Initialize $\langle\kappa'{,}p'\rangle$ with $\langle\mathtt{S}{,}p\rangle$.
- ii Reduce each child edge of p' using a recursive call to ReduceEdge$(l(p')-1{,}p'[i])$.
- iii If p' matches a replacement pattern (duplicate, redundant, low-zero or high-zero), replace $\langle\kappa'{,}p'\rangle$ with the result of the pattern replacement.

2 *Merge κ with $\langle\kappa'{,}p'\rangle$:*

- i If $p = \mathbf{0}$, then $\langle\mathtt{X}{,}\mathbf{0}\rangle$ is the reduced edge.
- ii If $l(p) = l(p')$, then $\langle\kappa{,}p'\rangle$ is the reduced edge.
- iii Otherwise, $l(p)$ must be greater than $l(p')$. If $\kappa = \kappa'$, then $\langle\kappa'{,}p'\rangle$ is the reduced edge.
- iv Otherwise, we have a κ that is not compatible with κ', We resolve it by building a node p'' representing pattern κ, one level above p. One of edges of p'' is $\langle\kappa'{,}p'\rangle$, and depending on

κ, the other is either $\langle \kappa', p' \rangle$ (if $\kappa = \mathtt{X}$), or $\langle \mathtt{X}, \mathbf{0} \rangle$ (if $\kappa \in \{\mathtt{L_0}, \mathtt{H_0}\}$). p'' by construction is not reducible, and the reduced edge is $\langle \kappa, p'' \rangle$.

v Finally, if the resulting edge does not skip any levels, we label it S.

Algorithm 5.1 Depth-first reduction of an ESRBDD.

```
1: procedure REDUCE(ESRBDD $\langle \kappa^\star, p^\star \rangle$)
2:     return REDUCEEDGE($L$, $\langle \kappa^\star, p^\star \rangle$)

3: procedure REDUCEEDGE(Level $n$, ESRBDD $\langle \kappa, p \rangle$)
4:     $\langle \kappa', p' \rangle \leftarrow$ REDUCENODE($p$)
5:     return MERGEEDGE($n$, $\kappa$, $l(p)$, $\langle \kappa', p' \rangle$)
```

Algorithm 5.2 Reduce an ESRBDD node.

```
1: procedure REDUCENODE(Node $p$)
2:     if $l(p) = 0$ then return $\langle \mathtt{S}, p \rangle$;
3:     $p' \leftarrow$ empty node at $l(p)$
4:     for all $i \in \{0, 1\}$ do $p'[i] \leftarrow$ REDUCEEDGE($l(p) - 1$, $p[i]$);
5:     if $p'$ is a redundant node then
6:         $\langle \kappa', p' \rangle \leftarrow \langle \mathtt{X}, p'[0].p \rangle$
7:     else if $p'$ is a low-zero node then
8:         $\langle \kappa', p' \rangle \leftarrow \langle \mathtt{L_0}, p'[1].p \rangle$
9:     else if $p'$ is a high-zero node then
10:        $\langle \kappa', p' \rangle \leftarrow \langle \mathtt{H_0}, p'[0].p \rangle$
11:    else
12:        if $p'$ is a duplicate of $q$ then
13:            $p' \leftarrow q$
14:        $\langle \kappa', p' \rangle \leftarrow \langle \mathtt{S}, p' \rangle$
15:    return $\langle \kappa', p' \rangle$
```

The time-complexity of the depth-first algorithm is linearly proportional to the number of reducible nodes in the DD, i.e. the same as Algorithm 4.1. But, the depth-first algorithm may be up to twice as fast: consider a node that is redundant, and whose child is a high-zero node, illustrated in Figure 5.1(a). The chaotic reduction might eliminate the redundant node first resulting in Figure 5.1(b), before eliminating the high-zero node resulting in Figure 5.1(c) – which cannot be

Algorithm 5.3 Merge an ESRBDD edge $\langle\kappa,p\rangle$ originating at level n with an incoming edge labelled $\hat{\kappa}$ originating at level $\hat{n}$.

```
1:  procedure MergeEdge(Level $\hat{n}$, Rule $\hat{\kappa}$, Level $n$, ESRBDD $\langle\kappa,p\rangle$)
2:      if $p = \mathbf{0} \vee \hat{n} = n \vee \hat{\kappa} = \mathtt{S} \vee \hat{\kappa} = \kappa$ then
3:          $\langle\kappa',p'\rangle \leftarrow \langle\kappa,p\rangle$
4:      else if $\kappa = \mathtt{S}$ then
5:          $\langle\kappa',p'\rangle \leftarrow \langle\hat{\kappa},p\rangle$
6:      else                                                    • $\hat{\kappa} \neq \mathtt{S} \wedge \hat{\kappa} \neq \kappa' \wedge \kappa' \neq \mathtt{S}$
7:          $\langle\kappa',p'\rangle \leftarrow$ BuildPattern($n+1$, $\hat{\kappa}$, $\langle\kappa,p\rangle$)
8:          $\kappa' \leftarrow \hat{\kappa}$
9:      if $\hat{n} - l(p') < 2$ then $\kappa' \leftarrow \mathtt{S}$
10:     return $\langle\kappa',p'\rangle$
```

Algorithm 5.4 Build ESRBDD node at level n according to pattern $\hat{\kappa}$.

```
1:  procedure BuildPattern(Level $n$, Pattern $\hat{\kappa}$, ESRBDD $\langle\kappa,p\rangle$)
2:      /*Require: $\hat{\kappa} \neq \mathtt{S} \wedge \hat{\kappa} \neq \kappa$ */
3:      $p' \leftarrow$ new node at level $n$
4:      if $\hat{\kappa} = \mathtt{X}$ then
5:          $p'[0] \leftarrow \langle\kappa,p\rangle$ $p'[1] \leftarrow \langle\kappa,p\rangle$
6:      else if $\hat{\kappa} = \mathtt{L_0}$ then
7:          $p'[0] \leftarrow \langle\mathtt{X},\mathbf{0}\rangle$ $p'[1] \leftarrow \langle\kappa,p\rangle$
8:      else                                                    • $\hat{\kappa} = \mathtt{H_0}$
9:          $p'[0] \leftarrow \langle\kappa,p\rangle$ $p'[1] \leftarrow \langle\mathtt{X},\mathbf{0}\rangle$
10:     if $p'$ is a duplicate of $q$ then $p' \leftarrow q$;
11:     return $\langle\mathtt{S},p'\rangle$
```

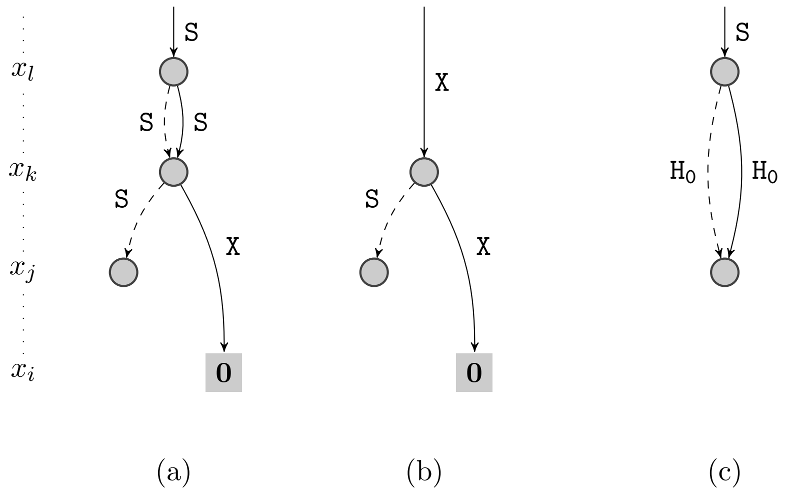


Figure 5.1 Chaotic vs Bottom-Up Reduction: (a) Unreduced ESRBDD, (b) Removing Redundant first, (c) Removing Highzero first.

further reduced. But, Algorithm 5.1 will reduce the child edges first, and will therefore eliminate the high-zero node first, and get to the final reduced ESRBDD in single step.

5.2 *Apply* operation

The *Apply* operation is an essential tool for manipulating decision diagrams (Bryant, 1986). In its most common form, it can be used to perform binary boolean operations over two decision diagrams, for example, conjunction and disjunction. The Apply operation over BDDs (Apply-BDD) is outlined in Algorithm 5.5, and it works recursively by applying the operation over corresponding child nodes before applying the operation over the parent nodes. Note that Apply-BDD is able to skip over levels that are skipped by both operands due to the semantics of a long edge in a BDD.

The Apply operation over ZDDs (Apply-ZDD) is outlined in Algorithm 5.6, and works similar to Apply-BDD except that it recurses one level at-a-time. ZDD operations such as OR and AND are based on an alternate algorithm (Apply-ZDD-0) that allows skipping of levels. What sets operations such as OR and AND apart from general boolean operations for ZDDs, is that the result of $\langle \mathbf{0} \oplus \mathbf{0} \rangle$ is $\mathbf{0}$, and therefore, the result of applying such an operation over long edges to p_0 and p_1 is a long edge to $p_0 \oplus p_1$. When $\langle \mathbf{0} \oplus \mathbf{0} \rangle$ is $\mathbf{1}$, such as the boolean operation NOR, we must either use Apply-ZDD or perform a negation on the result of the complementary operation (for NOR, perform OR and negate the result).

Algorithm 5.5 Apply for BDDs.

```
procedure BddEdge(Level $n$, BDD $p$, int $i$)                • $i$: index, $i \in \{0, 1\}$
    if $n = l(p)$ then return $p[i]$
    /* interpreting a long edge */
    return $p$

procedure Apply-BDD(Opcode $\oplus$, BDD $p_0$, BDD $p_1$)
    if $l(p_0) = 0 \wedge l(p_1) = 0$ then return $p_0 \oplus p_1$
    if "$\oplus$, $p_0$, $p_1$, $p$" $\in$ CT then return $p$
    $n \leftarrow$ Max$(l(p_0), l(p_1))$
    $p \leftarrow$ new node at level $n$
    $p[0] \leftarrow$ Apply-BDD$(\oplus$, BddEdge$(n, p_0, 0)$, BddEdge$(n, p_1, 0))$
    $p[1] \leftarrow$ Apply-BDD$(\oplus$, BddEdge$(n, p_0, 1)$, BddEdge$(n, p_1, 1))$
    $p \leftarrow$ Reduce$(p)$
    CT $\leftarrow$ CT $\cup$ "$\oplus$, $p_0$, $p_1$, $p$"
    return $p$
```

5.2.1 Time-complexity

The run-time complexity of Apply-BDD is $\mathcal{O}(|\text{BDD}(A)|.|\text{BDD}(B)|)$ (Bryant, 1986), where A and B are boolean functions over L variables, $\mathbb{B}^L \to \mathbb{B}$, BDD$(f)$ is the BDD representing a boolean function f, and $|G|$ is the number of nodes and edges in graph G.

Algorithm 5.6 Apply for ZDDs.

```
procedure ZddEdge(Level $n$, ZDD $p$, int $i$)                    • $i$: index, $i \in \{0, 1\}$
    if $n = l(p)$ then return $p[i]$
    /* interpreting a long edge */
    if $i = 0$ then return $p$
    return 0

procedure Apply-ZDD(Opcode $\oplus$, Level n, ZDD $p_0$, ZDD $p_1$)
    if $l(p_0) = 0 \wedge l(p_1) = 0$ then return $p_0 \oplus p_1$
    if "$\oplus$, $n$, $p_0$, $p_1$, $p$" $\in$ CT then return $p$
    $p \leftarrow$ new node at level $n$
    $p[0] \leftarrow$ Apply-ZDD($\oplus, n-1,$ ZddEdge$(n, p_0, 0),$ ZddEdge$(n, p_1, 0)$)
    $p[1] \leftarrow$ Apply-ZDD($\oplus, n-1,$ ZddEdge$(n, p_0, 1),$ ZddEdge$(n, p_1, 1)$)
    $p \leftarrow$ Reduce($p$)
    CT $\leftarrow$ CT $\cup$ "$\oplus$, $n$, $p_0$, $p_1$, $p$"
    return $p$

procedure Apply-ZDD-0(Opcode $\oplus$, ZDD $p_0$, ZDD $p_1$)
    /* Requires: $\mathbf{0} \oplus \mathbf{0}$ is $\mathbf{0}$ */
    if $l(p_0) = 0 \wedge l(p_1) = 0$ then return $p_0 \oplus p_1$
    if "$\oplus$, $p_0$, $p_1$, $p$" $\in$ CT then return $p$
    $n' \leftarrow$ Max($l(p_0), l(p_1)$)
    $p \leftarrow$ new node at level $n'$
    $p[0] \leftarrow$ Apply-ZDD-0($\oplus,$ ZddEdge$(n', p_0, 0),$ ZddEdge$(n', p_1, 0)$)
    $p[1] \leftarrow$ Apply-ZDD-0($\oplus,$ ZddEdge$(n', p_0, 1),$ ZddEdge$(n', p_1, 1)$)
    $p \leftarrow$ Reduce($p$)
    CT $\leftarrow$ CT $\cup$ "$\oplus$, $p_0$, $p_1$, $p$"
    return $p$
```

The run-time complexity of APPLY-ZDD is $\mathcal{O}(|\text{ZDD}(A)|.|\text{ZDD}(B)|)$ for boolean operations $\oplus$ that satisfy $\mathbf{0} \oplus \mathbf{0} = \mathbf{0}$, and is $\mathcal{O}(|\text{ZDD}(A)|.|\text{ZDD}(B)|.L)$ otherwise (Minato, 2001), where $\text{ZDD}(f)$ is the ZDD representing a boolean function f.

The run-time complexity of BDDs (and ZDDs) is based on the use of memoization of computation results, and is the count of the number of unique recursive calls made by Apply. Let $\Psi_{j,\oplus}$ represent number of unique recursive calls made by the Apply algorithm when performing operation $\oplus$ over decision diagrams of type j, where $\oplus$ is a binary boolean operation, and $j \in \{\text{BDD}, \text{ZDD}, \text{CBDD}, \text{CZDD}, \text{TBDD}, \text{ESR}\}$. We know that

$$\Psi_{\text{BDD},\oplus} \leq |\text{BDD}(A)|.|\text{BDD}(B)| \text{ for any } \oplus\,, \tag{5.1}$$

$$\Psi_{\text{ZDD},\oplus} \leq |\text{ZDD}(A)|.|\text{ZDD}(B)| \text{ when } \mathbf{0} \oplus \mathbf{0} = \mathbf{0}\text{, and} \tag{5.2}$$

$$\Psi_{\text{ZDD},\oplus} \leq |\text{ZDD}(A)|.|\text{ZDD}(B)|.L \text{ when } \mathbf{0} \oplus \mathbf{0} \neq \mathbf{0}. \tag{5.3}$$

Algorithm 5.7 Return the low or high edge of an ESRBDD edge w.r.t to a given level.

```
1: procedure EDGE(Level $n$, ESRBDD $\langle\kappa,p\rangle$, Index $i$)
2:     if $n = l(p)$ then return $\langle\kappa',p'\rangle \leftarrow p[i]$
3:     if $\kappa = \mathtt{X} \vee (i = 0 \wedge \kappa = \mathtt{H_0}) \vee (i = 1 \wedge \kappa = \mathtt{L_0})$ then
4:         $\langle\kappa',p'\rangle \leftarrow \langle\kappa,p\rangle$
5:     else
6:         $\langle\kappa',p'\rangle \leftarrow \langle\mathtt{X},\mathbf{0}\rangle$
7:     if $n = l(p') + 1$ then $\kappa' \leftarrow \mathtt{S}$
8:     return $\langle\kappa',p'\rangle$
```

The Apply operation over ESRBDDs is described in Algorithm 5.8. It works in similar fashion to the Apply over BDDs, but in addition to the operands $\text{ESR}(A)$ and $\text{ESR}(B)$, APPLY-ESR requires level n, resulting in

$$\Psi_{\text{ESR},\oplus} \leq |\text{ESR}(A)|.|\text{ESR}(B)|.L$$

unique recursive calls, and a time complexity of

$$\Phi_{\text{ESR}} = \mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|.L).$$

Algorithm 5.8 Apply for ESRBDDs.

```
procedure APPLY-ESR(Opcode $\oplus$, Level $n$, ESRBDD $\langle\kappa_0, p_0\rangle$, ESRBDD $\langle\kappa_1, p_1\rangle$)
    if $n = 0$ then return $\langle \mathtt{S}, p_0 \oplus p_1\rangle$
    if "$\oplus$, $n$, $\langle\kappa_0,p_0\rangle$, $\langle\kappa_1,p_1\rangle$, $\langle\kappa,p\rangle$" $\in \mathtt{CT}$ then return $\langle\kappa,p\rangle$
    $p \leftarrow$ new node at level $n$
    for all $i \in \{0,1\}$ do
        $\langle\kappa'_0,p'_0\rangle \leftarrow$ EDGE($n$, $\langle\kappa_0,p_0\rangle$, $i$)
        $\langle\kappa'_1,p'_1\rangle \leftarrow$ EDGE($n$, $\langle\kappa_1,p_1\rangle$, $i$)
        $n' \leftarrow n - 1$
        if $l(p'_0) < n' \wedge l(p'_1) < n' \wedge \kappa'_0 = \kappa'_1 \wedge (\kappa'_0 = \mathtt{X} \vee \mathbf{0} \oplus \mathbf{0} = \mathbf{0})$ then
            $n' \leftarrow$ MAX($l(p'_0)$, $l(p'_1)$);                              • operation can skip levels
            $p[i] \leftarrow$ APPLY-ESR($\oplus$, $n'$, $\langle\kappa'_0,p'_0\rangle$, $\langle\kappa'_1,p'_1\rangle$)
            $p[i] \leftarrow$ MERGEEDGE($n-1$, $\kappa'_0$, $n'$, $p[i]$)            • merge $\kappa'_0$ with $p[i]$
        else
            $p[i] \leftarrow$ APPLY-ESR($\oplus$, $n'$, $\langle\kappa'_0,p'_0\rangle$, $\langle\kappa'_1,p'_1\rangle$)
    $\langle\kappa,p\rangle \leftarrow$ REDUCEEDGE($n$, $\langle \mathtt{S},p\rangle$)
    $\mathtt{CT} \leftarrow \mathtt{CT} \cup$ "$\oplus$, $n$, $\langle\kappa_0,p_0\rangle$, $\langle\kappa_1,p_1\rangle$, $\langle\kappa,p\rangle$"
    return $\langle\kappa,p\rangle$
```

This time-complexity is better than it seems at first-glance, and we prove that for any operation $\oplus$, the number of unique recursive calls in APPLY-ESR is less than twice the number of unique recursive calls in APPLY-BDD and APPLY-ZDD.

Theorem 5.2.1

The time complexity of Algorithm 5.8 is $\mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|.L)$.

Proof: We observe that for each pair of edges (e_i, e_j) and non-negative integer n where $e_i \in \text{ESR}(A)$, $e_j \in \text{ESR}(B)$ and $n \in [0, L]$ we can construct $|\text{ESR}(A)| \times |\text{ESR}(B)| \times L$ distinct tuples to pass as arguments to Apply. Any other tuple would be a duplicate call whose result has been previously computed and stored in the compute cache. □

Definition 5.2.1

We define a *simple-long-edge* to be a long-edge in a BDD (or a ZDD) whose destination node does not fit any of the reduction rules of ESRBDDs.

Long-edges that are not simple-long-edges can be further reduced by ESRBDDs since they end in nodes that satisfy an ESRBDD reduction pattern. For example, consider the edge $\langle \mathtt{X}, a \rangle$. If a is a high-zero node, then, after reduction, a is eliminated and a node a' at level $l(a)+1$ is created such that $a'[0] = a'[1] = \langle \mathtt{H}_0, a \rangle$. We point out that a' cannot be further reduced. As discussed previously, this reduction rule in ESRBDDs is used to enforce canonicity. We have previously shown that ESR(A) is never larger than BDD(A), and there are situations when they are of the same size, but even in those situations they are not guaranteed to be isomorphic. BDD(A) and ESR(A) are isomorphic only when there is no node in BDD(A) that satisfies an ESRBDD reduction pattern. The above implies that all the long-edges in BDD(A) must be simple-long-edges.

Theorem 5.2.2

The number of unique recursive calls to Apply-ESR in Apply-ESR is less than twice the number of unique recursive calls to Apply-BDD in Apply-BDD.

$$\Psi_{\mathrm{ESR},\oplus} \leq 2 \times \Psi_{\mathrm{BDD},\oplus}$$

Proof: First, we assume that BDD(A) and BDD(B) are isomorphic with ESR(A) and ESR(B). This implies that there are no high-zero or low-zero nodes in BDD(A) and BDD(B), are therefore all long edges in BDD(A), BDD(B), ESR(A) and ESR(B) are also simple-long-edges. For every pair of edges (e_i, e_j) in BDD(A) and BDD(B), there is at most one unique recursive call to Apply-BDD (since any duplicate will cause a cache hit). Similarly, for every pair of simple-long-edges (e_i, e_j) in ESR(A) and ESR(B), due to lines 9–12 in Algorithm 5.8, there is at most one unique recursive call to Apply-ESR and any duplicate will cause a cache hit. Therefore, when all long edges are simple-long-edges, $\Psi_{\mathrm{ESR},\oplus} \leq \Psi_{\mathrm{BDD},\oplus}$.

Next, we allow BDD(A) and BDD(B) to contain low-zero and high-zero nodes, but we retain the assumption that any long-edge is a simple-long-edge. From the above discussion we know that when restricted to simple-long-edges $\Psi_{\mathrm{ESR},\oplus} \leq \Psi_{\mathrm{BDD},\oplus}$. Therefore, we consider only those cases where at least one of the operands to Apply-BDD and Apply-ESR is a low-zero or high-zero node, and a $\mathtt{L_0}$ or $\mathtt{H_0}$ edge respectively. In the discussion that follows, the operands to Apply-ESR are edges $\langle \kappa_i, p_i \rangle$ and $\langle \kappa_j, p_j \rangle$, n is the level at which the edges originate (or are being evaluated), and n' is the *higher* of two nodes p_i and p_j and is defined as $n' = \textsc{Max}(l(p_i), l(p_j))$.

- If $\kappa_i = \kappa_j = \mathtt{H_0}$, or $\kappa_i = \kappa_j = \mathtt{L_0}$, there is at most one unique call to Apply-ESR per level, between levels n and n'. This is because at every such level, one of the two calls is a duplicate call that is equivalent to $\langle \oplus, \langle \mathtt{X}, \mathbf{0} \rangle, \langle \mathtt{X}, \mathbf{0} \rangle \rangle$. Apply-BDD will require at least one unique recursive call per level for levels n to n', because these long edges represent *unreducible* high-zero or low-zero node at each level in BDDs. It follows that $\Psi_{\text{ESR},\oplus} \leq \Psi_{\text{BDD},\oplus}$ in this case.

- If $\kappa_i = \mathtt{X}$ and $\kappa_j = \mathtt{H_0}$, there is at most one unique call to Apply-ESR per level, between levels n and n' since at least one of the two calls is a duplicate call that is equivalent to $\langle \oplus, \langle \mathtt{X}, p_i \rangle, \langle \mathtt{X}, \mathbf{0} \rangle \rangle$. Apply-BDD will require at least one unique recursive call per level for levels n to n', because the $\mathtt{H_0}$ edge represents *unreducible* high-zero nodes at each level in the corresponding BDD. It follows that $\Psi_{\text{ESR},\oplus} \leq \Psi_{\text{BDD},\oplus}$ in this case. The cases, $\kappa_i = \mathtt{X}$ and $\kappa_j = \mathtt{L_0}$, $\kappa_j = \mathtt{X}$ and $\kappa_i = \mathtt{H_0}$, and $\kappa_j = \mathtt{X}$ and $\kappa_i = \mathtt{L_0}$ are symmetric.

- If $\kappa_i = \mathtt{L_0}$ and $\kappa_j = \mathtt{H_0}$, there are two calls generated at level n for Apply-ESR and Apply-BDD, namely, $\langle \oplus, \langle \mathtt{X}, \mathbf{0} \rangle, \langle \mathtt{H_0}, p_j \rangle \rangle$ and $\langle \oplus, \langle \mathtt{L_0}, p_i \rangle, \langle \mathtt{X}, \mathbf{0} \rangle \rangle$. Both these calls have been shown above to satisfy $\Psi_{\text{ESR},\oplus} \leq \Psi_{\text{BDD},\oplus}$.

