

RICE UNIVERSITY

**BDD-Based Boolean Synthesis**

by

**Lucas Martinelli Tabajara**

A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

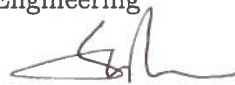
**Master of Science**

APPROVED, THESIS COMMITTEE:



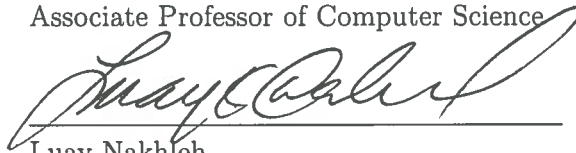
---

Moshe Y. Vardi, Chair  
Karen Ostrum George Distinguished  
Service Professor in Computational  
Engineering



---

Swarat Chaudhuri  
Associate Professor of Computer Science



Luay Nakhleh  
Professor of Computer Science, of Ecology  
and Evolutionary Biology, and of  
Biochemistry and Cell Biology, Chair of  
the Department of Computer Science



---

Dror Fried  
Postdoctoral Research Associate and  
Part-time Lecturer in Computer Science

Houston, Texas

April, 2018

## ABSTRACT

### BDD-Based Boolean Synthesis

by

Lucas Martinelli Tabajara

Synthesizing a Boolean function satisfying a given relation between inputs and outputs is a problem with many applications in the verification and design of hardware and software systems. In digital logic, Boolean synthesis can be used to automatically design circuits that produce the desired behavior. In program synthesis, Boolean functions can represent programs manipulating bit vectors and other data over finite domains. Additionally, Boolean synthesis is an essential component of reactive synthesis from temporal specifications, a problem that can be applied to automate the design of safety-critical systems. Binary Decision Diagrams (BDDs) have historically been popular data structures for representing Boolean functions, and BDDs are especially useful for the application of reactive synthesis, where they are particularly well-suited for fixpoint computations over sets of states. However, recent works in Boolean synthesis have raised concerns about the scalability of BDDs and chosen to use alternative approaches, such as SAT solvers.

In this thesis, we show that BDDs remain viable structures for Boolean synthesis, by developing a BDD-based synthesis framework that can in many cases outperform alternative approaches. For cases where efficient BDD representations are hard to construct, we demonstrate that techniques for decomposing a Boolean relation into multiple smaller BDDs can be used to make BDD-based approaches competitive.

# Contents

Abstract	ii
List of Illustrations	v
List of Tables	vii
<b>1 Introduction</b>	<b>1</b>
1.1 Program Synthesis, Reactive Synthesis and Functional Synthesis . . .	3
1.2 Synthesis from Factored Formulas . . . . .	6
1.3 Organization . . . . .	8
<b>2 Preliminaries</b>	<b>9</b>
2.1 Boolean Functions . . . . .	9
2.2 Quantified Boolean Formulas . . . . .	10
2.3 Binary Decision Diagrams . . . . .	10
<b>3 Related Work</b>	<b>14</b>
3.1 SAT-based Approaches to Boolean Synthesis . . . . .	14
3.2 Skolem Functions in QBF . . . . .	15
3.3 Boolean Unification . . . . .	17
<b>4 Boolean Synthesis by Self-Substitution</b>	<b>19</b>
4.1 Realizability and Synthesis . . . . .	21
4.2 Realizability and Function-Construction Using BDDs . . . . .	25
4.2.1 Realizability . . . . .	25
4.2.2 Function Construction . . . . .	26

4.3	Synthesis of Input-First BDD . . . . .	27
4.4	Benchmark Selection . . . . .	31
4.4.1	Relational Operations . . . . .	32
4.4.2	Addition . . . . .	32
4.4.3	Sorting . . . . .	33
4.5	Experimental Evaluation . . . . .	34
4.5.1	Scalability Comparison with Previous Approaches . . . . .	35
4.5.2	Shannon Expansion vs. Self-Substitution . . . . .	38
4.5.3	Synthesis for Input-First BDDs . . . . .	41
4.6	Experimental Comparison with CUDD v3.0.0 . . . . .	42
4.7	Conclusion . . . . .	45
<b>5</b>	<b>Synthesis from Factored Formulas</b>	<b>47</b>
5.1	Synthesis from Factored Formulas . . . . .	50
5.1.1	Factored Representation of Boolean Formulas . . . . .	50
5.1.2	Synthesis from Factored Specifications . . . . .	52
5.1.3	Clustering and Reordering . . . . .	57
5.1.4	BDD Variable Ordering . . . . .	58
5.2	Experimental Evaluation . . . . .	60
5.2.1	Heuristics for Factor Reordering . . . . .	61
5.2.2	Factored vs. Monolithic . . . . .	63
5.2.3	Comparison with CEGARSKOLEM and CADET . . . . .	65
5.2.4	Unrealizable Specifications . . . . .	68
5.2.5	QBF for Unrealizable Formulas . . . . .	69
5.3	Conclusion . . . . .	70
<b>6</b>	<b>Conclusion and Future Work</b>	<b>72</b>
	<b>Bibliography</b>	<b>75</b>

## Illustrations

2.1	Comparison between the binary decision tree and BDD for the formula $((x_1 \leftrightarrow x_2) \wedge (x_1 \leftrightarrow x_3)) \oplus x_1$ . . . . .	12
2.2	BDDs for the expression $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$ for two different variable orderings. Missing positive edges are understood to lead to the terminal 1 and missing negative edges to the terminal 0. . . . .	13
4.1	Example of the TRIMSUBSTITUTE method for a BDD representing the formula $((x_1 \rightarrow \neg y_1) \wedge (x_1 \oplus x_2) \wedge (x_1 \oplus y_2)) \vee ((x_1 \leftrightarrow x_2) \wedge (y_1 \oplus y_2))$ . Nodes with bold outlines are in $Fringe(B'_i)$ , and are either white if they should be replaced by the leaf node 0 or gray if they should be replaced by the leaf node 1. . . . .	30
4.2	Comparison of running time of RSYNTH against MONOSKOLEM . . . . .	36
4.3	Comparison of running time using non-deterministic benchmarks. . . . .	38
4.4	Comparison of Shannon Expansion and Self-Substitution, for the realizability step of the <i>Sorting</i> benchmark class. . . . .	39
4.5	Comparison of Self-Substitution and CUDD's native quantifier-elimination method, for the realizability step of the <i>Sorting</i> benchmark class. . . . .	40
4.6	Comparison of methods for synthesis using <i>input-first</i> BDDs for the <i>Sorting</i> benchmark class . . . . .	41

4.7	Comparison of running time of RSYNTH and the CUDD implementation <code>SolveEqn</code> on deterministic benchmarks. . . . .	43
4.8	Comparison of running time of RSYNTH and the CUDD implementation <code>SolveEqn</code> on non-deterministic benchmarks. . . . .	44
4.9	Comparison between TRIMSUBSTITUTE and the CUDD implementation <code>SolveEqn</code> using <i>input-first</i> BDDs for the <i>Sorting</i> benchmark class . . . . .	45
5.1	Synthesis from Factored Specifications . . . . .	54
5.2	Performance of the factored algorithm using different reordering heuristics, in log scale. The values include both the time spent reordering the factors and time running the algorithm. Bars of maximum height indicate instances that timed out. Bars not displayed mean that the instance took less than 1ms. . . . .	62
5.3	Performance of the monolithic and factored synthesis algorithms, in log scale. Bars of maximum height indicate instances that timed out. . . . .	64
5.4	Comparison of running time for monolithic and factored algorithms on the deterministic benchmarks of Section 4.5. . . . .	65
5.5	Comparison of running time between RSYNTH, CEGARSKOLEM and CADET, in log scale. Bars of maximum height indicate instances that timed out. . . . .	66
5.6	Comparison of function size, in number of nodes, between RSYNTH, CEGARSKOLEM and CADET, in log scale. The size is not displayed for those instances in which the tool timed out. . . . .	67
5.7	Comparison of running time between RSYNTH and CEGARSKOLEM tools, in log scale, over unrealizable benchmarks. Bars of maximum height indicate instances that timed out. . . . .	68

## Tables

4.1	Equivalent formulas using each method of quantifier elimination . . .	24
4.2	Benchmark classes used for synthesis. . . . .	32
4.3	Non-deterministic benchmark classes . . . . .	37

# Chapter 1

## Introduction

Boolean functions appear in all levels of computing, and can fairly be considered one of the most fundamental building blocks of modern digital computers. Often, the most intuitive way of defining a Boolean function is not *constructively*, describing how the outputs can be computed from the inputs, but rather *declaratively*, as a *relation* between input and output values that must be satisfied [1]. Nevertheless, in order to implement a function in a practical format, such as in a circuit or program, a declarative definition is not enough, and a constructive description of how to compute the output from the input is necessary. The process of converting a declarative formalization to a constructive one is called *functional synthesis* [2].

Synthesizing a Boolean function satisfying a given relation between inputs and outputs is a problem with applications in many areas of formal methods. Boolean functions can both represent logical circuits and encode programs over finite domains, and Boolean synthesis is an essential component of reactive synthesis from temporal specifications [3, 4, 5], where the goal is to synthesize a model of a system that interacts continuously with an environment.

When developing a Boolean-synthesis procedure, an important choice to be made is of how to represent Boolean functions and relations, since this representation has significant consequences on what operations can be efficiently performed. One representation of Boolean functions that has been popular in many applications is Binary Decision Diagrams (BDDs), data structures which represent Boolean functions as di-



rected acyclic graphs. BDDs are widely used in existing tools for verification and synthesis, especially in the context of reactive synthesis, where BDDs can be used to symbolically represent sets of states of the system. One advantage of BDDs in this context is that they are *canonical*, meaning that for a fixed ordering of the Boolean variables, two BDDs representing the same function are identical. This property makes equivalence checking extremely efficient for BDDs, making them particularly well-suited for computing fixpoints, which are the basis for many reactive synthesis algorithms. This fact motivates us to pursue BDD-based techniques for Boolean synthesis, in order to develop Boolean-synthesis algorithms that can directly interface with such tools, especially since Boolean synthesis has direct applications in reactive synthesis.

However, to effectively use BDDs as tools for Boolean synthesis, it is necessary to address some drawbacks of this data structure. In particular, the size of a BDD is heavily dependent on the ordering chosen for the Boolean variables during its construction, and computing an ordering that leads to an efficient representation is not always easy, or even possible.

In this thesis, we show that when an efficient ordering for the input-output relation is known, BDD-based Boolean synthesis excels over alternative approaches. Meanwhile, in the case when an efficient representation is not available, we demonstrate how techniques for decomposing the relation into smaller formulas allow us to handle cases that BDDs would otherwise be unable to, making BDD-based approaches competitive with other tools. The main contributions of this thesis are as follows:

1. Introducing a technique called *Self-Substitution* for synthesizing functions from Boolean formulas.

2. Designing a Boolean synthesis framework based on Self-Substitution using BDDs as the underlying data structures. When the synthesis problem can be efficiently represented by a BDD, this approach outperforms non-BDD-based approaches.
3. Providing a specialization of the synthesis framework to the class of *input-first* BDDs, for which we obtain significant improvements over the approach for general BDDs.
4. Adapting techniques used in model checking for processing *factored representations* to synthesize cases which would otherwise be unfeasible for BDDs to represent.

These contributions result in a flexible synthesis framework that can be applied to automate the design of a variety of systems, including serial programs, circuits and reactive systems.

To better understand the problem of Boolean functional synthesis and the role played by BDDs in the context of this problem, in the next section we present an overview of synthesis and the different variants of this problem within the area of formal methods, starting from the origins of program synthesis from theorem-proving procedures in first-order logic, to modern synthesis approaches focusing on specific domains, including the synthesis of reactive systems, which forms one of the main practical applications of Boolean synthesis.

## 1.1 Program Synthesis, Reactive Synthesis and Functional Synthesis

Program synthesis is the problem of algorithmically constructing a program given a specification of the program's expected behavior. The roots of program synthesis

can be traced back to deductive systems [6] where a program is constructed as a byproduct of proving a theorem of the form  $\forall x.\exists y.f(x,y)$ . Such a theorem can be interpreted as saying that for every input  $x$  there exists an output  $y$  that satisfies some relation  $f(x,y)$ . Then, from a constructive proof of the theorem it would be possible to extract a function that, given an  $x$ , outputs a  $y$  satisfying  $f(x,y)$ . Early approaches attempted to prove general theorems in first-order logic, a problem which is in general undecidable. Therefore, for an arbitrary theorem, there is no guarantee that such a procedure will terminate. To be able to guarantee that the synthesis procedure will produce a result for well-defined classes of formulas, more recent works on synthesis tend to specialize in more restricted but decidable logic systems. We now look at examples of such works representing different points of view on the synthesis problem and how they relate to the work developed in this thesis.

Possibly the most prominent example of such a specialization is *reactive synthesis* [7], which applies synthesis to specifications in temporal logic. In contrast with the previous deductive systems, this form of synthesis does not intend to produce a simple sequential program, but rather a system that continuously interacts with an environment. This system can be, for example, a controller for a power plant, which receives input signals from various sensors and responds by sending output signals to physical actuators in order to maintain the correct operation of the plant. Reactive synthesis has since become a well-studied problem in the field of formal methods, due to its applicability in the automated design of safety-critical systems.

In contrast to reactive synthesis, the term *functional synthesis* was introduced in [8] to denote the original application of synthesis to sequential programs. That work avoids the incompleteness of the previous systems based on first-order logic by proposing the development of fully decidable synthesis procedures for specific

domains, such as Presburger arithmetic or sets with size constraints.

Following the above terminology, this thesis uses the term *Boolean functional synthesis* to refer to the problem of synthesizing a function from Boolean inputs to Boolean outputs. Since many data types can be encoded as bit vectors, including integers, enumerations and sets, programs in a number of interesting domains can be generated by this form of synthesis. Additionally, Boolean synthesis is equally useful for designing logic circuits, since these are effectively physical implementations of Boolean functions.

Furthermore, the applications of Boolean synthesis are not limited to sequential programs and combinational circuits. In fact, Boolean functional synthesis has an important application as a component of reactive synthesis. By encoding input and output signals as well as the state of the system as Boolean values, a Boolean formula can be computed that encodes the behavior of a reactive system in a single cycle, which can then be synthesized using Boolean-synthesis techniques. A common way of deriving this Boolean formula is through a fixpoint computation, for which BDDs are particularly suited due to the efficiency of equivalence-checking for BDDs. With this application in mind, BDDs are a natural choice for data structures for Boolean synthesis.

Despite having been used in the past to synthesize Boolean functions [9, 10], more recently BDDs have lost popularity due to the drawbacks mentioned earlier, which often cause small BDD representations to be hard or even impossible to find for certain formulas. For example, it has been shown that a Boolean formula encoding the multiplication of two numbers in binary can only be represented by a BDD of exponential size in the number of bits. In such a case, BDD-based methods might not be applicable because the BDD for the relation cannot be constructed within

reasonable time and memory constraints.

Given these observations, this thesis seeks to demonstrate two main claims. The first is that for cases where an efficient representation of the relation as a BDD can be obtained, BDD-based techniques significantly outperform alternative approaches. The second is that when such a representation cannot be constructed, BDDs can remain competitive by using *factored representations*. When employing a factored representation, rather than using a single BDD, the relation is decomposed into multiple BDDs, such that the sum of the sizes of the individual BDDs can be much smaller than the size of the BDD for the entire relation would be. A function that satisfies the relation can then be synthesized by considering only some of the smaller BDDs at a time, thus avoiding constructing the entire relation and significantly reducing the size of the BDDs that must be manipulated. The next section presents a more in-depth motivation and description of this approach.

## 1.2 Synthesis from Factored Formulas

The idea of decomposing, or *factoring*, Boolean formulas to allow for efficient representation has been applied to a number of problems that can be solved using BDDs, including model checking [11] and symbolic satisfiability [12]. This technique was successful in allowing these problems to be applied to instances that would be infeasible when using a monolithic representation, that is, representing the entire formula as a single BDD. By employing factored formulas, previous works were able to obtain representations that were over 100 times smaller than the monolithic representation would be [11].

Factored formulas take advantage of the fact that formulas used in practice often are in the form of a conjunction (logical *and*) of constraints, such as  $f_1(\vec{x}, \vec{y}) \wedge f_2(\vec{x}, \vec{y}) \wedge$

$\dots \wedge f_n(\vec{x}, \vec{y})$ . Instead of representing the entire formula in a single data structure, one can instead represent each subformula  $f_i(\vec{x}, \vec{y})$  of the conjunction individually, while leaving the conjunction implicit. In this context, the subformulas are called *factors*, and a formula is satisfied if and only if all of its factors are satisfied. For BDDs, using factored formulas can significantly improve the efficiency of representation, since BDDs can in the worst case have size exponential in the number of variables of the formula. If each factor uses only a small subset of the variables, the size of each BDD can be exponentially smaller than if all factors were combined.

However, model checking, symbolic satisfiability and synthesis all tend to process formulas using existential quantification, which does not distribute over conjunction. In other words,  $\exists \vec{y}.(f_1(\vec{x}, \vec{y}) \wedge \dots \wedge f_n(\vec{x}, \vec{y}))$  is not equivalent to  $(\exists \vec{y}.f_1(\vec{x}, \vec{y})) \wedge \dots \wedge (\exists \vec{y}.f_n(\vec{x}, \vec{y}))$ . In synthesis, for example, existentially quantifying a variable corresponds to synthesizing a function that outputs a satisfying assignment for that variable, so this limitation translates to the fact that a function that satisfies one of the factors might not satisfy the others. This means that each factor cannot simply be considered individually.

Following the approach employed in model checking and symbolic satisfiability, to be able to perform synthesis using factored formulas while avoiding combining all the factors, this thesis takes advantage of the fact that existential quantification of a variable only needs to be applied to those factors in which the variable appears. Therefore, by combining the factors one by one, a variable can be existentially quantified as soon as all factors in which it appears have been processed. Although combining factors tends to produce larger BDDs, existential quantification removes the variable from the BDD, which tends to reduce its size. Therefore, if factors are processed in the right order the size of the formula can be kept under control. Since in synthesis exis-

tential quantification also corresponds to function construction, this also means that it is likely that the synthesized functions will be smaller if quantification is performed early.

### 1.3 Organization

In this thesis, we propose techniques for Boolean synthesis and evaluate their performance using BDDs as the underlying data structure. We implement these algorithms into a tool called `RSYNTH`, which we evaluate against non-BDD-based tools for Boolean synthesis.

In Chapter 2, we introduce definitions and notation that will be used throughout this thesis.

In Chapter 3, we review related work, survey different approaches developed for Boolean synthesis in the past, and identify competing tools.

In Chapter 4 we formally define the Boolean functional synthesis problem and present a technique called *Self-Substitution* that can be used to solve this problem. We also provide a specialization of this technique to BDDs following a specific variable ordering. We then compare our BDD-based implementation with pre-existing tools and techniques.

In Chapter 5, we show how the performance of the synthesis algorithm can be improved by applying techniques originated in the model-checking literature for factoring a relation into several smaller BDDs. We extend our experimental evaluation by comparing against other tools that operate on factored formulas.

Finally, Chapter 6 presents concluding remarks and future work.

## Chapter 2

### Preliminaries

We start by presenting basic definitions and notation that will be used throughout this work.

#### 2.1 Boolean Functions

We denote by  $\mathbb{B} = \{0, 1\}$  the set of Boolean values. We use the notation  $\vec{x}$  to represent a Boolean vector  $(x_1, \dots, x_m) \in \mathbb{B}^m$ , for some positive integer  $m$ . For simplicity, we often conflate a Boolean formula  $f$  over Boolean variables  $x_1, \dots, x_m$  with the Boolean function  $f : \mathbb{B}^m \rightarrow \mathbb{B}$  such that  $f(\vec{x}) = 1$  if and only if  $\vec{x}$  is a satisfying assignment of formula  $f$ .

We use  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\rightarrow$ ,  $\leftrightarrow$  and  $\oplus$  to denote the usual Boolean operations. Two formulas  $f$  and  $f'$  are *logically equivalent*, denoted  $f \equiv f'$ , if  $f(\vec{x}) = f'(\vec{x})$  for every assignment  $\vec{x}$ . Given formulas  $f(x_1, \dots, x_m)$  and  $f'(y_1, \dots, y_n)$ , we use  $f[x_i \mapsto f']$  to denote the formula  $f(x_1, \dots, x_{i-1}, f'(y_1, \dots, y_n), x_{i+1}, \dots, x_m)$ , representing the functional composition of  $f$  in variable  $x_i$  with  $f'$ . The sets of variables  $\{x_1, \dots, x_m\}$  and  $\{y_1, \dots, y_n\}$  need not necessarily be disjoint. We say that a variable  $x_i$  is in the *support* of a formula  $f$  if  $x_i$  determines the value of  $f$ , that is,  $f[x_i \mapsto 0] \neq f[x_i \mapsto 1]$ .