- The remaining cases are symmetric.

Finally, we discuss the general case, where BDD(A) and BDD(B) may contain long edges that are not simple-long-edges. Without loss of generality, we restrict our discussion to the cases where at least one of the operands to Apply-BDD is a long edge ending in a high-zero node (since all other cases are symmetric or have been dealt with above). Let p_i be the high-zero node, and p_j be a node that is neither high-zero nor low-zero. In ESR(A), p_i is eliminated and a node p'_i is created at level $l(p_i) + 1$ such that $p'_i[0] = p'_i[1] = \langle \mathtt{H_0}, p_i[0] \rangle$ (see Figures 5.1(b) and 5.1(c)). There are three cases depending on $l(p_i)$ and $l(p_j)$:

1. $l(p_i) = l(p_j) + 1$:

   Three unique recursive calls to Apply-BDD are generated:

   (a) $\langle \oplus, p_i, p_j \rangle$.

   (b) $\langle \oplus, p_i[0], p_j \rangle$.

   (c) $\langle \oplus, p_i[1], p_j \rangle$.

   Four unique recursive calls to Apply-ESR are generated:

   (a) $\langle l(p'_i), \oplus, p'_i, p_j \rangle$.

   (b) $\langle l(p'_i) - 1, \oplus, p'_i[0], p_j \rangle$.

   (c) $\langle l(p'_i) - 1, \oplus, p'_i[1], p_j \rangle$ which is equivalent to $\langle l(p'_i) - 1, \oplus, p'_i[0], p_j \rangle$ since $p'_i[0] = p'_i[1]$. This generates a cache hit and is not counted as a unique recursive call.

   (d) $\langle l(p'_i) - 2, \oplus, p'_i[0], p_j \rangle$.

   (e) $\langle l(p'_i) - 2, \oplus, p'_i[0], p_j \rangle$.

2. $l(p_i) = l(p_j)$:

   Three unique recursive calls to Apply-BDD are generated:

   (a) $\langle \oplus, p_i, p_j \rangle$.

   (b) $\langle \oplus, p_i[0], p_j[0] \rangle$.

   (c) $\langle \oplus, p_i[1], p_j[1] \rangle$.

Four unique recursive calls to Apply-ESR are generated:

(a) $\langle l(p'_i), \oplus, p'_i, p_j \rangle$.

(b) $\langle l(p'_i) - 1, \oplus, p'_i[0], p_j \rangle$.

(c) $\langle l(p'_i) - 1, \oplus, p'_i[1], p_j \rangle$ which is equivalent to $\langle l(p'_i) - 1, \oplus, p'_i[0], p_j \rangle$ since $p'_i[0] = p'_i[1]$. This generates a cache hit and is not counted as a unique recursive call.

(d) $\langle l(p'_i) - 2, \oplus, p'_i[0], p_j[0] \rangle$.

(e) $\langle l(p'_i) - 2, \oplus, p'_i[0], p_j[1] \rangle$.

3. $l(p_i) = l(p_j) - 1$:

At most seven unique recursive calls to Apply-BDD are generated:

(a) $\langle \oplus, p_i, p_j \rangle$.

(b) $\langle \oplus, p_i, p_j[0] \rangle$.

(c) $\langle \oplus, p_i, p_j[1] \rangle$.

(d) Two calls for children of p_i and $p_j[0]$.

(e) Two calls for children of p_i and $p_j[1]$.

We note that since $p_j[0] \neq p_j[1]$, there can be at most one duplicate recursive call among these seven. There are exactly the same number of unique recursive calls to Apply-ESR generated:

(a) $\langle l(p'_i), \oplus, p'_i, p_j \rangle$.

(b) $\langle l(p'_i) - 1, \oplus, p'_i[0], p_j[0] \rangle$.

(c) $\langle l(p'_i) - 1, \oplus, p'_i[1], p_j[1] \rangle$.

(d) Two calls for children of p_i and $p_j[0]$.

(e) Two calls for children of p_i and $p_j[1]$.

4. The cases where p_i and p_j are more than one level apart are analogous.

The case when both long edges end in high-zero or low-zero nodes are similar to the case above with single a non-simple-long-edge, so we restrict our discussion to the worst-case for Apply-ESR. Let p_i and p_j be high-zero or low-zero nodes. In ESR(A), p_i is eliminated and a node p'_i is created at level $l(p_i)+1$ such that $p'_i[0] = p'_i[1] = \langle \kappa_i, p_i[0] \rangle$ and $\kappa_i \in \{\mathtt{L_0}, \mathtt{H_0}\}$. Similarly, in ESR($B$), p_j is eliminated and a node p'_j is created at level $l(p_j)+1$ such that $p'_j[0] = p'_j[1] = \langle \kappa_j, p_i[0] \rangle$ and $\kappa_j \in \{\mathtt{L_0}, \mathtt{H_0}\}$. We discuss the case where $l(p_i) = l(p_j)$ since, the rest of the cases do not increase the ratio $\Psi_{\mathrm{ESR},\oplus} : \Psi_{\mathrm{BDD},\oplus}$.

Three unique recursive calls to Apply-BDD are generated (illustrated in Figure 5.2(c)):

1. $\langle \oplus, p_i, p_j \rangle$.
2. $\langle \oplus, p_i[0], p_j[0] \rangle$.
3. $\langle \oplus, p_i[1], p_j[1] \rangle$.

Four unique recursive calls to Apply-ESR are generated (illustrated in Figure 5.2(f)):

1. $\langle l(p'_i), \oplus, p'_i, p'_j \rangle$.
2. $\langle l(p'_i)-1, \oplus, p'_i[0], p'_j[0] \rangle$.
3. $\langle l(p'_i)-1, \oplus, p'_i[1], p'_j[1] \rangle$ which is equivalent to $\langle l(p'_i)-1, \oplus, p'_i[0], p'_j[0] \rangle$ since $p'_i[0] = p'_i[1]$ and $p'_j[0] = p'_j[1]$. This generates a cache hit and is not counted as a unique recursive call.
4. $\langle l(p'_i)-2, \oplus, p'_i[0], p'_j[0] \rangle$.
5. $\langle l(p'_i)-2, \oplus, p'_i[0], p'_j[1] \rangle$.

Therefore, in every case $\Psi_{\mathrm{ESR},\oplus} \leq 2 \times \Psi_{\mathrm{BDD},\oplus}$

□

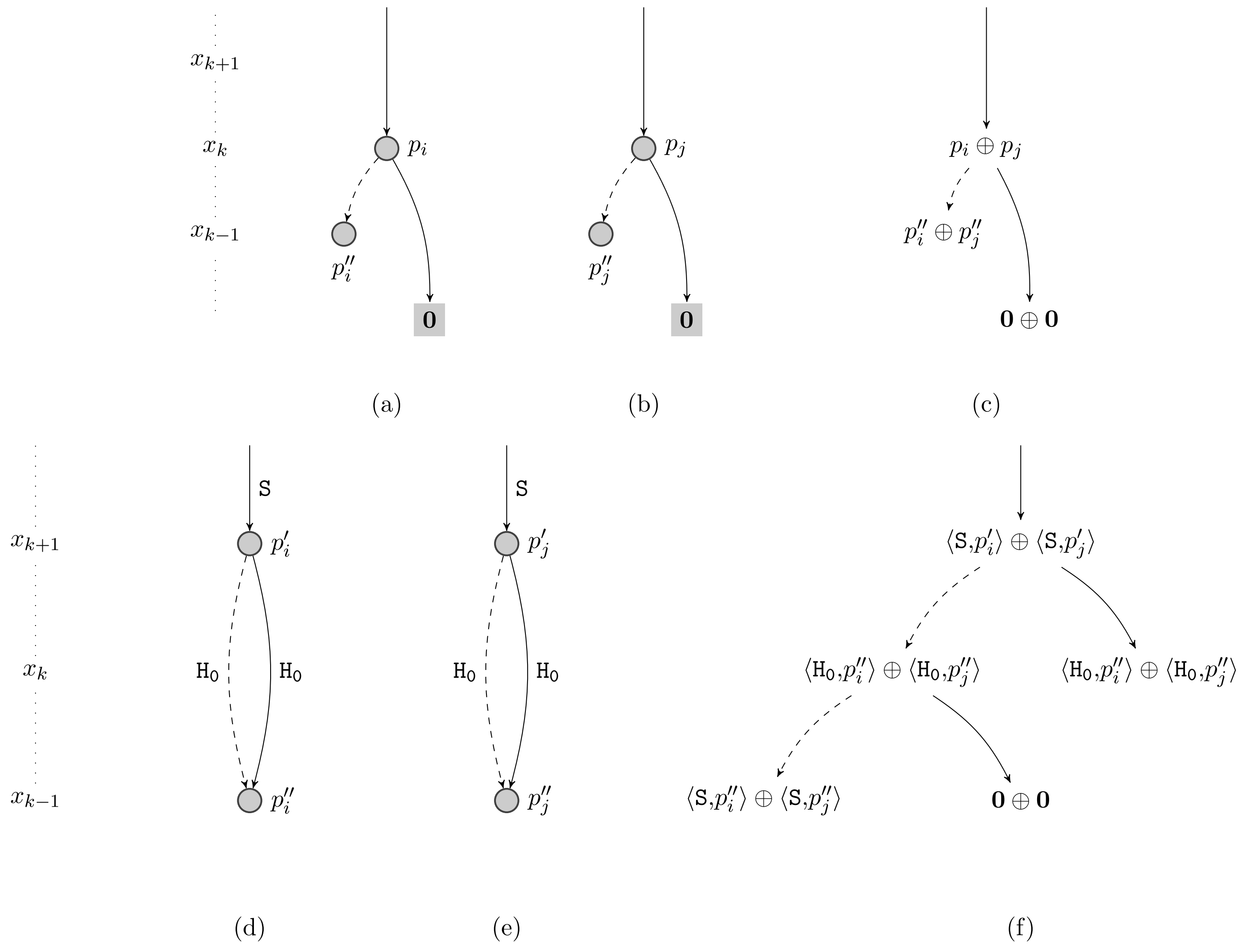

x_{k+1}
x_k
x_{k-1}
p_i
p_i''
$\mathbf{0}$
(a)
p_j
p_j''
$\mathbf{0}$
(b)
$p_i \oplus p_j$
$p_i'' \oplus p_j''$
$\mathbf{0} \oplus \mathbf{0}$
(c)
S
p_i'
$\mathtt{H}_0$
p_i''
(d)
S
p_j'
$\mathtt{H}_0$
p_j''
(e)
$\langle \mathtt{S}, p_i' \rangle \oplus \langle \mathtt{S}, p_j' \rangle$
$\langle \mathtt{H}_0, p_i'' \rangle \oplus \langle \mathtt{H}_0, p_j'' \rangle$
$\langle \mathtt{S}, p_i'' \rangle \oplus \langle \mathtt{S}, p_j'' \rangle$
$\mathbf{0} \oplus \mathbf{0}$
(f)

Figure 5.2 Worst-case time-complexity for Apply-ESR with non-simple-long-edges. Apply-BDD (a), (b), (c). Apply-ESR (d), (e), (f).

Theorem 5.2.3

The number of unique recursive calls in Apply-ESR is less than twice the number of unique recursive calls in Apply-ZDD.

$$\Psi_{\text{ESR},\oplus} \leq 2.\Psi_{\text{ZDD},\oplus}$$

Proof: The proof is analogous to the proof for BDDs but with simple-long-edges representing ZDD reductions ending in nodes that are neither redundant nor low-zero. Lines 9–12 in Apply-ESR ensure that it skips the levels that are skipped by Apply-ZDD-0 for operations that satisfy $\mathbf{0} \oplus \mathbf{0} = \mathbf{0}$. □

Theorem 5.2.4

The number of unique recursive calls in Apply-ESR is less than twice the number of unique recursive calls in Apply-BDD and Apply-ZDD.

$$\Psi_{\text{ESR},\oplus} \leq \text{Min} \begin{cases} 2.\Psi_{\text{BDD},\oplus} \\ 2.\Psi_{\text{ZDD},\oplus} \end{cases}$$

Proof: From Theorem 5.2.1 and Theorem 5.2.1. □

5.3 Conclusions

In Chapter 4, ESRBDDs were shown to be simpler and more compact than existing works. In this chapter, we have shown that the Apply operation on ESRBDDs is at least as fast as those on BDDs and ZDDs. Indeed, for any binary function, ESRBDDs are at least as compact as the smaller of the two representations (BDDs and ZDDs); and for Apply operations over such functions, ESRBDDs are at least as fast as the faster of the two representations.

CHAPTER 6. ESRBDDS WITH COMPLEMENTARY EDGES

In Chapter 4, we defined ESRBDDs to eliminate duplicate, redundant, low-zero and high-zero nodes; proved that they were canonical and gave experimental evidence for their compactness. In Chapter 5, we proved that the Apply operation over ESRBDDs has a time-complexity at most that of BDDs and ZDDs. In this chapter, we demonstrate the extendability of ESRBDDs, as claimed in (Babar et al., 2019), by eliminating complementary, low-one, and high-one nodes and their counterparts.

The rest of this chapter is organized as follows. Section 6.1 gives a brief introduction to the additional reduction rules we intend to integrate into ESRBDDs. Section 6.2 defines ESRBDDs with complement edges, *CESRBDDs*. Section 6.3 defines reduced CESRBDDs and provides a reduction algorithm. Section 6.4 proves the canonicity of reduced CESRBDDs. Section 6.5 makes a comparison between the size of CESRBDDs and BDDs, ZDDs, CBDDs, CZDDs, TBDDs and ESRBDDs, and Section 6.6 concludes.

6.1 Introduction

We have defined a few node patterns in Chapter 4, namely, redundant, high-zero and low-zero nodes, and replaced them in ESRBDDs with long edges labelled $\mathtt{X}$, $\mathtt{H}_0$ and $\mathtt{L}_0$ respectively. We now expand the set of patterns that we would like to eliminate in ESRBDDs.

The *high-one* pattern is derived from the high-zero pattern by replacing high edges to **0** with **1**. Further, for any high-zero edge going to node p, there is an implicit reference to terminal node **0**. Swapping these two nodes in the high-zero pattern, produces the *high-swap-zero* pattern. Similarly, for any high-one edge going to node p, there is an implicit reference to terminal node **1**. Swapping these two nodes in the high-one pattern, produces the *high-swap-one* pattern. Figure 6.1 illustrates these patterns.

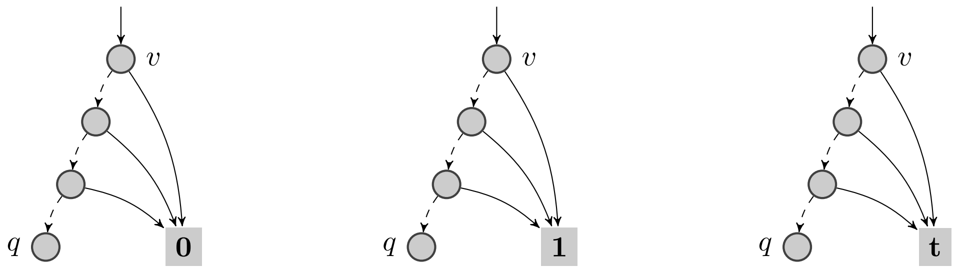


Figure 6.1 High patterns.
Left to right: High-zero $\mathtt{H_0}$, High-one $\mathtt{H_1}$, High-t $\mathtt{H_t}$.

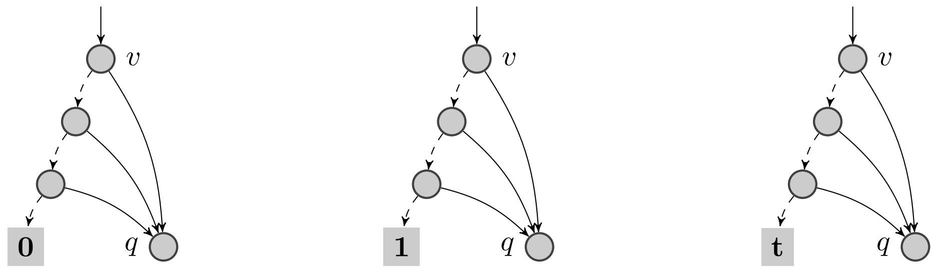


Figure 6.2 High-Swap patterns.
Left to right: High-swap-zero $\tilde{\mathtt{H}}_0$, High-swap-one $\tilde{\mathtt{H}}_1$, High-swap-t $\tilde{\mathtt{H}}_\mathtt{t}$.

Following the same process for the low-zero pattern we derive the *low-one*, *low-swap-zero* and *low-swap-one* patterns illustrated in Figure 6.3.

Complementary edges introduced in Brace et al. (1990) for BDDs and in S. Minato et al. (1990) for ZDDs, can reduce the number of nodes in the decision diagram by a factor of two by representing a node and its complement with the same node. To ensure canonicity Brace et al. and S. Minato et al. described the idea of node normalization: a normalized node cannot have a complemented low edge, and a reduced decision diagram cannot contain non-normalized nodes. A node can be normalized by complementing both child edges and any incoming edge. From here on we refer to ESRBDDs that incorporate all of the above patterns as "ESRBDDs with complementary edges", or CESRBDDs for short.

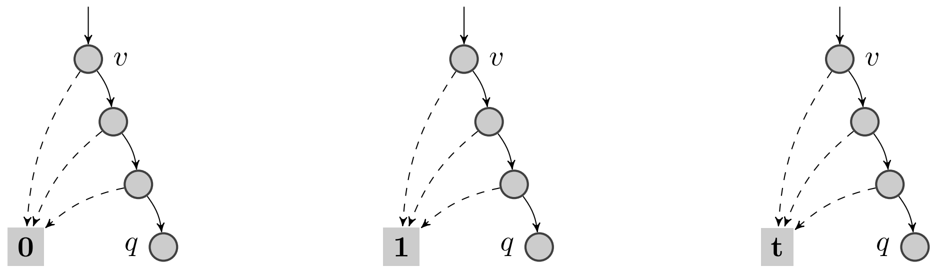


Figure 6.3 Low patterns.
Left to right: Low-zero $\mathtt{L}_0$, Low-one $\mathtt{L}_1$, Low-t $\mathtt{L}_\mathbf{t}$.

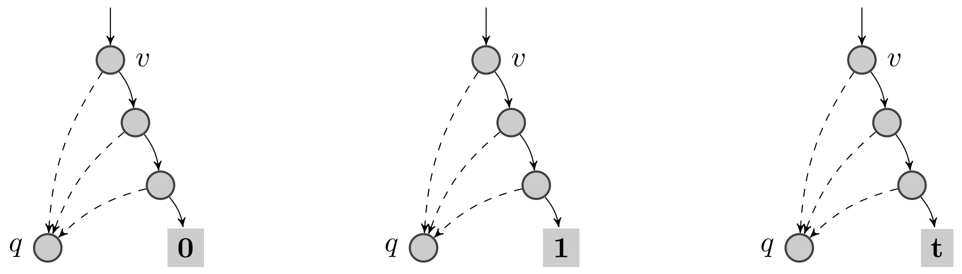


Figure 6.4 Low-Swap patterns.
Left to right: Low-swap-zero $\tilde{\mathtt{L}}_0$, Low-swap-one $\tilde{\mathtt{L}}_1$, Low-swap-t $\tilde{\mathtt{L}}_\mathbf{t}$.

We note that for certain functions these patterns overlap. For example, a high-zero pattern for a function over a single boolean variable may be match a low-one pattern. In the rest of this chapter, we give more precise definitions for these patterns and show that reduced CESRBDDs are canonical.

6.2 Definition of CESRBDDs

An L-level *(ordered) complementary edge-specified reduction* binary decision diagram (CESRBDD) is a directed acyclic graph where the only *terminal* node $\mathbf{\Omega}$ is at level 0, $l(\mathbf{\Omega}) = 0$, while each *nonterminal* node p belongs to a level $l(p) \in \{1, ..., L\}$ and has two outgoing edges, $p[0]$ and $p[1]$, pointing to nodes at lower levels. An edge is a triple $e = \langle e.r, e.c, e.p \rangle$, where $e.r$ is a *reduction rule* in $\{\mathtt{S}, \mathtt{X}, \mathtt{L}_0, \mathtt{H}_0, \mathtt{L}_1, \mathtt{H}_1, \tilde{\mathtt{H}}_0, \tilde{\mathtt{L}}_0, \tilde{\mathtt{H}}_1, \tilde{\mathtt{L}}_1\}$, $e.p$ is the node to which edge e points, and $e.c$ is a boolean ($\mathtt{0}$ and $\mathtt{1}$ representing *false* and *true* respectively), that indicates whether the function encoded by $e.p$ should be complemented. For $i \in \{0, 1\}$, if $l(p[i].p) = l(p) - 1$, we say that $p[i]$ is a *short* edge and require that $p[i].r = \mathtt{S}$. If instead $l(p[i].p) < l(p) - 1$, the only other possibility, we say $p[i]$ is a *long* edge, since it "skips over" one or more levels, and require that $p[i].r \in \{\mathtt{X}, \mathtt{L}_0, \mathtt{H}_0, \mathtt{L}_1, \mathtt{H}_1, \tilde{\mathtt{H}}_0, \tilde{\mathtt{L}}_0, \tilde{\mathtt{H}}_1, \tilde{\mathtt{L}}_1\}$.

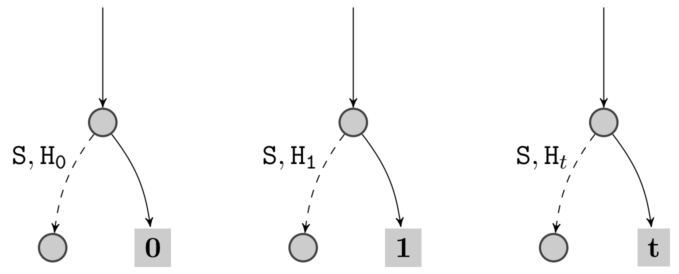

$\mathtt{S}, \mathtt{H}_0$
0
$\mathtt{S}, \mathtt{H}_1$
1
$\mathtt{S}, \mathtt{H}_t$
t

Figure 6.5 High-zero, high-one and high-t nodes.

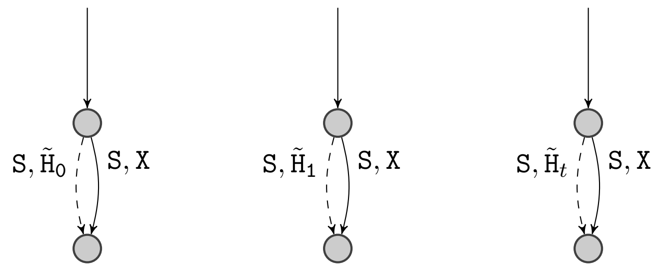

$\mathtt{S}, \tilde{\mathtt{H}}_0$
$\mathtt{S}, \mathtt{X}$
$\mathtt{S}, \tilde{\mathtt{H}}_1$
$\mathtt{S}, \mathtt{X}$
$\mathtt{S}, \tilde{\mathtt{H}}_t$
$\mathtt{S}, \mathtt{X}$

Figure 6.6 High-swap-zero, high-swap-one and high-swap-t nodes.

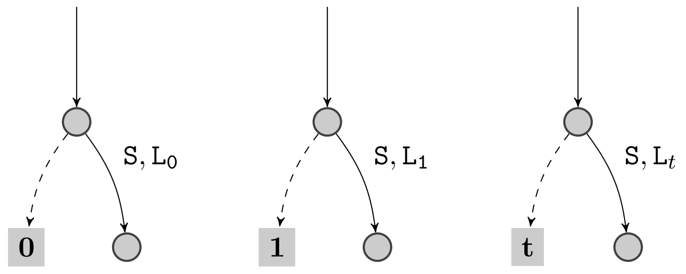

$\mathtt{S}, \mathtt{L}_0$
0
$\mathtt{S}, \mathtt{L}_1$
1
$\mathtt{S}, \mathtt{L}_t$
t

Figure 6.7 Low-zero, low-one and low-t nodes.

The reduction rule on an edge specifies its meaning when skipping levels, thus for short edges it is just $\mathtt{S}$. Chapter 4 defined redundant (Figure 4.2), high-zero (Figure 4.3) and low-zero (Figure 4.4) nodes and represented them as edges labelled $\mathtt{X}$, $\mathtt{H}_0$ and $\mathtt{L}_0$ respectively. We also define new rules: $\mathtt{H}_1$ (high-one) and $\mathtt{L}_1$ (low-one) are analogous to $\mathtt{H}_0$ and $\mathtt{L}_0$ respectively, $\tilde{\mathtt{L}}_0$ (low-swap-zero), $\tilde{\mathtt{L}}_1$ (low-swap-one), $\tilde{\mathtt{H}}_0$ (high-swap-zero), and $\tilde{\mathtt{H}}_1$ (high-swap-one) are duals of $\mathtt{H}_0$, $\mathtt{H}_1$, $\mathtt{L}_0$, and $\mathtt{L}_1$ respectively.

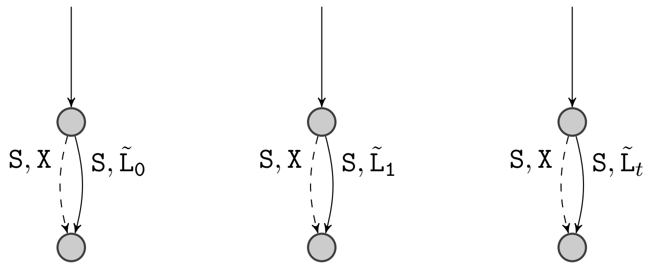


Figure 6.8 Low-swap-zero, low-swap-one and low-swap-t nodes.

Figures 6.5, 6.6, 6.7, and 6.8 illustrate these reductions. In addition, we allow edges to be *complemented* (Brace et al., 1990; S. Minato et al., 1990).

To make this more precise, we recursively define the boolean function $F^n_{\langle\kappa,c,p\rangle} : \mathbb{B}^n \to \mathbb{B}$ encoded by an CESRBDD edge $\langle\kappa,c,p\rangle$ with respect to a level $n \in \{0,...,L\}$, subject to $l(p) \leq n$, as

$$F^n_{\langle\kappa,c,p\rangle}(x_{1:n}) = \begin{cases} \text{if } l(p) = n \land n = 0, & c \\ \text{if } l(p) = n \land n > 0, & c \oplus F^{n-1}_{p[x_n]}(x_{1:n-1}) \\ \text{if } l(p) < n \land \kappa = \mathtt{X}, & F^{n-1}_{\langle\kappa,c,p\rangle}(x_{1:n-1}) \\ \text{if } l(p) < n \land \kappa = \mathtt{H}_t, & x_n \; ? \; \mathtt{t} \; : \; F^{n-1}_{\langle\kappa,c,p\rangle}(x_{1:n-1}) \\ \text{if } l(p) < n \land \kappa = \mathtt{L}_t, & x_n \; ? \; F^{n-1}_{\langle\kappa,c,p\rangle}(x_{1:n-1}) \; : \; \mathtt{t} \\ \text{if } l(p)+1 < n \land \kappa = \tilde{\mathtt{H}}_t, & x_n \; ? \; F^{n-1}_{\langle\mathtt{X},c,p\rangle}(x_{1:n-1}) \; : \; F^{n-1}_{\langle\kappa,c,p\rangle}(x_{1:n-1}) \\ \text{if } l(p)+1 = n \land \kappa = \tilde{\mathtt{H}}_t, & x_n \; ? \; F^{n-1}_{\langle\mathtt{S},c,p\rangle}(x_{1:n-1}) \; : \; \mathtt{t} \\ \text{if } l(p)+1 < n \land \kappa = \tilde{\mathtt{L}}_t, & x_n \; ? \; F^{n-1}_{\langle\kappa,c,p\rangle}(x_{1:n-1}) \; : \; F^{n-1}_{\langle\mathtt{X},c,p\rangle}(x_{1:n-1}) \\ \text{if } l(p)+1 = n \land \kappa = \tilde{\mathtt{L}}_t, & x_n \; ? \; \mathtt{t} \; : \; F^{n-1}_{\langle\mathtt{S},c,p\rangle}(x_{1:n-1}) \end{cases}$$

where $(x_{1:n})$ is a shorthand for the tuple $(x_1, ..., x_n)$, $\mathtt{t}$ is a boolean ($\mathtt{0}$ and $\mathtt{1}$ representing *false* and *true* respectively), and the if-then-else operator $(x_n)?f_1{:}f_0$ is a shorthand for $(\neg x_n \land f_0) \lor (x_n \land f_1)$.

We defined an CESRBDD as a directed acyclic graph, so it can potentially have multiple *roots* (nodes with no incoming edges). However, since our focus is on the size of the encoding for a given function, we assume from now on that our CESRBDDs have a single root. Furthermore, we defined the function encoded by an edge $\langle\kappa,c,p\rangle$ with respect to a level n, because the function encoded by a node p would be well-defined only with respect to level $l(p)$, not levels above that. Since this applies also when p is the root node and we are interested in discussing functions over L booleans, we assume a *dangling edge* $\langle\kappa^\star,c^\star,p^\star\rangle$, which encodes the function $F^L\langle\kappa^\star,c^\star,p^\star\rangle$; this function is independent of $\kappa^\star$ only if $l(p^\star) = L$, in which case we require $\kappa^\star = \mathtt{S}$. Finally, we will informally say "CESRBDD $\langle\kappa^\star,c^\star,p^\star\rangle$" to mean "the CESRBDD whose nodes are the nodes reachable from root node $p^\star$, pointed to by a dangling edge with rule $\langle\kappa^\star,c^\star\rangle$".

6.2.1 Equivalence relationships

Based on the definition of CESRBDDs we observe the following equivalences.

- The constant function `1` can be represented in multiple ways:

$$F^n_{\langle\mathtt{X},\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{L}_1,\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{H}_1,\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\tilde{\mathtt{L}}_1,\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\tilde{\mathtt{H}}_1,\mathtt{1},\mathbf{\Omega}\rangle} \equiv \mathbf{1} \tag{6.1}$$

- The constant function `0` can be represented in multiple ways:

$$F^n_{\langle\mathtt{X},\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{L}_0,\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{H}_0,\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\tilde{\mathtt{L}}_0,\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^n_{\langle\tilde{\mathtt{H}}_0,\mathtt{0},\mathbf{\Omega}\rangle} \equiv \mathbf{0} \tag{6.2}$$

- There are 2^{2^n} functions over n boolean variables. Therefore there are four functions over the lowest variable in a CESRBDD. The functions `1` and `0` are covered in equations 6.1 and 6.2. The remaining are covered by equations 6.3 and 6.4 below.

$$F^1_{\langle\mathtt{L}_0,\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^1_{\langle\mathtt{H}_1,\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^1_{\langle\tilde{\mathtt{L}}_0,\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^1_{\langle\tilde{\mathtt{H}}_1,\mathtt{0},\mathbf{\Omega}\rangle} \tag{6.3}$$

and,

$$F^1_{\langle\mathtt{L}_1,\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^1_{\langle\mathtt{H}_0,\mathtt{1},\mathbf{\Omega}\rangle} \equiv F^1_{\langle\tilde{\mathtt{L}}_1,\mathtt{0},\mathbf{\Omega}\rangle} \equiv F^1_{\langle\tilde{\mathtt{H}}_0,\mathtt{1},\mathbf{\Omega}\rangle} \tag{6.4}$$

- We also observe that all long edges to $\mathbf{\Omega}$ labelled with $\tilde{\mathtt{L}}_0$, $\tilde{\mathtt{L}}_1$, $\tilde{\mathtt{H}}_0$, and $\tilde{\mathtt{H}}_1$ have equivalents in terms of $\mathtt{X}$, $\mathtt{L}_0$, $\mathtt{L}_1$, $\mathtt{H}_0$, and $\mathtt{H}_1$.

$$F^n_{\langle\tilde{\mathtt{L}}_0,0,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{X},0,\mathbf{\Omega}\rangle} \tag{6.5}$$

$$F^n_{\langle\tilde{\mathtt{L}}_0,1,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{L}_1,0,\mathbf{\Omega}\rangle} \tag{6.6}$$

$$F^n_{\langle\tilde{\mathtt{L}}_1,0,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{L}_0,1,\mathbf{\Omega}\rangle} \tag{6.7}$$

$$F^n_{\langle\tilde{\mathtt{L}}_1,1,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{X},1,\mathbf{\Omega}\rangle} \tag{6.8}$$

$$F^n_{\langle\tilde{\mathtt{H}}_0,0,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{X},0,\mathbf{\Omega}\rangle} \tag{6.9}$$

$$F^n_{\langle\tilde{\mathtt{H}}_0,1,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{H}_1,0,\mathbf{\Omega}\rangle} \tag{6.10}$$

$$F^n_{\langle\tilde{\mathtt{H}}_1,0,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{H}_0,1,\mathbf{\Omega}\rangle} \tag{6.11}$$

$$F^n_{\langle\tilde{\mathtt{H}}_1,1,\mathbf{\Omega}\rangle} \equiv F^n_{\langle\mathtt{X},1,\mathbf{\Omega}\rangle} \tag{6.12}$$

- If a *swap* edge skips a single level n, we observe that there is an equivalent non-swap edge for it.

$$F^n_{\langle\tilde{\mathtt{L}}_0,c,p\rangle}(x_{n-1:n}) \equiv F^n_{\langle\mathtt{H}_0,c,p\rangle}(x_{n-1:n}) \tag{6.13}$$

$$F^n_{\langle\tilde{\mathtt{L}}_1,c,p\rangle}(x_{n-1:n}) \equiv F^n_{\langle\mathtt{H}_1,c,p\rangle}(x_{n-1:n}) \tag{6.14}$$

$$F^n_{\langle\tilde{\mathtt{H}}_0,c,p\rangle}(x_{n-1:n}) \equiv F^n_{\langle\mathtt{L}_0,c,p\rangle}(x_{n-1:n}) \tag{6.15}$$

$$F^n_{\langle\tilde{\mathtt{H}}_1,c,p\rangle}(x_{n-1:n}) \equiv F^n_{\langle\mathtt{L}_1,c,p\rangle}(x_{n-1:n}) \tag{6.16}$$

- The complement of a function represented as a CESRBDD can be constructed by complementing any edges to the terminal node, i.e. any edge $\langle\kappa,c,\mathbf{\Omega}\rangle$ is replaced with $\langle\kappa,\neg c,\mathbf{\Omega}\rangle$. Applying this to the patterns discussed so far, produces the following equivalences.

$$\neg F^n_{\langle \mathtt{X},c,p\rangle} \equiv F^n_{\langle \mathtt{X},\neg c,p\rangle} \tag{6.17}$$

$$\neg F^n_{\langle \mathtt{L}_0,c,p\rangle} \equiv F^n_{\langle \mathtt{L}_1,\neg c,p\rangle} \tag{6.18}$$

$$\neg F^n_{\langle \mathtt{L}_1,c,p\rangle} \equiv F^n_{\langle \mathtt{L}_0,\neg c,p\rangle} \tag{6.19}$$

$$\neg F^n_{\langle \mathtt{H}_0,c,p\rangle} \equiv F^n_{\langle \mathtt{H}_1,\neg c,p\rangle} \tag{6.20}$$

$$\neg F^n_{\langle \mathtt{H}_1,c,p\rangle} \equiv F^n_{\langle \mathtt{H}_0,\neg c,p\rangle} \tag{6.21}$$

$$\neg F^n_{\langle \tilde{\mathtt{L}}_0,c,p\rangle} \equiv F^n_{\langle \tilde{\mathtt{L}}_1,\neg c,p\rangle} \tag{6.22}$$

$$\neg F^n_{\langle \tilde{\mathtt{L}}_1,c,p\rangle} \equiv F^n_{\langle \tilde{\mathtt{L}}_0,\neg c,p\rangle} \tag{6.23}$$

$$\neg F^n_{\langle \tilde{\mathtt{H}}_0,c,p\rangle} \equiv F^n_{\langle \tilde{\mathtt{H}}_1,\neg c,p\rangle} \tag{6.24}$$

$$\neg F^n_{\langle \tilde{\mathtt{H}}_1,c,p\rangle} \equiv F^n_{\langle \tilde{\mathtt{H}}_0,\neg c,p\rangle} \tag{6.25}$$

6.3 Reduced CESRBDDs

Before we define reduced CESRBDDs and show that they are canonical, we need some more terminology. We say that a CESRBDD nonterminal node q is:

normalized if $q[0].c = \mathtt{0}$.

Normalization of nodes is a standard technique discussed in Brace et al. (1990) and S. Minato et al. (1990) to canonicalize complement edges. A normalized node cannot have a low edge that is complemented, and a node that is not normalized (or *unnormalized*) must be normalized before it can be used in a canonical representation. Normalization of a node q is performed by:

1. Replacing edge $q[0]$ with its complement (see equations 6.17–6.25).
2. Replacing edge $q[1]$ with its complement.
3. Replacing any edge $\langle\kappa,c,q\rangle$ with edge $\langle\kappa,\neg c,q\rangle$.

From here on, we assume that a node is normalized, i.e. $q[0].c = \mathtt{0}$, before it is reduced.

duplicate if there exists a node p such that $l(p) = l(q)$, $p[0] = q[0]$, and $p[1] = q[1]$.

If q is a duplicate of p, then q can be eliminated and any edge $\langle\kappa, c, q\rangle$ can be replaced with $\langle\kappa, c, p\rangle$.

redundant if $q[0] = q[1] = \langle\kappa, \mathtt{0}, p\rangle$, with $\kappa \in \{\mathtt{S}, \mathtt{X}\}$.

If q is redundant it can be eliminated and replaced with the edge $\langle\mathtt{X}, \mathtt{0}, p\rangle$.

low-zero if $q[0] = \langle\{\mathtt{S}, \mathtt{X}\}, \mathtt{0}, \mathbf{\Omega}\rangle$, and $q[1] = \langle\{\mathtt{S}, \mathtt{L_0}\}, c, p\rangle$.

If q is low-zero it can be eliminated and replaced the edge $\langle\mathtt{L_0}, c, p\rangle$.

low-one if $q[0] = \langle\{\mathtt{S}, \mathtt{X}\}, \mathtt{1}, \mathbf{\Omega}\rangle$, and $q[1] = \langle\{\mathtt{S}, \mathtt{L_1}\}, c, p\rangle$.

If q is low-one it can be eliminated and replaced with the edge $\langle\mathtt{L_1}, c, p\rangle$. Note that a normalized node cannot also be a low-one node. But edges labelled $\mathtt{L_1}$ can be produced when $\mathtt{L_0}$ edges are complemented during node normalization.

high-zero if

1. $q[0] = \langle\{\mathtt{S}, \mathtt{H_0}\}, \mathtt{0}, p\rangle$, and $q[1] = \langle\mathtt{X}, \mathtt{0}, \mathbf{\Omega}\rangle$, or,
2. $l(p) = 2$, $q[0] = \langle\mathtt{L_1}, \mathtt{0}, \mathbf{\Omega}\rangle$, and $q[1] = \langle\mathtt{X}, \mathtt{0}, \mathbf{\Omega}\rangle$.

If q satisfies the first rule, it can be eliminated and replaced with the edge $\langle\mathtt{H_0}, \mathtt{0}, p\rangle$. If q is satisfies the second rule, it can be eliminated and replaced with the edge $\langle\mathtt{H_0}, \mathtt{1}, \mathbf{\Omega}\rangle$.

The first rules ensures that edges labelled $\mathtt{H_0}$ are not produced for edges that have equivalent $\mathtt{X}$ and $\mathtt{L_1}$ edges. Since the first rule eliminates edges labelled $\mathtt{H_0}$ over the bottom-most variable, the second rule (written using a $\mathtt{L_1}$ edge that is equivalent to $\langle\mathtt{H_0}, \mathtt{1}, \mathbf{\Omega}\rangle$) is needed to construct edges labelled $\mathtt{H_0}$ that skip the lowest n variables where $n > 1$.

high-one if

1. $q[0] = \langle\{\mathtt{S}, \mathtt{H_1}\}, \mathtt{0}, p\rangle$, and $q[1] = \langle\mathtt{X}, \mathtt{1}, \mathbf{\Omega}\rangle$, or,
2. $l(p) = 2$, $q[0] = \langle\mathtt{L_0}, \mathtt{1}, \mathbf{\Omega}\rangle$, and $q[1] = \langle\mathtt{X}, \mathtt{1}, \mathbf{\Omega}\rangle$.

If q satisfies the first rule, it can be eliminated and replaced with the edge $\langle \mathtt{H}_1,0,p\rangle$. If q is satisfies the second rule, it can be eliminated and replaced with the edge $\langle \mathtt{H}_1,0,\mathbf{\Omega}\rangle$.

The first rules ensures that edges labelled $\mathtt{H}_1$ are not produced for edges that have equivalent $\mathtt{X}$ and $\mathtt{L}_0$ edges. Since the first rule eliminates edges labelled $\mathtt{H}_1$ over the bottom-most variable, the second rule (written using a $\mathtt{L}_0$ edge that is equivalent to $\langle \mathtt{H}_1,0,\mathbf{\Omega}\rangle$) is needed to construct edges labelled $\mathtt{H}_1$ that skip the lowest n variables where $n > 1$.

low-swap-one if

1. $q[0] = \langle \mathtt{X},0,p\rangle$, $q[1] = \langle \tilde{\mathtt{L}}_1,0,p\rangle$, and $p \neq \mathbf{\Omega}$, or,
2. $q[0] = \langle \mathtt{X},0,p\rangle$, $q[1] = \langle \mathtt{H}_1,0,p\rangle$, $l(q) - l(p) = 2$, and $p \neq \mathbf{\Omega}$.

If q satisfies one of these rules, it can be eliminated and replaced with the edge $\langle \tilde{\mathtt{L}}_1,0,p\rangle$. Note that p cannot be terminal node $\mathbf{\Omega}$.

These rules ensure that edges labelled $\tilde{\mathtt{L}}_1$ are not produced for edges that have equivalent $\mathtt{X}$, $\mathtt{L}_t$, $\mathtt{H}_t$ edges, including edges over the lowest two variables as well as edges that skip a single level. The second rule (written using a $\mathtt{H}_1$ edge that is equivalent to $\langle \tilde{\mathtt{L}}_1,0,p\rangle$) is needed to construct edges labelled $\tilde{\mathtt{L}}_1$ that skip the lowest n variables where $n > 2$, and for edges labelled $\tilde{\mathtt{L}}_1$ that skips more than one level.

low-swap-zero if

1. $q[0] = \langle \mathtt{X},0,p\rangle$, $q[1] = \langle \tilde{\mathtt{L}}_0,0,p\rangle$, and $p \neq \mathbf{\Omega}$, or,
2. $q[0] = \langle \mathtt{X},0,p\rangle$, $q[1] = \langle \mathtt{H}_0,0,p\rangle$, $l(q) - l(p) = 2$, and $p \neq \mathbf{\Omega}$.

If q satisfies one of these rules, it can be eliminated and replaced with the edge $\langle \tilde{\mathtt{L}}_0,0,p\rangle$. Note that p cannot be terminal node $\mathbf{\Omega}$.

These rules ensure that edges labelled $\tilde{\mathtt{L}}_0$ are not produced for edges that have equivalent $\mathtt{X}$, $\mathtt{L}_t$, $\mathtt{H}_t$ edges, including edges over the lowest two variables as well as edges that skip a single level. The second rule (written using a $\mathtt{H}_0$ edge that is equivalent to $\langle \tilde{\mathtt{L}}_0,0,p\rangle$) is needed to construct edges labelled $\tilde{\mathtt{L}}_0$ that skip the lowest n variables where $n > 2$, and for edges labelled $\tilde{\mathtt{L}}_0$ that skips more than one level.

high-swap-one if

1. $q[0] = \langle\tilde{\mathtt{H}}_1,\mathtt{0},p\rangle$, and $q[1] = \langle\mathtt{X},\mathtt{0},p\rangle$, or,
2. $q[0] = \langle\mathtt{L}_1,\mathtt{0},p\rangle$, $q[1] = \langle\mathtt{X},\mathtt{0},p\rangle$, $l(q) - l(p) = 2$, and $p \neq \mathbf{\Omega}$.

If q satisfies one of these rules, it can be eliminated and replaced with the edge $\langle\tilde{\mathtt{H}}_1,\mathtt{0},p\rangle$. Note that p cannot be terminal node $\mathbf{\Omega}$.

These rules ensure that edges labelled $\tilde{\mathtt{H}}_1$ are not produced for edges that have equivalent $\mathtt{X}$, $\mathtt{L}_t$, $\mathtt{H}_t$ edges, including edges over the lowest two variables as well as edges that skip a single level. The second rule (written using a $\mathtt{L}_1$ edge that is equivalent to $\langle\tilde{\mathtt{H}}_1,\mathtt{0},p\rangle$) is needed to construct edges labelled $\tilde{\mathtt{H}}_1$ that skip the lowest n variables where $n > 2$, and for edges labelled $\tilde{\mathtt{H}}_1$ that skips more than one level.

high-swap-zero if

1. $q[0] = \langle\tilde{\mathtt{H}}_0,\mathtt{0},p\rangle$, and $q[1] = \langle\mathtt{X},\mathtt{0},p\rangle$, or,
2. $q[0] = \langle\mathtt{L}_0,\mathtt{0},p\rangle$, $q[1] = \langle\mathtt{X},\mathtt{0},p\rangle$, $l(q) - l(p) = 2$, and $p \neq \mathbf{\Omega}$.

If q satisfies one of these rules, it can be eliminated and replaced with the edge $\langle\tilde{\mathtt{H}}_0,\mathtt{0},p\rangle$. Note that p cannot be terminal node $\mathbf{\Omega}$.

These rules ensure that edges labelled $\tilde{\mathtt{H}}_0$ are not produced for edges that have equivalent $\mathtt{X}$, $\mathtt{L}_t$, $\mathtt{H}_t$ edges, including edges over the lowest two variables as well as edges that skip a single level. The second rule (written using a $\mathtt{L}_0$ edge that is equivalent to $\langle\tilde{\mathtt{H}}_0,\mathtt{0},p\rangle$) is needed to construct edges labelled $\tilde{\mathtt{H}}_0$ that skip the lowest n variables where $n > 2$, and for edges labelled $\tilde{\mathtt{H}}_0$ that skips more than one level.

We say that a CESRBDD edge e is a *one-level-skip-edge* if it:

- starts from node q and ends at node p such that, $l(q) - l(p) = 2$, or

- is a long edge in a CESRBDD with a single level.

Definition 6.3.1

A CESRBDD is *reduced* if the following restrictions hold:

R1. There are no unnormalized nodes.

R2. There are no duplicate nodes.

R3. There are no redundant nodes.

R4. There are no low-zero nodes.

R5. There are no low-one nodes.

R6. There are no high-zero nodes.

R7. There are no high-one nodes.

R8. There are no low-swap-zero nodes.

R9. There are no low-swap-one nodes.

R10. There are no high-swap-zero nodes.

R11. There are no high-swap-one nodes.

R12. For any edge $e = \langle \kappa, \mathtt{0}, \mathbf{\Omega} \rangle$, $\kappa \notin \{\mathtt{L_0}, \mathtt{H_0}, \tilde{\mathtt{L}}_\mathtt{0}, \tilde{\mathtt{H}}_\mathtt{0}\}$.

R13. For any edge $e = \langle \kappa, \mathtt{1}, \mathbf{\Omega} \rangle$, $\kappa \notin \{\mathtt{L_1}, \mathtt{H_1}, \tilde{\mathtt{L}}_\mathtt{1}, \tilde{\mathtt{H}}_\mathtt{1}\}$.

R14. For any edge $e = \langle \kappa, c, \mathbf{\Omega} \rangle$, $\kappa \notin \{\tilde{\mathtt{L}}_\mathtt{0}, \tilde{\mathtt{L}}_\mathtt{1}, \tilde{\mathtt{H}}_\mathtt{0}, \tilde{\mathtt{H}}_\mathtt{1}\}$.

R15. For any one-level-skip-edge $e = \langle \kappa, c, p \rangle$, $\kappa \notin \{\tilde{\mathtt{H}}_\mathtt{0}, \tilde{\mathtt{H}}_\mathtt{1}, \tilde{\mathtt{L}}_\mathtt{0}, \tilde{\mathtt{L}}_\mathtt{1}\}$.

R16. For any one-level-skip-edge $e = \langle \kappa, c, \mathbf{\Omega} \rangle$, $\kappa \notin \{\mathtt{H_0}, \mathtt{H_1}\}$.