## 2.2 Quantified Boolean Formulas

We use  $\forall$  and  $\exists$  to denote universal and existential quantification over Boolean variables. A *Quantified Boolean Formula*, or QBF, is a Boolean formula in which some or all of the variables are universally or existentially quantified. Unless stated otherwise, we assume that all QBFs in this work are in *prenex normal form*, that is, in the form  $Q_1X_1\dots Q_kX_k.f(x_1,\dots,x_m)$ , where  $Q_i \in \{\forall, \exists\}$ , all  $X_i$  are disjoint lists over the variables  $x_1,\dots,x_m$ , and  $f(x_1,\dots,x_m)$  is quantifier-free.

Every QBF can be converted into a logically equivalent quantifier-free formula through a process of *quantifier elimination*. This is often performed through the technique of *Shannon Expansion* [13], using the fact that  $\forall x.f \equiv f[x \mapsto 0] \wedge f[x \mapsto 1]$ , and  $\exists x.f \equiv f[x \mapsto 0] \vee f[x \mapsto 1]$ . Given a QBF in prenex normal form, we can obtain its equivalent quantifier-free formula by eliminating the quantifiers from the inside out.

Given a formula  $f(\vec{x}, y)$ , where  $\vec{x} = (x_1, \dots, x_m)$ , and a function  $g : \mathbb{B}^m \rightarrow \mathbb{B}$ , if  $f(\vec{x}, g(\vec{x})) \equiv \exists y.f(\vec{x}, y)$ , we say that  $g$  is a *witness* for  $y$  in  $f$ . Intuitively, if for a given  $\vec{x}$  there exists a  $y$  that satisfies  $f(\vec{x}, y)$ , then  $g$  provides evidence of this fact by producing such a  $y$ . In this context, witnesses are also called *Skolem functions*. Computing and then substituting a Skolem function in a formula leads to an alternative method for eliminating existential quantifiers. This technique is commonly called *Skolemization*.

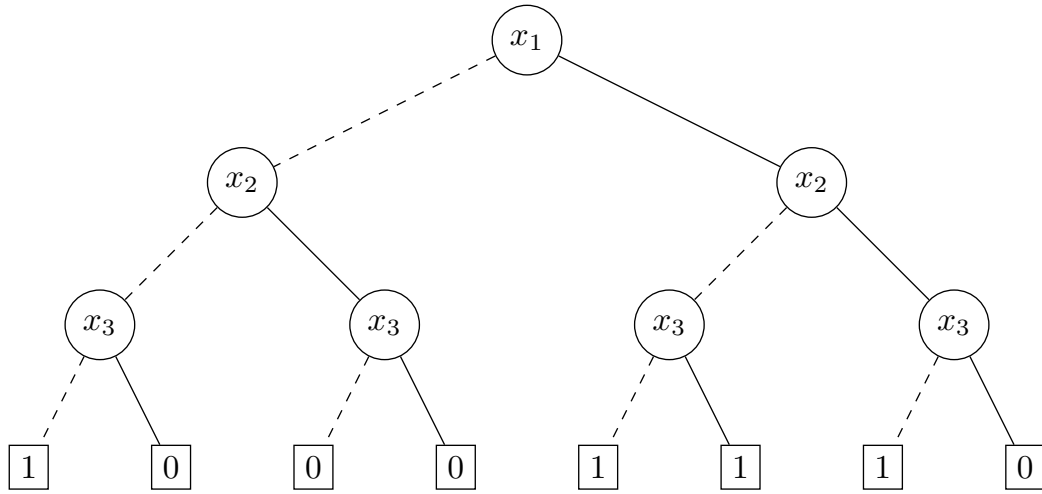
## 2.3 Binary Decision Diagrams

A (*Reduced Ordered*) *Binary Decision Diagram* [13], or BDD for short, is a data structure that represents a Boolean function as a directed acyclic graph consisting of *variable nodes* and *terminal nodes*. A variable node is labeled by a variable in the sup-

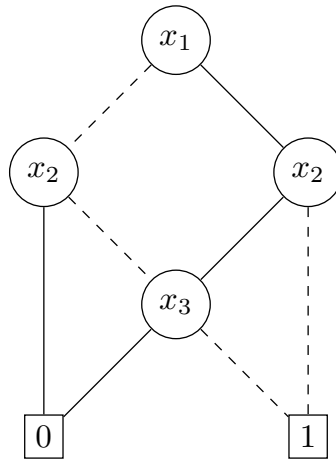
port of the function and has two outgoing edges leading to different subgraphs. These edges are respectively called the *positive* and *negative edge*, with the corresponding subgraph being called the *positive* or *negative child*. In diagrams, we use solid edges to denote positive edges and dashed edges to denote negative edges. Terminal nodes are labeled by either the constant 1 or the constant 0.

To evaluate a BDD on an assignment to the variables, one starts at the root and follows a path down the BDD by taking, at each variable node, the positive edge if the variable is assigned 1 and the negative edge if the variable is assigned 0. The evaluation of the assignment is given by the label of the terminal node that is reached. Since BDDs represent Boolean functions, they can be manipulated using standard Boolean operations. These operations can be implemented using a recursive traversal of the DAG structure of the BDD. We overload the notation of all Boolean operators as well as functional composition (e.g.  $B[x_i \mapsto B']$ ) with equivalent semantics to their counterparts for Boolean formulas.

A BDD can be seen as a reduced representation of a binary decision tree of a Boolean function. While the size of a binary decision tree always is always exponential in the number of variables, a BDD can be significantly smaller. Figure 2.1 compares the binary decision tree representation of a formula with its BDD. Like in a binary decision tree, variables must follow the same ordering along every path of the BDD. In a BDD, however, redundant nodes are eliminated by merging two nodes if they are identical (same label and same children) and removing a node if both its edges point to the same child. These compose the *ordered* and *reduced* properties of BDDs. For a given variable ordering, the reduced BDD is *canonical*, meaning that two BDDs that represent the same function are structurally identical. In practical implementations, identical BDDs can be represented by the same object in memory, making checking



(a)



(b)

Figure 2.1 : Comparison between the binary decision tree and BDD for the formula  $((x_1 \leftrightarrow x_2) \wedge (x_1 \leftrightarrow x_3)) \oplus x_1$ .

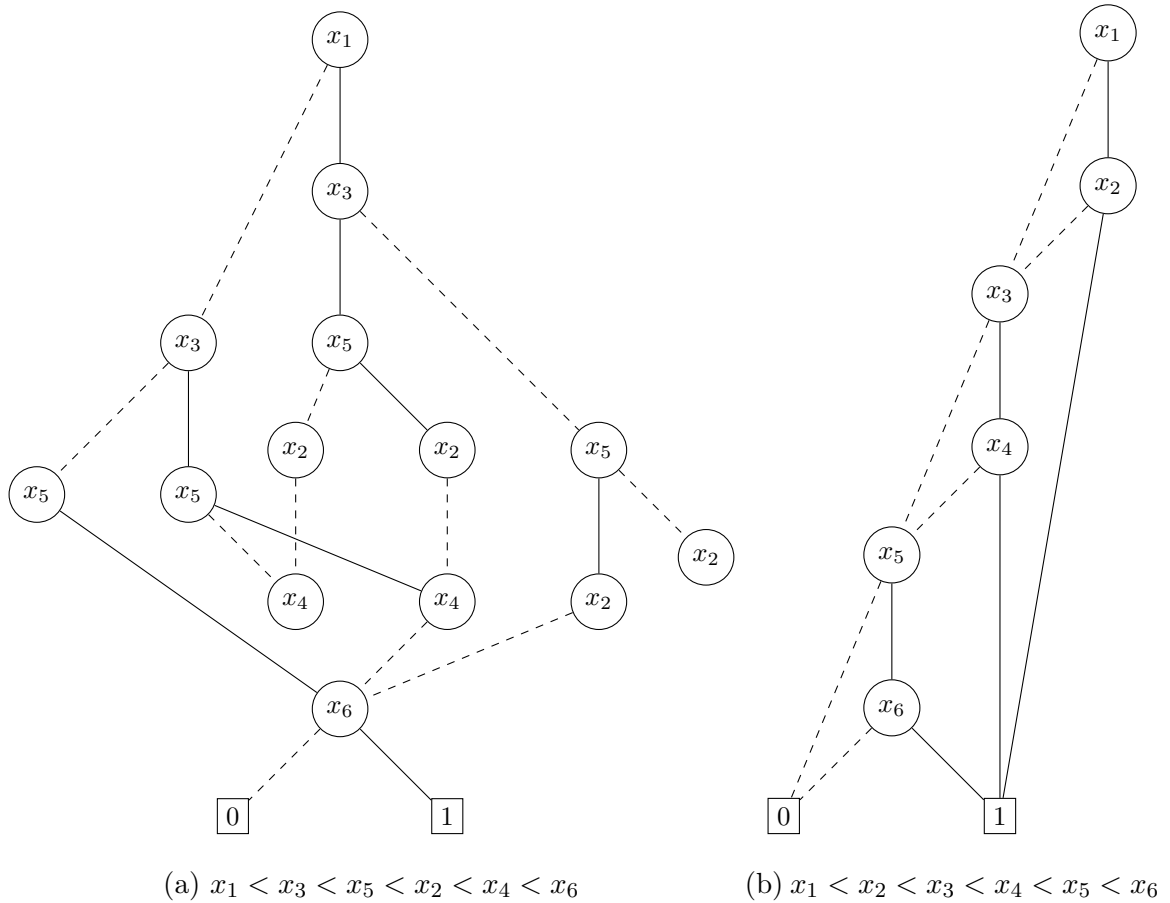


Figure 2.2 : BDDs for the expression  $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$  for two different variable orderings. Missing positive edges are understood to lead to the terminal 1 and missing negative edges to the terminal 0.

equivalence between two BDDs very efficient.

The variable ordering used can have a major impact on the size of a BDD, and two BDDs representing the same function but with different orderings can have an exponential difference in size. Consequently, finding a good variable ordering is essential for BDD-based Boolean reasoning. Figure 2.2 shows the difference between two BDDs representing the same function but with different variable ordering.

## Chapter 3

### Related Work

Due to its wide-ranging applications, including program synthesis, circuit design and QBF certification, Boolean synthesis appears under many guises within and occasionally outside the field of formal methods. In this chapter we present different perspectives on the Boolean synthesis problem and identify previous approaches.

#### 3.1 SAT-based Approaches to Boolean Synthesis

Possibly the clearest application of Boolean synthesis is in the synthesis of hardware systems, since circuits are essentially physical realizations of Boolean functions. In this context, it is natural for techniques commonly used for hardware verification, such as BDDs, And-Inverter Graphs (AIGs) and SAT, to be leveraged for synthesis.

One of the main works in this vein is that of Jiang, Lin and Hung [14]. Due to the limitations of BDDs discussed in Chapter 1, that work chooses to avoid BDDs and instead construct Boolean functions in the form of AIGs using a SAT-based interpolation procedure. Their experimental evaluation shows that this choice of data structure allows functions to be constructed for cases where the relation cannot be efficiently represented by BDDs. However, the same experiments showed that, when the BDD for the relation could be constructed within the time and memory constraints, the resulting functions were often smaller and had fewer logic levels. In Chapter 4 we perform an experimental comparison with an implementation of this

interpolation-based approach.

A more recent work by John, et al. [15] follows the previous approach by Jiang, et al. in using SAT-based techniques and AIGs to represent Boolean functions. However, they also attempt to take advantage of factored formulas for synthesis. Compared to the work in this thesis, their approach uses a different solution to the problem of performing existential quantification on factored formulas. First, functions are synthesized independently for each factor. Although these functions satisfy the factors from which they were extracted, they might not satisfy the entire formula. These functions are then given as input to a counterexample-guided abstraction refinement (CEGAR) loop, which combines the individual functions into a function satisfying the entire formula. This CEGAR loop makes repeated calls to a SAT solver, which can have a high cost in running time. In contrast, BDD-based approaches do not require SAT calls, and BDDs can be very compact for small formulas, which leads to the question of whether they can perform better and produce smaller functions than AIGs when using factored representation. Therefore, that work forms the main baseline of comparison in Chapter 5, where we focus on techniques for factored representation.

In their experimental comparison to other works, John, et al. also bring attention to an alternative approach to Boolean synthesis. In this approach, Boolean functional synthesis is interpreted as solving a quantified Boolean formula, allowing recent developments in solvers for such formulas to be leveraged for this problem. The following section explores this connection.

## 3.2 Skolem Functions in QBF

The original view of synthesis as the process of deriving a constructive proof for a theorem of the form  $\forall x.\exists y.f(x,y)$  suggests a connection between Boolean synthesis

and solving Quantified Boolean Formulas (QBF). In fact, in the same way that SAT solvers can emit a satisfying assignment to the variables of a satisfiable formula, many modern QBF solvers are able, when a quantified formula is true, to produce witnesses in the form of *Skolem functions* [16, 17, 18]. As explained in Section 2.2, these are functions for each of the existential variables in terms of the universal variables, such that replacing each variable for its corresponding function satisfies the formula.

In the QBF literature, the primary importance of Skolem functions is as certificates which can be used to validate the truth of the QBF formula. However, the ability of emitting Skolem functions allows QBF solvers to effectively be employed as synthesizers for Boolean functions. The evaluation by John, et al. shows that when used for this purpose they can often be very efficient. Additionally, QBF solvers are not necessarily limited to formulas of the form  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y})$ , but are able to solve and construct Skolem functions for formulas with arbitrary combinations of quantifiers, although at a higher cost as the number of quantifier alternations increases. However, QBF solvers have the limitation that they can produce Skolem functions only if the quantified formula is true, meaning that for every assignment of the universal variables there is a satisfying assignment to the existential variables. This is a problem for synthesis, since specifications can often have unsatisfiable corner cases, that is, particular inputs for which no output can satisfy the formula. For such specifications, we would still like to construct a (partial) function that works on all satisfiable inputs.

The approach presented in this thesis has no such limitations. Rather, the relation between inputs and outputs given as specification to the synthesis algorithm can be *unrealizable*, that is, the relation can be such that for some inputs there is no satisfying output. In this case, the synthesized function is still guaranteed to be correct for all inputs that are satisfiable. In addition, the synthesis procedure also

produces a *precondition*, a formula that defines exactly the set of inputs for which the specification can be satisfied. This precondition can be used to test inputs before they are passed to the function, guaranteeing that invalid cases do not occur.

QBF solvers, along with BDDs, AIGs, SAT and factored formulas, are familiar tools in formal methods. These are techniques that are widely employed in the field not only for synthesis but also for a number of different verification problems. However, a careful investigation reveals that the problem of finding Boolean functions that satisfy a formula had previously appeared independently from formal methods in the field of symbolic computation, under the name of *Boolean unification*. The next section concludes this chapter by presenting this alternative point of view of the problem, and how it can be explored for synthesis applications.

### 3.3 Boolean Unification

The problem of *Boolean unification* [19] is concerned with finding the solutions to a Boolean equation of the form  $f(y_1, \dots, y_n) = 0$ . If  $f$  is allowed to contain constant symbols  $x_1, \dots, x_m$ , then finding a solution is the same as synthesizing expressions for  $y_1, \dots, y_n$  in terms of  $x_1, \dots, x_m$ . Since every Boolean formula can be written in this equational form, Boolean unification is in a sense equivalent to Boolean functional synthesis.

In contrast with works originating from the formal methods community, works on Boolean unification are less concerned with scaling their techniques to large formulas as required for practical application in circuit synthesis, for example. There have been attempts to directly apply techniques from Boolean unification to synthesis [20], but so far these two fields have largely remained separate.

Nevertheless, many of the basic concepts and algorithms in one field have analogs



in the other, which makes translating between the two straightforward. Furthermore, research in Boolean unification has revealed many theoretical insights that can be leveraged for synthesis applications. For example, Boolean unification usually attempts to generate not only one solution to a Boolean equation, but the set of all solutions. Methods for constructing and representing such sets can be used to synthesize programs that enumerate all values that satisfy a Boolean expression. Boolean unification algorithms have also been generalized beyond Boolean values to a wider class of structures called Boolean rings. This insight might be a key for translating Boolean-synthesis techniques to more general domains. Therefore, the theory behind Boolean unification might lead to interesting directions of future research in Boolean synthesis and generalizations of the problem.

## Chapter 4

### Boolean Synthesis by Self-Substitution

In this chapter, we follow a framework proposed in [14, 21] for algorithmically synthesizing a correct-by-construction implementation of a desired Boolean function from a relational specification. This relation is given as a propositional formula that relates input and output variables. Our construction ensures that the function that we synthesize produces the correct output for every input for which a corresponding output exists. Our framework consists of two phases. The first phase is the *realizability* phase, which includes the computation of the precondition  $p$ . The second phase is the *function-construction* phase, in which we construct the function  $g$  implementing the specification.

The proposed framework in [14] is based on representing Boolean functions by means of And-Inverter Graphs (AIGs) [22]. In this work we adapt this framework to the setting of Reduced Ordered Binary Decision Diagrams (BDDs) [13] BDDs provide easy-to-manipulate canonical (and minimal) representations of Boolean functions in which Boolean operations can be implemented efficiently. BDDs have found numerous applications in a variety of settings, including model checking [23], equivalence checking [24], and others. As mentioned in Chapter 1, our main motivation for using BDDs is that Boolean functional synthesis is also a basic step in *temporal synthesis*, a fundamental technique for constructing control software/hardware from temporal specifications [7], which is most often implemented by using BDDs, cf. [25]. Thus, our approach can be easily incorporated to temporal-synthesis tools. We discuss the

differences between the AIG-based and BDD-based approaches below.

At the heart of our approach is a technique we call *Self-Substitution*, a simplification of the Craig Interpolation-based approach that appears in [14]. A single-step Self-Substitution enables us to extract a function  $g$  *syntactically* from the function  $f$  for the case in which there is a single output variable. When there are multiple output variables, we iterate the single-step for each of the output variables. We can use Self-Substitution in this way both for quantifier elimination, in the realizability phase, and for constructing the function  $g$ , one output variable at a time. We use the CUDD [26] package for our implementation, and show that Self-Substitution can be efficiently implemented through basic BDD operations by using the CUDD API. Thus, Self-Substitution provides a novel way to perform quantifier elimination for BDDs, where the standard technique has been *Shannon Expansion* [13].

We begin the synthesis process by converting the relational specification  $f$  into a BDD. To obtain  $p$  we quantify out the output variables existentially one by one, which can be done by either Shannon Expansion or Self-Substitution. Eliminating the output variables one by one yields a *realizability sequence*  $f_n, \dots, f_0$  of BDDs, where  $f_n$  is the specification  $f$ , and  $f_0$  is the desired precondition  $p$ . In the function-construction phase again we use Self-Substitution, leveraging the realizability sequence  $f_1, \dots, f_n$  to construct a function, represented as a BDD, for each output variable. At the end, we obtain an  $n$ -rooted BDD for the implementation function  $g$ , where each root represents a single output variable. Motivated by [8], we also study Self-Substitution on a specific BDD order called *input-first*. In input-first BDDs, all input variables precede output variables. We develop a novel method, TRIMSUBSTITUTE, which tailors Self-Substitution to input-first BDDs.

Our experimental evaluation relies on *scalable* problem instances rather than a

random collection of problem instances, so we can evaluate the scalability of our techniques. Our evaluations demonstrate that the proposed framework scales well when the problem admits a good variable order, which is a well-established property of BDDs [13]. Our comparisons also showed that our method outperforms the previous AIG-based approach and other state-of-the-art tools by orders of magnitude. We also compare Self-Substitution as a quantifier-elimination technique against the standard Shannon-Expansion technique, and show that in many cases Self-Substitution scales better than Shannon Expansion. In addition, we show that TRIMSUBSTITUTE outperforms Self-Substitution on input-first BDDs.

The contributions of this chapter are as follows: We offer a BDD-based approach for Boolean Synthesis that is simple and requires only basic BDD procedures. We show that our method outperforms other synthesis tools on scalable benchmarks. Our method also suggests a novel way for BDD-based quantifier elimination. In addition we also offer a technique for input-first BDD in which we tailor our method specifically to this BDD order and show that we outperform all others tools.

## 4.1 Realizability and Synthesis

The problem of synthesis of Boolean functions is formally defined as follows.