We observe that the restrictions **R12** and **R13** force long edges that represent $F^n_{\langle \kappa, c, p \rangle} \equiv \mathbf{0}$ and $F^n_{\langle \kappa, c, p \rangle} \equiv \mathbf{1}$ to be of the form $\langle \mathtt{X}, \mathtt{0}, \mathbf{\Omega} \rangle$ and $\langle \mathtt{X}, \mathtt{1}, \mathbf{\Omega} \rangle$, respectively. Based on equations 6.1 and 6.2 we know that are multiple ways to represent **1** and **0** respectively. But we must assign one reduction for such long edges if we want to enforce canonicity, and we choose X. We can also choose any of the other equivalent representations and rewrite any dependent rules accordingly.

Restriction **R14** ensures that there are no *swap* edges to $\mathbf{\Omega}$. This is also to enforce canonicity, since equations 6.5–6.12 show that all such edges can also be represented using $\mathtt{X}$, $\mathtt{L_0}$, $\mathtt{L_1}$, $\mathtt{H_0}$, or $\mathtt{H_1}$ edges. Again, our choice of replacing these swap edges with non-swap edges is arbitrary and has no impact on which nodes are reducible, since swap nodes have been defined with respect to swap edges as well as non-swap edges.

Restriction **R15** ensures that there are no one-level-skip-edges labelled with $\tilde{\mathtt{L}}_1$, $\tilde{\mathtt{L}}_0$, $\tilde{\mathtt{H}}_1$ or $\tilde{\mathtt{H}}_0$, since (as shown in equations 6.13–6.16) they can be represented by equivalent $\mathtt{H_1}$, $\mathtt{H_0}$, $\mathtt{L_1}$ and $\mathtt{L_0}$ edges respectively.

Restriction **R16** ensures that there are no one-level-skip-edges labelled with $\mathtt{H_1}$ or $\mathtt{H_0}$ that skip over the bottom level, since (as shown in equations 6.3 and 6.4 respectively) they can be represented by equivalent $\mathtt{L_0}$ and $\mathtt{L_1}$ edges respectively.

We note that there are implied restrictions in the node definitions: redundant nodes are defined in terms of $\mathtt{S}$ and $\mathtt{X}$ edges; low-t nodes are defined in terms of $\mathtt{S}$, $\mathtt{X}$ and $\mathtt{L}_t$ edges; high-t nodes are defined in terms of $\mathtt{S}$, $\mathtt{X}$, $\mathtt{L}_t$ and $\mathtt{H}_t$ edges; and finally, swap nodes are defined in terms of their respective swap edges in addition to $\mathtt{S}$, $\mathtt{X}$, $\mathtt{L}_t$ and $\mathtt{H}_t$ edges. These implied restrictions ensure canonicity by eliminating the possibility of multiple reduction rules being applicable to the same node. At the same time, they are not biased towards any particular reduction rule since the hierarchy of reduction rules only impacts nodes that fit more than one pattern, and in every such case the node is eliminated and the representative edge is labelled based on the implicit rule hierarchy.

6.3.1 Reducing a CESRBDD

A CESRBDD can be converted into a reduced CESRBDD using Algorithm 6.1, which reduces the CESRBDD by traversing it in depth-first order, recursively reducing edges to child nodes before reducing the parent node.

The procedure ReduceEdge reduces an edge $\langle \kappa, c, p \rangle$ with respect to level n as follows:

- Reduces node p using the procedure ReduceNode. This produces the edge $\langle \kappa', c', p' \rangle$.

Algorithm 6.1 Reduce a CESRBDD.

```
1: procedure REDUCE(CESRBDD $\langle\kappa^\star,c^\star,p^\star\rangle$)
2:     return REDUCEEDGE($n$, $\langle\kappa^\star,c^\star,p^\star\rangle$)

3: procedure REDUCEEDGE(Level $n$, Edge $\langle\kappa,c,p\rangle$)
4:     $\langle\kappa',c',p'\rangle \leftarrow$ REDUCENODE($p$);
5:     if $c = \mathtt{1}$ then $\langle\kappa',c',p'\rangle \leftarrow$ COMPLEMENTEDGE($\langle\kappa',c',p'\rangle$);
6:     $\langle\kappa',c',p'\rangle \leftarrow$ MERGEEDGE($n$, $\kappa$, $l(p)$, $\langle\kappa',c',p'\rangle$);
7:     return $\langle\kappa',c',p'\rangle$;
```

- If p is a complement node (i.e. $c = \mathtt{1}$), REDUCEEDGE complements $\langle\kappa',c',p'\rangle$ (the edge representing p) using procedure COMPLEMENTEDGE.

- If $\kappa \neq \kappa'$, an intermediate node may need to be created since we cannot have different reduction rules on the same edge. Procedure MERGEEDGE checks for compatibility between κ and κ' and merges them into a single edge if possible. Otherwise, it creates a node at level $l(p) + 1$ representing the reduction rule κ, and returns an edge with reduction rule κ. Merge also checks if merged edge violates any of the restrictions **R12**–**R16**, and if so converts the edge into an equivalent edge that does not violate these restrictions (see Definition 6.3)

Algorithm 6.2 Reduce a CESRBDD node.

```
 1: procedure ReduceNode(Node $p$)
 2:     if $p = \mathbf{\Omega}$ then return $\langle \mathtt{S},0,\mathbf{\Omega}\rangle$;
 3:     if Cache contains “Reduce, $p$, $\langle \kappa',c',p'\rangle$” then return $\langle \kappa',c',p'\rangle$;
 4:     $p' \leftarrow$ empty node at $l(p)$;
 5:     $c' \leftarrow \mathtt{0}$;
 6:     for all $i \in \{0,1\}$ do $p'[i] \leftarrow$ ReduceEdge($l(p)$, $p[i]$);
 7:     if $p'$ is not a normalized node then
 8:         $c' \leftarrow \mathtt{1}$;
 9:         for all $i \in \{0,1\}$ do $p'[i] \leftarrow$ ComplementEdge($p'[i]$);
10:     if $p'$ is a redundant node then
11:         $e' \leftarrow p'[0]$; $e'.r \leftarrow \mathtt{X}$;
12:     else if $p'$ is a low-zero node then
13:         $e' \leftarrow p'[1]$; $e'.r \leftarrow \mathtt{L_0}$;
14:     else if $p'$ is a high-zero node then
15:         $e' \leftarrow p'[0]$; $e'.r \leftarrow \mathtt{H_0}$;
16:     else if $p'$ is a high-one node then
17:         $e' \leftarrow p'[0]$; $e'.r \leftarrow \mathtt{H_1}$;
18:     else if $p'$ is a low-zero-swap node then
19:         $e' \leftarrow p'[1]$; $e'.r \leftarrow \tilde{\mathtt{L}}_\mathtt{0}$;
20:     else if $p'$ is a low-one-swap node then
21:         $e' \leftarrow p'[1]$; $e'.r \leftarrow \tilde{\mathtt{L}}_\mathtt{1}$;
22:     else if $p'$ is a high-zero-swap node then
23:         $e' \leftarrow p'[0]$; $e'.r \leftarrow \tilde{\mathtt{H}}_\mathtt{0}$;
24:     else if $p'$ is a high-one-swap node then
25:         $e' \leftarrow p'[0]$; $e'.r \leftarrow \tilde{\mathtt{H}}_\mathtt{1}$;
26:     else if $p'$ is a duplicate of $q$ then
27:         $e' \leftarrow \langle \mathtt{S},0,q\rangle$;
28:     else
29:         $e' \leftarrow \langle \mathtt{S},0,p'\rangle$;
30:     if $c' = \mathtt{1}$ then $e' \leftarrow$ ComplementEdge($e'$);
31:     Save “Reduce, $p$, $e'$” in Cache;
32:     return $e'$;
```

Algorithm 6.3 Merge a CESRBDD edge with an incoming rule.

```
1:  procedure MergeEdge(Level $\hat{n}$, Rule $\kappa$, Level $n$, Edge $e$)
2:      if $\hat{n} = n \vee \kappa = \mathtt{S} \vee \kappa = e.r$ then
3:          $e' \leftarrow e$;
4:      else if $e.r = \mathtt{S}$ then
5:          $e' \leftarrow e$;
6:          $e'.r \leftarrow \kappa$;
7:      else                                        $\bullet\ \kappa \neq \mathtt{S} \wedge \kappa \neq \kappa' \wedge \kappa' \neq \mathtt{S}$
8:          $e' \leftarrow$ BuildPattern($n + 1$, $\kappa$, $e$);
9:          $e'.r \leftarrow \kappa$;
10:     if $e'$ violates restriction R12 then
11:         $e' \leftarrow \langle \mathtt{X},\mathtt{0},\mathbf{\Omega} \rangle$;
12:     if $e'$ violates restriction R13 then
13:         $e' \leftarrow \langle \mathtt{X},\mathtt{1},\mathbf{\Omega} \rangle$;
14:     if $e'$ violates restrictions R14 or R15 then
15:         Convert $e'$ to non-swap edge;
16:     if $e'$ violates restriction R16 then
17:         Convert $e'$ to equivalent $\mathtt{L}_t$ edge;
18:     if $e'$ is not a long edge then $e'.r \leftarrow \mathtt{S}$;
19:     return $e'$;
```

Algorithm 6.4 Build CESRBDD node at given level and according to given pattern.

```
procedure BuildPattern(Level $n$, Pattern $\hat{\kappa}$, CESRBDD $\langle\kappa,c,p\rangle$)
    /* Require: $\hat{\kappa} \neq \mathtt{S} \wedge \hat{\kappa} \neq \kappa$ */
    $p' \leftarrow$ new node at $n$;
    if $\hat{\kappa} = \mathtt{X}$ then
        $p'[0] \leftarrow \langle\kappa,c,p\rangle$; $p'[1] \leftarrow \langle\kappa,c,p\rangle$;
    else if $\hat{\kappa} = \mathtt{L}_0$ then
        $p'[0] \leftarrow \langle\mathtt{X},\mathtt{0},\mathbf{\Omega}\rangle$; $p'[1] \leftarrow \langle\kappa,c,p\rangle$;
    else if $\hat{\kappa} = \mathtt{L}_1$ then
        $p'[0] \leftarrow \langle\mathtt{X},\mathtt{1},\mathbf{\Omega}\rangle$; $p'[1] \leftarrow \langle\kappa,c,p\rangle$;
    else if $\hat{\kappa} = \mathtt{H}_0$ then
        $p'[0] \leftarrow \langle\kappa,c,p\rangle$; $p'[1] \leftarrow \langle\mathtt{X},\mathtt{0},\mathbf{\Omega}\rangle$;
    else if $\hat{\kappa} = \mathtt{H}_1$ then
        $p'[0] \leftarrow \langle\kappa,c,p\rangle$; $p'[1] \leftarrow \langle\mathtt{X},\mathtt{1},\mathbf{\Omega}\rangle$;
    else if $\hat{\kappa} = \tilde{\mathtt{L}}_0$ then
        $p'[0] \leftarrow \langle\mathtt{X},c,p\rangle$; $p'[1] \leftarrow \langle\kappa,c,p\rangle$;
    else if $\hat{\kappa} = \tilde{\mathtt{L}}_1$ then
        $p'[0] \leftarrow \langle\mathtt{X},c,p\rangle$; $p'[1] \leftarrow \langle\kappa,c,p\rangle$;
    else if $\hat{\kappa} = \tilde{\mathtt{H}}_0$ then
        $p'[0] \leftarrow \langle\kappa,c,p\rangle$; $p'[1] \leftarrow \langle\mathtt{X},c,p\rangle$;
    else                                                        • $\hat{\kappa} = \tilde{\mathtt{H}}_1$
        $p'[0] \leftarrow \langle\kappa,c,p\rangle$; $p'[1] \leftarrow \langle\mathtt{X},c,p\rangle$;
    $c' \leftarrow \mathtt{0}$
    if $p'$ is a not a normalized node then
        $c' \leftarrow \mathtt{1}$
        for all $i \in \{0,1\}$ do $p'[i] \leftarrow$ ComplementEdge($p'[i]$);
    if $p'$ is a duplicate of $q$ then $p' \leftarrow q$;
    return $\langle\mathtt{S},c',p'\rangle$;
```

Algorithm 6.5 Complement a CESRBDD edge.

```
procedure ComplementEdge(Edge $\langle\kappa,c,p\rangle$)
    if $\kappa = \mathtt{L}_0$ then $\kappa' \leftarrow \mathtt{L}_1$;
    else if $\kappa = \mathtt{L}_1$ then $\kappa' \leftarrow \mathtt{L}_0$;
    else if $\kappa = \mathtt{H}_0$ then $\kappa' \leftarrow \mathtt{H}_1$;
    else if $\kappa = \mathtt{H}_1$ then $\kappa' \leftarrow \mathtt{H}_0$;
    else if $\kappa = \tilde{\mathtt{L}}_0$ then $\kappa' \leftarrow \tilde{\mathtt{L}}_1$;
    else if $\kappa = \tilde{\mathtt{L}}_1$ then $\kappa' \leftarrow \tilde{\mathtt{L}}_0$;
    else if $\kappa = \tilde{\mathtt{H}}_0$ then $\kappa' \leftarrow \tilde{\mathtt{H}}_1$;
    else if $\kappa = \tilde{\mathtt{H}}_1$ then $\kappa' \leftarrow \tilde{\mathtt{H}}_0$;
    else $\kappa' \leftarrow \kappa$;
    return $\langle\kappa',\neg c,p\rangle$
```

The procedure ReduceNode reduces a node p as follows:

1. Reduce each child edge of p.
2. If p is not normalized, complement its edges.
3. If p satisfies one of reduction rules κ', create an edge labelled $\langle\kappa',c_i,p_i\rangle$ where c_i and p_i corresponded to the i^{th} child of p, and is determined by the reduction rule κ'.
4. If p is not reducible and if it duplicates an existing node q, use q instead of p in the steps that follow.
5. If p is not reducible, create an edge labelled $\langle\mathtt{S},0,p\rangle$.
6. If p's edges were complemented in Step 2, complement the edge representing p (edge e' in ReduceNode) before returning it.

The procedure MergeEdge combines compatible edge rules. If the rules are not compatible it builds an intermediate node using procedure BuildPattern at one level above the start of edge e (i.e. level $n+1$) using the pattern dictated by the incoming reduction rule κ. MergeEdge also ensures that any returned edge does not violate restrictions **R12**–**R16** as follows:

- It first replaces edges $\langle\mathtt{H}_0,0,\mathbf{\Omega}\rangle$, $\langle\mathtt{L}_0,0,\mathbf{\Omega}\rangle$, $\langle\tilde{\mathtt{H}}_0,0,\mathbf{\Omega}\rangle$, and $\langle\tilde{\mathtt{L}}_0,0,\mathbf{\Omega}\rangle$ with $\langle\mathtt{X},0,\mathbf{\Omega}\rangle$, to satisfy restriction R12.
- Next, it replaces edges $\langle\mathtt{H}_1,1,\mathbf{\Omega}\rangle$, $\langle\mathtt{L}_1,1,\mathbf{\Omega}\rangle$, $\langle\tilde{\mathtt{H}}_1,1,\mathbf{\Omega}\rangle$, and $\langle\tilde{\mathtt{L}}_1,1,\mathbf{\Omega}\rangle$ with $\langle\mathtt{X},1,\mathbf{\Omega}\rangle$, to satisfy restriction R13.
- Next it satisfies restriction R14 by: replacing $\langle\tilde{\mathtt{H}}_0,1,\mathbf{\Omega}\rangle$ with $\langle\mathtt{H}_1,0,\mathbf{\Omega}\rangle$, replacing $\langle\tilde{\mathtt{H}}_1,0,\mathbf{\Omega}\rangle$ with $\langle\mathtt{H}_0,1,\mathbf{\Omega}\rangle$, replacing $\langle\tilde{\mathtt{L}}_0,1,\mathbf{\Omega}\rangle$ with $\langle\mathtt{L}_1,0,\mathbf{\Omega}\rangle$, and replacing $\langle\tilde{\mathtt{L}}_1,0,\mathbf{\Omega}\rangle$ with $\langle\mathtt{L}_0,1,\mathbf{\Omega}\rangle$.
- Next, it satisfies restriction R15 by replacing swap edges that skip only one level (i.e. they originate from a node at level n and end at a node at level $n-2$) with equivalent non-swap edges: replacing $\langle\tilde{\mathtt{H}}_0,c,p\rangle$ with $\langle\mathtt{L}_0,c,p\rangle$, replacing $\langle\tilde{\mathtt{H}}_1,c,p\rangle$ with $\langle\mathtt{L}_1,c,p\rangle$, replacing $\langle\tilde{\mathtt{L}}_0,c,p\rangle$ with $\langle\mathtt{H}_0,c,p\rangle$, and replacing $\langle\tilde{\mathtt{L}}_1,c,p\rangle$ with $\langle\mathtt{H}_1,c,p\rangle$.

- Next it satisfies restriction R16 by: replacing $\langle \texttt{H}_0,1,\mathbf{\Omega}\rangle$ edges that originate from a level 2 node, with $\langle \texttt{L}_1,0,\mathbf{\Omega}\rangle$; and replacing $\langle \texttt{H}_1,0,\mathbf{\Omega}\rangle$ edges that originate from a level 2 node, with $\langle \texttt{L}_0,1,\mathbf{\Omega}\rangle$.

- Finally, it checks if the edge produced thus far is a long edge, and if it is not, sets its reduction rule to `S`.

In the above discussion, note that any edge that is replaced by the algorithm encodes the same function as its replacement, giving us the following lemma.

Lemma 6.3.1

In Algorithm 6.1, each edge replacement preserves the function encoded by the CESRBDD.

It remains to show that the algorithm always terminates and produces a reduced CESRBDD.

Lemma 6.3.2

Algorithm 6.1 terminates in $\mathcal{O}(|Nodes(p^*)|)$ steps.

Proof: The algorithm performs a depth-first traversal of the nodes that are reachable from edge $\langle \kappa^*,c^*,p^*\rangle$. The use of a compute cache guarantees that each node in $Nodes(p^*)$ is reduced at most once. Furthermore, each call to ReduceEdge may eliminate a node via ReduceNode and may create a node via MergeEdge. Any node created by MergeEdge is, by construction, not reducible. Therefore, the algorithm terminates in time linear to the number of edges in the graph rooted by p^*, which is $\mathcal{O}(|Nodes(p^*)|)$. $\square$

Theorem 6.3.1

Algorithm 6.1 converts any CESRBDD to an equivalent reduced CESRBDD in $\mathcal{O}(|Nodes(p^*)|)$ steps.

Proof: Lemma 6.3.1 establishes that Algorithm 6.1 terminates in $\mathcal{O}(|Nodes(p^*)|)$ steps. In a CESRBDD, the reduction rules for any node q are independent of the incoming edge(s) to q, and once $q[0]$ and $q[1]$ are reduced, reducing q does not change the reducibility of $q[0].p$ and $q[1].p$. Further, any node created via MergeEdge is, by construction, not reducible. MergeEdge also converts any edge $\langle\kappa,c,q\rangle$ that violates restrictions **R12–R16** into an equivalent compliant edge $\langle\kappa',c',q'\rangle$ with $q = q'$. Therefore, once ReduceEdge has been called on every edge in the CESRBDD, every node and edge complies with the restrictions in Definition 6.3, producing a reduced CESRBDD. Lemma 6.3.1 establishes that Algorithm 6.1 produces an equivalent (in terms of encoded function) CESRBDD. □

We note here that, similar to ESRBDDs, and unlike most other decision diagrams (including BDDs, ZDDs, CBDDs, CZDDs, and TDDs), a reduced CESRBDD is not necessarily a minimum size CESRBDD encoding of a function, even for a fixed variable order, as elimination of some node q during the reduction could trigger the creation of eight new nodes. An example of this is shown in Figure 6.9, where redundant node q is eliminated. Edges $\langle\mathtt{S},c,q\rangle$ and $\langle\mathtt{X},c,q\rangle$ can be simply redirected as $\langle\mathtt{X},c,p\rangle$, but the edges $\langle\kappa_i,c,q\rangle$, where $\kappa_i \in \{\mathtt{L_0},\mathtt{H_0},\mathtt{L_1},\mathtt{H_1},\tilde{\mathtt{H}}_0,\tilde{\mathtt{L}}_0,\tilde{\mathtt{H}}_1,\tilde{\mathtt{L}}_1\}$, require the creation of a new per edge, q_{κ_i}.

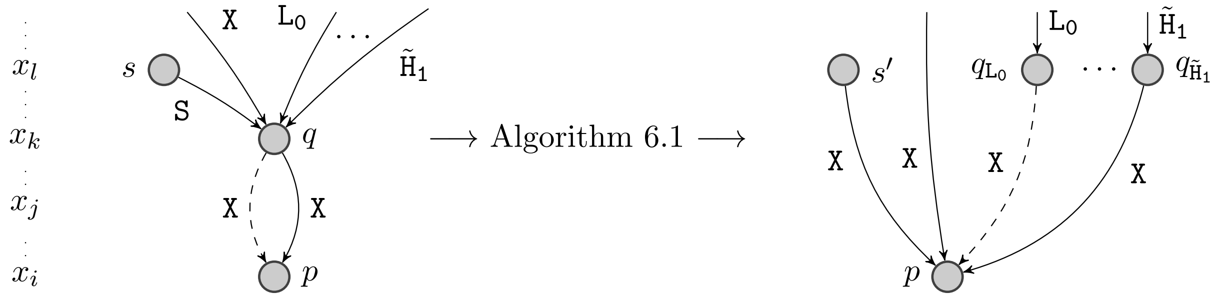


Figure 6.9 A worst-case example for CESRBDDs where elimination of node q creates 8 nodes.

6.4 Canonicity of CESRBDDs

We are now ready to discuss the *canonicity* of reduced CESRBDDs, i.e., to show that a function has a unique encoding as a reduced CESRBDD. In the following, we say that functions $F^n_{\langle\kappa,c,p\rangle}$ and $F^n_{\langle\kappa',c',p'\rangle}$ are *equivalent*, written $F^n_{\langle\kappa,c,p\rangle} \equiv F^n_{\langle\kappa',c',p'\rangle}$, if $F^n_{\langle\kappa,c,p\rangle}(x_1,\ldots,x_n) = F^n_{\langle\kappa',c',p'\rangle}(x_1,\ldots,x_n)$ for all possible inputs $(x_1,\ldots,x_n) \in \mathbb{B}^n$.

Theorem 6.4.1

In a reduced CESRBDD, for any $n \in \mathbb{N}$, for any two edges $e = \langle\kappa,c,p\rangle$, $e' = \langle\kappa',c',p'\rangle$ with $l(p) \leq n$, $l(p') \leq n$, if $F^n_e \equiv F^n_{e'}$ then (1) $p = p'$, (2) $c = c'$, and (3) if $l(p) < n$ then $\kappa = \kappa'$.

Proof: The proof is by induction on n. For the base case, we use $n = 0$ and from the definition of F we have $F^0_e \equiv F^0_{e'} \;\rightarrow\; (p = p' = \mathbf{\Omega} \wedge c = c')$.

Now, suppose the theorem holds for $n = m \geq 0$, and we will prove it holds for $n = m+1$. Regardless of $\langle\kappa,c,p\rangle$, we have

$$F^n_{\langle\kappa,c,p\rangle}(x_{1:n}) = (x_n)?f_1(x_{1:n-1}){:}f_0(x_{1:n-1})$$

for some functions f_0 and f_1. Similarly, we have

$$F^n_{\langle\kappa',c',p'\rangle}(x_{1:n}) = (x_n)?f'_1(x_{1:n-1}){:}f'_0(x_{1:n-1}).$$

It follows that $F^n_{\langle\kappa,c,p\rangle} \equiv F^n_{\langle\kappa',c',p'\rangle}$ if and only if $f_0 \equiv f'_0$ and $f_1 \equiv f'_1$.