**Problem 4.1.** *Given a relation between two vectors of Boolean variables represented by the characteristic function  $f : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}$ , obtain a function  $p : \mathbb{B}^m \rightarrow \mathbb{B}$  such that  $p(\vec{x}) = 1$  exactly for the inputs  $\vec{x} \in \mathbb{B}^m$  for which  $\exists \vec{y}. f(\vec{x}, \vec{y})$ , and a function  $g : \mathbb{B}^m \rightarrow \mathbb{B}^n$  such that  $f(\vec{x}, g(\vec{x})) = 1$  if and only if  $p(\vec{x}) = 1$ .*

In the context of this problem,  $f$  is called the *specification*,  $g$  is called the *implementation* or *witness function*, and  $p$  is called a *precondition*. The specification

is interpreted as describing a desired relationship between inputs and outputs of a function, and the implementation describes how to obtain an output from an input such that this relationship is maintained. The precondition indicates for which inputs the specification can be satisfied. In the expression  $f(\vec{x}, \vec{y})$ ,  $\vec{x} = (x_1, \dots, x_m)$  are called the *input variables*, and  $\vec{y} = (y_1, \dots, y_n)$  the *output variables*. The function that gives the  $i$ -th bit of  $g(\vec{x})$  is called a *witness-bit function*, and is denoted by  $g_i(\vec{x})$ . A Boolean function  $f(\vec{x}, \vec{y})$  is said to be *realizable for an input  $\vec{\sigma}$*  if  $\exists \vec{y}. f(\vec{\sigma}, \vec{y}) \equiv 1$ . We say that  $f$  is *realizable* if  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y}) \equiv 1$ . For every assignment  $\vec{\sigma}$  for  $\vec{x}$  such that  $f$  is realizable for  $\vec{\sigma}$  we will have that  $p(\vec{\sigma}) = 1$ . In this case,  $g(\vec{\sigma})$  is called a *witness* for  $\vec{\sigma}$ . In case  $p(\vec{\sigma}) = 0$  we are not concerned about the output of  $g$  since  $f$  is unrealizable for  $\vec{\sigma}$ .

Following [14], the structure of our solution takes two steps, 1) Realizability, where we obtain  $p$  by constructing a sequence of formulas with progressively fewer output variables, and 2) Function construction, in which we synthesize a witness-bit function from every formula in the sequence obtained in the realizability step.

To perform both steps, we suggest a novel method called *Self-Substitution*. In Chapter 2 we observed that Boolean quantifier elimination is usually performed via Shannon Expansion. More recently, it was proposed to use Craig Interpolation for quantifier elimination (see [21]). Other quantifier elimination techniques (see [27]) require the formulas to be in CNF form, rather than the BDD representation we use. We now introduce Self-Substitution as an alternative quantifier-elimination technique.

**Lemma 4.1.** (*Self-Substitution for Quantifier Elimination*) *Let  $\varphi = Qy.f(\vec{x}, y)$  be a QBF formula, where  $Q$  is either a universal or existential quantifier and  $f$  is quantifier-free. Let  $q$  be 0 if  $Q$  is universal and 1 if  $Q$  is existential. Then,  $Qy.f(\vec{x}, y)$  is logically equivalent to  $f(\vec{x}, f(\vec{x}, q))$ , and is also logically equivalent to  $f(\vec{x}, \neg f(\vec{x}, \neg q))$ .*

*Proof.* If  $Q$  is an existential quantifier, we prove that for every assignment  $\vec{\sigma}$  for  $\vec{x}$ ,  $\exists y.f(\vec{\sigma}, y) = 0$  iff  $f(\vec{\sigma}, f(\vec{\sigma}, 1)) = 0$ : If  $\exists y.f(\vec{\sigma}, y) = 0$ , then  $f(\vec{\sigma}, y) = 0$  for all possible assignments of  $y$ . Since this includes  $f(\vec{\sigma}, 1)$ , then  $f(\vec{\sigma}, f(\vec{\sigma}, 1)) = 0$ . On the other hand, if  $f(\vec{\sigma}, f(\vec{\sigma}, 1)) = 0$ , then it cannot be the case that  $f(\vec{\sigma}, 1) = 1$  (otherwise  $f(\vec{\sigma}, f(\vec{\sigma}, 1)) = f(\vec{\sigma}, 1) = 1$ ). Therefore,  $f(\vec{\sigma}, 1) = 0$ , and so  $f(\vec{\sigma}, 0) = f(\vec{\sigma}, f(\vec{\sigma}, 1)) = 0$ . Since both  $f(\vec{\sigma}, 1) = 0$  and  $f(\vec{\sigma}, 0) = 0$ , then  $\exists y.f(\vec{\sigma}, y) = 0$ . The claim that for every assignment  $\vec{\sigma}$ ,  $\exists y.f(\vec{\sigma}, y) = 0$  iff  $f(\vec{\sigma}, \neg f(\vec{\sigma}, 0)) = 0$  is proved analogously. The proof when  $Q$  is a universal quantifier is derived by using the identity  $\forall y.f(\vec{x}, y) \equiv \neg \exists y.\neg f(\vec{x}, y)$ .  $\square$

Following Lemma 4.1, quantifier elimination can be performed by replacing quantified formulas by their quantifier-free equivalents. Table 4.1 compares the formulas produced by quantifier elimination using Shannon Expansion and Self-Substitution.

The Self-Substitution method looks surprising at first glance. In the Shannon-Expansion method it is easy to see that the size of the quantifier-free formula becomes exponential compared to its quantified version, as it is a disjunction of all possible assignments. In Self-Substitution such a blowup also takes place, but the encapsulation of all assignments is more subtle. The depth of the nested functions for a formula with  $n$  quantified variables is  $n + 1$ . Therefore all the possible assignments for the quantified variables can be obtained recursively. For example let  $q_i = 1$  if the quantifier  $Q_i$  is existential, and  $q_i = 0$  if  $Q_i$  is universal. Then a possible expansion for two quantified variables is  $Q_1 y_1. Q_2 y_2. f(\vec{x}, y_1, y_2) = Q_1. y_1 f(\vec{x}, y_1, f(\vec{x}, y_1, q_2)) = f(\vec{x}, f(\vec{x}, q_1, f(\vec{x}, q_1, q_2)), f(\vec{x}, f(\vec{x}, q_1, f(\vec{x}, q_1, q_2)), q_2))$ .

The following lemma which appeared in many forms in various places, e.g. [28, 14, 9], is derived from Lemma 4.1 and shows how Self-Substitution can be used for synthesis purposes.

Table 4.1 : Equivalent formulas using each method of quantifier elimination

	$\forall y.f(\vec{x}, y)$	$\exists y.f(\vec{x}, y)$
Shannon Expansion	$f(\vec{x}, 0) \wedge f(\vec{x}, 1)$	$f(\vec{x}, 0) \vee f(\vec{x}, 1)$
Self-Substitution 1	$f(\vec{x}, f(\vec{x}, 0))$	$f(\vec{x}, f(\vec{x}, 1))$
Self-Substitution 2	$f(x, \neg f(\vec{x}, 1))$	$f(\vec{x}, \neg f(\vec{x}, 0))$

**Lemma 4.2.** (*Synthesis by Self-Substitution*) Let  $f(\vec{x}, y)$  be a Boolean formula with free variables  $\vec{x}$  and  $y$ . Then  $f(\vec{x}, 1)$  and  $\neg f(\vec{x}, 0)$  are witness functions to  $f(\vec{x}, y)$ .

*Proof.* By Lemma 1,  $\exists y.f(\vec{x}, y)$  is logically equivalent to  $f(\vec{x}, f(\vec{x}, 1))$  and to  $f(\vec{x}, \neg f(\vec{x}, 0))$ . Since  $p(\vec{x}) = 1$  exactly for those  $\vec{x}$  for which  $\exists y.f(\vec{x}, y)$  holds, both  $f(\vec{x}, f(\vec{x}, 1))$  and  $f(\vec{x}, \neg f(\vec{x}, 0))$  return 1 if and only if  $p(\vec{x}) = 1$ . Thus,  $f(\vec{x}, 1)$  and  $\neg f(\vec{x}, 0)$  are witness functions to  $f(\vec{x}, y)$ .  $\square$

The witness  $f(\vec{x}, 1)$  is called the *default-1* witness, while the witness  $\neg f(\vec{x}, 0)$  is called the *default-0* witness. The observation in [14] is that when  $f$  is realizable for all  $\vec{x}$ , the conjunction of the two formulas  $\neg f(\vec{x}, 0)$  and  $\neg f(\vec{x}, 1)$  is unsatisfiable. From a resolution proof of this unsatisfiability, one can extract a Craig interpolant, which may be smaller than either  $f(\vec{x}, 1)$  or  $\neg f(\vec{x}, 0)$ . Our experimental evaluation for our benchmarks does not support this expectation, where we show the advantage of using the witness function  $f(\vec{x}, 1)$  for synthesis and  $f(\vec{x}, f(\vec{x}, 1))$  for existential-quantifier elimination.

## 4.2 Realizability and Function-Construction Using BDDs

Similarly to [14], we separate the synthesis approach into two phases. We call the first the *realizability* phase, and the second the *function construction* phase. We assume that the input is in the form of a BDD  $B_f$  that describes the function  $f(\vec{x}, \vec{y})$ . When  $f$  is obvious from the reference, we denote  $B_f$  by  $B$ .

### 4.2.1 Realizability

Our definition of  $p$  requires that  $p$  returns 1 exactly for those assignments of  $\vec{x}$  for which  $\exists \vec{y}. f(\vec{x}, \vec{y})$ . This means  $p$  can be obtained by applying quantifier elimination on the output variables. Recall that  $f$  has  $n$  output variables  $y_1, \dots, y_n$ . Typically, the order of variables makes a major difference in constructing a BDD. However, in this section we assume no specific order.

The basic idea is as follows: from the input BDD  $B$ , we construct a sequence  $\vec{B} = \langle B_n, B_{n-1}, \dots, B_1, B_0 \rangle$  of BDDs, where  $B_n = B$ , such that  $B_{i-1}$  is logically equivalent to  $\exists y_i. B_i$ . Therefore, the BDD  $B_{i-1}$  is constructed from  $B_i$  by eliminating the existentially quantified variable  $y_i$ . The elimination process guarantees that  $B_0$  represents the precondition  $p$ .

The elimination of  $y_i$  from  $B_i$  can be done via either Shannon Expansion, or via Self-Substitution. For Shannon Expansion, we define  $B_{i-1} = B_i[y_i \mapsto 0] \vee B_i[y_i \mapsto 1]$ . To use the Self-Substitution method, we define either  $B_{i-1} = B_i[y_i \mapsto B_i[y_i \mapsto 1]]$  to construct the default-1 witness for  $y_i$  or  $B_{i-1} = B_i[y_i \mapsto \neg B_i[y_i \mapsto 0]]$  to construct the default-0 witness for  $y_i$ .



### 4.2.2 Function Construction

We next use the BDD sequence obtained in the realizability process to construct a sequence of BDDs  $\vec{W} = \langle W_n, W_{n-1}, \dots, W_1 \rangle$ , each emitting an output bit.

By using Lemma 4.2, we perform the function-construction step as follows. Let  $\vec{B} = \langle B_n, B_{n-1}, \dots, B_1, B_0 \rangle$  be the BDD sequence obtained in the realizability step, and note that the output variables in the BDD  $B_i$  are  $y_1, \dots, y_i$ . We first construct  $W_1$  from  $B_1$  by setting  $W_1 = B_1[y_1 \mapsto 1]$  for a default-1 witness for  $y_1$  or  $W_1 = \neg B_1[y_1 \mapsto 0]$  for a default-0 witness for  $y_1$ . The structure of BDDs allows us to define both  $B_1$  and  $\neg B_1$  without extra effort. Next, we inductively define either  $W_i = B_i[y_1 \mapsto W_1, \dots, y_{i-1} \mapsto W_{i-1}, y_i \mapsto 1]$  for a default 1 witness for  $y_i$ , or  $W_i = \neg B_i[y_1 \mapsto W_1, \dots, y_{i-1} \mapsto W_{i-1}, y_i \mapsto 0]$  for a default 0 witness for  $y_i$ . Thus, every  $W_i$  has only the input variables  $\vec{x}$ , and represents the witness-bit function  $g_i(\vec{x})$ . Thus, the proof for the following theorem follows from Lemma 4.2.

**Theorem 4.1.** *For every assignment  $\vec{\sigma}$  for  $\vec{x}$ , the sequence  $\langle g_1(\vec{\sigma}), \dots, g_n(\vec{\sigma}) \rangle$  is a witness to  $\vec{\sigma}$ . Thus  $\vec{W}$  describes a witness function for  $B$ .*

In practice, we chose, for simplicity, to use only the default-1 witnesses. In principle, one could always choose the best among the default-0 and default-1 witnesses. Since, however, we have  $n$  output variables, and the assignment of one of them affects the others, finding the optimal combination of bit-witness functions requires optimizing over an exponentially large space, which is an expensive undertaking. Finding such combinations of functions is a matter of future work.

### 4.3 Synthesis of Input-First BDD

An *input-first* BDD is a BDD in which all the input variables precede all the output variables. Synthesis using input-first BDDs was suggested in [8], but an explicit way to do it was not provided. This specific order of variables of input-first BDDs has led us to develop a method called TRIMSUBSTITUTE for synthesis of input-first BDDs, in which we tailor Self-Substitution specifically for the input-first order. Given the input BDD, the running time of TRIMSUBSTITUTE is at most quadratic in its size. In Section 4.5 we show that TRIMSUBSTITUTE indeed outperforms Self-Substitution on input-first BDDs. In this section we give an outline of our method, together with a proof of correctness and example. For simplicity TRIMSUBSTITUTE produces default-1 witnesses. With minor modification the TRIMSUBSTITUTE method can produce any desired combination of bit-witness functions.

An *output node* (resp. *input node*) in a BDD  $B$  is a node labeled with an output (resp. input) variable. Recall that every non-terminal node in  $B$  has exactly two children called *high-child* and *low-child*. Let  $B$  be an input-first BDD. We define  $Fringe(B)$  to be the collection of all output nodes and terminal nodes in  $B$  that have an input node as an immediate parent.  $Fringe(B)$  can be found by performing standard graph-search operations (e.g. Depth-First-Search) on  $B$ . Note that every assignment to the input variables defines a node in  $Fringe(B)$  obtained by following that assignment.  $B$  is realizable exactly for those assignments for which the corresponding node in  $Fringe(B)$  is not the terminal node 0.

Given an input-first BDD  $B$ , we assume without loss of generality that the order of the output variables in  $B$  is  $y_1, \dots, y_n$ . We construct a sequence of witness BDDs  $\vec{W} = \langle W_1, \dots, W_n \rangle$ , in which every  $W_i$  contains only input variables, and represents the witness-bit function  $g_i(\vec{x})$ . To obtain  $\vec{W}$ , we first construct a sequence of BDDs

$\vec{B}' = \langle B'_1, \dots, B'_n \rangle$  in which every  $B'_i$  is an input-first BDD that contains all input variables, plus only output variables from  $y_i, \dots, y_n$ . We obtain  $W_i$  from  $B'_i$  by an operation called “trim”, and obtain  $B'_{i+1}$  from  $B'_i$  and  $W'_i$  by an operation called “substitute”, hence our method’s name TRIMSUBSTITUTE. We next describe how  $\vec{B}'$  and  $\vec{W}$  are obtained.

We assume by induction on  $i \leq n$  that  $B'_i$  is an input-first BDD that is realizable for exactly the same inputs as  $B$ , and that contains input variables plus only output variables from  $y_i, \dots, y_n$ . Setting  $B'_1 = B$ , we already satisfy these assumptions for the base case. We first construct  $W_i$  by “trimming”  $B'_i$ , which means replacing each node  $v$  in  $Fringe(B'_i)$  with either the terminal node 0 or 1. Intuitively we construct  $W_i$  to produce an output bit for  $y_i$  in the “default-1” sense, i.e.,  $W_i$  always produces 1 unless 1 is not a possible output bit for  $y_i$ . Formally, this is done as follows.

Note that if a  $v \in Fringe(B'_i)$  is the terminal node 0, then the assignment to  $y_i$  is irrelevant since the path to  $v$  corresponds to an unrealizable input, and so it can be left as 0. If  $v$  is a variable node, it cannot be that both children of  $v$  are the terminal node 0, as otherwise  $v$  itself would be reduced to 0. Therefore, if  $v$  is labeled by  $y_i$ , and the high-child of  $v$  is not the terminal node 0, replace  $v$  with the terminal node 1. Otherwise, if  $v$  is labeled by  $y_i$  and the high-child of  $v$  is 0 (then the low-child of  $v$  is not 0), replace  $v$  with the terminal node 0. For all other cases ( $v$  is labeled  $y_j$ , where  $j > i$ , or  $v$  is the terminal node 1), replace  $v$  with the terminal node 1. Note that  $W_i$  has only input variables.

Finally, we use  $B'_i$  and  $W_i$  to construct  $B'_{i+1}$ . To do that, we define  $B'_{i+1} = B'_i[y_i \mapsto W_i]$ . That is,  $B'_{i+1}$  is constructed from  $B'_i$  by “substituting”  $y_i$  with  $W_i$ . By construction we have that  $B'_{i+1}$  is an input-first BDD that is realizable for the same inputs as  $B$  and that contains input variables plus only output variables from

$y_{i+1}, \dots, y_n$ . Therefore, the induction assumption is maintained. An example of the construction can be found in Figure 4.1.

We now prove the correctness of the TRIMSUBSTITUTE method. Let  $B$  be an input-first BDD, and let  $\vec{B}' = \langle B'_1, \dots, B'_n \rangle$  and  $\vec{W} = \langle W_1, \dots, W_n \rangle$  as defined above. Given a BDD  $D$ , and a node  $v$  in  $D$ , let  $D_v$  be the subgraph of  $D$  rooted on  $v$ . Assume  $z_1, \dots, z_k$  are the variables of  $D$ . Then by following a partial assignment  $\vec{v}$  to the variables of  $z_1, \dots, z_i$  for some  $i$ , we follow a unique path in  $D$  that ends up in a node  $v$ . Then the subgraph  $D_v$  is called the subgraph *reached* by following  $\vec{v}$  in  $D$ .

**Theorem 4.2.** *The BDD sequence  $\vec{W} = (W_1, \dots, W_n)$  describes a witness function for  $B$ .*

*Proof.* Let  $g_i : \mathbb{B}^m \rightarrow \mathbb{B}$  be the function that describes  $W_i$ . The following facts are easily proved by induction on  $i$ .

1. Following the construction of  $W_i$ , for every realizable assignment  $\vec{\sigma}$  to the input variables, the path followed by  $(\vec{\sigma}, g_i(\vec{\sigma}))$  in  $B'_i$  does not end in the terminal node 0.
2. Following fact (1), and the construction of  $B'_{i+1}$ , we have that for every realizable assignment  $\vec{\sigma}$  to the input variables, the subgraph reached by following  $\vec{\sigma}$  in  $B'_{i+1}$  is identical to the subgraph reached by following  $(\vec{\sigma}, g_i(\vec{\sigma}))$  in  $B'_i$ . Therefore  $B'_{i+1}$  is realizable for  $\vec{\sigma}$  as well.

As a result, we specifically have that for every realizable assignment  $\vec{\sigma}$  to the input variables, the assignment  $(\vec{\sigma}, g_1(\vec{\sigma}), \dots, g_n(\vec{\sigma}))$  leads to the terminal node 1. This means that the BDD sequence  $\vec{W} = (W_1, \dots, W_n)$  describes a witness function for  $B$ . □

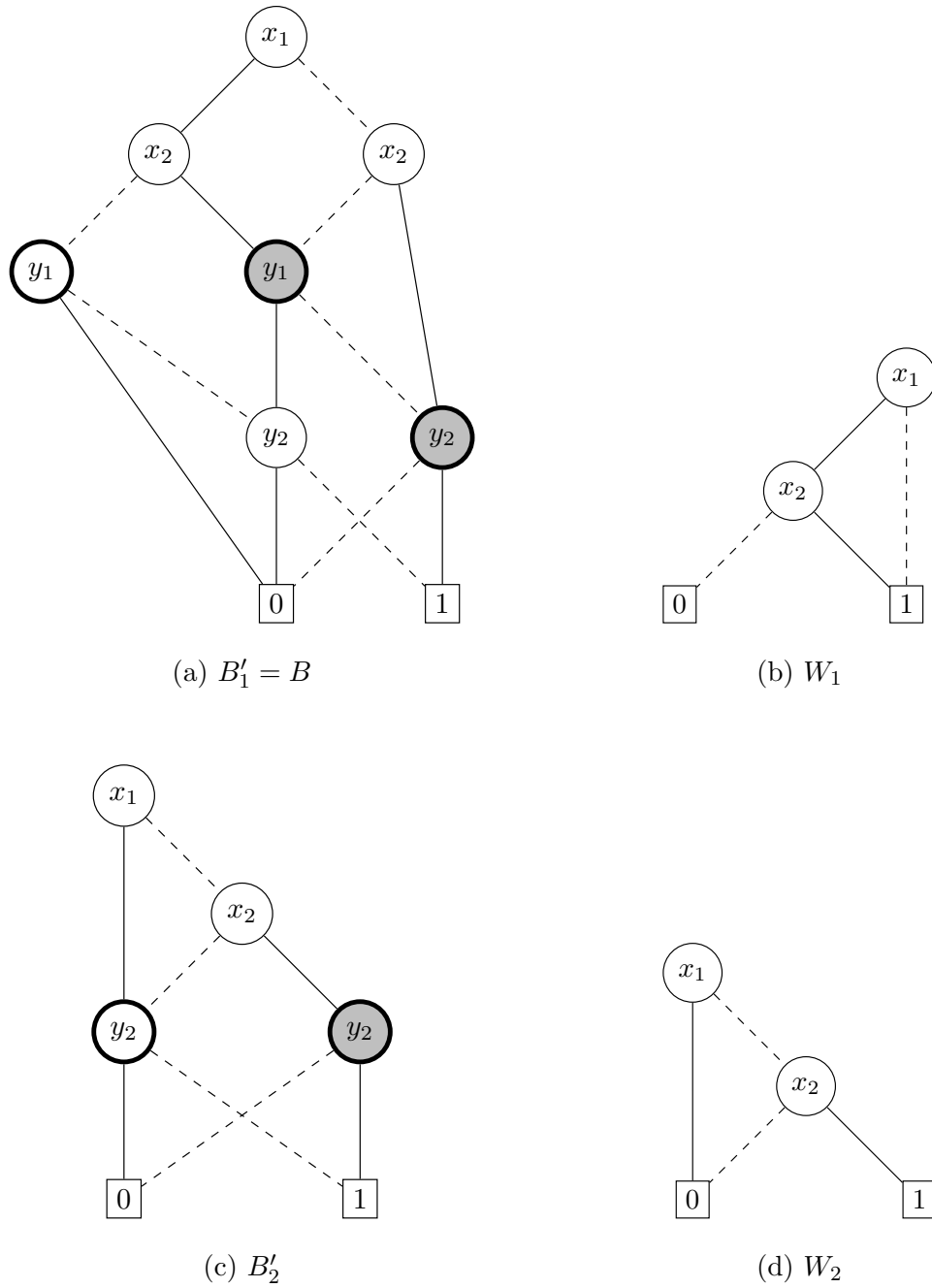


Figure 4.1 : Example of the TRIMSUBSTITUTE method for a BDD representing the formula  $((x_1 \rightarrow \neg y_1) \wedge (x_1 \oplus x_2) \wedge (x_1 \oplus y_2)) \vee ((x_1 \leftrightarrow x_2) \wedge (y_1 \oplus y_2))$ . Nodes with bold outlines are in  $Fringe(B'_i)$ , and are either white if they should be replaced by the leaf node 0 or gray if they should be replaced by the leaf node 1.

In the last induction step we obtain an additional BDD  $B'_{n+1}$  which is realizable for the same inputs as  $B$ , but contains only input variables. As such,  $B'_{n+1}$  encodes the precondition  $p$ .

## 4.4 Benchmark Selection

Rather than using a random collection of problem instances for our experiments, we selected a collection of *scalable* benchmarks, presented in Table 4.2, that operate over vectors of Boolean variables. Each entry in the table represents a class of benchmarks parameterized by the length  $n$  of the vectors. This allows us to produce benchmarks of different size to measure how our techniques scale. For our experiments we vary  $n$  in powers of 2 between 8 and 1024, totaling 42 benchmark instances. The first five benchmark classes represent linear-arithmetic functions in which the vectors encode the binary representation of integers in  $n$  bits, while the sixth represents the sorting of a bit array of size  $n$ . The first column in Table 4.2 describes the function we synthesize, where  $\vec{x}$  and  $\vec{x}'$  are vectors of input variables and  $\vec{y}$  is a vector of output variables. The relational specification of these functions are shown in the second column. These specifications are translated to propositional-logic formulas and given as input to the algorithm, which then constructs a BDD for the relational specification and synthesizes the implementation function. All benchmarks are realizable for every input, therefore the precondition is the constant function 1.

For completeness we show how to encode the specifications given in Table 4.2 into propositional logic formulas (later represented as a BDD). We assume that an integer is described by a vector of variables  $\vec{z} = (z_n, z_{n-1}, \dots, z_2, z_1)$ , where  $z_n$  represents the most significant bit and  $z_1$  the least significant bit. We now describe how specific operations used in the high-level specifications are encoded in propositional logic.

Table 4.2 : Benchmark classes used for synthesis.

	Function to synthesize	Specification
Subtraction	$\vec{y} = \vec{x}' - \vec{x}$	$\vec{y} + \vec{x} = \vec{x}'$
Maximum	$\vec{y} = \max(\vec{x}, \vec{x}')$	$(\vec{y} \geq \vec{x}) \wedge (\vec{y} \geq \vec{x}') \wedge ((\vec{y} = \vec{x}) \vee (\vec{y} = \vec{x}'))$
Minimum	$\vec{y} = \min(\vec{x}, \vec{x}')$	$(\vec{y} \leq \vec{x}) \wedge (\vec{y} \leq \vec{x}') \wedge ((\vec{y} = \vec{x}) \vee (\vec{y} = \vec{x}'))$
Floor of average	$\vec{y} = \left\lfloor \frac{\vec{x} + \vec{x}'}{2} \right\rfloor$	$(2\vec{y} = \vec{x} + \vec{x}') \vee (2\vec{y} + 1 = \vec{x} + \vec{x}')$
Ceiling of average	$\vec{y} = \left\lceil \frac{\vec{x} + \vec{x}'}{2} \right\rceil$	$(2\vec{y} = \vec{x} + \vec{x}') \vee (2\vec{y} = \vec{x} + \vec{x}' + 1)$
Sorting	$\vec{y} = \text{sort}(\vec{x})$	$\text{sorted}(\vec{y}) \wedge (\sum_{i=1}^n x_i = \sum_{j=1}^n y_j)$

#### 4.4.1 Relational Operations

The formulas  $(z = z')$ ,  $(z \leq z')$  and  $(z \geq z')$  are encoded respectively as  $\varphi^=$ ,  $\varphi^{\leq}$  and  $\varphi^{\geq}$ , as follows:

$$\varphi^= := \bigwedge_{i=1}^n (z_i \leftrightarrow z'_i) \quad (4.1)$$

$$\varphi^{\leq} := \varphi_n, \text{ where } \varphi_i := (\neg z_i \wedge z'_i) \vee ((z_i \leftrightarrow z'_i) \wedge \varphi_{i-1}) \text{ and } \varphi_0 := 1 \quad (4.2)$$

$$\varphi^{\geq} := \varphi_n, \text{ where } \varphi_i := (z_i \wedge \neg z'_i) \vee ((z_i \leftrightarrow z'_i) \wedge \varphi_{i-1}) \text{ and } \varphi_0 := 1 \quad (4.3)$$

#### 4.4.2 Addition

Since addition is an operation that returns an integer rather than a Boolean, it cannot be implemented as a single Boolean formula. Rather, it produces  $n$  formulas  $\varphi_n^+, \dots, \varphi_1^+$  representing a new integer, which can be later combined into a single formula through one of the relational operators above. The encoding for the  $+$  operator

follows the usual representation of addition in binary:

$$\varphi_i^+ := z_i \oplus z'_i \oplus c_{i-1}$$

where  $c_i := (z_i \wedge z'_i) \vee (z_i \wedge c_{i-1}) \vee (z'_i \wedge c_{i-1})$ .

In this encoding,  $c_i$  represents the carry-out from the addition in the  $i$ -th position. The carry-in for the first position,  $c_0$ , is normally 0, but can be set to 1 to add an extra term of 1 to the sum, which is useful in the formulas for average.

In the *Subtraction* benchmark class,  $+$  is interpreted as addition modulo  $n$ . On the other hand, in the high-level formulas for average we need the result of the addition with an extra bit added if necessary. This extra bit can be obtained by simply taking  $c_n$ . Therefore the comparisons in these formulas are actually performed over  $(n + 1)$ -bit integers.

#### 4.4.3 Sorting

The specification for the *Sorting* benchmark class requires a more careful encoding. In Table 4.2, its high-level specification is given as  $sorted(\vec{y}) \wedge (\sum_{i=1}^n x_i = \sum_{j=1}^n y_j)$  where  $\vec{x}$  and  $\vec{y}$  are interpreted as bit arrays. The first conjunct says that the output must be sorted, meaning that all 0 bits must precede all 1 bits. This is defined recursively for a range of consecutive positions  $y_i, \dots, y_j$  by saying that either all the variables are assigned to 1, or the first is 0 and the rest are sorted. The function  $sorted(\vec{y})$  is defined as:

$$sorted(y_i, \dots, y_j) = \begin{cases} 1, & \text{if } i = j \\ \left( \bigwedge_{k=i}^j y_k \right) \vee (\neg y_i \wedge sorted(y_{i+1}, \dots, y_j)), & \text{if } i \neq j \end{cases}$$



The second conjunct in the *Sorting* specification says that the output must have the same number of bits set to 1 as the input. In the high-level representation, this can be represented by  $\sum_{i=1}^n x_i = \sum_{j=1}^n y_j$ , but in practice it is not necessary to use summation in the encoding. Instead, the propositional logic formula for this property can be represented by a recurrence and constructed using dynamic programming:

$$\begin{aligned}
 \varphi_{0,0} &= 1 \\
 \varphi_{i,0} &= \neg x_i \wedge \varphi_{i-1,0} \\
 \varphi_{0,j} &= \neg y_j \wedge \varphi_{0,j-1} \\
 \varphi_{i,j} &= ((x_i \leftrightarrow y_j) \wedge \varphi_{i-1,j-1}) \vee (x_i \wedge \neg y_j \wedge \varphi_{i,j-1}) \vee (\neg x_i \wedge y_j \wedge \varphi_{i-1,j}) \quad (4.4)
 \end{aligned}$$

In this encoding,  $\varphi_{i,j}$  means that  $x_1, \dots, x_i$  has the same number of 1s as  $y_1, \dots, y_j$ . This is obtained by matching each bit that is set to 1 in the input with a bit that is set to 1 in the output, and skipping bits that are set to 0.

Note that some of these encodings can be optimized, for example the specification for *Sorting* can be reduced by testing at the same time if the input is sorted and it has the same number of 1s as the output. This can shorten the construction time, but since it is logically equivalent to the original formula, by the canonicity property of BDDs, the resulting BDD for the specification will be the same.

## 4.5 Experimental Evaluation

We compare our approach with two current state-of-the-art methods: the Craig Interpolation-based approach [14] and SKETCH [29]. In addition, we compare between Shannon Expansion and Self-Substitution as quantifier-elimination methods to be used for the realizability phase. Finally, we see how the TRIMSUBSTITUTE method, specialized to input-first BDDs, compares with the generic Self-Substitution

method when using this type of BDD.

For purposes of evaluation we have constructed a tool, called `RSYNTH`, implemented in C++11 using the CUDD BDD library [26]. Self-Substitution was implemented using the built-in method `Compose` for BDD composition. That way, for a BDD  $B$  representing a function  $f(\vec{x}, y)$ , the BDD for  $f(\vec{x}, f(\vec{x}, 1))$  is computed as `B.Compose(i, B.Compose(i, bddOne))`, where `bddOne` is the BDD for the constant 1 and `i` is the index of variable  $y$ . All the experiments in this paper were carried out on a computer cluster consisting of 192 Westmere nodes of 12 processor cores each, running at 2.83 GHz with 4 GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2 GHz with 8 GB of RAM per core. Since the algorithm has not been parallelized, the cluster was used solely to run different experiments simultaneously. The execution of each benchmark for a given  $n$  had a maximum time limit of 8 hours.

#### 4.5.1 Scalability Comparison with Previous Approaches

We compared the performance of `RSYNTH` with the Craig Interpolation approach from [14] that synthesizes functions in the format of AIGs, and the `SKETCH` synthesis tool [29] that uses syntax-guided search-based synthesis. The original tool for Craig Interpolation from [14] was not available, therefore we used an implementation of the same method, which is called `MONOSKOLEM`, from [15].

Since BDD sizes can blow up if a poor variable order is chosen, causing initial BDD construction time to dominate the overall running time, we selected a variable order that can be expected to produce efficient BDDs for our benchmarks. For that, we chose an order called *fully interleaved*, in which the variables are ordered according to their index, alternating input and output variables.

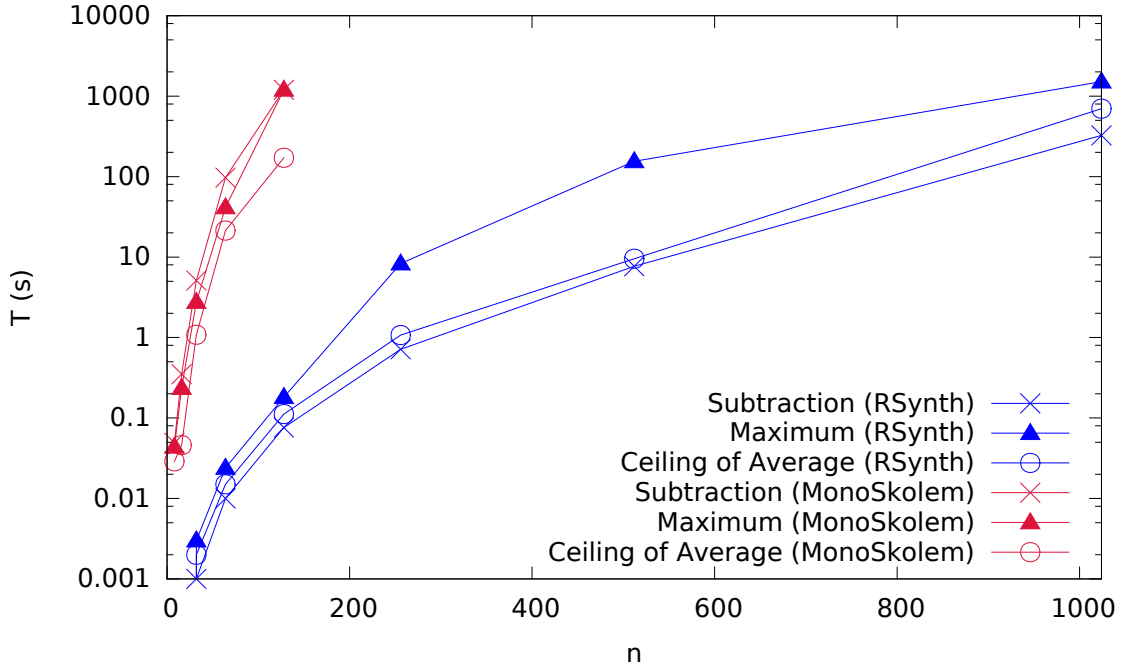


Figure 4.2 : Comparison of running time of RSYNTH against MONOSKOLEM

We show the results of the comparison for the *Subtraction*, *Maximum* and *Ceiling of Average* benchmark classes in Fig. 4.2. Similar results were obtained for the other arithmetic benchmarks. Recall that  $n$  is the number of variables in each vector  $\vec{x}$ ,  $\vec{x}'$  and  $\vec{y}$ , therefore the total number of variables in each case is  $3n$ , with  $2n$  input variables and  $n$  output variables.

SKETCH is omitted from Fig. 4.2 because it was unable to synthesize the benchmarks for any  $n$  greater than 3, in all cases either timing out or running out of memory. For the two remaining approaches, it is noticeable that RSYNTH outperformed MONOSKOLEM by orders of magnitude, and scaled significantly better.

Although these results seem to lean considerably in favor of our approach, note that the benchmark classes used so far are deterministic (relations that have a unique implementation), while Craig Interpolation is reported to produce better results for

Table 4.3 : Non-deterministic benchmark classes

	Input	Output	Specification
Decomposition	$\vec{x}$	$\vec{y}, \vec{y}'$	$\vec{x} = \vec{y} + \vec{y}'$
Equalization	$\vec{x}, \vec{x}'$	$\vec{y}, \vec{y}'$	$\vec{x} + \vec{y} = \vec{x}' + \vec{y}'$
Intermediate value	$\vec{x}, \vec{x}'$	$\vec{y}$	$(\vec{x} \leq \vec{y} \wedge \vec{y} \leq \vec{x}') \vee (\vec{x}' \leq \vec{y} \wedge \vec{y} \leq \vec{x})$

non-deterministic relations by exploiting the flexibility in the choice of witness. To address these factors, we added to the same setting an additional collection of linear arithmetic operations, represented in Table 4.3, this time of *non-deterministic* benchmarks.

Contrary to expectations, as Fig. 4.3 shows, our method gives better performance for the non-deterministic benchmark classes as well. From this we can conclude that despite the flexibility that Craig Interpolation provides, it does not necessarily exploit the don't-cares of the input specification efficiently. These results are supported by the ones obtained in [15], which reported that the quality of the results obtained when using Craig Interpolation depended strongly on the interpolation procedure of finding good interpolants, something which is not guaranteed to happen. Comparison of the size of the implementation between RSYNTH and MONOSKOLEM also showed that the functions constructed by Craig Interpolation are much larger.

These results allow us to conclude that with a good variable order to the function being synthesized, our method scales well and outperforms previous approaches. For linear arithmetic operations, we can identify fully-interleaved to be such an order.

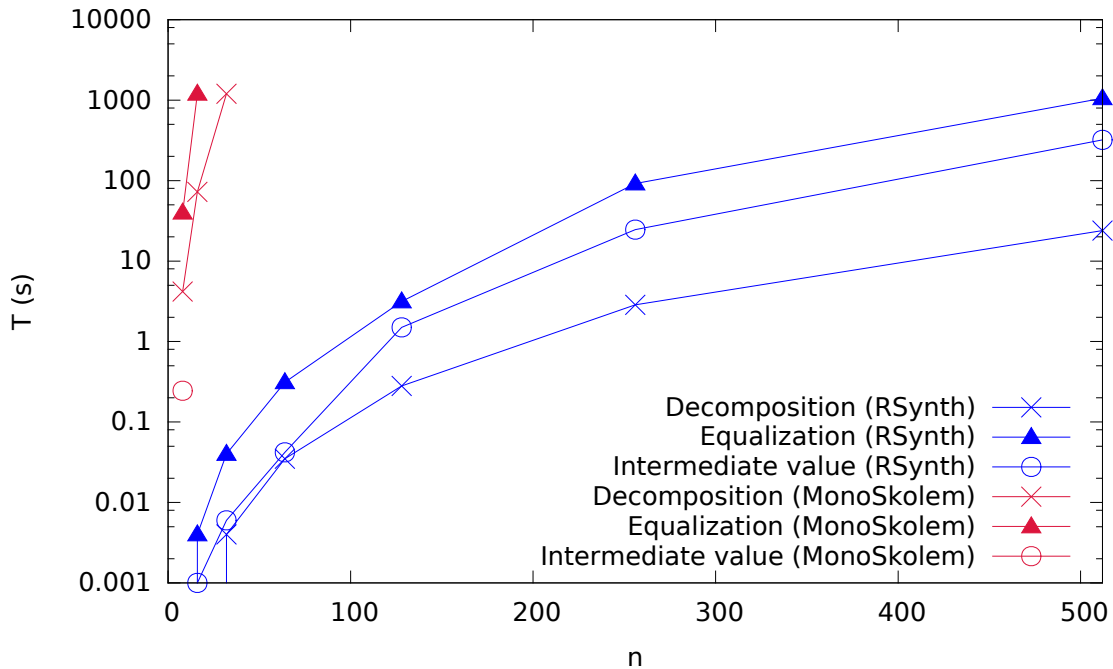


Figure 4.3 : Comparison of running time using non-deterministic benchmarks.