First, suppose $l(p) = n$ and $l(p') = n$. From the definition of F, it follows that $F^{n-1}_{p[0]} \equiv F^{n-1}_{p'[0]}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{p'[1]}$. By inductive hypothesis, $p[0].p = p'[0].p$, $p[0].c = p'[0].c$, $p[1].p = p'[1].p$, and $p[1].c = p'[1].c$. If $l(p[0].p) < n-1$, then by inductive hypothesis, $p[0] = p'[0]$; otherwise, $l(p[0].p) = n-1$ and we must have $p[0].r = \texttt{S}$ and $p'[0].r = \texttt{S}$, thus either $(p[0] = p'[0] \wedge c = c')$ or $(p[0] = \neg p'[0] \wedge c = \neg c')$. By a similar argument, it follows that either $p[1] = p'[1] \wedge c = c'$ or $p[1] = \neg p'[1] \wedge c = \neg c'$. We therefore have two cases:

- $p = p'$ and $c = c'$. In which case, either p and p' are the same node and the theorem holds, or p and p' are duplicates which is impossible because of restriction R2.

- $p = \neg p'$ and $c = \neg c'$. In which case either p or p' is not a normalized node which is impossible because of restriction R1.

Next, suppose $l(p) < n$ and $l(p') < n$. If $\kappa = \kappa'$, then in all cases for F we conclude that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$ and by inductive hypothesis we have that $p = p'$ and $c = c'$, so the theorem holds. We now show that $\kappa \neq \kappa'$ is impossible, by contradiction. Consider the possible cases for $\kappa \neq \kappa'$:

1. $\kappa = \mathtt{X}$:

   (a) $\kappa' \in \{\mathtt{L}_\mathbf{0}, \mathtt{H}_\mathbf{0}\}$: From the definition of F we conclude that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$ and that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv \mathbf{0}$. Therefore, $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv \mathbf{0}$, and $F^{n-1}_{\langle\kappa',c',p'\rangle} \equiv \mathbf{0}$. By the inductive hypothesis, we have that $p = \mathbf{\Omega}$ and $c = \mathbf{0}$, and $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$. But, according to restrictions R12 if $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$ then $\kappa' \notin \{\mathtt{L}_\mathbf{0}, \mathtt{H}_\mathbf{0}, \tilde{\mathtt{L}}_\mathbf{0}, \tilde{\mathtt{H}}_\mathbf{0}\}$. contradicting our assumption that $\kappa' \in \{\mathtt{L}_\mathbf{0}, \mathtt{H}_\mathbf{0}\}$.

   (b) $\kappa' \in \{\mathtt{L}_\mathbf{1}, \mathtt{H}_\mathbf{1}\}$: From the definition of F we conclude that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$ and that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv \mathbf{1}$. Therefore, $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv \mathbf{1}$, and $F^{n-1}_{\langle\kappa',c',p'\rangle} \equiv \mathbf{1}$. By the inductive hypothesis, we have that $p = \mathbf{\Omega}$ and $c = \mathbf{1}$, and $p' = \mathbf{\Omega}$ and $c' = \mathbf{1}$. But, according to restrictions R13 if $p' = \mathbf{\Omega}$ and $c' = \mathbf{1}$ then $\kappa' \notin \{\mathtt{L}_\mathbf{1}, \mathtt{H}_\mathbf{1}, \tilde{\mathtt{L}}_\mathbf{1}, \tilde{\mathtt{H}}_\mathbf{1}\}$. contradicting our assumption that $\kappa' \in \{\mathtt{L}_\mathbf{1}, \mathtt{H}_\mathbf{1}\}$.

   (c) $\kappa' \in \{\tilde{\mathtt{L}}_\mathbf{t}, \tilde{\mathtt{H}}_\mathbf{t}\}$: From the definition of F we conclude that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$ and that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\mathtt{X},c',p'\rangle}$. If $l(p) > l(p')$, when the function reaches a level n' such that $n' = l(p)$, from the definition of $\mathtt{X}$ and the above conclusion, $F^{n'}_{p[0]} \equiv F^{n'}_{p[1]} \equiv F^{n'}_{\langle\mathtt{X},c',p'\rangle}$, violating restriction restriction R3. If $l(p) \leq l(p')$, when the function reaches a level n' such that $n' = l(p)$, $F^{n'}_{\langle\kappa,c,p\rangle} \equiv F^{n'}_{\langle\mathtt{S},c',p'\rangle}$ and $F^{n'}_{\langle\kappa,c,p\rangle} \equiv \mathbf{t}$. Therefore, $F^{n'}_{\langle\mathtt{S},c',p'\rangle} \equiv \mathbf{t}$, $p' = \mathbf{\Omega}$ and $c' = \mathbf{t}$. But, according to restriction R14, if $p' = \mathbf{\Omega}$ and $c' = \mathbf{t}$ then $\kappa' \notin \{\tilde{\mathtt{L}}_\mathbf{t}, \tilde{\mathtt{H}}_\mathbf{t}\}$. contradicting our assumption that $\kappa' \in \{\tilde{\mathtt{L}}_\mathbf{t}, \tilde{\mathtt{H}}_\mathbf{t}\}$.

2. $\kappa = \mathtt{L}_\mathbf{t}$ and $\kappa' = \mathtt{L}_{\neg\mathbf{t}}$:

   From the definition of F we conclude that $\mathbf{t} \equiv \neg\mathbf{t}$, a contradiction.

3. $\kappa = \mathtt{H}_{\mathbf{t}}$ and $\kappa' = \mathtt{H}_{\neg\mathbf{t}}$:

   Analogous to case $\kappa = \mathtt{L}_{\mathbf{t}}$ and $\kappa' = \mathtt{L}_{\neg\mathbf{t}}$.

4. $\kappa = \tilde{\mathtt{L}}_{\mathbf{t}}$ and $\kappa' = \tilde{\mathtt{L}}_{\neg\mathbf{t}}$:

   From the definition of F we conclude that $F^{n-1}_{\langle \mathtt{X},c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$ and that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \kappa',c',p'\rangle}$. If $l(p) \geq l(p')$, when the function reaches a level n' such that $n' = l(p)$, from the definition of $\tilde{\mathtt{L}}_{\mathbf{t}}$ and the above conclusion, $F^{n'}_{p[0]} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$, and $\mathbf{t} \equiv F^{n'}_{\langle \kappa',c',p'\rangle}$. But, according to restriction R14, if $p' = \mathbf{\Omega}$ and $c' = \mathbf{t}$ then $\kappa' \notin \{\tilde{\mathtt{L}}_{\mathbf{t}}, \tilde{\mathtt{H}}_{\mathbf{t}}\}$, contradicting our assumption that $\kappa' = \tilde{\mathtt{L}}_{\neg\mathbf{t}}$. The case $l(p) \leq l(p')$ is symmetric.

5. $\kappa = \tilde{\mathtt{H}}_{\mathbf{t}}$ and $\kappa' = \tilde{\mathtt{H}}_{\neg\mathbf{t}}$:

   Analogous to the case $\kappa = \tilde{\mathtt{L}}_{\mathbf{t}}$ and $\kappa' = \tilde{\mathtt{L}}_{\neg\mathbf{t}}$.

6. $\kappa = \mathtt{L}_{\mathbf{t}}$ and $\kappa' = \tilde{\mathtt{L}}_{\mathbf{t}}$:

   From the definition of F we conclude that $\mathbf{t} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$ and that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \kappa',c',p'\rangle}$. By the inductive hypothesis, $p' = \mathbf{\Omega}$ and $c' = \mathbf{t}$. But, according to restriction R14, if $p' = \mathbf{\Omega}$ and $c' = \mathbf{t}$ then $\kappa' \notin \{\tilde{\mathtt{L}}_{\mathbf{t}}, \tilde{\mathtt{H}}_{\mathbf{t}}\}$. contradicting our assumption that $\kappa' = \tilde{\mathtt{L}}_{\mathbf{t}}$.

7. $\kappa = \mathtt{H}_{\mathbf{t}}$ and $\kappa' = \tilde{\mathtt{H}}_{\mathbf{t}}$:

   Analogous to the case $\kappa = \mathtt{L}_{\mathbf{t}}$ and $\kappa' = \tilde{\mathtt{L}}_{\mathbf{t}}$.

8. $\kappa = \mathtt{L}_{\mathbf{0}}$:

   (a) $\kappa' = \mathtt{H}_{\mathbf{0}}$: From the definition of F we conclude that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv \mathbf{0}$ and that $F^{n-1}_{\langle \kappa',c',p'\rangle} \equiv \mathbf{0}$. By the inductive hypothesis, we have that $p = \mathbf{\Omega}$ and $c = \mathbf{0}$, and $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$. But, according to restrictions R12 if $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$ then $\kappa' \notin \{\mathtt{L}_{\mathbf{0}}, \mathtt{H}_{\mathbf{0}}, \tilde{\mathtt{L}}_{\mathbf{0}}, \tilde{\mathtt{H}}_{\mathbf{0}}\}$. contradicting our assumption that $\kappa' = \mathtt{H}_{\mathbf{0}}$.

   (b) $\kappa' = \mathtt{H}_{\mathbf{1}}$: From the definition of F we conclude that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv \mathbf{1}$, and that $F^{n-1}_{\langle \kappa',c',p'\rangle} \equiv \mathbf{0}$. By the inductive hypothesis, we have that $p = \mathbf{\Omega}$ and $c = \mathbf{1}$, and $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$. From the definition of F, this function is satisfied only when $n = 2$, since $F^{2}_{\langle \mathtt{L}_{\mathbf{0}},\mathbf{1},\mathbf{\Omega}\rangle} \equiv$

$F^{2}_{\langle \mathtt{H}_1,0,\mathbf{\Omega}\rangle}$. But, according to restriction R16, when $n = 2$, $\kappa' \notin \{\mathtt{H}_\mathbf{0}, \mathtt{H}_\mathbf{1}\}$. contradicting our assumption that $\kappa' = \mathtt{H}_\mathbf{1}$.

(c) $\kappa' = \tilde{\mathtt{H}}_\mathbf{0}$: From the definition of F we conclude that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$ and that $\mathbf{0} \equiv F^{n-1}_{\langle \kappa',c',p'\rangle}$. By the inductive hypothesis, we have $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$. But, according to restrictions R12 if $p' = \mathbf{\Omega}$ and $c' = \mathbf{0}$ then $\kappa' \notin \{\mathtt{L}_\mathbf{0}, \mathtt{H}_\mathbf{0}, \tilde{\mathtt{L}}_\mathbf{0}, \tilde{\mathtt{H}}_\mathbf{0}\}$. contradicting our assumption that $\kappa' = \tilde{\mathtt{H}}_\mathbf{0}$.

(d) $\kappa' = \tilde{\mathtt{H}}_\mathbf{1}$: From the definition of F we conclude that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$ and that $\mathbf{1} \equiv F^{n-1}_{\langle \kappa',c',p'\rangle}$. By the inductive hypothesis, we have $p' = \mathbf{\Omega}$ and $c' = \mathbf{1}$. But, according to restrictions R13 if $p' = \mathbf{\Omega}$ and $c' = \mathbf{1}$ then $\kappa' \notin \{\mathtt{L}_\mathbf{1}, \mathtt{H}_\mathbf{1}, \tilde{\mathtt{L}}_\mathbf{1}, \tilde{\mathtt{H}}_\mathbf{1}\}$. contradicting our assumption that $\kappa' = \tilde{\mathtt{H}}_\mathbf{1}$.

9. $\kappa = \mathtt{L}_1$:

Analogous to $\kappa = \mathtt{L}_0$.

10. $\kappa = \mathtt{H}_0$:

(a) $\kappa' = \tilde{\mathtt{L}}_0$: From the definition of F we conclude that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$, and that $F^{n-1}_{\langle \kappa',c',p'\rangle} \equiv \mathbf{0}$. If $l(p) > l(p')$, when the function reaches a level n' such that $n' = l(p)$, from the definition of $\mathtt{H}_0$ and the above conclusions, $F^{n'}_{\langle \mathtt{S},c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$, and $F^{n'}_{\langle \kappa',c',p'\rangle} \equiv \mathbf{0}$. This is impossible since it violates the restriction R14. If $l(p) \geq l(p')$, when the function reaches a level n' such that $n' = l(p')$, from the definition of $\tilde{\mathtt{L}}_0$ and the above conclusions, $F^{n'}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{S},c',p'\rangle}$, and that $\mathbf{0} \equiv \mathbf{0}$. From the definition of F, these conditions are satisfied when $l(p) = l(p')$ and $n - l(p) \leq 2$. But, according to restriction R15, if $n - l(p) \leq 2$ then $\kappa \notin \{\tilde{\mathtt{H}}_0, \tilde{\mathtt{H}}_1, \tilde{\mathtt{L}}_0, \tilde{\mathtt{L}}_1\}$ contradicting our assumption that $\kappa' = \tilde{\mathtt{L}}_0$.

(b) $\kappa' = \tilde{\mathtt{L}}_1$: From the definition of F we conclude that $F^{n-1}_{\langle \kappa,c,p\rangle} \equiv F^{n-1}_{\langle \mathtt{X},c',p'\rangle}$, and that $F^{n-1}_{\langle \kappa',c',p'\rangle} \equiv \mathbf{1}$. By the inductive hypothesis, we have $p' = \mathbf{\Omega}$ and $c' = \mathbf{1}$. But, according to restriction R13 if $p' = \mathbf{\Omega}$ and $c' = \mathbf{1}$ then $\kappa' \notin \{\mathtt{L}_\mathbf{1}, \mathtt{H}_\mathbf{1}, \tilde{\mathtt{L}}_\mathbf{1}, \tilde{\mathtt{H}}_\mathbf{1}\}$. contradicting our assumption that $\kappa' = \tilde{\mathtt{L}}_1$.

11. $\kappa = \mathtt{H}_1$:

    Analogous to $\kappa = \mathtt{H}_0$.

12. $\kappa = \tilde{\mathtt{L}}_0$:

    (a) $\kappa' = \tilde{\mathtt{H}}_0$: From the definition of F we conclude that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\mathtt{X},c',p'\rangle}$ and that $F^{n-1}_{\langle\kappa',c',p'\rangle} \equiv F^{n-1}_{\langle\mathtt{X},c,p\rangle}$. By the inductive hypothesis, we have $p = p'$ and $c = c'$. From the definition of F and $\mathtt{X}$, and the above conclusions, it follows that $p = p' = \mathbf{\Omega}$ and $c = c'$. But, according to restriction R14, if $p' = \mathbf{\Omega}$ then $\kappa \notin \{\tilde{\mathtt{L}}_0, \tilde{\mathtt{L}}_1, \tilde{\mathtt{H}}_0, \tilde{\mathtt{H}}_1\}$. contradicting our assumption that $\kappa' = \tilde{\mathtt{L}}_1$.

    (b) $\kappa' = \tilde{\mathtt{H}}_1$: From the definition of F we conclude that $F^{n-1}_{\langle\kappa,c,p\rangle} \equiv F^{n-1}_{\langle\mathtt{X},c',p'\rangle}$ and that $F^{n-1}_{\langle\kappa',c',p'\rangle} \equiv F^{n-1}_{\langle\mathtt{X},c,p\rangle}$. By the inductive hypothesis, we have $p = p'$ and $c = c'$. From the definition of F and $\mathtt{X}$, and the above conclusions, it follows that $p = p' = \mathbf{\Omega}$ and $c = c'$. But, according to restriction R14, if $p' = \mathbf{\Omega}$ then $\kappa \notin \{\tilde{\mathtt{L}}_0, \tilde{\mathtt{L}}_1, \tilde{\mathtt{H}}_0, \tilde{\mathtt{H}}_1\}$. contradicting our assumption that $\kappa' = \tilde{\mathtt{L}}_1$.

13. $\kappa = \tilde{\mathtt{L}}_1$:

    Analogous to $\kappa = \tilde{\mathtt{L}}_0$.

14. Remaining cases are either symmetric or analogous to those discussed.

Finally, suppose $l(p) = n$ and $l(p') < n$ (the case $l(p) < n$ and $l(p') = n$ is symmetric). We show that this is impossible, by contradiction. Consider the possible cases for κ':

1. $\kappa' = \mathtt{X}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$. By the inductive hypothesis, we conclude that $p[0].p = p[1].p = p'$ and $p[0].c = p[1].c$. If $l(p') = n-1$, then we have $p[0] = p[1] = \langle\mathtt{S},c,p'\rangle$; otherwise, we have $l(p') < n-1$ and by inductive hypothesis, $p[0] = p[1] = \langle\kappa',c',p'\rangle = \langle\mathtt{X},c',p'\rangle$. Either way, node p is redundant, and from R3 we have a contradiction.

2. $\kappa' = \mathtt{L}_\mathbf{t}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv \mathbf{t}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{\langle\kappa',c',p'\rangle}$. By the inductive hypothesis, we conclude that $p[0].p = \mathbf{\Omega}$, $p[0].c = \mathbf{t}$, $p[1].p = p'$ and $p[1].c = c'$. If

$l(p') = n-1$, then we have $p[1] = \langle \mathtt{S}, c', p' \rangle$; otherwise, we have $l(p') < n-1$ and by inductive hypothesis, $p[1] = \langle \kappa', c', p' \rangle = \langle \mathtt{L}_{\mathbf{t}}, c', p' \rangle$. Either way, node p is low-t, and from R4 and R5 we have a contradiction.

3. $\kappa' = \mathtt{H}_{\mathbf{t}}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv F^{n-1}_{\langle \kappa', c', p' \rangle}$ and $F^{n-1}_{p[1]} \equiv \mathbf{t}$. By the inductive hypothesis, we conclude that $p[0].p = p'$, $p[0].c = c'$, $p[1].p = \mathbf{\Omega}$ and $p[1].c = \mathbf{t}$. If we have $l(p) = 1$, from R16 we have a contradiction. If $l(p') = n-1$, then we have $p[0] = \langle \mathtt{S}, c', p' \rangle$; otherwise, we have $l(p') < n-1$ and by inductive hypothesis, $p[0] = \langle \kappa', c', p' \rangle = \langle \mathtt{H}_{\mathbf{t}}, c', p' \rangle$. Either way, node p is high-t, and from R6 and R7 we have a contradiction.

4. $\kappa' = \tilde{\mathtt{L}}_{\mathbf{t}}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv F^{n-1}_{\langle \mathtt{X}, c', p' \rangle}$ and $F^{n-1}_{p[1]} \equiv F^{n-1}_{\langle \kappa', c', p' \rangle}$. By the inductive hypothesis, we conclude that $p[0].p = p[1].p = p'$, and $p[0].c = p[1].c = c'$. If we have $l(p) - l(p') < 2$, from R15 we have a contradiction. If we have $p' = \mathbf{\Omega}$, from R14 we have a contradiction. Since we have $l(p') < n-1$, by inductive hypothesis, $p[1] = \langle \kappa', c', p' \rangle = \langle \tilde{\mathtt{L}}_{\mathbf{t}}, c', p' \rangle$. Therefore, node p is low-swap-t, and from R8 and R9 we have a contradiction.

5. $\kappa' = \tilde{\mathtt{H}}_{\mathbf{t}}$: From the definition of F, we must have $F^{n-1}_{p[0]} \equiv F^{n-1}_{\langle \kappa', c', p' \rangle}$, and $F^{n-1}_{p[1]} \equiv F^{n-1}_{\langle \mathtt{X}, c', p' \rangle}$. By the inductive hypothesis, we conclude that $p[0].p = p[1].p = p'$, and $p[0].c = p[1].c = c'$. If we have $l(p) - l(p') < 2$, from R15 we have a contradiction. If we have $p' = \mathbf{\Omega}$, from R14 we have a contradiction. Since we have $l(p') < n-1$, by inductive hypothesis, $p[0] = \langle \kappa', c', p' \rangle = \langle \tilde{\mathtt{H}}_{\mathbf{t}}, c', p' \rangle$. Therefore, node p is high-swap-t, and from R10 and R11 we have a contradiction.
$\square$

The canonicity result establishes that, regardless of how a reduced CESRBDD is constructed for a given function, the resulting reduced CESRBDD is guaranteed to be unique (assuming the variable order is fixed). From now on, unless otherwise specified, we assume that all CESRBDDs are reduced.

6.5 Comparing CESRBDDs to other types of decision diagrams

Chapter 4 compared the sizes of ESRBDDs to BDDs, ZDDs, CBDDs, CZDDs, and TBDDs. We extend this comparison to CESRBDDs. The process of converting these decision diagrams to CESRBDDs remains the same as shown in Section 4.3.3, since any ESRBDD is also a CESRBDD. But the CESRBDD thus constructed may need to be reduced by applying Algorithm 6.1. For example, to convert a BDD to a CESRBDD, we can annotate long edges in the BDD with $\mathtt{X}$ and short edges with $\mathtt{S}$, to obtain an unreduced CESRBDD, before applying the reduction algorithm. We now show that the reduction will not increase the CESRBDD size, and thus the resulting CESRBDD cannot be larger than the original BDD.

Lemma 6.5.1

Suppose we have an unreduced CESRBDD where, for every node q, there exists a rule $\kappa \in \{\mathtt{X}, \mathtt{L_0}, \mathtt{H_0}, \mathtt{L_1}, \mathtt{H_1}, \tilde{\mathtt{H}}_0, \tilde{\mathtt{L}}_0, \tilde{\mathtt{H}}_1, \tilde{\mathtt{L}}_1\}$. such that every edge to q is either $\langle \mathtt{S},c,q \rangle$ or $\langle \kappa,c,q \rangle$. Then reducing the CESRBDD will not increase the number of nodes.

Proof: Algorithm 6.1 always reduces child nodes before reducing the parent. When reducing a node q, all incoming edges to q will be labeled either with $\mathtt{S}$ or with κ. Normalizing q by complementing its edges does not create any new nodes. Any incoming edge $\langle \mathtt{S},c,q \rangle$ will not cause any node to be created, since reduction will only eliminate a node. Any incoming edge $\langle \kappa,c,q \rangle$ edges will cause at most one node to be created, if κ is not compatible with κ' (see MergeEdge), where κ' is the label of the reduced edge representing q. But then node q is eliminated. Thus, the overall number of nodes cannot increase. $\square$

$$\begin{tikzcd}
 & \text{TBDD} \arrow[dd, bend right, "3"'] & \\
\text{BDD} \arrow[dr, bend right, "1"'] & & \text{ZDD} \arrow[dl, bend right, "1"'] \\
 & \text{CESR} \arrow[uu, bend right, "7L/9"'] \arrow[ul, bend right, "8L/9"'] \arrow[ur, bend right, "8L/9"'] \arrow[dl, bend right, "7L/9"'] \arrow[dr, bend right, "7L/9"'] \arrow[dd, bend right, "2L/3"'] & \\
\text{CBDD} \arrow[ur, bend right, "2"'] & & \text{CZDD} \arrow[ul, bend right, "2"'] \\
 & \text{ESR} \arrow[uu, bend right, "3/2"'] &
\end{tikzcd}$$

An edge from A to B with weight n says, "Converting from A to B could increase the size of the representation by a factor of n".

Figure 6.10 CESRBDD size comparison.

Lemma 6.5 applies to any decision diagram that has two kinds of edges, short edges indicating `S`, and long edges representing a single type of reduction. For instance, in BDDs a long edge represents `X`, and in ZDDs a long edge represents $\texttt{H}_0$.

The worst-case for converting BDDs, ZDDs, CBDDs, CZDDs, and TBDDs to CESRBDDs is no different from that ESRBDDs. The conversion from ESRBDDs to CESRBDDs can produce a larger CESRBDD, similar to how an ESRBDD could be larger than a ESRBDD$-\texttt{L}_0$. The worst-case ratio for the size of an ESRBDD representing a ESRBDD$-\texttt{L}_0$ is 3/2, as shown in Section 4.3.3, and is based on the observation that for an ESRBDD$-\texttt{L}_0$ node q with outgoing `X` and $\texttt{H}_0$ edges to a low-zero node, the ESRBDD reduction will eliminate the low-zero node and create one node for each outgoing edge of q. This is also the worst-case scenario for a ESRBDD node being converted to a CESRBDD. Therefore, we get a worst-case ratio of 3/2 for the size of a CESRBDD representing an ESRBDD.

The worst-case for converting CESRBDDs to BDDs, ZDDs, CBDDs, CZDDs, and TBDDs changes due to CESRBDDs having more edge labels than ESRBDDs.