#### 4.5.2 Shannon Expansion vs. Self-Substitution

As mentioned in Section 4.2, the first step of the synthesis, realizability, requires quantifier elimination, which can be performed by either Shannon Expansion or Self-Substitution. We compared these two techniques by measuring the running time of the realizability phase using each of them. Our experiments show that the realizability step is responsible for only a small fraction of the running time of the synthesis. For the arithmetic benchmarks with fully-interleaved order, this step is performed in under 1s in all cases, even for  $n = 1024$ . In order to better observe the difference between the two quantifier-elimination techniques, we measured them using the *Sorting* benchmark class, for which the BDD representation is not as efficient.

As can be seen in Fig. 4.4, as  $n$  grows Self-Substitution tends to perform bet-

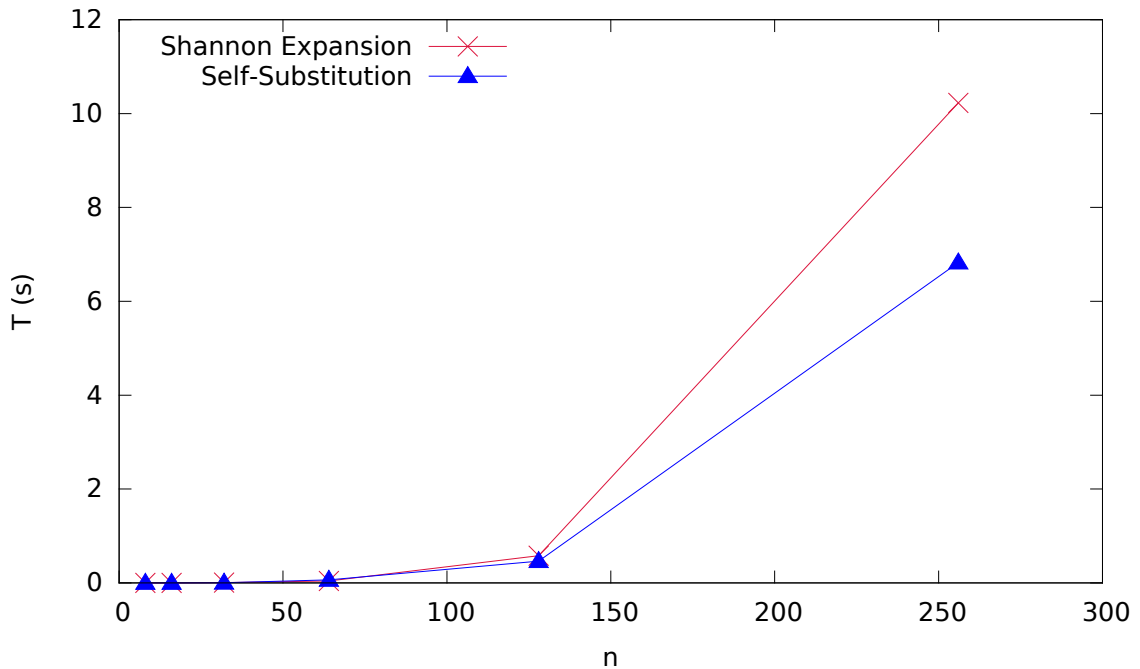


Figure 4.4 : Comparison of Shannon Expansion and Self-Substitution, for the realizability step of the *Sorting* benchmark class.

ter than Shannon Expansion, taking approximately 30% less time to perform the realizability step for  $n = 256$  (the same behavior was observed on the arithmetical benchmarks, using different variable orders). Thus, our experiments show an advantage in using Self-Substitution for quantifier elimination in the realizability step. Note that both Self-Substitution and Shannon Expansion are semantically equivalent, and thus produce identical BDDs. Therefore, the difference in performance between the two methods originates solely from the application of the CUDD operation itself over the constructed BDD. One reason for this might be because Shannon Expansion produces two intermediate BDDs for the cofactors  $f(\vec{x}, 0)$  and  $f(\vec{x}, 1)$ , while Self-Substitution produces only one for  $f(\vec{x}, 1)$ . Shannon Expansion is currently the standard way of performing quantifier elimination on Boolean formulas, but our ex-

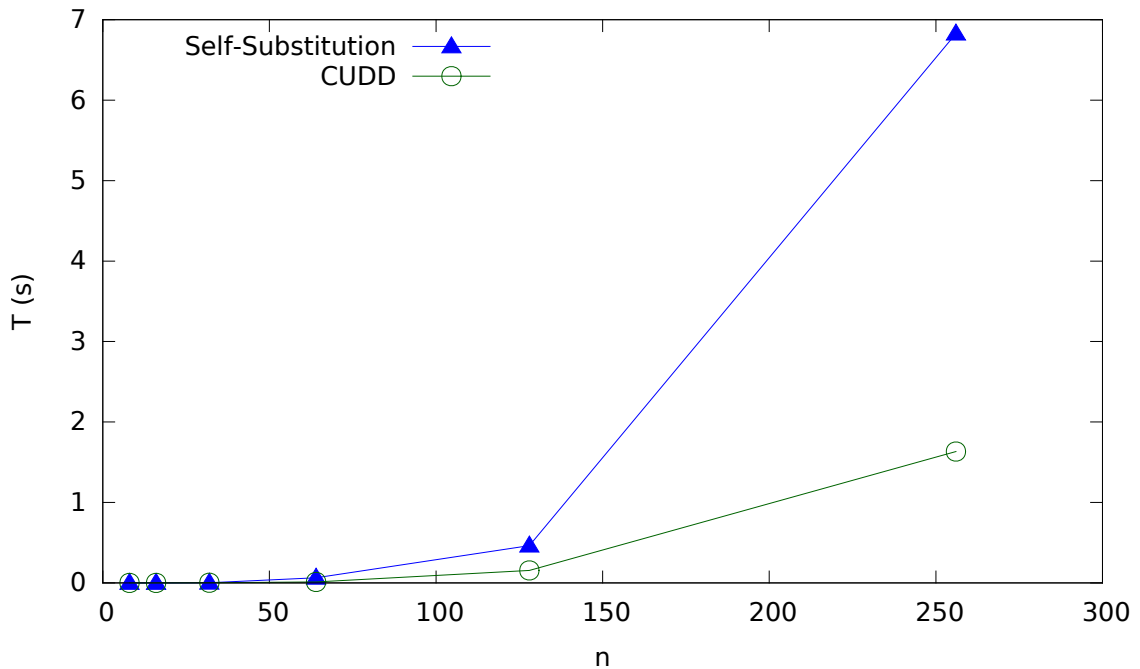


Figure 4.5 : Comparison of Self-Substitution and CUDD’s native quantifier-elimination method, for the realizability step of the *Sorting* benchmark class.

periments indicate that Self-Substitution can be often more efficient and should be considered for practical applications.

In practice, however, the CUDD package includes a native implementation of quantifier elimination tailored specifically for BDDs, in the form of a method called **ExistAbstract**. The logic behind this method follows similar principles to Shannon Expansion and Self-Substitution. However, it performs quantifier elimination by directly manipulating the internal structure of the BDD, allowing better performance than implementations which have to go through the CUDD API. Fig. 4.5 shows the comparison of Self-Substitution and **ExistAbstract** for the realizability step, and unsurprisingly we can see that the native implementation is able to achieve significantly better running time.

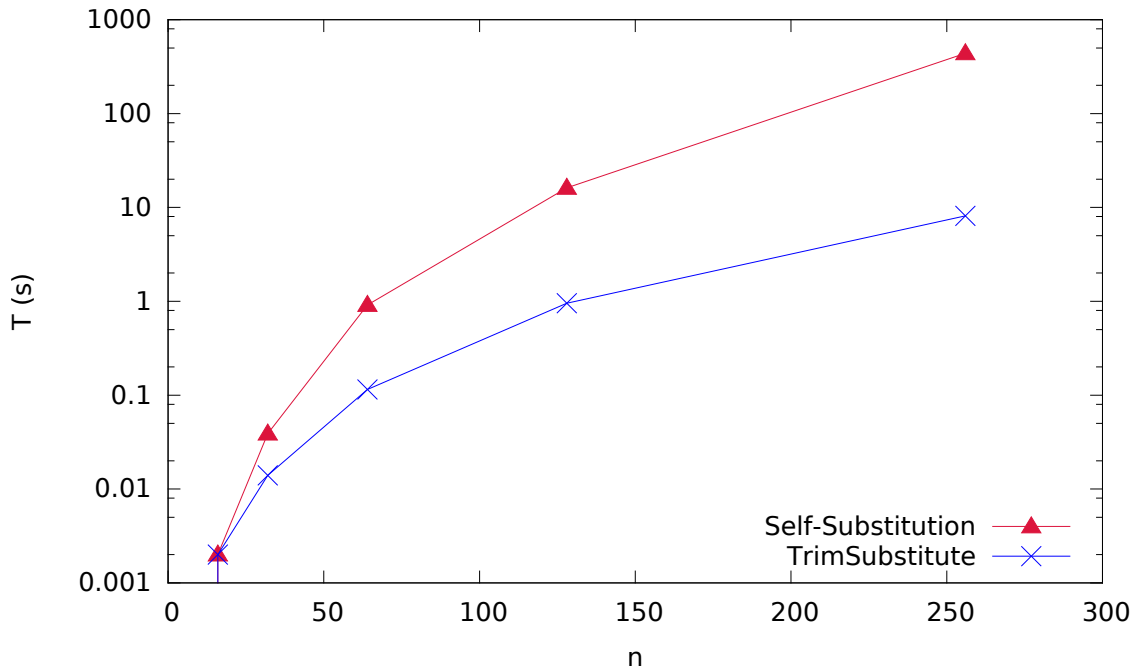


Figure 4.6 : Comparison of methods for synthesis using *input-first* BDDs for the *Sorting* benchmark class

### 4.5.3 Synthesis for Input-First BDDs

Following a suggestion in [8] for synthesis of propositional logic, we presented in Section 4.3 the TRIMSUBSTITUTE method for BDDs that follow an input-first order. We compared the performance of TRIMSUBSTITUTE with Self-Substitution (using Self-Substitution for both realizability and function construction) on input-first BDDs.

We first observed that construction time of the input-first BDD for the arithmetic benchmark classes scales poorly and was very large even for a relatively small  $n$ . The reason is that in the input-first order, the BDD is forced to keep track of all relevant information about the input before looking at the output variables. Thus, the constructed BDD must have a path for every possible output of the function being



synthesized. Since in the arithmetic benchmarks, the number of such paths is  $2^n$ , it does not pay off to use an input-first order for these benchmarks, regardless of the efficiency of the synthesis algorithm used.

On the other hand, for other classes of specifications the amount of information that must be memorized about the input can be polynomial or even linear in size. An example for that is the *Sorting* benchmark class, in which it is only necessary to keep track of the number of 1s in the input; thus, only  $n$  paths are required in the constructed BDD. In this case, although the construction time of the initial BDD still dominates the running time (experiments showed construction to take around 1200s for  $n = 256$ ), the size of the constructed BDD scales much better and makes synthesis feasible for a larger number of bits. The development of techniques to lessen the impact of construction time is a matter of future work.

Fig. 4.6 shows a comparison of running time between the Self-Substitution and TRIMSUBSTITUTE methods for *Sorting*. We can see that TRIMSUBSTITUTE greatly improves over Self-Substitution, performing around 50 times faster for  $n = 256$ . These results imply that when the specification can be efficiently represented as an *input-first* BDD, TRIMSUBSTITUTE can be used to obtain a significant improvement in synthesis time.

## 4.6 Experimental Comparison with CUDD v3.0.0

The results from the previous section were published in the 28th International Conference on Computer-Aided Verification (CAV 2016) [30]. It later came to our attention that the latest version of the CUDD package (3.0.0), released shortly before the work's submission, includes a method called `SolveEqn` for solving Boolean equations, based on techniques for the problem of Boolean unification discussed in Section 3.3. Because

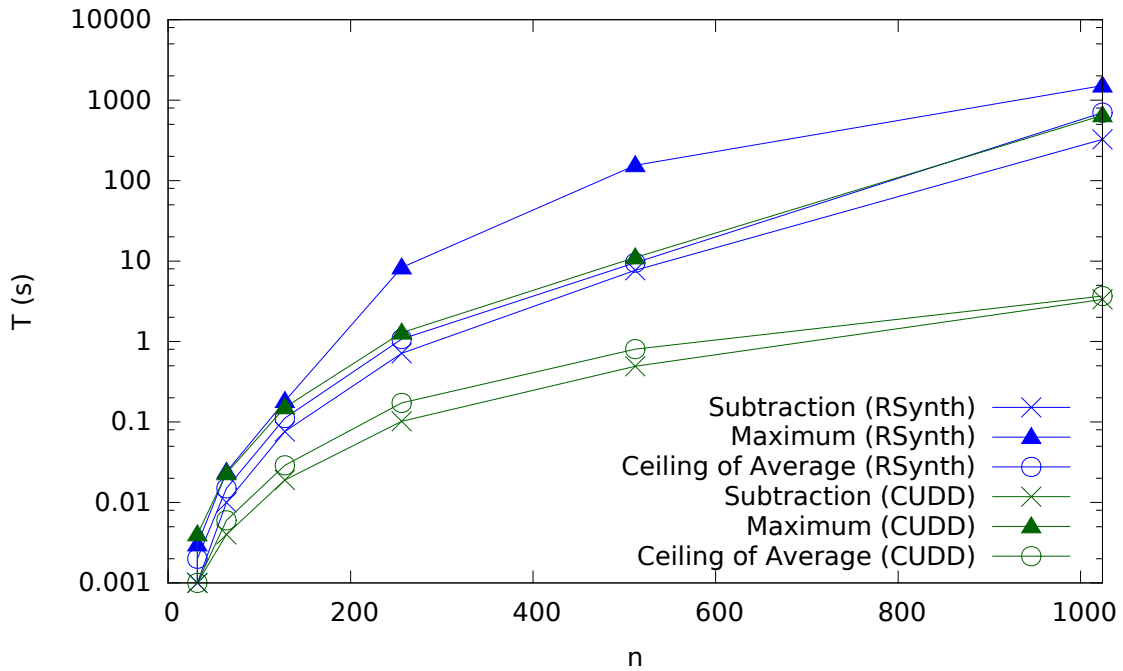


Figure 4.7 : Comparison of running time of RSYNTH and the CUDD implementation `SolveEqn` on deterministic benchmarks.

of the relationship between Boolean unification and synthesis, this operation can be used to perform Boolean synthesis, and although developed independently it corresponds in a way to a package-level implementation of the techniques presented in this chapter. Since `SolveEqn` is implemented at a lower level, it can exploit the internal structure of the BDD representation and perform optimizations in the algorithm that our implementation cannot.

With that in mind, we perform an additional experimental comparison, between the Self-Substitution approach implemented in RSYNTH and the `SolveEqn` method from CUDD. We use the same arithmetic benchmarks from Section 4.5. Fig. 4.7 shows the results for the deterministic benchmarks, while Fig. 4.8 shows the results for the non-deterministic benchmarks. As expected, the CUDD implementation outperforms

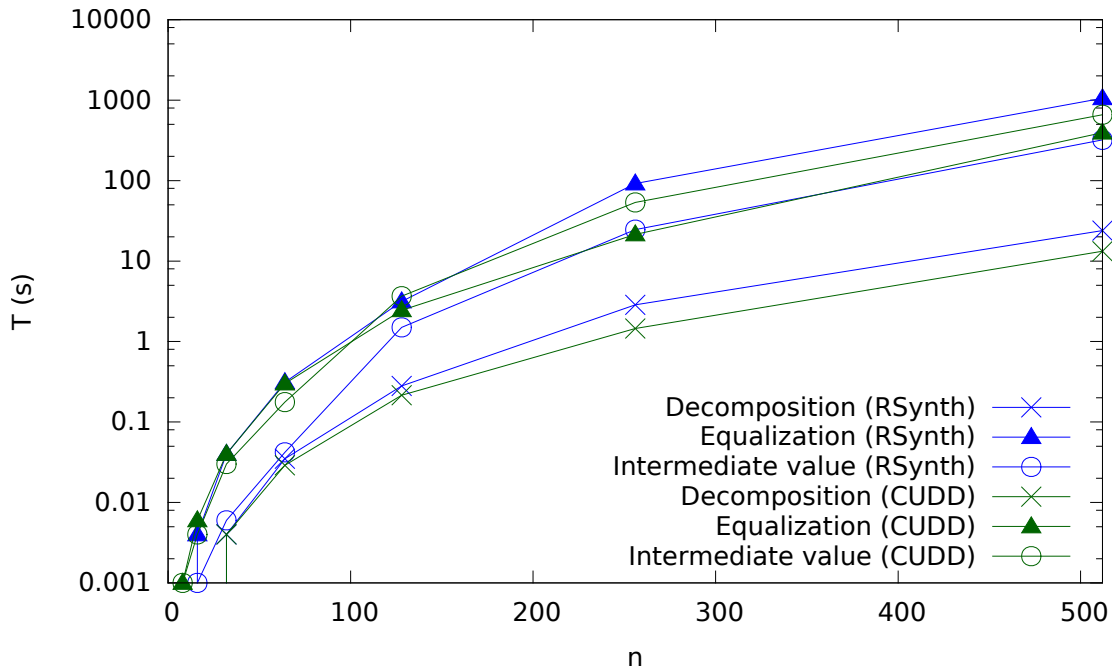


Figure 4.8 : Comparison of running time of RSYNTH and the CUDD implementation `SolveEqn` on non-deterministic benchmarks.

RSYNTH in most of the benchmarks. The *Intermediate-value* benchmark was the only one for which RSYNTH is able to obtain a slightly better performance. These results are not surprising, and match the ones for quantifier elimination in Section 4.5.2. Methods implemented at the package level can make better use of the structure of BDDs to improve performance.

However, the picture changes when we focus on input-first BDDs, in which case RSYNTH can use the `TRIMSUBSTITUTE` method. Fig. 4.9 shows a comparison of `SolveEqn` and `TRIMSUBSTITUTE` for the *Sorting* benchmark using input-first BDDs, where we can see that `TRIMSUBSTITUTE` is able to outperform the CUDD implementation by a full order of magnitude. This shows that the specialized nature of `TRIMSUBSTITUTE` is able to produce a non-trivial improvement in performance, enough to

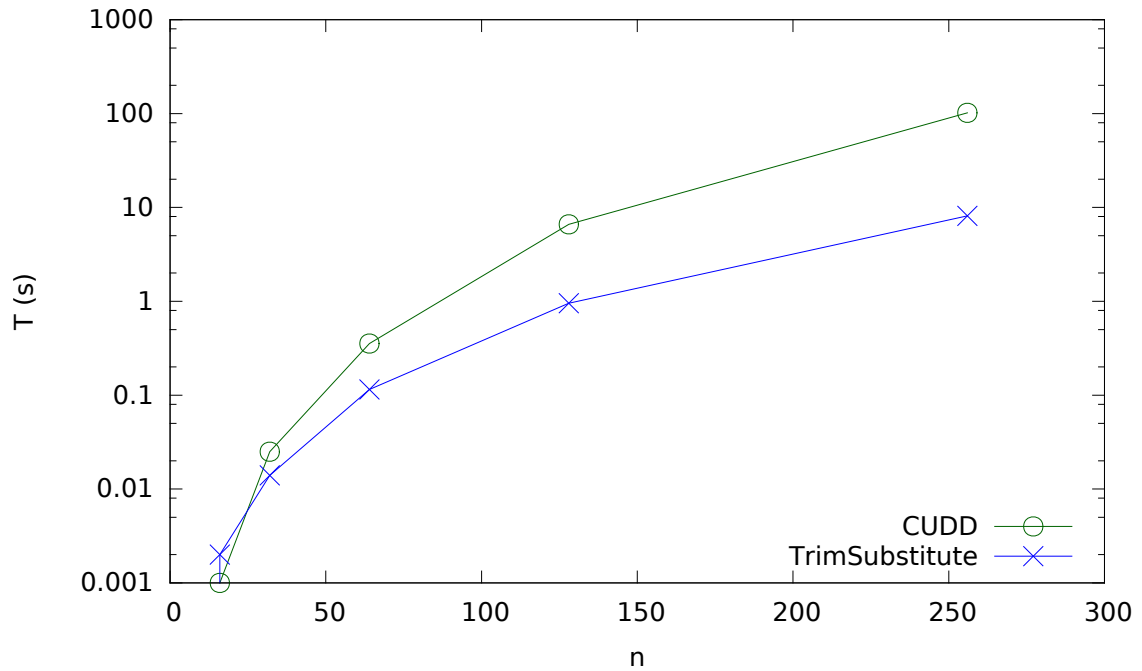


Figure 4.9 : Comparison between TRIMSUBSTITUTE and the CUDD implementation SolveEqn using *input-first* BDDs for the *Sorting* benchmark class

surpass the native implementation. Differently from Self-Substitution, TRIMSUBSTITUTE makes use of the BDD structure, which might have also contributed to being able to outperform a package-level method.