Consider the set of 8 ESRBDD edges $\langle\kappa_i,c,q\rangle$ where q is a node at level 2 and $\kappa_i \in \{\mathtt{L}_0, \mathtt{H}_0, \mathtt{L}_1, \mathtt{H}_1, \tilde{\mathtt{H}}_0, \tilde{\mathtt{L}}_0, \tilde{\mathtt{H}}_1, \tilde{\mathtt{L}}_1\}$. A CESRBDD can be constructed using a node at level L, with 2 children at level $L-1$, and 4 grand children at level $L-2$, such that the 8 outgoing edges from the grandchildren correspond to the 8 distinct $\langle\kappa_i,c,q\rangle$. This CESRBDD has 9 nodes (counting the terminal). Because, BDDs can only exploit $\mathtt{X}$ edges, this will produce a BDD with at least $8(L-5)+9 = 8L-31$ nodes, giving a worst-case ratio of $8L/9$. The ZDD worst-case is similar, using $\mathtt{X}$ instead of $\mathtt{H}_0$ in set of labels for κ_i. For, CZDDs, CBDDs and TBDDs, i.e. decision diagram that can exploit both $\mathtt{X}$ and $\mathtt{H}_0$ edges, we can construct CESRBDDs with 7 distinct edge labels that will correspond to a CBDD, CZDD or TBDD with at least $7(L-5)+9 = 7L-26$ nodes, giving a worst-case ratio of $7L/9$. Finally, ESRBDDs can exploit $\mathtt{X}$, $\mathtt{L}_0$ and $\mathtt{H}_0$ edges, and we can construct a CESRBDD with 6 distinct edge labels that will correspond to an ESRBDD with at least $6(L-5)+9 = 6L-21$ nodes giving a worst-case ratio of $2L/3$.

6.6 Conclusions

In this chapter, we have extended the notion of ESRBDDs to include edge complementation and analogues to $\mathtt{L}_0$ and $\mathtt{H}_0$ edges: $\mathtt{L}_1$, $\mathtt{H}_1$, $\tilde{\mathtt{H}}_0$, $\tilde{\mathtt{L}}_0$, $\tilde{\mathtt{H}}_1$ and $\tilde{\mathtt{L}}_1$. We have provided the definition for CESRBDDs and reduced CESRBDDs, a depth-first reduction algorithm, and proved that reduced CESRBDDs are canonical. We have also provided a theoretical comparison between CESRBDDs and existing decision diagrams (including ESRBDDs), and highlighted the potential for producing smaller representations.

Acknowledgments

This work was supported in part by National Science Foundation grant ACI-1642397.

CHAPTER 7. BINARY DECISION DIAGRAMS WITH NODE SPECIFIED REDUCTIONS

In Chapter 4, we defined ESRBDDs to eliminate duplicate, redundant, low-zero and high-zero nodes; proved that they were canonical and gave experimental evidence for their compactness. In Chapter 5, we proved that the Apply operation over ESRBDDs has a time-complexity at most that of BDDs and ZDDs. In Chapter 6, we defined a canonical extension of ESRBDDs that included complemented edges in addition to the reductions $\mathtt{X}$, $\mathtt{L}_0$, $\mathtt{H}_0$, $\mathtt{L}_1$, $\mathtt{H}_1$, $\tilde{\mathtt{H}}_0$, $\tilde{\mathtt{L}}_0$, $\tilde{\mathtt{H}}_1$, $\tilde{\mathtt{L}}_1$. In this chapter, we look at the complexity of the Apply operation for CESRBDDs, and define an extension for ESRBDDs designed to improve the time-complexity for Apply over ESRBDDs.

The rest of this chapter is organized as follows. We discuss the time-complexity of Apply-ESR and Apply-CESR in Section 7.1, and highlight the patterns in these computations that lead to a factor of L in the time-complexity. In Section 7.2, we define NSRBDDs and reduced NSRBDDs. In Section 7.3, we discuss the time-complexity for the Apply operation over NSRBDDs, and Section 7.4 concludes.

7.1 Introduction

From Theorem 5.2.1, the time-complexity of Apply-ESR, is $\mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|.L)$. In addition, from Theorem 5.2.1, we have

$$\Psi_{\text{ESR},\oplus} \leq \text{Min} \begin{cases} 2.\Psi_{\text{BDD},\oplus} \\ 2.\Psi_{\text{ZDD},\oplus} \end{cases}$$

Let us consider the Apply operation over long edges in CESRBDDs. The additional reduction rules in CESRBDDs improve the size of the representation with respect to ESRBDDs by (at most) a factor of $2L/3$. From our analysis of Apply-ESR, and with respect to Apply-BDD, it is straightforward to design an Apply algorithm for Apply-CESR that skips the same levels as

Apply-ESR, and that requires (in the worst-case) as many recursive calls as Apply-ESR, i.e.

$$\Psi_{\text{CESR},\oplus} \leq \Psi_{\text{ESR},\oplus}$$

There are operations wherein some of the sub-cases can benefit from the presence of additional reduction rules in CESRBDDs. For example, for the OR operation:

1. $\langle \texttt{X},0,p_0\rangle \vee \langle \texttt{L}_1,0,p_1\rangle = \langle \texttt{L}_1,c,p\rangle$, where c and p are computed from the result of $\langle p_0 \vee p_1\rangle$. In an ESRBDD on the other hand, this operation would take up to $n - n'$ additional steps where n is the level at which the call to Apply-ESR originated, and n' is $\textsc{Max}(l(p_0), l(p_1))$.

   The case $\langle \texttt{X},0,p_0\rangle \vee \langle \texttt{H}_1,0,p_1\rangle$ also benefits similarly and reduces the the number of recursive calls by a factor of $n - n'$, which in the worst-case is L.

2. $\langle \texttt{L}_1,0,p_0\rangle \vee \langle \texttt{H}_1,0,p_1\rangle = \mathbf{1}$ can be computed in constant-time in a CESRBDD, and (in the worst-case) will take an additional L steps in an ESRBDD.

In the next section, we describe the patterns that may arise during the computation of $\langle \kappa_0,c_0,p_0\rangle \oplus \langle \kappa_1,c_1,p_1\rangle$. For the sake of convenience, from here on we assume that there are two terminals $\mathbf{0}$ and $\mathbf{1}$, there are no complement edges, and edges are represented as $\langle \kappa,p\rangle$, i.e. the convention used for ESRBDDs.

We define the following generic patterns:

Low-Pattern, $\texttt{L}_{\langle p_0,p_1\rangle}$ A low-pattern to nodes p_0 and p_1 mimics a $\texttt{L}_0$ edge to node p_1 with node p_0 taking the place of terminal $\mathbf{0}$. The low-pattern is defined over levels x_n to $l(p_1)$, with $l(p_1) \geq l(p_0)$.

High-Pattern, $\texttt{H}_{\langle p_0,p_1\rangle}$ A high-pattern to nodes p_0 and p_1 mimics a $\texttt{H}_0$ edge to node p_0 with node p_1 taking the place of terminal $\mathbf{0}$. The high-pattern is defined over levels x_n to $l(p_0)$, with $l(p_0) \geq l(p_1)$.

We will define these patterns formally in short order.

The following is the list of patterns that arise for $\langle \kappa_0,p_0\rangle \oplus \langle \kappa_1,p_1\rangle$ where $\kappa_0, \kappa_1 \in \{\texttt{X}, \texttt{L}_0, \texttt{H}_0, \texttt{L}_1, \texttt{H}_1\}$. x_n is the level at which the call to Apply is made, and $x_m = \textsc{Max}(l(p_0), l(p_1))$.

1. $\langle \mathtt{X}, p_0 \rangle \oplus \langle \mathtt{L}_{\mathbf{t}}, p_1 \rangle$:

   A low-pattern is produced by this computation as shown in Figure 7.1 that spans levels x_n to x_m. Between levels x_n and $x_m + 1$ no computations are needed, i.e. these levels can be skipped by the corresponding APPLY operation. At level x_m, two recursive calls to APPLY are needed, to compute the following:

   (a) $\langle \kappa'_0, p'_0 \rangle = \langle \mathtt{X}, p_0 \rangle \oplus \langle \mathtt{X}, \mathbf{t} \rangle$, and

   (b) $\langle \kappa'_1, p'_1 \rangle = \langle \kappa_0, p_0 \rangle \oplus \langle \kappa_1, p_1 \rangle$, where κ_0 is $\mathtt{S}$ if $l(p_0) \geq l(p_1)$, and $\mathtt{X}$ otherwise; and, κ_1 is $\mathtt{S}$ if $l(p_0) \leq l(p_1)$, and $\mathtt{L}_{\mathbf{t}}$ otherwise.

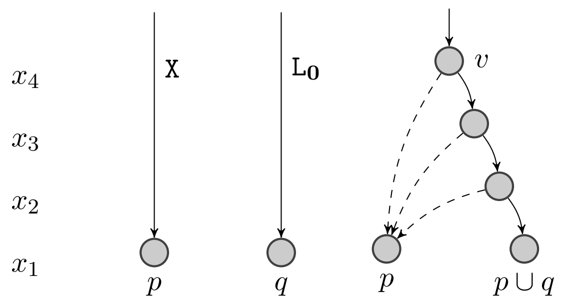


Figure 7.1 $\langle \mathtt{X}, p_0 \rangle \cup \langle \mathtt{L}_{\mathbf{0}}, p_1 \rangle$.

2. $\langle \mathtt{X}, p_0 \rangle \oplus \langle \mathtt{H}_{\mathbf{t}}, p_1 \rangle$:

   A high-pattern is produced by this computation as shown in Figure 7.2 that spans levels x_n to x_m. Between levels x_n and $x_m + 1$ no computations are needed, i.e. these levels can be skipped by the corresponding APPLY operation. At level x_m, two recursive calls to APPLY are needed, to compute the following:

   (a) $\langle \kappa'_0, p'_0 \rangle = \langle \kappa_0, p_0 \rangle \oplus \langle \kappa_1, p_1 \rangle$, where κ_0 is $\mathtt{S}$ if $l(p_0) \geq l(p_1)$, and $\mathtt{X}$ otherwise; and, κ_1 is $\mathtt{S}$ if $l(p_0) \leq l(p_1)$, and $\mathtt{H}_{\mathbf{t}}$ otherwise, and

   (b) $\langle \kappa'_1, p'_1 \rangle = \langle \mathtt{X}, p_0 \rangle \oplus \langle \mathtt{X}, \mathbf{t} \rangle$.

3. $\langle \mathtt{L}_{\mathbf{t}}, p_0 \rangle \oplus \langle \mathtt{X}, p_1 \rangle$:

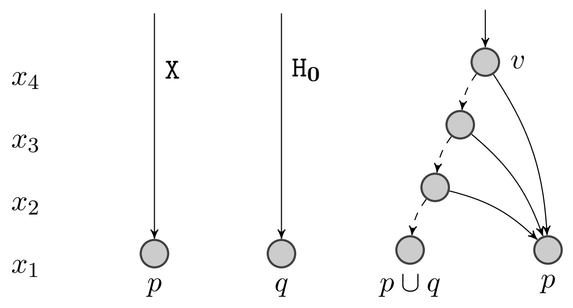


Figure 7.2 $\langle \mathtt{X}, p_0 \rangle \cup \langle \mathtt{H}_{\mathbf{0}}, p_1 \rangle$.

A low-pattern is produced by this computation as shown in Figure 7.3 that spans levels x_n to x_m. Between levels x_n and $x_m + 1$ no computations are needed, i.e. these levels can be skipped by the corresponding APPLY operation. At level x_m, two recursive calls to APPLY are needed, to compute the following:

(a) $\langle \kappa'_0, p'_0 \rangle = \langle \mathtt{X}, \mathbf{t} \rangle \oplus \langle \mathtt{X}, p_1 \rangle$, and

(b) $\langle \kappa'_1, p'_1 \rangle = \langle \kappa_0, p_0 \rangle \oplus \langle \kappa_1, p_1 \rangle$, where κ_0 is S if $l(p_0) \geq l(p_1)$, and $\mathtt{L}_{\mathbf{t}}$ otherwise; and, κ_1 is S if $l(p_0) \leq l(p_1)$, and X otherwise.

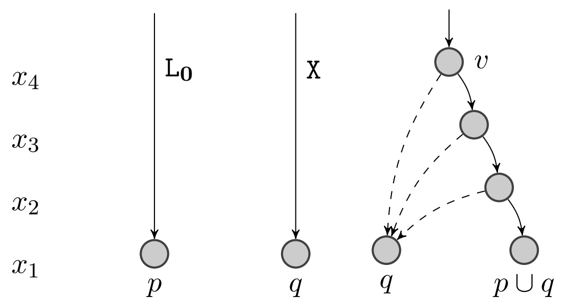


Figure 7.3 $\langle \mathtt{L}_{\mathbf{0}}, p_1 \rangle \cup \langle \mathtt{X}, p_0 \rangle$.

4. $\langle \mathtt{H}_{\mathbf{t}}, p_0 \rangle \oplus \langle \mathtt{X}, p_1 \rangle$:

A high-pattern is produced by this computation as shown in Figure 7.4 that spans levels x_n to x_m. Between levels x_n and $x_m + 1$ no computations are needed, i.e. these levels can be skipped by the corresponding APPLY operation. At level x_m, two recursive calls to APPLY are needed, to compute the following:

(a) $\langle \kappa_1', p_1' \rangle = \langle \kappa_0, p_0 \rangle \oplus \langle \kappa_1, p_1 \rangle$, where κ_0 is $\mathtt{S}$ if $l(p_0) \geq l(p_1)$, and $\mathtt{H}_{\mathbf{t}}$ otherwise; and, κ_1 is $\mathtt{S}$ if $l(p_0) \leq l(p_1)$, and $\mathtt{X}$ otherwise, and

(b) $\langle \kappa_0', p_0' \rangle = \langle \mathtt{X}, \mathbf{t} \rangle \oplus \langle \mathtt{X}, p_1 \rangle$.

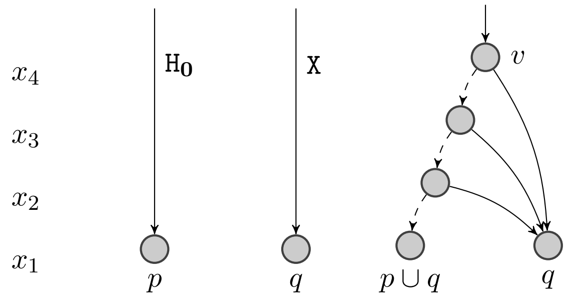


Figure 7.4 $\langle \mathtt{H}_{\mathbf{0}}, p_1 \rangle \cup \langle \mathtt{X}, p_0 \rangle$.

5. $\langle \mathtt{L}_{\mathbf{t}_0}, p_0 \rangle \oplus \langle \mathtt{H}_{\mathbf{t}_1}, p_1 \rangle$:

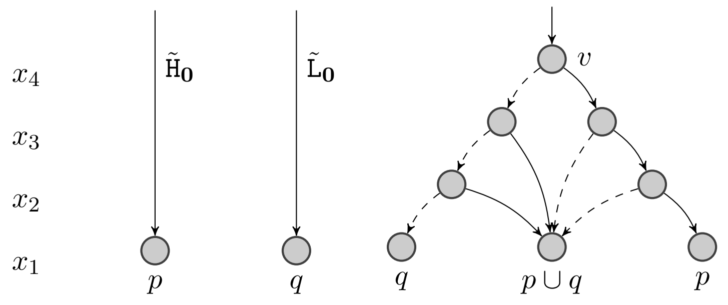

x_4
x_3
x_2
x_1
$\tilde{\mathtt{H}}_{\mathbf{0}}$
$\tilde{\mathtt{L}}_{\mathbf{0}}$
v
p
q
q
$p \cup q$
p

Figure 7.5 $\langle \tilde{\mathtt{H}}_{\mathbf{0}}, p_1 \rangle \cup \langle \tilde{\mathtt{L}}_{\mathbf{0}}, p_0 \rangle$.

A combination of a low-pattern and a high-pattern are produced in this computation as shown in Figure 7.5 that spans levels x_{n-1} to x_m. At level x_n two recursive calls to Apply are needed, to compute the following:

(a) $\langle \kappa_0', p_0' \rangle = \langle \mathtt{X}, \mathbf{t}_0 \rangle \oplus \langle \mathtt{H}_{\mathbf{t}_1}, p_1 \rangle$ produces a high-pattern, and

(b) $\langle \kappa_1', p_1' \rangle = \langle \mathtt{L}_{\mathbf{t}_0}, p_0 \rangle \oplus \langle \mathtt{X}, \mathbf{t}_1 \rangle$ produces a low-pattern.

In general, the above instances induce a factor of L steps in Apply-ESR and Apply-CESR since their long edges can only represent low- or high-patterns in which at least one of the nodes is a terminal. Therefore, the time-complexity for ESRBDDs and CESRBDDs remains

$$\mathcal{O}(|\mathrm{ESR}(A)|.|\mathrm{ESR}(B)|.L), \text{ and}$$

$$\mathcal{O}(|\mathrm{CESR}(A)|.|\mathrm{CESR}(B)|.L)$$

respectively. To eliminate the factor of L, the Apply algorithm must:

1. Skip levels that are skipped by long edges, i.e. for any operation $\oplus$ and long edges $\langle\kappa_0,p_0\rangle$, $\langle\kappa_1,p_1\rangle$, if Apply can compute $\langle\kappa_0,p_0\rangle \oplus \langle\kappa_1,p_1\rangle$ at level $x_m = \textsc{Max}(l(p_0), l(p_1))$ in β steps, then it must be able to compute $\langle\kappa_0,p_0\rangle \oplus \langle\kappa_1,p_1\rangle$ at any level x_n *s.t.* $L \geq x_n > x_m$, in $k \times \beta$ steps, where k is a constant.

2. Represent the low- and high-pattern result in constant-time.

7.2 Definition of NSRBDDs

Definition 7.2.1

An L-level *(ordered) node-specified reduction* binary decision diagram (NSRBDD) is a directed acyclic graph where the two *terminal* nodes $\mathbf{0}$ and $\mathbf{1}$ are at level 0, $l(\mathbf{0}) = l(\mathbf{1}) = 0$, while each *nonterminal* node p belongs to a level $l(p) \in \{1, ..., L\}$ and a *reduction pattern* in $p.r \in \{\mathtt{S}, \mathtt{L}, \mathtt{H}\}$ and has two outgoing edges, $p[0]$ and $p[1]$, pointing to nodes at lower levels. For $i \in \{0, 1\}$, if $l(p[i]) = l(p) - 1$, we say that $p[i]$ is a *short* edge. If instead $l(p[i]) < l(p) - 1$, the only other possibility, we say that $p[i]$ is a *long* edge, since it "skips over" one or more levels. □

Long edges in NSRBDDs correspond to `X` edges. Reduction patterns `L` and `H` correspond to the low- and high-patterns discussed in the Section 7.1. We define a function $m(p)$ for an NSRBDD node p as

$$m(p) = \begin{cases} 1 + \textsc{Max}(l(p[0]), l(p[1])) & l(p) > 0, p.r \in \{\mathtt{L}, \mathtt{H}\} \\ l(p) & \text{otherwise.} \end{cases}$$

To make definition of NSRBDDs more precise, we recursively define the boolean function $F_p^n : \mathbb{B}^n \to \mathbb{B}$ encoded by an NSRBDD node p with respect to a level $n \in \{0, ..., L\}$, subject to $n \geq m(p)$, as

$$F_p^n(x_{1:n}) = \begin{cases} \text{if } l(p) = 0, & p \\ \text{if } n > l(p) > 0, & F_p^{l(p)}(x_{1:l(p)}) \\ \text{if } n > 0 \wedge n = m(p), & F_{p[x_n]}^{n-1}(x_{1:n-1}) \\ \text{if } l(p) \geq n > m(p) \wedge p.r = \mathtt{L}, & x_n \ ? \ F_p^{n-1}(x_{1:n-1}) \ : \ F_{p[0]}^{n-1}(x_{1:n-1}) \\ \text{if } l(p) \geq n > m(p) \wedge p.r = \mathtt{H}, & x_n \ ? \ F_{p[1]}^{n-1}(x_{1:n-1}) \ : \ F_p^{n-1}(x_{1:n-1}) \end{cases}$$

Before we formally define a reduced NSRBDD, we need some terminology. We say that a node p is:

duplicate if there exists a node q such that, $l(p) = l(q)$, $p.r = q.r$, $p[0] = q[0]$, and $p[1] = q[1]$.

redundant if $p[0] = p[1]$.

quasi-node if p a nonterminal node with both children at level $l(p) - 1$.

low-pattern if $p[0] \neq p[1]$, and at least one of the following statements is true:

1. $p.r \neq \mathtt{L}$, $l(p) > l(p[0]) + 1$, $l(p) = l(p[1]) + 1$.

   That is, the edge to $p[0]$ is long, and the edge to $p[1]$ is short.

2. $p.r = p[1].r = \mathtt{L}$, $p[0] = p[1][0]$.

   That is, $p[1]$ is a low-pattern node that can be merged with p.

3. $p.r = \mathtt{L}$, $p[1]$ is a quasi-node, and $p[0] = p[1][0]$.

   That is, $p[1]$ is a quasi-node that can be merged with p.

high-pattern if $p[1] \neq p[0]$, and at least one of the following statements is true:

1. $p.r \neq \mathtt{H}$, $l(p) = l(p[0]) + 1$, $l(p) > l(p[1]) + 1$.

   That is, the edge to $p[0]$ is short, and the edge to $p[1]$ is long.

2. $p.r = p[0].r = \mathtt{H}$, $p[1] = p[0][1]$.

   That is, $p[0]$ is a high-pattern node that can be merged with p.

3. $p.r = \texttt{H}$, $p[0]$ is a quasi-node, and $p[1] = p[0][1]$.

   That is, $p[0]$ is a quasi-node that can be merged with p.

Definition 7.2.2

An NSRBDD is *reduced* if the following restrictions hold:

R1. There are no duplicate nodes.

R2. There are no redundant nodes.

R3. There are no high-pattern nodes.

R4. There are no low-pattern nodes.

We do not give a proof for its canonicity here, but point out that the reduction rules have been constructed with the same strategy as those for ESRBDDs and CESRBDDs:

- If pattern P_a matching reduction rule R_a and pattern P_b matching reduction rules R_b occur adjacent to each other (along a path in the graph) with P_a occurring immediately prior to P_b, we define the reduced form as that which is produced when pattern P_a is eliminated first followed by the elimination of pattern P_b (if it still exists). For example, see Figure 5.1.

- If the set of node patterns P_a and P_b that match the reduction rules R_a and R_b respectively, are not disjoint, we define a restriction that disallows any pattern in $P_a \cap P_b$ to be represented using R_b, and add a *production* rule for R_b in terms of R_a specifically for building patterns that match R_b and build on patterns in $P_a \cap P_b$. For example, since level 1 nodes in CESRBDDs can be represented using $\texttt{L}_\texttt{t}$ and $\texttt{H}_\texttt{t}$ edges, we restricted $\texttt{H}_\texttt{t}$ from representing level 1 nodes in reduced CESRBDDs, and added rules for $\texttt{H}_\texttt{t}$ edges based on $\texttt{L}_\texttt{t}$ edges that would allow long edges labelled $\texttt{H}_\texttt{t}$ that end in a terminal node. See Section 6.3.

We note that NSRBDDs can be seen as a generalization of CBDDs since "chained" CBDD nodes represent a chain of ZDD nodes, i.e. chained nodes in CBDDs corresponds to low-pattern nodes q in NSRBDDs with $q[0] = \mathbf{0}$.

7.3 Time-complexity for Apply operations on NSRBDDs

In this section, we revisit the cases in Section 7.1 that produce a factor of L in the time-complexity of Apply for ESRBDDs and CESRBDDs. To recall, these are the list of patterns that arise for $\langle \kappa_0, p_0 \rangle \oplus \langle \kappa_1, p_1 \rangle$ where $\kappa_0, \kappa_1 \in \{\mathtt{X}, \mathtt{L}_0, \mathtt{H}_0, \mathtt{L}_1, \mathtt{H}_1\}$. x_n is the level at which the call to Apply-NSR is made, and $x_m = \textsc{Max}(l(p_0), l(p_1))$. For the sake of clarity, we initially assume $x_m = l(p_0) = l(p_1)$.

To maintain our convention with ESRBDDs, we represent long edges in NSRBDDs as $\langle \mathtt{X}, q \rangle$ for some NSRBDD node q; blank-pattern nodes as $\mathtt{S}_{\langle p,q \rangle}$, low-pattern nodes as $\mathtt{L}_{\langle p,q \rangle}$, and high-pattern nodes as $\mathtt{H}_{\langle p,q \rangle}$. Note that this representation is only complete where a "starting" level x_n for is provided.