## 4.7 Conclusion

In this chapter we introduced BDD-based methods for synthesizing Boolean functions from relational specifications. We suggested a method called Self-Substitution for both quantifier elimination and function construction. We also suggested a method called TRIMSUBSTITUTE, which outperforms Self-Substitution on input-first BDDs. We demonstrated that our methods scale well for benchmarks for which we have good

BDD variable order, and outperform prior techniques.

Although our implementation of Self-Substitution is outperformed by the native implementation of `SolveEqn` in the latest version of the CUDD package, the theory behind it still holds value, and because it does not depend on the structure of the representation it can be easily applied to other data structures, including the large number of existing BDD variants and alternative representations such as AIGs.

The performance of the synthesis methods for BDDs depends crucially on the choice of variable order. However, finding a good variable order is not only hard in general, but also impossible for some specifications. It is known that there exist formulas for which there is no order that produces a polynomial-size BDD [13]. Therefore, a key challenge is to lessen the impact of the BDD size in the synthesis process. For this purpose, in the next chapter we explore the use of factored formulas, which have been successfully used in both symbolic model checking [11] and satisfiability testing [12], and which are able to produce significantly more compact representations of specifications.

## Chapter 5

### Synthesis from Factored Formulas

In Chapter 4, we presented a framework for performing Boolean synthesis using Binary Decision Diagrams (BDDs). The efficiency of BDDs, however, is highly dependent on finding a good variable ordering, which is a hard optimization problem. Furthermore, it is well-known that there are interesting Boolean formulas that cannot be represented by a polynomial-sized BDD. This is part of the reason why recent works have avoided BDDs in favor of approaches using SAT solvers, such as the Craig-Interpolation approach from [14].

Nevertheless, the evaluation that we presented in Section 4.5 shows that BDD-based techniques can be very competitive when the specification can be efficiently represented by a BDD and a good variable ordering is known. This raises the question of whether it is possible to find a way to employ BDD-based techniques even in cases when a BDD for the specification cannot be constructed. In other applications where the use of BDDs is common, the solution to this problem came in the form of *factored representations* of formulas, which allow a much wider range of instances to be effectively computed [11].

Factored representations are based on the fact that it is common for Boolean formulas of practical importance to be represented by conjunctions of constraints. In other words, a Boolean formula  $f(\vec{x}, \vec{y})$  might be written in the format  $f_1(\vec{x}, \vec{y}) \wedge \dots \wedge f_k(\vec{x}, \vec{y})$ . In this case, rather than constructing a single *monolithic* BDD  $B$  for  $f$ , we can instead represent the formula as a collection of BDDs  $B_1, \dots, B_k$  for each of the

*factors*  $f_1, \dots, f_k$ , implicitly interpreted as a conjunction. Since conjoining multiple BDDs can lead to a combinatorial explosion, the factored representation is usually significantly more compact.

In symbolic model checking, where the idea of factored BDD representations originated, this approach was able to reduce the size of representations of transition relations by an order of magnitude [11]. Since then, different techniques have been developed for further improving performance, including heuristics for clustering and reordering factors [31]. Similar techniques have been used for processing factored formulas in the context of symbolic satisfiability [12]. In this approach to the satisfiability problem, a CNF formula is encoded by partitioning the set of clauses and representing each partition as a BDD. Then, symbolic quantifier elimination is used to find if there is a satisfying assignment to the formula. In this paper we show how techniques and heuristics used in these applications can be adapted to perform synthesis from factored specifications.

Other approaches have been developed for synthesis of factored formulas that do not employ BDDs. A recent work [15] uses And-Inverter Graphs (AIGs) for representing Boolean formulas and a counterexample-guided abstraction refinement (CEGAR) loop for synthesizing the function. A downside to this approach is that the CEGAR loop requires repeated calls to a SAT solver, which can have a high cost in running time. Furthermore, BDDs can be very compact for small formulas, which poses the question of whether they can produce smaller functions than AIGs when using factored representation.

A different synthesis approach is based on the close relation between Boolean synthesis and QBF solving. The CNF formulas given as input to QBF solvers are special cases of factored formulas, and a number of modern solvers are capable of

computing Skolem functions for the existential variables in terms of the universal variables [16, 17]. Therefore, by writing a specification  $f(\vec{x}, \vec{y})$  in CNF, the synthesis problem can be encoded as a QBF  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y})$ .

Although QBF solvers can be very efficient in solving these formulas, they are able to synthesize Skolem functions only when the QBF  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y})$  evaluates to *true*. This corresponds to the case when the specification  $f(\vec{x}, \vec{y})$  is *realizable*, that is, when every input  $\vec{x}$  has an output  $\vec{y}$  that satisfies  $f$ , and consequently  $p(\vec{x}) \equiv 1$ . In many applications of Boolean synthesis we are interested, however, in unrealizable specifications as well. One such a case is LTL<sub>f</sub> synthesis using DFA games [32], in which a winning strategy might not exist for every state of the automaton, but we would like to synthesize this strategy for all states for which it exists.

In our experimental evaluation, we first compare our implementation using factored representation with the monolithic approach, allowing us to confirm that indeed factoring the specification allows us to synthesize a number of instances that would be otherwise intractable. We then compare our implementation using BDDs to two other tools: CEGARSKOLEM [15, 33], which uses the CEGAR-based approach, and the QBF solver CADET [17]. The results show that no approach is universally better, and every tool outperforms the others in some subset of the benchmarks. Although the QBF approach has a clear advantage for realizable specifications, being unable to handle unrealizable instances limits its applicability in a number of practical cases.

Beyond performance, an advantage of using BDDs is that this makes the approach easier to integrate in temporal synthesis applications, such as [32]. This is because such applications usually employ some kind of fixpoint computation, for which BDDs are particularly suited due to the ease of checking if two BDDs are equivalent. Using other representations for Boolean formulas, such as AIGs or CNF, it becomes harder



to perform such computations.

## 5.1 Synthesis from Factored Formulas

In this section, we start by formally defining the notion of factored representations and present some of their properties. We then describe a method for performing synthesis over factored representations.

### 5.1.1 Factored Representation of Boolean Formulas

Let the specification  $f$  for an instance of the Boolean synthesis problem be of the form  $f(\vec{x}, \vec{y}) = f_1(\vec{x}, \vec{y}) \wedge f_2(\vec{x}, \vec{y}) \wedge \dots \wedge f_k(\vec{x}, \vec{y})$ . Each formula  $f_i$  is called a *factor* of  $f$ . The sequence of BDDs  $\langle B_1, B_2, \dots, B_k \rangle$ , where  $B_i$  is the BDD encoding of  $f_i$ , is called the *factored representation* of  $f$ . In contrast, the representation of  $f$  as a single BDD  $B$  is called the *monolithic representation*.

Note that it is possible for a formula to have an exponential monolithic representation and a polynomial factored representation. In particular, the factored representation of a formula in CNF can always be linear, since the BDD of a single clause is linear in size.

Although factored representations can be exponentially more compact than monolithic representations, they introduce complications into the synthesis procedure. To understand why, first note from the definition of Boolean synthesis that there is a close connection between Boolean synthesis and quantifier elimination. In fact, substituting the implementation  $g(\vec{x})$  in the specification  $f(\vec{x}, \vec{y})$  is equivalent to existentially quantifying  $\vec{y}$ , and the precondition  $p(\vec{x})$  is exactly the result of this quantification. Then, recall that existential quantifiers do not distribute over conjunction. That is,

in general

$$\exists y_1, \dots, y_n. \bigwedge_{i=1}^k f_i(\vec{x}, \vec{y}) \not\equiv \bigwedge_{i=1}^k \exists y_1, \dots, y_n. f_i(\vec{x}, \vec{y})$$

More precisely, as pointed out in [15], the right-hand side is an over-approximation of the left-hand side, meaning that every assignment of  $\vec{x}$  that satisfies the left-hand side satisfies the right-hand side, but not vice-versa.

As a consequence, if we are given a factored representation of a Boolean formula, it is not clear how to perform existential quantifier elimination, and consequently synthesis, without conjoining the factors. However, the insight first employed in [11] is that it is possible to move conjuncts outside an existential quantifier if the quantified variable does not appear in the support of the conjunct. Formally, let  $F_j \subseteq \{1, \dots, k\}$  be the set of indices  $i$  such that  $y_j$  is in the support of  $f_i$ . Then,

$$\begin{aligned} & \exists y_1, \dots, y_n. \bigwedge_{i=1}^k f_i(\vec{x}, \vec{y}) \\ & \equiv \exists y_1, \dots, y_{n-1}. \left( \exists y_n. \bigwedge_{i \in F_n} f_i(\vec{x}, \vec{y}) \right) \wedge \bigwedge_{i \notin F_n} f_i(\vec{x}, \vec{y}) \end{aligned}$$

Using the relation between synthesis and existential quantification, we obtain the following result:

**Lemma 5.1.** *Let  $f(\vec{x}, \vec{y}) = f_1(\vec{x}, \vec{y}) \wedge f_2(\vec{x}, \vec{y}) \wedge \dots \wedge f_k(\vec{x}, \vec{y})$  be a specification and  $g_j(\vec{x})$  be a witness to  $y_j$  in  $\bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y})$ . Then,  $g_j(\vec{x})$  is a witness to  $y_j$  in  $f(\vec{x}, \vec{y})$ .*

*Proof.* Since  $g_j(\vec{x})$  is a witness to  $y_j$  in  $\bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y})$ , then by definition

$$\left( \bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y}) \right) [y_j \mapsto g_j(\vec{x})] \equiv \exists y_j. \bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y})$$

To prove that  $g_j(\vec{x})$  is also a witness of  $f(\vec{x}, \vec{y})$ , it needs to be shown that  $f(\vec{x}, \vec{y})[y_j \mapsto$

$g_j(\vec{x})] \equiv \exists y_j. f(\vec{x}, \vec{y})$ . But

$$\begin{aligned}
& f(\vec{x}, \vec{y})[y_j \mapsto g_j(\vec{x})] \\
& \equiv \left( \bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y}) \wedge \bigwedge_{i \notin F_j} f_i(\vec{x}, \vec{y}) \right) [y_j \mapsto g_j(\vec{x})] \\
& \equiv \left( \bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y}) \right) [y_j \mapsto g_j(\vec{x})] \wedge \bigwedge_{i \notin F_j} f_i(\vec{x}, \vec{y}) \\
& \equiv \left( \exists y_j. \bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y}) \right) \wedge \bigwedge_{i \notin F_j} f_i(\vec{x}, \vec{y}) \\
& \equiv \exists y_j. \left( \bigwedge_{i \in F_j} f_i(\vec{x}, \vec{y}) \wedge \bigwedge_{i \notin F_j} f_i(\vec{x}, \vec{y}) \right) \\
& \equiv \exists y_j. f(\vec{x}, \vec{y})
\end{aligned}$$

Therefore,  $g_j(\vec{x})$  is a witness of  $f(\vec{x}, \vec{y})$ .  $\square$

From Lemma 5.1 we have that a witness for a variable in a factored formula can be constructed from only the factors in which that variable appears. Since in practice each variable will only be in the support of a small subset of the factors, this insight means that it is possible to perform synthesis without converting entirely from the factored to the monolithic representation. Instead, we can design a strategy for synthesis directly over factored formulas.

### 5.1.2 Synthesis from Factored Specifications

Algorithm 5.1 presents a synthesis framework that takes advantage of the factored representation of the specification, using the insight from Lemma 5.1 to avoid conjoining all factors at once. Instead, we conjoin the factors one-by-one, and after each conjunction synthesize and eliminate the variables that do not appear in the support

of any of the remaining factors. This strategy is similar to the ones followed in model checking [11] and symbolic satisfiability [12] from factored representations.

We assume the existence of a monolithic Boolean synthesis procedure, such as the ones presented in Chapter 4, denoted by  $\text{synth}(B, X, Y)$ , which receives a BDD  $B$ , a set of input variables  $X$  and a set of output variables  $Y$ , and returns a BDD  $P$  representing the precondition and a sequence of BDDs  $(W_j)_{y_j \in Y}$  representing the implementation.

We start, in line 2, by partitioning the output variables into sets  $Y_1, \dots, Y_k$  such that  $y_j \in Y_i$  if and only if  $B_i$  is the last factor where  $y_j$  appears. In other words,  $y_j \in Y_i$  if and only if  $\max F_j = i$ . We maintain a BDD  $B$  which accumulates the factors. In line 3,  $B$  is initialized to the empty conjunction, which is equivalent to the constant 1. We then iterate over the factors, conjoining the next factor to  $B$  at every iteration in line 5. Once  $B_i$  is conjoined, none of the output variables in  $Y_i$  appear in any of the remaining factors. The monolithic synthesis procedure is then called in line 6 to synthesize witnesses for every variable in  $Y_i$ , in terms of the input variables  $x_1, \dots, x_m$  and the output variables in  $Y_{i+1}, \dots, Y_k$ . Then, in line 7,  $B$  is updated to the precondition  $P_i$ , which corresponds to the conjunction of the first  $i$  factors with the output variables in  $Y_1 \cup \dots \cup Y_i$  existentially quantified.

After the end of the loop, every witness  $W_j$  for  $y_j \in Y_i$  has the variables from  $Y_{i+1}, \dots, Y_k$  in its support set. In the last step, performed by the loop in lines 9-13, these extra variables are eliminated by substituting their respective witnesses, making every  $W_j$  dependent only in the input variables  $x_1, \dots, x_m$ .

The following theorem states the correctness of Algorithm 5.1, which follows from the correctness of the monolithic synthesis procedure and Lemma 5.1.

Figure 5.1 : Synthesis from Factored Specifications

**Input:** Factored representation  $\langle B_1, \dots, B_k \rangle$  of the specification.

**Output:** Precondition BDD  $P$ , witness BDDs  $\langle W_1, \dots, W_n \rangle$ .

```

1:  $X \leftarrow \{x_1, \dots, x_m\}$ 
2:  $Y_i \leftarrow \{y_j \mid B_i \text{ is the last factor where } y_j \text{ appears}\}$ 
3:  $B \leftarrow 1$ 
4: for  $i \leftarrow 1 \dots k$  do
5:    $B \leftarrow B \wedge B_i$ 
6:    $P_i, (W_j)_{y_j \in Y_i} \leftarrow \text{synth}(B, X \cup Y_{i+1} \cup \dots \cup Y_k, Y_i)$ 
7:    $B \leftarrow P_i$ 
8: end for
9: for  $i \leftarrow 1 \dots k$  do
10:  for  $i' \leftarrow (i + 1) \dots k$  do
11:     $W_\ell \leftarrow W_\ell[y_j \mapsto W_j]$ , for all  $y_\ell \in Y_i, y_j \in Y_{i'}$ 
12:  end for
13: end for
14:  $P \leftarrow B$ 
15: return  $P, W_1, \dots, W_n$ 

```

**Theorem 5.1.** *If  $P, W_1, \dots, W_n$  are computed according to Algorithm 5.1, then*

$$\exists y_1, \dots, y_n. (B_1 \wedge \dots \wedge B_k) \equiv P \equiv (B_1 \wedge \dots \wedge B_k)[y_1 \mapsto W_1, \dots, y_n \mapsto W_n]$$

*Proof.* We assume the correctness of the monolithic synthesis procedure *synth*, meaning that  $\text{synth}(B, X, Y)$  returns a precondition  $P$ , and a witness  $W_j$  for each variable  $y_j \in Y$  in terms of the variables in  $X$ , such that  $\exists Y. B \equiv P \equiv B[y_j \mapsto W_j]_{y_j \in Y}$ .

Consider the loop in lines 4-8. We will first prove that if at the start of the  $i$ -th iteration  $B \equiv \exists Y_1, \dots, Y_{i-1}. (B_1 \wedge \dots \wedge B_{i-1})$ , then at the end of the  $i$ -th iteration  $B \equiv \exists Y_1, \dots, Y_i. (B_1 \wedge \dots \wedge B_i)$ . We will use this to prove that  $P \equiv \exists y_1, \dots, y_n. (B_1 \wedge \dots \wedge B_k)$

Assume that at the start of the  $i$ -th iteration  $B \equiv \exists Y_1, \dots, Y_{i-1}. (B_1 \wedge \dots \wedge B_{i-1})$ . Then, after line 5,  $B \equiv (\exists Y_1, \dots, Y_{i-1}. (B_1 \wedge \dots \wedge B_{i-1})) \wedge B_i$ . Since  $Y_1, \dots, Y_{i-1}$  do not appear in  $B_i$ , the quantifier can be moved outside the conjunction, so  $B \equiv \exists Y_1, \dots, Y_{i-1}. (B_1 \wedge \dots \wedge B_{i-1} \wedge B_i)$ .

Then, in line 6 the monolithic synthesis procedure is called on  $B$ , with input variables  $X \cup Y_{i+1} \cup \dots \cup Y_k$  and output variables  $Y_i$ . By the correctness of the monolithic procedure,  $P_i \equiv \exists Y_i. B \equiv \exists Y_1, \dots, Y_{i-1}, Y_i. (B_1 \wedge \dots \wedge B_{i-1} \wedge B_i)$ . Then, after line 7, when  $B$  is updated to  $P_i$ ,  $B \equiv \exists Y_1, \dots, Y_{i-1}, Y_i. (B_1 \wedge \dots \wedge B_{i-1} \wedge B_i)$ .

Therefore, if  $B \equiv \exists Y_1, \dots, Y_{i-1}. (B_1 \wedge \dots \wedge B_{i-1})$  at the start of the  $i$ -th iteration,  $B \equiv \exists Y_1, \dots, Y_i. (B_1 \wedge \dots \wedge B_i)$  at the end of the  $i$ -th iteration. Taking  $i = 1$ , this means that if  $B \equiv 1$  (the empty conjunction) before the loop then at the end of the first iteration  $B \equiv \exists Y_1. B_1$ . Since the invariant  $B \equiv \exists Y_1, \dots, Y_i. (B_1 \wedge \dots \wedge B_i)$  is maintained, at the end of the last iteration  $B \equiv \exists Y_1, \dots, Y_k. (B_1 \wedge \dots \wedge B_k)$ . Therefore, after line 14,  $P \equiv \exists Y_1, \dots, Y_k. (B_1 \wedge \dots \wedge B_k)$ , as desired.

We now prove that  $(B_1 \wedge \dots \wedge B_k)[y_1 \mapsto W_1, \dots, y_n \mapsto W_n] \equiv \exists Y_1, \dots, Y_k. (B_1 \wedge \dots \wedge B_k)$ . In iteration  $i$ , we construct  $W_j$  for every  $y_j \in Y_i$ . Since at this time

$B \equiv \exists Y_1, \dots, Y_{i-1}.(B_1 \wedge \dots \wedge B_{i-1} \wedge B_i)$ , by the correctness of the monolithic synthesis procedure,  $(\exists Y_1, \dots, Y_{i-1}.(B_1 \wedge \dots \wedge B_{i-1} \wedge B_i))[y_j \mapsto W_j]_{y_j \in Y_i} \equiv \exists Y_1, \dots, Y_{i-1}, Y_i.(B_1 \wedge \dots \wedge B_{i-1} \wedge B_i)$ . Then, since no variables in  $Y_1, \dots, Y_{i-1}$  appear in  $B_i$ ,

$$\begin{aligned} & \exists Y_1, \dots, Y_i.(B_1 \wedge \dots \wedge B_i) \\ & \equiv (\exists Y_1, \dots, Y_{i-1}.(B_1 \wedge \dots \wedge B_i))[y_j \mapsto W_j]_{y_j \in Y_i} \\ & \equiv ((\exists Y_1, \dots, Y_{i-1}.(B_1 \wedge \dots \wedge B_{i-1})) \wedge B_i)[y_j \mapsto W_j]_{y_j \in Y_i} \end{aligned}$$

Applying this transformation recursively to  $\exists Y_1, \dots, Y_k.(B_1 \wedge \dots \wedge B_k)$  results in  $(\dots (B_1[y_j \mapsto W_j]_{y_j \in Y_1} \wedge B_2)[y_j \mapsto W_j]_{y_j \in Y_2} \wedge \dots \wedge B_k)[y_j \mapsto W_j]_{y_j \in Y_k}$ . Applying Lemma 5.1, we can move the composition operators outside the conjunction, giving

$$\begin{aligned} & \exists Y_1, \dots, Y_k.(B_1 \wedge \dots \wedge B_k) \\ & \equiv (B_1 \wedge \dots \wedge B_k)[y_j \mapsto W_j]_{y_j \in Y_1} \dots [y_j \mapsto W_j]_{y_j \in Y_k} \end{aligned}$$

Recall that each  $W_j$  for  $y_j \in Y_i$  might contain variables from  $Y_{i+1}, \dots, Y_k$  in its support set. Because of this, we cannot change the order of the composition operators. However, the loop in lines 9-13 performs the composition of each witness with the ones that succeed it, making every  $W_j$  dependent only on  $x_1, \dots, x_m$ . This allows the compositions to be performed in any order, so that  $\exists Y_1, \dots, Y_k.(B_1 \wedge \dots \wedge B_k) \equiv (B_1 \wedge \dots \wedge B_k)[y_1 \mapsto W_1, \dots, y_n \mapsto W_n]$ .  $\square$

A problem with Algorithm 5.1 is that performance will be very dependent on the order of the factors. Consider for example a specification in which for every  $i$ , the output support of  $f_i$  is  $\{y_1, \dots, y_i\}$ . Then,  $Y_1 = Y_2 = \dots = Y_{k-1} = \{\}$  and  $Y_k = \{y_1, \dots, y_n\}$ . Processing the factors in order will result in all factors being conjoined before any witness can be synthesized, thus degenerating into the monolithic synthesis procedure. On the other hand, processing the factors in the reverse order would allow one variable to be synthesized immediately after each conjunction. Therefore, it is

clear that the algorithm can benefit from reordering the factors before starting the synthesis. Finding the optimal order is a combinatorially hard problem, but a number of heuristics can be used instead. Another possible improvement in the algorithm is clustering, a technique that has been employed in other applications which use factored representations of formulas [34, 35, 12]. In clustering, the set of factors is first partitioned, and the factors in each partition are conjoined into monolithic clusters. The algorithm is then applied over the clusters rather than the individual factors. The next section explores different heuristics for clustering and reordering.

### 5.1.3 Clustering and Reordering

As noted in [34], if the individual BDDs for each factor are small, it is often better to combine different factors into monolithic clusters. If the clusters are constructed so that they remain of reasonable size, clustering reduces the number of iterations while not excessively increasing the cost in space.

Formally, given a factored formula  $f(\vec{x}, \vec{y}) = f_1(\vec{x}, \vec{y}) \wedge f_2(\vec{x}, \vec{y}) \wedge \dots \wedge f_k(\vec{x}, \vec{y})$  a *clustering heuristic* partitions the set of factors  $\{f_1, f_2, \dots, f_k\}$  into  $\kappa$  disjoint non-empty subsets  $C_1, \dots, C_\kappa$ , called the *clusters*. In practice, each cluster  $C_\iota$  is represented by a BDD  $\mathcal{B}_\iota$  encoding the formula  $\bigwedge_{f_i \in C_\iota} f_i(\vec{x}, \vec{y})$ . Since conjunction is associative and commutative,  $\langle \mathcal{B}_1, \dots, \mathcal{B}_\kappa \rangle$  is itself a factored representation of the original formula  $f$ . Therefore, Algorithm 5.1 can be applied normally to this representation.

The goal of clustering is to create a balance between the number of factors and size of the factors. An example of clustering strategy is *rank-based clustering*, employed in [12]. In this strategy, for every variable  $y_j$ , cluster  $C_j = \{f_i \mid \text{rank}(f_i) = j\}$ , where  $\text{rank}(f_i)$  is the highest index among the variables in the support of  $f_i$ .

Rank-based clustering naturally gives rise to some reordering heuristics, in which



clusters are ordered either by increasing or decreasing rank. Two more options for reordering factors appear in the context of model checking in [31]. In that work, factored formulas are used to represent transition relations, and different reordering heuristics are used in the forward and backward simulation steps. The following are the four heuristics used in this work:

**Bouquet’s method** [12] Order clusters by increasing rank.

**Bucket elimination** [12] Order clusters by decreasing rank.

**Forward** [31] Greedily order factors by number of variables that can be eliminated once the factor is conjoined. In other words, at every step choose the factor that has the greatest number of output variables that do not appear in any of the remaining factors.

**Backward** [31] Order factors such that at every step the next factor will be the one that has the fewest new variables, that is, variables that have not appeared in any of the previous factors. This heuristic tries to avoid as much as possible increasing the size of the conjoined BDD.

All of the above heuristics for clustering and reordering can be applied to synthesis from factored representations, but it is unclear which would give better results. Section 5.2 describes an experimental evaluation of the different techniques.

#### 5.1.4 BDD Variable Ordering

The size of BDDs is strongly influenced by the ordering of the variables. Part of the goal of using factored representations is to be able to represent specifications

for which a good variable ordering is not known beforehand. Rather than using an arbitrary variable ordering for these cases, it would be good to be able to compute one by analyzing the structure of the formula. Similarly to clustering, finding the optimal variable ordering is a hard combinatorial problem, but numerous heuristics have been developed to find good enough approximations.

One such heuristic is the inverse *maximum cardinality search* (MCS) ordering [36]. This variable ordering is constructed based on the *Gaifman graph* of the formula  $f(\vec{x}, \vec{y}) = f_1(\vec{x}, \vec{y}) \wedge \dots \wedge f_k(\vec{x}, \vec{y})$ , defined as  $G = (V, E)$ , where  $V = \{x_1, \dots, x_m, y_1, \dots, y_n\}$  and  $E = \{(v_1, v_2) \mid \text{there exists an } i \text{ such that } v_1 \text{ and } v_2 \text{ are in the support of } f_i\}$ . In other words, the Gaifman graph of a factored formula has one vertex for each variable and has an edge between every pair of variables that share a factor.

The inverse MCS order can be computed from the Gaifman graph by the following procedure:

1. Initialize an empty list  $L$ .
2. At each step, select the vertex  $v \in V$  not in  $L$  with the largest number of neighbors in  $L$ . Add  $v$  to  $L$ .
3. After all vertices have been added, reverse  $L$ , so that vertices added later come first in the ordering.

Other heuristics for variable ordering were studied in [12], but among them the inverse MCS heuristic had the best results in that work. Therefore, this heuristic was chosen for the experiments in this paper.

## 5.2 Experimental Evaluation

We performed the experiments using QBF benchmarks taken from the QBFLIB collection [37]. All benchmarks selected were of the form  $\forall\vec{x}.\exists\vec{y}.f(\vec{x},\vec{y})$ , where  $f(\vec{x},\vec{y})$  is a CNF formula. In this case, synthesis corresponds to finding a Skolem function to the existential variables. Every clause in  $f(\vec{x},\vec{y})$  can be considered one factor.

We implemented the factored algorithm from Section 5.1 and the various heuristics for clustering and reordering factors in our tool `RSYNTH`, implemented in C++11 using the CUDD package. As mentioned in Section 4.6, the most recent version of CUDD includes a monolithic Boolean synthesis procedure `SolveEqn`, which uses a similar algorithm as the one presented in Section 4.2. We used this procedure in our implementation as the *synth* subroutine.

All experiments were executed in the DAVinCI cluster at Rice University, consisting of 192 Westmere nodes of 12 processor cores each, running at 2.83 GHz with 4 GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2 GHz with 8 GB of RAM per core. The algorithm has not been parallelized, so the cluster was solely used to run multiple experiments simultaneously.

Besides comparing the monolithic and factored algorithms and evaluating different reordering heuristics, we also compare our tool `RSYNTH` with two existing tools for Boolean synthesis. The first is the `CEGARSKOLEM` tool from [15], which uses a SAT-based CEGAR loop and AIGs to perform synthesis from factored formulas. The second is the 2QBF solver `CADET` [17].

All plots in this section are shown in log scale. Each benchmark was given a time limit of two hours. Only a subset of the total set of benchmarks is included in the plots. Benchmarks for which the results were similar to already-included benchmarks were omitted, as well as benchmarks for which all or almost all of the methods timed

out.

### 5.2.1 Heuristics for Factor Reordering

We first measure the performance of the factored algorithm using different reordering heuristics. The bar plots on Figure 5.2 show the running time of each heuristic on different benchmarks. Figure 5.2a shows the results for *Bouquet's Method* and *Bucket Elimination*, while Figure 5.2b shows the results for the *Forward* and *Backward* heuristics. The bars labeled *None* show the running time when no heuristic is used and the factors are simply processed in the order they are given in the input file.

Surprisingly, the results show that using no reordering is often preferable. In most of the instances, the best running time was achieved with no reordering. In fact, there are some benchmarks which none of the heuristics were able to synthesize in the time limit, but which succeeded when no reordering was used.

This result leads to the conclusion that for these benchmarks the clauses of the CNF formulas were already in a good order. This might be due to the fact that often CNF formulas are constructed in a way that places clauses with the same variables close to each other. To confirm that indeed the original clause order was a good one, we also ran experiments where the clauses were reordered randomly. In this case, all benchmarks from Figure 5.2 except for `mutex2`, `qshifter3` and `qshifter4` timed out. Considering that using the original ordering given in the input file allowed all benchmarks to be synthesized, we conclude that the order of the clauses in a typical CNF benchmark is not arbitrary, and rather it is often a very natural order for them to be processed. Additionally, although not performing as well as the original order, the performance of heuristics such as *Bouquet's Method* and *Bucket Elimination* show that they already bring significant improvements to an arbitrary ordering.

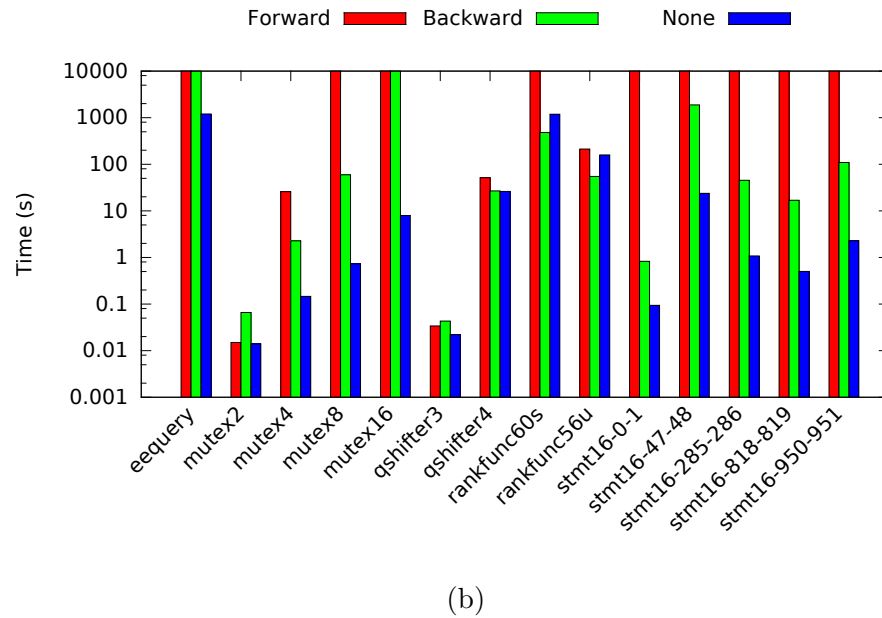
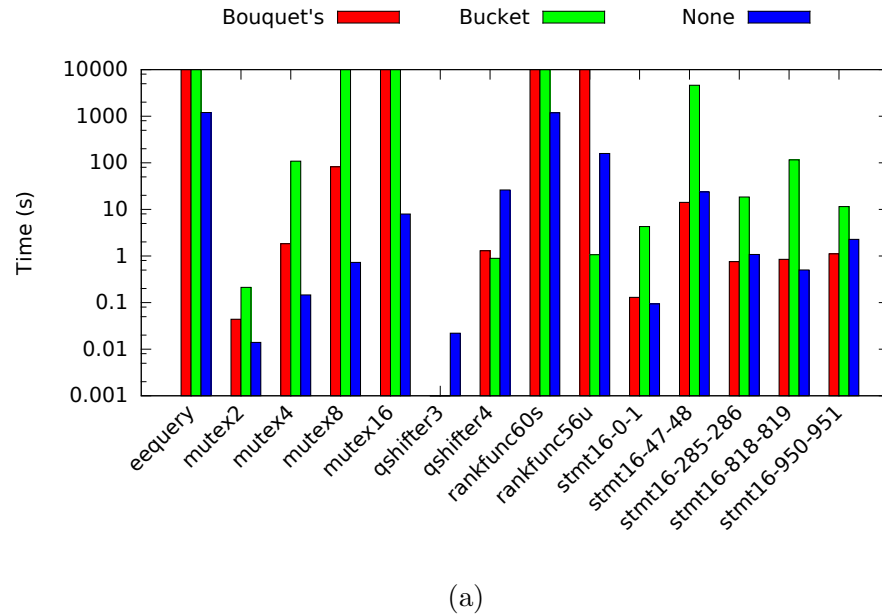


Figure 5.2 : Performance of the factored algorithm using different reordering heuristics, in log scale. The values include both the time spent reordering the factors and time running the algorithm. Bars of maximum height indicate instances that timed out. Bars not displayed mean that the instance took less than 1ms.

The performance of the other heuristics varied depending on the type of benchmark. Every heuristic performed better than all the others on at least one case. Overall, however, the *Forward* heuristic seems to have the worst scalability, timing out for most of the instances. This is likely due to it being a greedy heuristic which tries to synthesize as many variables as possible at each step, causing the size of the BDDs to quickly increase.

### 5.2.2 Factored vs. Monolithic

Next, we compare the running time of the factored algorithm with synthesis using the monolithic procedure. In the latter, the running time includes the time necessary to conjoin all the factors to create the monolithic representation. Given the previous results, no reordering was used for the factored approach. Results are shown in the bar plot on Figure 5.3.

It is immediately noticeable that the monolithic approach in most cases displays a much poorer performance compared with the factored one. In the few cases where the monolithic algorithm outperforms the factored algorithm, it is only by a small margin. On the other hand, there are several cases where the factored algorithm outperforms the monolithic one by an order of magnitude or more. There are additionally a number of cases synthesized by the factored algorithm which the monolithic algorithm is not able to solve in the time limit. This indicates that it is worthwhile to take advantage of factored representation for synthesis, and that it allows a number of instances to become feasible compared to a monolithic representation.

These results raise the question of whether one should always give preference to a factored algorithm, even when it is known that the specification can be efficiently represented by a monolithic BDD. To try to answer this question, we repeated the

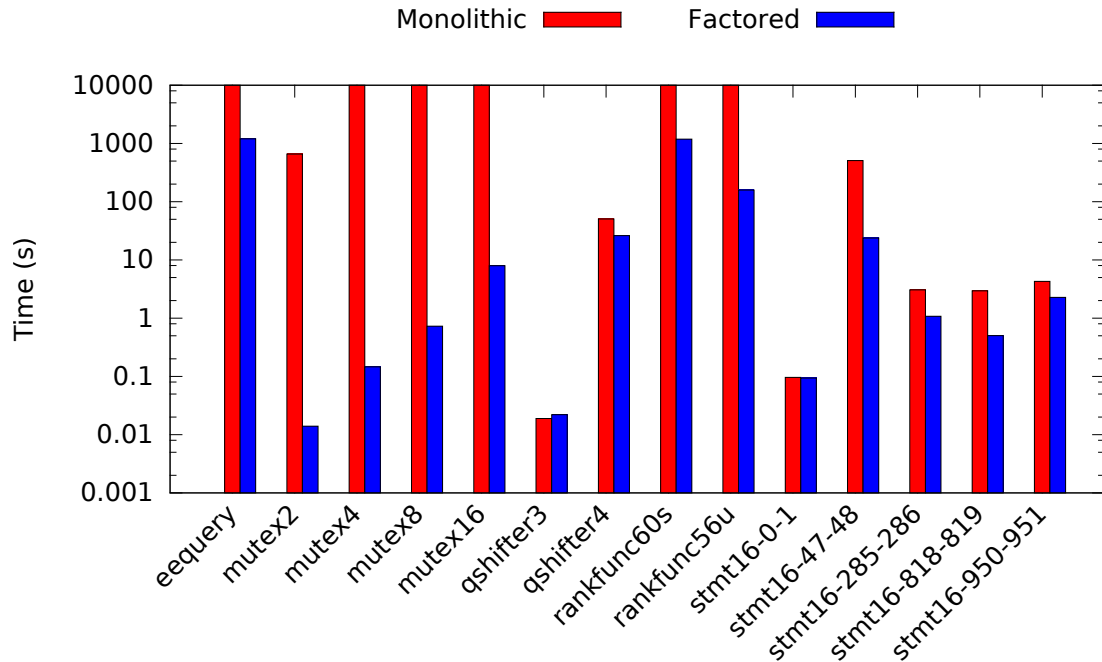


Figure 5.3 : Performance of the monolithic and factored synthesis algorithms, in log scale. Bars of maximum height indicate instances that timed out.

experiments performed in the previous chapter using the *Subtraction*, *Maximum*, *Minimum*, *Floor of Average* and *Ceiling of Average* classes of benchmarks, all of which have linear-sized monolithic BDD representations. This time, we ran each benchmark on both the monolithic and factored synthesis algorithms.

The results for these experiments can be found in Figure 5.4, which shows the running time of each algorithm on the *Subtraction*, *Maximum* and *Ceiling of Average* classes as a function of the length  $n$  of the vectors of Boolean variables. Although both algorithms scale similarly, for the *Subtraction* and *Maximum* classes the monolithic algorithm outperformed the factored algorithm by a significant margin, while for the *Ceiling of Average* class the factored algorithm performed slightly better. This demonstrates that it is not always preferable to use a factored algorithm when an

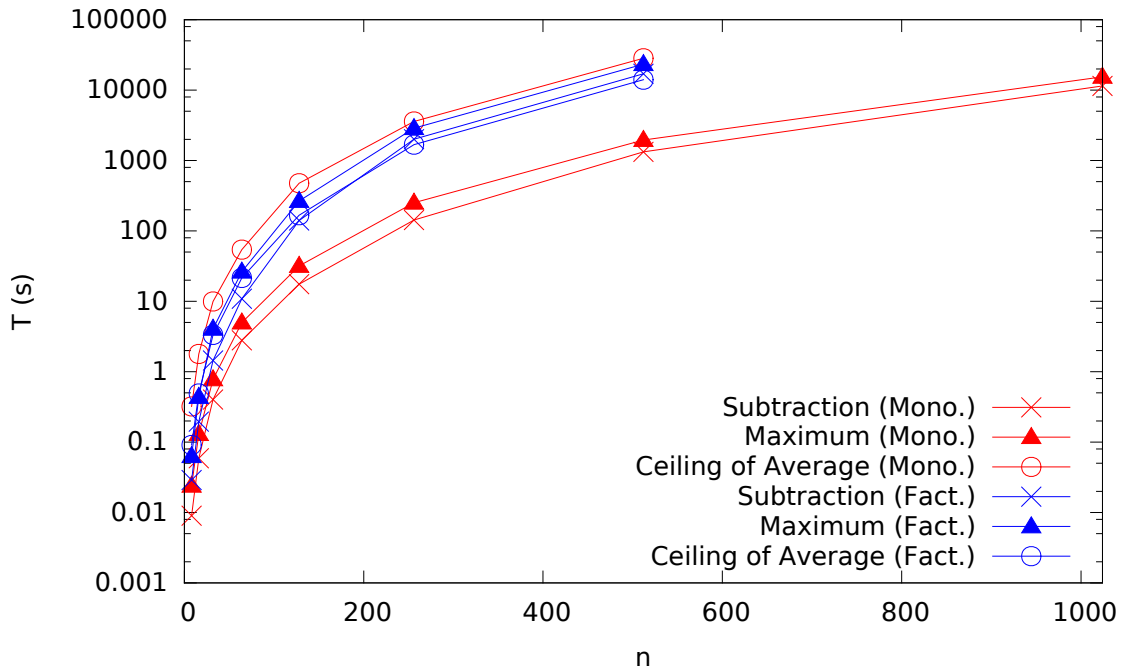


Figure 5.4 : Comparison of running time for monolithic and factored algorithms on the deterministic benchmarks of Section 4.5.

efficient monolithic representation is known. Note that the values of the running time are higher than the ones obtained in Section 4.5. This is due to the fact that the specifications were converted to CNF in order to obtain a factored representation. This conversion leads to the introduction of a high number of fresh variables which have to be existentially quantified during the algorithm, adding to the running time.

### 5.2.3 Comparison with CegarSkolem and CADET

Finally, we compare the performance of RSYNTH with the CEGAR-based approach used in the CEGARSKOLEM tool and the QBF solver CADET. Given the results of previous experiments, we select the factored algorithm with no reordering for the comparison.