1. $x_n, \langle \mathtt{X}, p_0 \rangle \oplus \langle \mathtt{X}, p_1 \rangle$:

   This produces a long edge $\langle \mathtt{X}, r \rangle$, where r is computed with a single recursive call to Apply-NSR: $r_0 = p_0 \oplus p_1$.

2. $x_n, \mathtt{L}_{\langle p_0,p_1\rangle} \oplus \mathtt{L}_{\langle p_2,p_3\rangle}$:

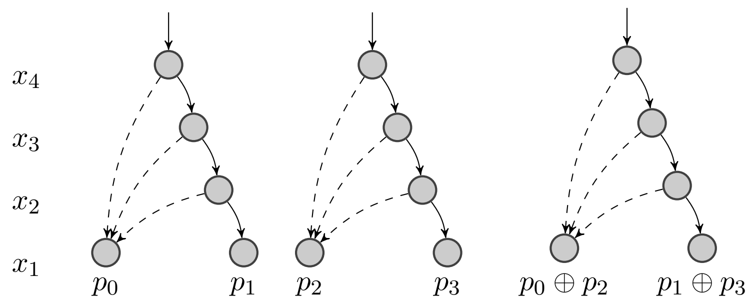


Figure 7.6 $\mathtt{L}_{\langle p_0,p_1\rangle} \oplus \mathtt{L}_{\langle p_2,p_3\rangle}$.

This produces a low-pattern $\mathtt{L}_{\langle r_0,r_1\rangle}$ (see Figure 7.6) that can be computed with two recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_2$, and

(b) $r_1 = p_1 \oplus p_3$.

Once these two recursions have completed, the result can be built in constant-time, i.e. independent of $x_n - x_m$.

3. $x_n, \mathtt{H}_{\langle p_0,p_1\rangle} \oplus \mathtt{H}_{\langle p_2,p_3\rangle}$:

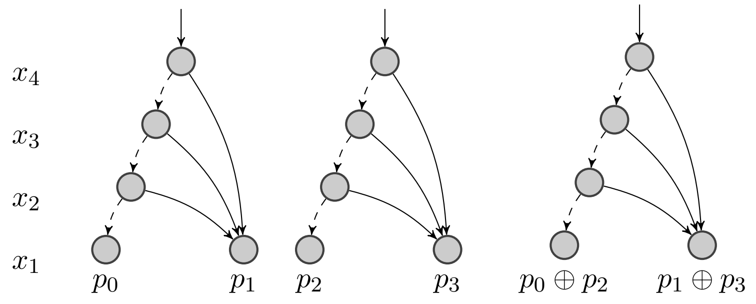


Figure 7.7 $\mathtt{H}_{\langle p_0,p_1\rangle} \oplus \mathtt{H}_{\langle p_2,p_3\rangle}$.

This produces a high-pattern $\mathtt{H}_{\langle r_0,r_1\rangle}$ (see Figure 7.7) that can be computed with two recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_2$, and

(b) $r_1 = p_1 \oplus p_3$.

Once these two recursions have completed, the result can be built in constant-time, i.e. independent of $x_n - x_m$.

4. $x_n, \langle \mathtt{X}, p_0 \rangle \oplus \mathtt{L}_{\langle p_1, p_2 \rangle}$:

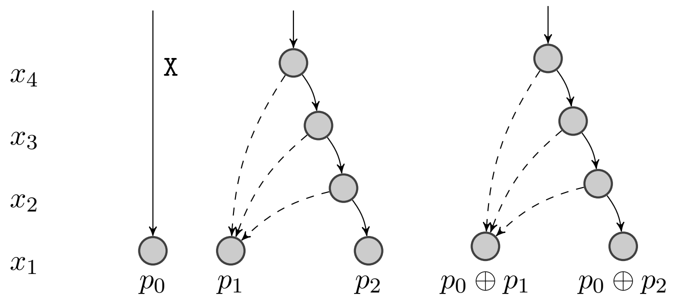


Figure 7.8 $\mathtt{X}p_0 \oplus \mathtt{L}_{\langle p_1, p_2 \rangle}$.

This produces a low-pattern $\mathtt{L}_{\langle r_0, r_1 \rangle}$ (see Figure 7.8) that can be computed with two recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_1$, and

(b) $r_1 = p_0 \oplus p_2$.

Once these two recursions have completed, the result can be built in constant-time, i.e. independent of $x_n - x_m$.

5. $x_n, \langle \mathtt{X}, p_0 \rangle \oplus \mathtt{H}_{\langle p_1, p_2 \rangle}$:

This produces a high-pattern $\mathtt{H}_{\langle r_0, r_1 \rangle}$ (see Figure 7.9) that can be computed with two recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_1$, and

(b) $r_1 = p_0 \oplus p_2$.

Once these two recursions have completed, the result can be built in constant-time, i.e. independent of $x_n - x_m$.

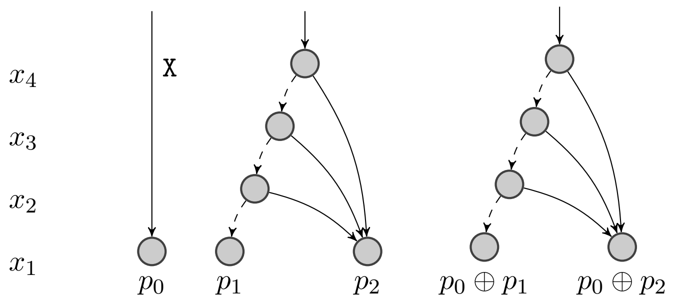


Figure 7.9 $\langle \mathtt{X}, p_0\rangle \oplus \mathtt{H}_{\langle p_1,p_2\rangle}$.

6. $x_n, \mathtt{L}_{\langle p_0,p_1\rangle} \oplus \langle \mathtt{X}, p_2\rangle$:

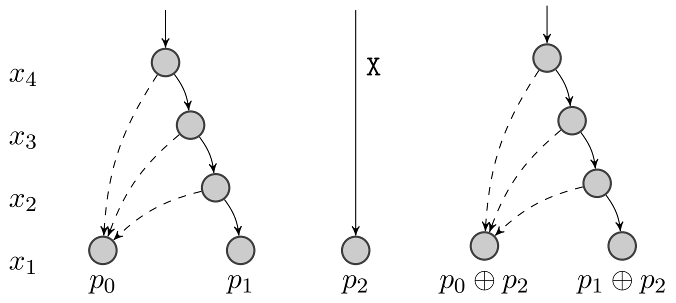


Figure 7.10 $\mathtt{L}_{\langle p_1,p_2\rangle} \oplus \mathtt{X}p_0$.

This produces a low-pattern $\mathtt{L}_{\langle r_0,r_1\rangle}$ (see Figure 7.10) that can be computed with two recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_2$, and

(b) $r_1 = p_1 \oplus p_2$.

Once these two recursions have completed, the result can be built in constant-time, i.e. independent of $x_n - x_m$.

7. $x_n, \mathtt{H}_{\langle p_0,p_1\rangle} \oplus \langle \mathtt{X}, p_2\rangle$:

This produces a high-pattern $\mathtt{H}_{\langle r_0,r_1\rangle}$ (see Figure 7.11) that can be computed with two recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_2$, and

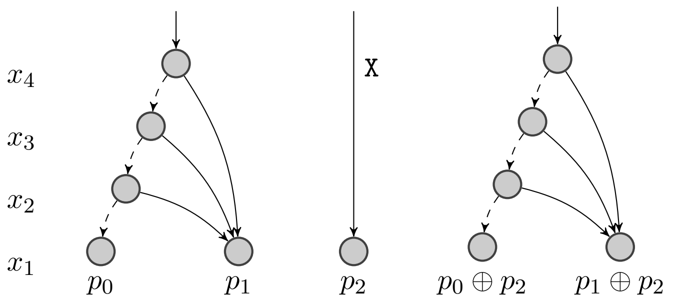


Figure 7.11 $\mathtt{H}_{\langle p_1,p_2\rangle} \oplus \langle \mathtt{X},p_0\rangle$.

(b) $r_1 = p_1 \oplus p_2$.

Once these two recursions have completed, the result can be built in constant-time, i.e. independent of $x_n - x_m$.

8. $x_n, \mathtt{H}_{\langle p_0,p_1\rangle} \oplus \mathtt{L}_{\langle p_2,p_3\rangle}$:

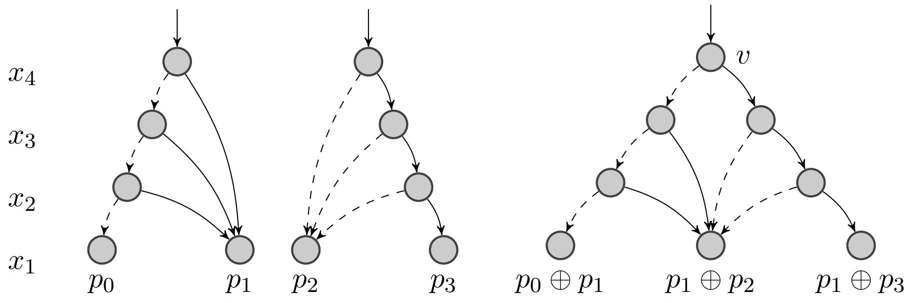


Figure 7.12 $\mathtt{H}_{\langle p_0,p_1\rangle} \oplus \mathtt{L}_{\langle p_2,p_3\rangle}$.

This produces a high-pattern $\mathtt{H}_{\langle r_0,r_1\rangle}$ (see Figure 7.12) and a low-pattern $\mathtt{L}_{\langle s_0,s_1\rangle}$ that spans levels x_{n-1} to x_m, and can be computed with recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_2$,

(b) $r_1 = s_0 = p_1 \oplus p_2$, and

(c) $s_1 = p_1 \oplus p_3$.

Once these three recursions have completed, the high-pattern and low-pattern nodes at level x_{n-1} can be built in constant-time, i.e. independent of $x_{n-1} - x_m$. Finally, a node v is constructed at level x_n with $v[0] = \mathtt{H}_{\langle r_0,r_1\rangle}$, and $v[1] = \mathtt{L}_{\langle s_0,s_1\rangle}$. Note that the high-pattern becomes the low-edge of v and the low-pattern becomes the high-edge of v.

9. $x_n, \mathtt{L}_{\langle p_0,p_1\rangle} \oplus \mathtt{H}_{\langle p_2,p_3\rangle}$:

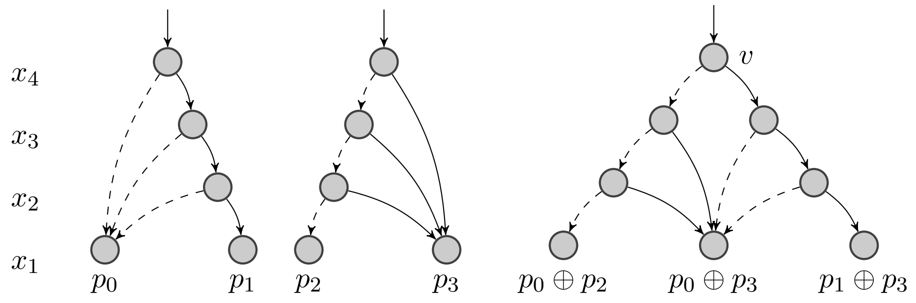


Figure 7.13 $\mathtt{L}_{\langle p_0,p_1\rangle} \oplus \mathtt{H}_{\langle p_2,p_3\rangle}$.

This produces a high-pattern $\mathtt{H}_{\langle r_0,r_1\rangle}$ (see Figure 7.13) and a low-pattern $\mathtt{L}_{\langle s_0,s_1\rangle}$ that spans levels x_{n-1} to x_m, and can be computed with recursive calls to Apply-NSR:

(a) $r_0 = p_0 \oplus p_2$,

(b) $r_1 = s_0 = p_0 \oplus p_3$, and

(c) $s_1 = p_1 \oplus p_3$.

Once these three recursions have completed, the high-pattern and low-pattern nodes at level x_{n-1} can be built in constant-time, i.e. independent of $x_{n-1} - x_m$. Finally, a node v is constructed at level x_n with $v[0] = \mathtt{H}_{\langle r_0,r_1\rangle}$, and $v[1] = \mathtt{L}_{\langle s_0,s_1\rangle}$. Note that, in this case as well, the high-pattern becomes the low-edge of v and the low-pattern becomes the high-edge of v.

We point out that it may not be possible to construct the pattern-nodes directly from the result of Apply-NSR. As was the case with ESRBDDs and CESRBDDs, only compatible edges can be "merged". ESRBDDs and CESRBDDs created an additional node for every pair of incompatible edges that needed merging. In NSRBDDs, the result of an Apply-NSR can either be a short edge or a long edge to some node q. Short edges pose no problems, but long edges can. For example, the low-pattern node $\mathtt{L}_{\langle p_0,p_1\rangle}$ requires $l(p_1) \geq l(p_0)$. If a low-pattern node must be built to span levels x_n to x_m for nodes p_0 and p_1, and if $l(p_1) < x_m$, a solution is to build a blank-pattern node $v = \mathtt{S}_{\langle p_0, p_1 \rangle}$ at level x_{m+1} and then build the low-pattern to span levels x_n to x_{m+1} using v instead of p_1. As was the case with ESRBDDs and CESRBDDs, this is a constant-time procedure and is performed (at most) once per pair of operand edges to Apply.

From the above, we see that, in the general case, Apply for NSRBDDs can compute the result for operands that span levels x_n to x_m by computing the result at level x_m in time β, and constructing the result at level x_n in time $k \times \beta$, where k is independent of x_n, x_m or L, i.e. a constant. Therefore, the time-complexity for Apply over NSRBDDs is $\mathcal{O}(|\text{NSR}(A)|.|\text{NSR}(B)|)$.

7.4 Conclusions

We have described another extension of ESRBDDs for the sole purpose of removing the factor of L in the time-complexity of Apply-ESR. We have described the cases in which the factor of L appears and shown that NSRBDDs are able to handle each of these cases without incurring the factor of L cost, thereby making the case that NSRBDDs have a time-complexity of $\mathcal{O}(|\text{NSR}(A)|.|\text{NSR}(B)|)$.

Our future work will focus on building a formal proof for NSRBDD canonicity, an algorithm that details the utilization of the short-circuit computations that are necessary for arriving at the above time-complexity, and integrating the rest of the rules from CESRBDDs into NSRBDDs.

Acknowledgments

This work was supported in part by National Science Foundation grant ACI-1642397.

CHAPTER 8. CONCLUSIONS AND FUTURE WORK

Explicit State Space Generation

In Chapter 2, we have shown how explicit generation algorithms can benefit from the use decision diagrams as data structures to store the set of reachable states, the list of reachable states to explore, and the reachability graph. We have also presented small modifications to existing tradition explicit generation algorithms to make the generation process more efficient. Our experimental results show that decision diagrams can greatly reduce the peak memory requirements of traditional explicit generation algorithms. There is very obvious tradeoff between BST based implementation and the MDD based implementation, with the BST implementation being up to 20 times faster but using up to 1000 times as much memory as the MDD implementation. Models for which traditional explicit implementations run low on available memory could be prime candidates for the MDD based method.

Deadlock Detection in Petri Nets

In Chapter 3, we have presented novel symbolic algorithms for the detection of deadlocks in Petri Nets, and have given empirical evidence of their effectiveness in relation to existing symbolic algorithms. We have also presented symbolic algorithms for the detection of deadlocks in potentially unbounded Petri Nets. Of the algorithms we presented, Interrupted Saturation outperformed all existing symbolic deadlock detection algorithms for Petri Nets. Interrupted Saturation with Chaining was the best among the symbolic algorithms for deadlock detection in potentially unbounded Petri Nets. It also demonstrated the effectiveness of a novel technique that combined the depth-first Saturation algorithm with the breadth-first Chaining algorithm to avoid the pitfalls of exploring a potentially infinite reachable state space.

Extensions to Decision Diagrams

In Chapter 4, we defined a new type of decision diagram, ESRBDD, that are a simple yet efficient, generalization of previous attempts at combining reduction rules. Unlike previous efforts, ESRBDDs are not biased towards any particular reduction rules and therefore eliminate the need for the user to prioritize the reduction rules. We demonstrated their effectiveness by comparing them empirically against BDDs, ZDDs, CBDDs, CZDDs and TBDDs over many different model types.

In Chapter 5, we defined a depth-first saturation algorithm for reducing a node, and provided an Apply algorithm for ESRBDDs. Having shown that, for any binary function, ESRBDDs are at least as compact as the smaller of the two representations (BDDs and ZDDs), we also proved that for Apply operations over such functions, ESRBDDs are at least as fast as the faster of the two representations.

In Chapter 6, we defined CESRBDDs to extend ESRBDDs with edge complementation and six additional reductions rules that are based on the ZDD reduction rule. We then provided a a depth-first reduction algorithm for CESRBDDs, proved that reduced CESRBDDs are canonical, and demonstrated the potential compactness of CESRBDDs through a theoretical comparison between CESRBDDs and BDDs, ZDDs, CBDDs, CZDDs, TBDDs and ESRBDDs.

In Chapter 7, we defined NSRBDDs to extend BDDs with pattern-nodes representing a superset of the rules of ESRBDDs, and showed that Apply operations for NSRBDDs, in the general case, improve upon the time-complexity of Apply for ESRBDDs by a factor of L, where L is the number of variables in the decision diagram.

Future Work

Explicit State Space Generation

We have made the case for the use of decision diagrams in explicit state space generation. It would be useful to study the impact of this work on algorithms that exploit model symmetry.

Deadlock Detection in Petri Nets

The symbolic deadlock detection techniques for Petri Nets that we have described were designed to complement structural analysis techniques. It would, therefore, be useful to integrate these symbolic techniques into tools that perform structural analysis on Petri Nets. Further, structural analysis of a Petri net may reveal information helpful to the symbolic analysis. A closer look is also needed at understanding the effectiveness of invariant analysis on Interrupted Saturation with Chaining.

Extensions to Decision Diagrams

Many areas of work remain: an empirical analysis of CESRBDDs would complement the theoretical analysis we have provided; a formal proof for NSDBDDs canonicity; a formal proof for the time-complexity of Apply over NSRBDDs; an empirical analysis of Apply operations over ESRBDDs, CESRBDDs and NSRBDDs; extending NSRBDDs with the reduction rules in CESRBDDs and analyzing the trade-offs if any; and building a software library for ESRBDDs and their extensions.

BIBLIOGRAPHY

Alur, R., Brayton, R. K., Henzinger, T. A., Qadeer, S., and Rajamani, S. K. (1997). Partial-order reduction in symbolic state space exploration. In *Proc. CAV*, pages 340–351. Springer.

Amparore, E. G., Ciardo, G., Donatelli, S., and Miner, A. (2019). irank: A variable order metric for deds subject to linear invariants. In Vojnar, T. and Zhang, L., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 285–302, Cham. Springer International Publishing.

Babar, J., Beccuti, M., Donatelli, S., and Miner, A. S. (2010). GreatSPN enhanced with decision diagram data structures. In *Proc. ATPN*, LNCS 6128, pages 308–317. Springer.

Babar, J., Jiang, C., Ciardo, G., and Miner, A. (2019). Binary decision diagrams with edge-specified reductions. In Vojnar, T. and Zhang, L., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–318, Cham. Springer International Publishing.

Babar, J. and Miner, A. (2014). Explicit state space and markov chain generation using decision diagrams. In Horváth, A. and Wolter, K., editors, *Computer Performance Engineering*, volume 8721 of *Lecture Notes in Computer Science*, pages 240–254. Springer International Publishing.

Babar, J. and Miner, A. S. (2010). Meddly: Multi-terminal and Edge-valued Decision Diagram LibrarY. In *Proc. QEST*, pages 195–196. IEEE Comp. Soc. Press.

Baier, C., Haverkort, B., Hermanns, H., and Katoen, J.-P. (2003). Model checking algorithms for continuous-time Markov chains. *IEEE Trans. Softw. Eng.*, 29(6):524–541.

Bollig, B. and Wegener, I. (1996). Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45(9):993–1002.

Brace, K. S., Rudell, R. L., and Bryant, R. E. (1990). Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 40–45. ACM Press.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691.

Bryant, R. E. (2018). Chain reduction for binary and zero-suppressed decision diagrams. In Beyer, D. and Huisman, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–98, Cham. Springer International Publishing.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170.

Chiola, G. (1989). Compiling techniques for the analysis of stochastic Petri nets. In *Proc. 4th Int. Conf. on Modelling Techniques and Tools for Performance Evaluation*, pages 13–27.

Chiola, G., Dutheillet, C., Franceschinis, G., and Haddad, S. (1993). Stochastic well-formed colored nets and symmetric modeling applications . *IEEE Trans. Comp.*, 42(11):1343–1360.

Chiola, G., Franceschinis, G., Gaeta, R., and Ribaudo, M. (1995). GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Perf. Eval.*, 24(1-2):47–68.

Chung, M.-Y., Ciardo, G., and Yu, A. J. (2006). A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 51—66.

Ciardo, G., Lüttgen, G., and Miner, A. S. (2007). Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100.

Ciardo, G., Lüttgen, G., and Siminiceanu, R. (2001). Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer.

Ciardo, G., Miner, A. S., and Wan, M. (2009). Advanced features in SMART: the Stochastic Model checking Analyzer for Reliability and Timing. *ACM SIGMETRICS Perf. Eval. Rev.*, 36(4):58–63.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999a). *Model Checking.* MIT Press.

Clarke, E. M., Long, D. E., and Mcmillan, K. L. (1999b). Compositional model checking. MIT Press.

Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J., and Zave, P. (1996). Formal methods: state of the art and future directions. *ACM Comp. Surv.*, 28(4):626–643.

Couvreur, J.-M. and Thierry-Mieg, Y. (2005). Hierarchical decision diagrams to exploit model structure. In *Proc. Formal Description Techniques, FORTE95*, volume 3731 of *LNCS*, pages 443–4572.

Delamare, C., Gardan, Y., and Moreaux, P. (2003). Performance evaluation with asynchronously decomposable SWN: implementation and case study. In *10th Int. Workshop on Petri Nets and Performance Models (PNPM'03)*, pages 20–29. IEEE Comp. Soc. Press.

Derisavi, S. (2007). A symbolic algorithm for optimal Markov chain lumping. In *Proc. TACAS*, LNCS 4424, pages 139–154. Springer.

Derisavi, S., Kemper, P., and Sanders, W. H. (2003). Symbolic state-space exploration and numerical analysis of state-sharing composed models. In *Numerical Solution of Markov Chains*, pages 167–189.

Drechsler, R. and Becker, B. (2006). Ordered Kronecker functional decision diagrams – a data structure for representation and manipulation of boolean functions. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 17(10):965–973.

Esparza, J. (1998). *Decidability and complexity of Petri net problems — An introduction*, pages 374–428. Springer Berlin Heidelberg, Berlin, Heidelberg.

Fleming, P. J. and Wallace, J. J. (1986). How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221.

Fujita, M., Matsunaga, Y., and Kakuda, T. (1991). On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Design Automation. EDAC., Proceedings of the European Conference on*, pages 50–54.

Hack, M. (1976). Decidability questions for Petri nets. Technical Report 161, Laboratory for Computer Science, Massachusetts Institute of Technology.

Hack, M. H. T. (1972). *Analysis of production schemata by Petri nets / Michel Henri Theodore Hack.* MAC T.R. 94. Project MAC, Massachusetts Institute of Technology, Cambridge.

Kam, T., Villa, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1998). Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62.

Karp, R. M. and Miller, R. E. (1969). Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147 – 195.

Kimura, S. and Clarke, E. M. (1990). A parallel algorithm for constructing binary decision diagrams. In *Proc. Int. Conf. on Computer Design (ICCD)*, pages 220–223. IEEE Comp. Soc. Press.

Knuth, D. E. (2011). *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part I*. Addison-Wesley.

Lai, Y.-T., Pedram, M., and Vrudhula, B. K. (1996). Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Comp.*, 45:247–255.

Lipton, R. (1976). The reachability problem requires exponential space. *Research Report 62. Department of Computer Science, Yale University.*

Mayr, E. W. (1984). An algorithm for the general petri net reachability problem. *SIAM Journal on Computing*, 13(3):441–460.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Mass 02061.

Miessler, D. and Haddix, J. SecLists. https://github.com/danielmiessler/SecLists.

Minato, S.-i. (2001). Zero-suppressed BDDs and their applications. *Software Tools for Technology Transfer*, 3:156–170.

Miner, A. S. (2001). Efficient solution of GSPNs using canonical matrix diagrams. In *Proc. PNPM*, pages 101–110. IEEE Comp. Soc. Press.

Miner, A. S. (2004). Implicit GSPN reachability set generation using decision diagrams. *Perf. Eval.*, 56(1-4):145–165.

Miner, A. S. and Ciardo, G. (1999). Efficient reachability set generation and storage using decision diagrams. In *Proc. ATPN*, LNCS 1639, pages 6–25. Springer.

Miner, A. S. and Parker, D. (2004). Symbolic representations and analysis of large state spaces. In *Validation of Stochastic Systems*, LNCS 2925, pages 296–338. Springer.

Molloy, M. K. (1982). Performance analysis using stochastic Petri nets. *IEEE Trans. Comp.*, 31(9):913–917.

Muppala, J. K., Ciardo, G., and Trivedi, K. S. (1993). Modeling using Stochastic Reward Nets. In *Proc. 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93)*, pages 367–372. IEEE Comp. Soc. Press.

Murata, T. (1989). Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579.

Norris IP, C. and Dill, D. L. (1996). Better verification through symmetry. *Formal Methods in System Design*, 9(1):41–75.

Pastor, E., Roig, O., Cortadella, J., and Badia, R. M. (1994). Petri net analysis using boolean manipulation. In *Proc. ATPN*, LNCS 815, pages 416–435. Springer.

Peled, D. (1994). Combining partial order reductions with on-the-fly model-checking. pages 377–390. Springer-Verlag.

Petri, C. (1962). *Kommunikation mit Automaten*. PhD thesis, University of Bonn.

Pnueli, A. (1977). The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pages 46–57. IEEE Comp. Soc. Press.

Queille, J. and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In *5th Intl. Symposium in Programming*, LNCS 137, pages 337–351. Springer.

Roig, O., Cortadella, J., and Pastor, E. (1995). Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *Proc. ATPN*, LNCS 935, pages 374–391. Springer.

Roig, O., Cortadella, J., and Pastor, E. (1996). Verification of asynchronous circuits by bdd-based model checking of petri nets. In *In 16th Int. Conf. on Application and Theory of Petri Nets, volume 935 of LNCS*, pages 374–391. Springer-Verlag.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. In *Int. Conference on CAD*, pages 139–144.

S. Minato, N. Ishiura, and S. Yajima (1990). Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 52–57. IEEE Comp. Soc. Press.

Smith, B. and Ciardo, G. (2018). SOUPS: a variable ordering metric for the saturation algorithm. In *Proc. International Conference on Application of Concurrency to System Design (ACSD)*. IEEE Comp. Soc. Press.

SPIN (1990). SPIN webpage. http://spinroot.com/spin.

van Dijk, T., Wille, R., and Meolic, R. (2017). Tagged BDDs: combining reduction rules from different decision diagram types. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, pages 108–115, Austin, TX. FMCAD Inc.

Vardi, M. and Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge.

Wan, M. and Ciardo, G. (2009). Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In Nielsen, M., Kučera, A., Miltersen, P., Palamidessi, C., Tůma, P., and Valencia, F., editors, *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 582–594. Springer Berlin Heidelberg.

Wimmer, R., Braitling, B., Becker, B., Hahn, E. M., Crouzen, P., Hermanns, H., Dhama, A., and Theel, O. (2010). Symblicit calculation of long–run averages for concurrent probabilistic systems. In *7th Int. Conf. on Quantitative Evaluation of Systems (QEST'10)*, pages 27–36.

Xie, A. and Beerel, P. A. (1998). Efficient state classification of finite-state Markov chains. *IEEE Trans. CAD of Integr. Circ. and Syst.*, 17(12):1334–1339.

Yang, S. (1991). *Logic synthesis and optimization benchmarks user guide: version 3.0.* Microelectronics Center of North Carolina (MCNC).

Yoneda, T., Hatori, H., Takahara, A., and Minato, S.-I. (1996). BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri nets. In *Proc. FMCAD*, LNCS 1166, pages 435–449.

Zhao, Y. and Ciardo, G. (2009). Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proc. ATVA*, LNCS 5799, pages 368–381. Springer.

APPENDIX. ESRBDD UNION AND INTERSECTION

The Apply algorithm described in Chapter 5 and Algorithm 5.8 is applicable to any binary boolean operation. In this appendix, we describe two specializations of this algorithm, namely, the union and intersection of ESRBDDs.

ESRBDD Union

The time-complexity of ESRBDD Union is the same as that of the generic Apply as described in Algorithm 5.8, i.e.

$$\mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|.L).$$

Figure A.1 illustrates the result of the Union operation over long-edges in ESRBDDs. As shown in this figure, the result of combining edges $\langle \texttt{X}, p \rangle$ and $\langle \kappa, q \rangle$ where $\kappa \in \{\texttt{L}_0, \texttt{H}_0\}$, needs upto $n - n'$ additional nodes, where, n is the level of node q' representing $\langle \kappa, q \rangle$ in a BDD or $\langle \texttt{X}, p \rangle$ in a ZDD (variable x_4 in the figure), and n' is the level for the higher of two nodes, p and q (variable x_1 in the figure). In the worst-case, $n - n'$ is L leading to a time-complexity of $\mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|.L)$ for ESRBDD Union. We note that the Union of BDDs and ZDDS, for the BDD and ZDD representation of these edges respectively, would require as many nodes in the result (as shown in Section 4.3.3).

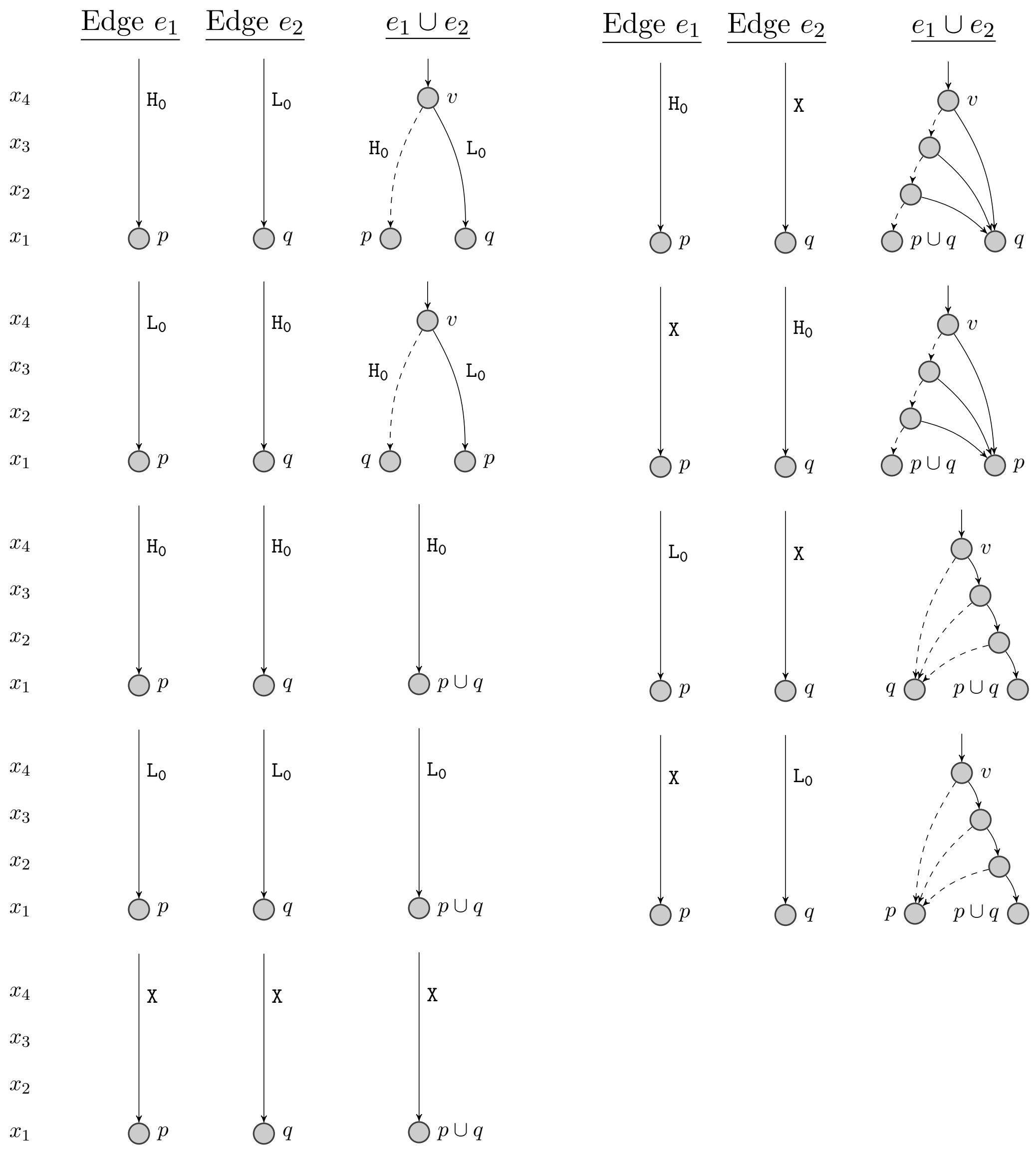


Figure A.1 Union over long edges.

Algorithm A.1 shows the specialization of Algorithm 5.8 based on the patterns in Figure A.1. In particular,

- BuildHighLow(n, $\langle\kappa_0$,$p_0\rangle$, $\langle\kappa_1$,$p_1\rangle$) builds the union of edges where $\kappa_0 = \mathtt{L}_0$ and $\kappa_1 = \mathtt{H}_0$ and vice-versa. We note that $p_0 \cup p_1$ is not computed since it is not a part of the final result.
- BuildHigh(n, $\langle\kappa_0$,$p_0\rangle$, $\langle\kappa_1$,$p_1\rangle$) builds the union of edges where $\kappa_0 = \mathtt{X}$ and $\kappa_1 = \mathtt{H}_0$ and vice-versa.
- BuildLow(n, $\langle\kappa_0$,$p_0\rangle$, $\langle\kappa_1$,$p_1\rangle$) builds the union of edges where $\kappa_0 = \mathtt{X}$ and $\kappa_1 = \mathtt{L}_0$ and vice-versa.

ESRBDD Intersection

The Intersection of ESRBDDs as defined in Chapter 4 has a time-complexity of

$$\mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|)$$

which is better that of the generic Apply as described in Algorithm 5.8.

Figure A.2 illustrates the result of the Intersection operation over long-edges in ESRBDDs. As shown in this figure, none of the combination long-edges in ESRBDDs requires the additional $n - n'$ nodes that are required in some cases for ESRBDD Union. Therefore the factor of L in ESRBDD Apply is reduced to a factor of 1 in the time-complexity of ESRBDD Intersection: $\mathcal{O}(|\text{ESR}(A)|.|\text{ESR}(B)|)$. Note that the Intersection of BDDs and ZDDS, for the BDD and ZDD representation of these edges respectively, would require as many nodes in the result.

Algorithm A.5 shows the specialization of Algorithm 5.8 based on the patterns in Figure A.2. Unlike the union of ESRBDDs described in Algorithm A.1, the intersection of ESRBDDs Algorithm A.5 does not require the use of BuildHighLow, BuildLow, and BuildHigh, since these patterns do not arise in the intersection of ESRBDDs.

Algorithm A.1 Union of ESRBDDs.

```
procedure UNION(Level $\hat{n}$, ESRBDD $\langle\kappa_0,p_0\rangle$, ESRBDD $\langle\kappa_1,p_1\rangle$)
    if $\hat{n} = 0$ then
        return $\langle \mathtt{S},p_0 \vee p_1\rangle$
    if $\langle\kappa_0,p_0\rangle = \langle \mathtt{X},\mathbf{1}\rangle \vee \langle\kappa_1,p_1\rangle = \langle \mathtt{X},\mathbf{1}\rangle$ then
        return $\langle \mathtt{X},\mathbf{1}\rangle$
    if $\langle\kappa_0,p_0\rangle = \langle\kappa_1,p_1\rangle \vee \langle\kappa_1,p_1\rangle = \langle \mathtt{X},\mathbf{0}\rangle$ then
        return $\langle\kappa_0,p_0\rangle$
    if $\langle\kappa_0,p_0\rangle = \langle \mathtt{X},\mathbf{0}\rangle$ then
        return $\langle\kappa_1,p_1\rangle$
    if $\kappa_0 = \mathtt{H_0} \wedge \kappa_1 = \mathtt{L_0}$ then
        return BUILDHIGHLOW($\hat{n}$, $\langle\kappa_0,p_0\rangle$, $\langle\kappa_1,p_1\rangle$)
    if $\kappa_0 = \mathtt{L_0} \wedge \kappa_1 = \mathtt{H_0}$ then
        return BUILDHIGHLOW($\hat{n}$, $\langle\kappa_1,p_1\rangle$, $\langle\kappa_0,p_0\rangle$)
    if "$\vee, \hat{n}, \langle\kappa_0,p_0\rangle, \langle\kappa_1,p_1\rangle, e$" $\in$ CT then
        return $\langle\kappa,p\rangle$
    $n \leftarrow$ MAX($l(\kappa_0)$, $l(\kappa_1)$)
    $p \leftarrow$ new node at level $n$
    $p[0] \leftarrow$ UNION($n-1$, EDGE($n$,$\langle\kappa_0,p_0\rangle$,0), EDGE($n$,$\langle\kappa_1,p_1\rangle$,0))
    $p[1] \leftarrow$ UNION($n-1$, EDGE($n$,$\langle\kappa_0,p_0\rangle$,1), EDGE($n$,$\langle\kappa_1,p_1\rangle$,1))
    if $\hat{n} = n$ then                                   • At least one incoming edge is short
        $\langle\kappa,p\rangle \leftarrow$ REDUCEEDGE($\hat{n}$,$\langle \mathtt{S},p\rangle$)
    else if $\kappa_0 = \kappa_1$ then
        $\langle\kappa,p\rangle \leftarrow$ REDUCEEDGE($\hat{n}$,$\langle\kappa_0,p\rangle$)
    else                          • for $i,j \in \{0,1\}, i \neq j, \exists i\ \kappa_i = \mathtt{X} \wedge \kappa_j \in \{\mathtt{L_0},\mathtt{H_0}\}$
        if $\kappa_0 = \mathtt{X}$ then
            $e_{\mathtt{X}} \leftarrow \langle\kappa_0,p_0\rangle$
        else
            $e_{\mathtt{X}} \leftarrow \langle\kappa_1,p_1\rangle$
        if $\kappa_0 = \mathtt{L_0} \vee \kappa_1 = \mathtt{L_0}$ then
            $\langle\kappa,p\rangle \leftarrow$ BUILDLOW($\hat{n}$,$n$,$e_{\mathtt{X}}$,$\langle \mathtt{S},p\rangle$)
        else
            $\langle\kappa,p\rangle \leftarrow$ BUILDHIGH($\hat{n}$,$n$,$\langle \mathtt{S},p\rangle$,$e_{\mathtt{X}}$)
    CT $\leftarrow$ CT $\cup$ "$\vee, \hat{n}, \langle\kappa_0,p_0\rangle, \langle\kappa_1,p_1\rangle, e$"
    return $\langle\kappa,p\rangle$
```

Algorithm A.2 Build an ESRBDD for the High-Low pattern.

```
1: procedure BuildHighLow(Level $n$, Edge $\langle\kappa_0,p_0\rangle$, Edge $\langle\kappa_1,p_1\rangle$)
2:     $p \leftarrow$ new node at level $n$
3:     $p[0] \leftarrow \langle \mathtt{H_0},p_0\rangle$
4:     $p[1] \leftarrow \langle \mathtt{L_0},p_1\rangle$
5:     return ReduceEdge($n$, $\langle\mathtt{S},p\rangle$)
```

Algorithm A.3 Build an ESRBDD for the High pattern.

```
1: procedure BuildHigh(Level $\hat{n}$, Level $n$, Edge $\langle\kappa_0,p_0\rangle$, Edge $\langle\kappa_1,p_1\rangle$)
2:     if $\hat{n} = n$ then return $\langle\kappa_0,p_0\rangle$
3:     $p \leftarrow$ new node at level $\hat{n}$
4:     $p[0] \leftarrow$ BuildHigh($\hat{n}-1$,$n$,$\langle\kappa_0,p_0\rangle$,$\langle\kappa_1,p_1\rangle$)
5:     $p[1] \leftarrow \langle\kappa_1,p_1\rangle$
6:     return ReduceEdge($\hat{n}$, $\langle\mathtt{S},p\rangle$)
```

Algorithm A.4 Build an ESRBDD for the Low pattern.

```
1: procedure BuildLow(Level $\hat{n}$, Level $n$, Edge $\langle\kappa_0,p_0\rangle$, Edge $\langle\kappa_1,p_1\rangle$)
2:     if $\hat{n} = n$ then return $\langle\kappa_1,p_1\rangle$
3:     $p \leftarrow$ new node at level $\hat{n}$
4:     $p[0] \leftarrow \langle\kappa_0,p_0\rangle$
5:     $p[1] \leftarrow$ BuildLow($\hat{n}-1$,$n$,$\langle\kappa_0,p_0\rangle$,$\langle\kappa_1,p_1\rangle$)
6:     return ReduceEdge($\hat{n}$, $\langle\mathtt{S},p\rangle$)
```

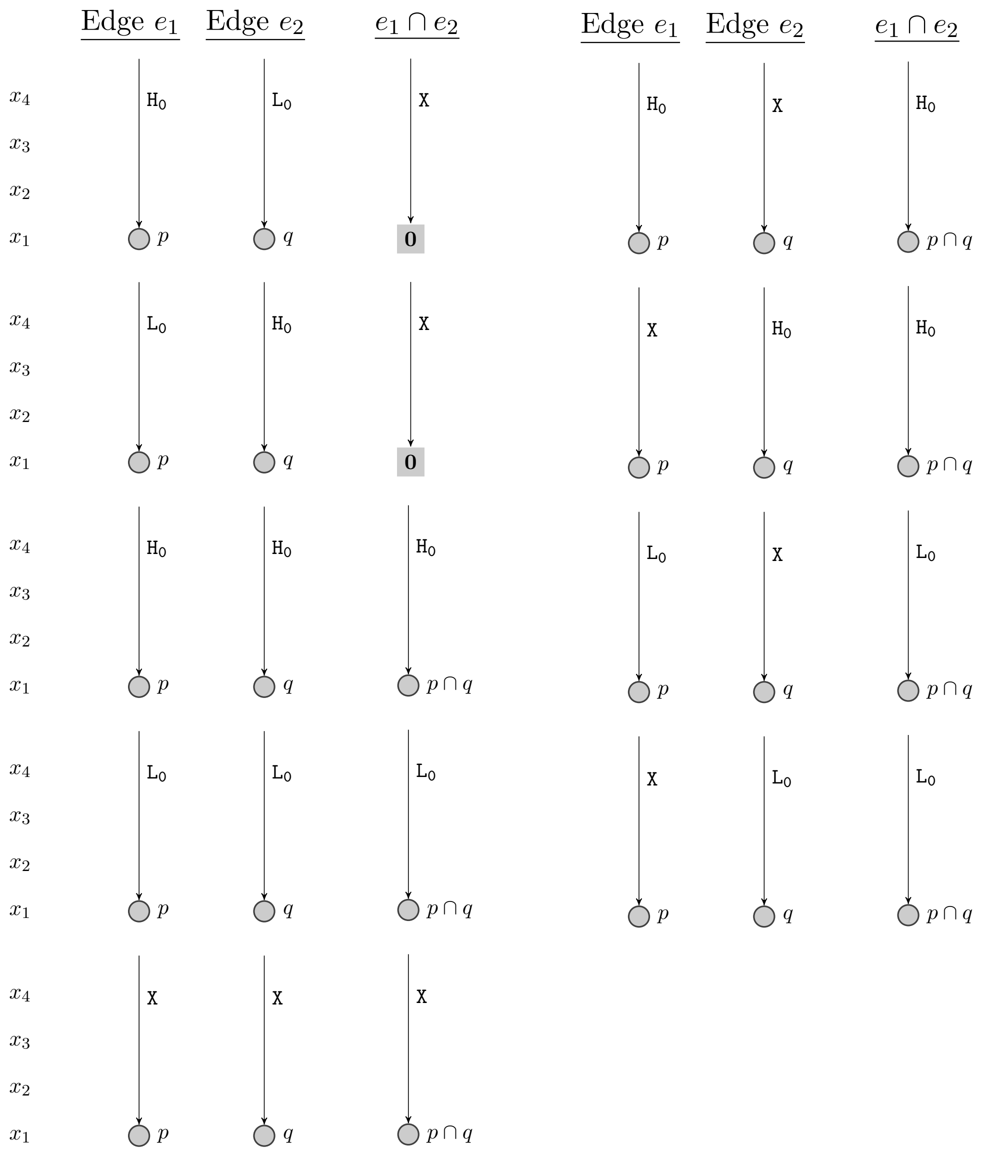


Figure A.2 Intersection of long-edges in ESRBDDs.

Algorithm A.5 Intersection of long-edges in ESRBDDs.

```
 1: procedure INTERSECT(Level $\hat{n}$, ESRBDD $\langle\kappa_0,p_0\rangle$, ESRBDD $\langle\kappa_1,p_1\rangle$)
 2:     if $\hat{n} = 0$ then
 3:         return $\langle \mathtt{S}, p_0 \wedge p_1\rangle$
 4:     if $\langle\kappa_0,p_0\rangle = \langle\kappa_1,p_1\rangle \vee \langle\kappa_1,p_1\rangle = \langle \mathtt{X},\mathbf{1}\rangle$ then
 5:         return $\langle\kappa_0,p_0\rangle$
 6:     if $\langle\kappa_0,p_0\rangle = \langle \mathtt{X},\mathbf{1}\rangle$ then
 7:         return $\langle\kappa_1,p_1\rangle$
 8:     if $\langle\kappa_0,p_0\rangle = \langle \mathtt{X},\mathbf{0}\rangle \vee \langle\kappa_1,p_1\rangle = \langle \mathtt{X},\mathbf{0}\rangle$ then
 9:         return $\langle \mathtt{X},\mathbf{0}\rangle$
10:     if $(\kappa_0 = \mathtt{L}_0 \wedge \kappa_1 = \mathtt{H}_0) \vee (\kappa_0 = \mathtt{H}_0 \wedge \kappa_1 = \mathtt{L}_0)$ then
11:         return $\langle \mathtt{X},\mathbf{0}\rangle$
12:     if "$\wedge, \hat{n}, \langle\kappa_0,p_0\rangle, \langle\kappa_1,p_1\rangle, e$" $\in \mathtt{CT}$ then
13:         return $\langle\kappa,p\rangle$
14:     $n \leftarrow$ MAX($l(\kappa_0)$, $l(\kappa_1)$)
15:     $p \leftarrow$ new node at level $n$
16:     $p[0] \leftarrow$ INTERSECT($n-1$, EDGE($n$,$\langle\kappa_0,p_0\rangle$,0), EDGE($n$,$\langle\kappa_1,p_1\rangle$,0))
17:     $p[1] \leftarrow$ INTERSECT($n-1$, EDGE($n$,$\langle\kappa_0,p_0\rangle$,1), EDGE($n$,$\langle\kappa_1,p_1\rangle$,1))
18:     if $\hat{n} = n$ then
19:         $\kappa \leftarrow \mathtt{S}$
20:     else if $\kappa_0 = \kappa_1$ then
21:         $\kappa \leftarrow \kappa_0$
22:     else if $\kappa_0 = \mathtt{L}_0 \vee \kappa_1 = \mathtt{L}_0$ then
23:         $\kappa \leftarrow \mathtt{L}_0$
24:     else                                                  • $\kappa_0 = \mathtt{H}_0 \vee \kappa_1 = \mathtt{H}_0$
25:         $\kappa \leftarrow \mathtt{H}_0$
26:     $\langle\kappa,p\rangle \leftarrow$ REDUCEEDGE($\hat{n}$,$\langle\kappa,0\rangle p$)
27:     $\mathtt{CT} \leftarrow \mathtt{CT} \cup$ "$\wedge, \hat{n}, \langle\kappa_0,p_0\rangle, \langle\kappa_1,p_1\rangle, e$"
28:     return $\langle\kappa,p\rangle$
```