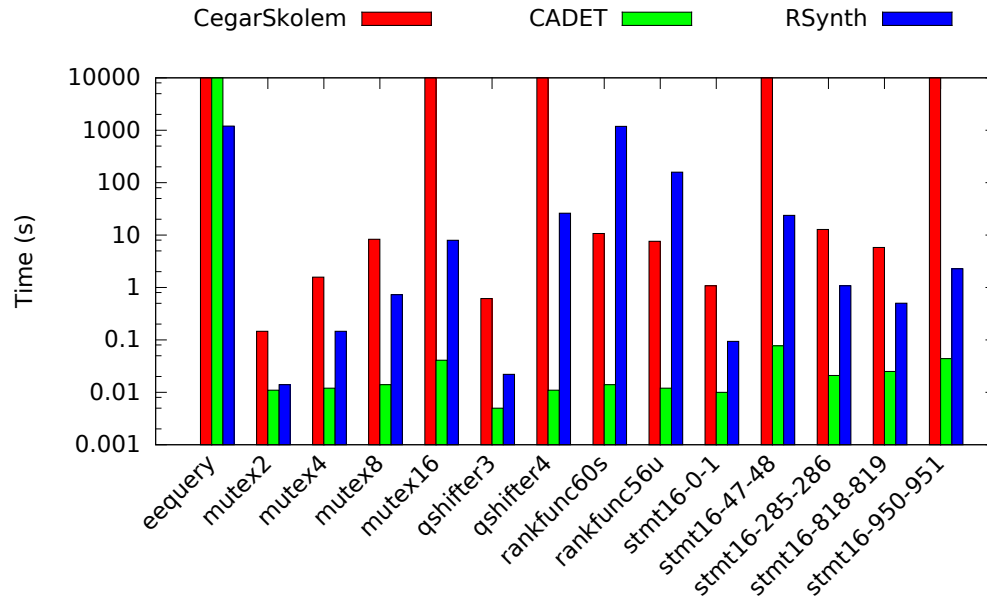


Figure 5.5 : Comparison of running time between RSYNTH, CEGARSKOLEM and CADET, in log scale. Bars of maximum height indicate instances that timed out.

Figure 5.5 shows a comparison of running time between RSYNTH, CEGARSKOLEM and CADET on the same benchmarks used in the previous experiments. All of the benchmarks are realizable, allowing CADET to be used for them. Out of 161 total benchmarks, RSYNTH was able to synthesize 87 and CEGARSKOLEM 52. There were only 6 benchmarks in which CEGARSKOLEM outperformed RSYNTH, all from the *rankfunc* class. However, CADET had by far the best performance in almost all instances, usually by orders of magnitude, and was able to synthesize all but one of the 161 benchmarks. This leads to the conclusion that the QBF approach is preferable when the specification is realizable.

Figure 5.6 shows a comparison of the size of the synthesized functions between the three tools. RSYNTH produces functions in the form of BDDs, while CEGARSKOLEM and CADET produce functions in the form of AIGs, therefore the comparison is in

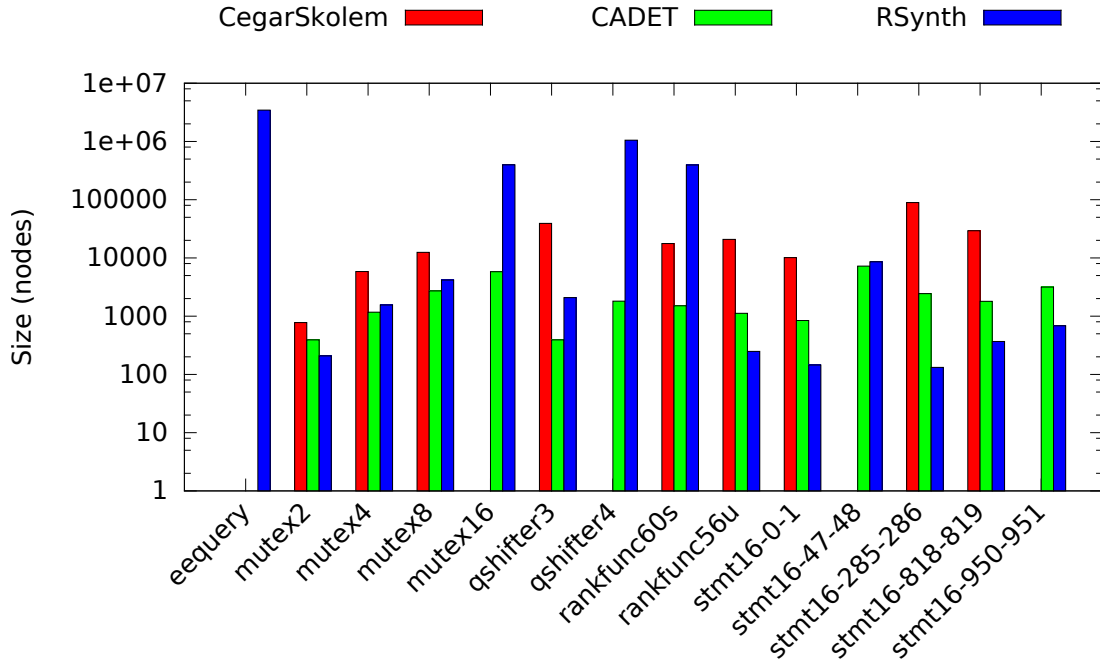


Figure 5.6 : Comparison of function size, in number of nodes, between RSYNTH, CEGARSKOLEM and CADET, in log scale. The size is not displayed for those instances in which the tool timed out.

number of nodes of these data structures. Missing bars mean that the tool timed out for that particular instance. RSYNTH produced smaller functions for about half of the benchmarks, while CADET had smaller functions for the other half. This demonstrates that in many cases BDDs are indeed able to produce a more compact representation than the one obtained by AIGs.

The main conclusion that we can draw from this comparison is that, for realizable specifications, synthesis approaches based on QBF will likely dominate in terms of running time. In general, however, QBF solvers do not support the generation of Skolem functions when the formula  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y})$  evaluates to *false*, i.e.,  $f(\vec{x}, \vec{y})$  is unrealizable. Therefore, for unrealizable specifications it becomes necessary to turn

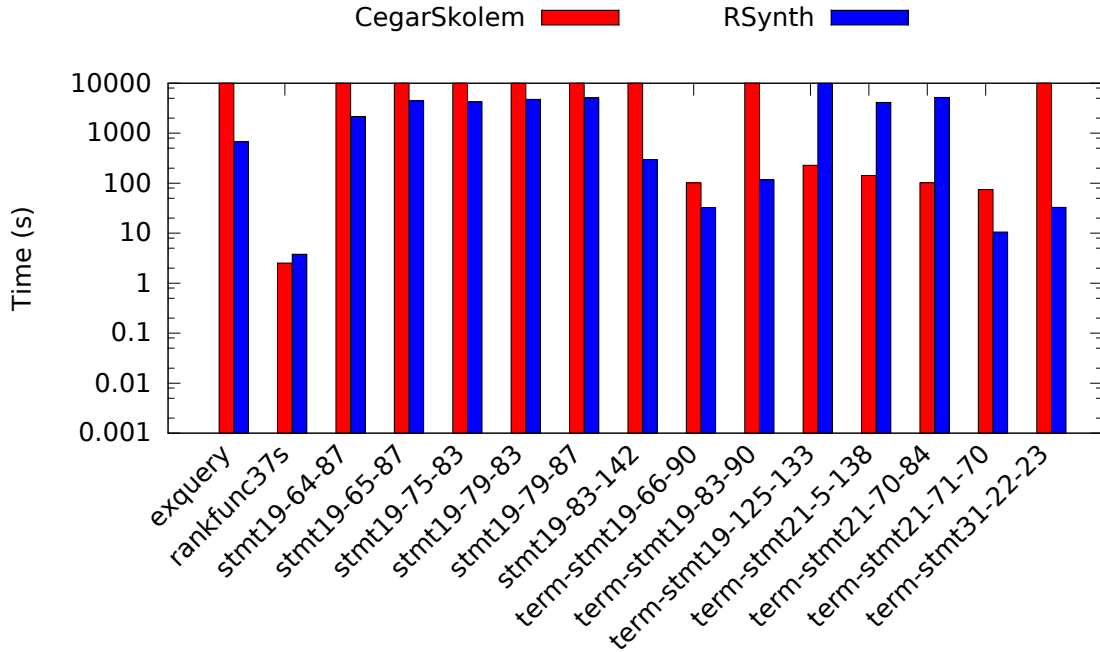


Figure 5.7 : Comparison of running time between RSYNTH and CEGARSKOLEM tools, in log scale, over unrealizable benchmarks. Bars of maximum height indicate instances that timed out.

to other synthesis approaches. This brings up the question of how RSYNTH and CEGARSKOLEM perform in synthesizing unrealizable instances. The next section presents an evaluation dedicated to answering this question.

#### 5.2.4 Unrealizable Specifications

For the next comparison, we also used QBF benchmarks of the form  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y})$  from QBFLIB. This time, however, the quantified formulas evaluate to *false*, meaning that  $f(\vec{x}, \vec{y})$  is unrealizable. Since CADET is unable to handle such cases, we only perform a comparison between RSYNTH and CEGARSKOLEM for these formulas.

Figure 5.7 shows the running time of each tool in a set of unrealizable benchmarks.

Comparing `RSYNTH` and `CEGARSKOLEM`, we see that the results vary depending on the instance, with either tool outperforming the other on a subset of the benchmarks. There are also many cases which one of the tools is able to synthesize while the other times out. In total, 227 benchmarks were solved by at least one of the tools, with `CEGARSKOLEM` performing best in 118 cases, and `RSYNTH` performing best in the remaining 109. This result suggests that no approach is strictly better than the other, and the best choice will likely depend on the specific instance of the problem.

### 5.2.5 QBF for Unrealizable Formulas

The performance of QBF solvers when the specification is realizable invites the question of whether we can find a way to exploit them in synthesizing unrealizable formulas as well. It turns out that it is possible to transform an unrealizable formula into a realizable one with the same witnesses by adding an additional quantifier alternation. This idea is well known in the context of arithmetic realizability [38]. In our case, given a quantified formula  $\forall \vec{x}. \exists \vec{y}. f(\vec{x}, \vec{y})$ , we can construct a formula  $\forall \vec{x}. \exists p. (p \leftrightarrow \exists \vec{y}. f(\vec{x}, \vec{y}))$ , which is always true. By a few simple transformations, we obtain  $\forall \vec{x}. \exists p. (\neg p \vee \exists \vec{y}. f(\vec{x}, \vec{y})) \wedge (p \vee \forall \vec{y}. \neg f(\vec{x}, \vec{y}))$ , and by renaming variables and moving the quantifiers to the front, the resulting formula is  $\forall \vec{x}. \exists p. \exists \vec{y}. \forall \vec{z}. ((\neg p \vee f(\vec{x}, \vec{y})) \wedge (p \vee \neg f(\vec{x}, \vec{z})))$ . Note that the Skolem function for the additional existentially quantified variable  $p$  now corresponds exactly to the precondition. Although this produces a QBF that is guaranteed to evaluate to *true*, note that the resulting formula is not in CNF. In particular, if  $f$  is originally in CNF, its negation in the second conjunct is now in DNF. Converting the formula back to CNF, as required by most modern QBF solvers, requires extra existential variables, which results in an additional quantifier alternation.

Because of the additional quantifiers, the formula is no longer in 2QBF, and therefore can no longer be handled by CADET. Therefore, to evaluate the feasibility of this method, we instead employed the DEPQBF [18] solver, which accepts formulas of arbitrary quantification depth. In an attempt to make solving the formulas easier for DEPQBF, we first ran them through the BLOQQER preprocessor [39], which uses a number of rewriting techniques to simplify the formula. Despite these efforts, for almost all benchmarks in Fig. 5.7, DEPQBF was unable to synthesize an implementation in the time limit. The only exception was the *rankfunc37s* benchmark, which was solved already in the preprocessing stage by BLOQQER, in about 1.3s.

This result suggests that the translation to realizable formulas is not an efficient way of using QBF solvers to perform synthesis of unrealizable specifications. A better approach would likely be to modify current QBF-solving algorithms, such as the one used by CADET, to also produce (partial) Skolem functions when the input formula is false. Further exploring QBF solving as a means for Boolean synthesis is left for future work.

### 5.3 Conclusion

In this chapter, we adapted techniques for processing factored representations of Boolean formulas using BDDs to the problem of Boolean functional synthesis. We show that these techniques allow synthesis from a number of specifications which cannot be handled when using a monolithic representation.

We performed an experimental comparison of our tool RSYNTH with other tools for Boolean synthesis, namely the CEGAR-based tool CEGARSKOLEM [15] and the QBF solver CADET [17]. Our experiments show the QBF approach to be very efficient when the specification is realizable, significantly outperforming the others.

However, QBF solvers are not generally able to synthesize functions for unrealizable specifications, which motivates the use of alternative approaches such as the one presented in this paper. For unrealizable specifications, the results of the comparison between `RSYNTH` and `CEGARSKOLEM` vary, with the best tool depending on the specific instance. Therefore, we conclude that there is no single approach that dominates over all cases, rather every tool is able to handle some specifications that the others cannot.

## Chapter 6

### Conclusion and Future Work

This thesis presented an exploration of Binary Decision Diagrams as data structures for Boolean functional synthesis. Our first contribution, in Chapter 4, was the introduction of the Self-Substitution technique, which can be used for both quantifier elimination and extraction of witnesses of Boolean functions. Using Self-Substitution as a foundation, we designed a Boolean synthesis framework using BDDs as the underlying data structure. Chapter 4 also introduced the `TRIMSUBSTITUTE` method, which gives rise to a more efficient synthesis algorithm for BDDs following a specific variable ordering that we call input-first. The synthesis framework presented in this thesis was implemented in a tool called `RSYNTH`. Through our experimental evaluation, we showed that when an efficient variable ordering is available for the BDDs, `RSYNTH` outperforms tools based on alternative approaches.

However, computing an efficient variable ordering is not always feasible. To handle such cases, in Chapter 5 we adapted techniques for manipulating factored representations of Boolean relations, originated in the context of model checking, to Boolean functional synthesis. By extending `RSYNTH` to handle factored representations, we were able to synthesize a number of cases for which the monolithic approach fails, showing that BDDs can be competitive with other approaches.

An advantage of BDD-based techniques lies on their applicability to reactive synthesis, which is one of the main applications of Boolean synthesis. BDDs are a popular symbolic representation for sets of states in this problem, since they allow for easy

equivalence checking when performing the fixpoint computations that are usually employed in reactive-synthesis algorithms. Furthermore, the Boolean formulas obtained in the context of this problem are often unrealizable, meaning that they define partial functions. This makes other approaches which do not handle unrealizable formulas, such as QBF-based ones, harder to apply. Finally, temporal specifications are often given as conjunctions of constraints, which might indicate that techniques based on factored representations could be applicable. These points suggest that a synthesis approach using a factored BDD representation might be a good choice for this problem. Therefore, our main direction for future research is in pursuing forms of integrating the techniques presented here in frameworks for reactive synthesis.

At the same time, the use of QBF-solving techniques for Boolean synthesis requires further exploration. QBF solvers give excellent results for realizable specifications, but current algorithms do not handle specifications that are unrealizable. How to modify these algorithms so they can be used for synthesizing unrealizable specifications is still an open question.

The Self-Substitution method also calls for further research. This technique can be of interest both in applied settings that make use of quantifier elimination, for example in symbolic model checking [11], and in theoretical settings. For example, the relation between Self-Substitution and function composition suggests that it might be relevant for the study of Post classes and algebraic clones [40], which relate to the result of composing Boolean functions.

It would also be interesting to find options for generalizing Boolean synthesis and the techniques used in this thesis to other domains and logic systems. As mentioned in Section 3.3, delving deeper into the problem of Boolean unification might be relevant for this goal, as from theoretical results from this area it can be derived that



Self-Substitution can be generalized to arbitrary Boolean rings. Results from Boolean unification also give rise to other kinds of generalization, such as finding alternative witnesses to the default-1 and default-0 ones presented in Chapter 4, or even synthesizing procedures for enumerating all possible outputs rather than computing a single one.

## Bibliography

- [1] R. Brayton and F. Somenzi, “Minimization of Boolean Relations,” in *IEEE Int’l Symp. on Circuits and Systems*, pp. 738–743, IEEE, 1989.
- [2] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.
- [3] R. Ehlers, “Symbolic Bounded Synthesis,” in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pp. 365–379, 2010.
- [4] B. Jobstmann, S. J. Galler, M. Weiglhofer, and R. Bloem, “Anzu: A Tool for Property Synthesis,” in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pp. 258–262, 2007.
- [5] N. Piterman, A. Pnueli, and Y. Sa’ar, “Synthesis of Reactive(1) Designs,” in *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pp. 364–380, 2006.
- [6] Z. Manna and R. Waldinger, “A Deductive Approach to Program Synthesis,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 90–121, 1980.

- [7] A. Pnueli and R. Rosner, “On the Synthesis of a Reactive Module,” in *Proc. 16th ACM Symp. on Principles of Programming Languages*, pp. 179–190, 1989.
- [8] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, “Complete Functional Synthesis,” in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010* (B. G. Zorn and A. Aiken, eds.), pp. 316–329, ACM, 2010.
- [9] E. Tronci, “Automatic Synthesis of Controllers from Formal Specifications,” in *ICFEM*, pp. 134–143, 1998.
- [10] J. Kukula and T. Shiple, “Building Circuits from Relations,” in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings* (E. Emerson and A. Sistla, eds.), vol. 1855 of *Lecture Notes in Computer Science*, pp. 113–123, Springer, 2000.
- [11] J. Burch, E. Clarke, and D. Long, “Representing Circuits More Efficiently in Symbolic Model Checking,” in *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 403–407, ACM, 1991.
- [12] G. Pan and M. Vardi, “Symbolic Techniques in Satisfiability Solving,” *J. Autom. Reasoning*, vol. 35, no. 1-3, pp. 25–50, 2005.
- [13] R. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Trans. Comput.*, vol. 35, pp. 677–691, Aug. 1986.
- [14] J. Jiang, H. Lin, and W. Hung, “Interpolating Functions From Large Boolean Relations,” in *2009 International Conference on Computer-Aided Design (ICCAD’09), November 2-5, 2009, San Jose, CA, USA*, pp. 779–784, IEEE, 2009.

- [15] A. K. John, S. Shah, S. Chakraborty, A. Trivedi, and S. Akshay, “Skolem Functions for Factored Formulas,” in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pp. 73–80, 2015.
- [16] M. Heule, M. Seidl, and A. Biere, “Efficient Extraction of Skolem Functions from QRAT Proofs,” in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pp. 107–114, 2014.
- [17] M. N. Rabe and S. A. Seshia, “Incremental Determinization,” in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 375–392, Springer, 2016.
- [18] F. Lonsing and A. Biere, “Integrating Dependency Schemes in Search-Based QBF Solvers,” in *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pp. 158–171, 2010.
- [19] U. Martin and T. Nipkow, “Boolean Unification - The Story so far,” *Journal of Symbolic Computation*, vol. 7, no. 3-4, pp. 275–293, 1989.
- [20] M. Fujita, Y. Tamiya, Y. Kukimoto, and K. chien Chen, “Application of Boolean Unification to Combinational Logic Synthesis,” in *in Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 510–513, 1991.
- [21] J. Jiang, “Quantifier Elimination via Functional Composition,” in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings* (A. Bouajjani and O. Maler, eds.), vol. 5643 of *Lecture Notes in Computer Science*, pp. 383–397, Springer, 2009.

- [22] A. Kuehlmann, M. Ganai, and V. Paruthi, “Circuit-Based Boolean Reasoning,” in *Proc. Design Automation Conference*, pp. 232–237, IEEE, 2001.
- [23] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic Model Checking:  $10^{20}$  States and Beyond,” *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.
- [24] V. Paruthi and A. Kuehlmann, “Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation,” in *Proc. Int’l Conf. on Computer Design*, pp. 459–464, IEEE, 2000.
- [25] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, “Automatic Hardware Synthesis from Specifications: a Case Study,” in *Proc. Conference on Design, Automation and Test in Europe*, pp. 1188–1193, ACM, 2007.
- [26] F. Somenzi, *CUDD: CU Decision Diagram Package Release 2.5.0*. University of Colorado at Boulder, 2012.
- [27] E. Goldberg and P. Manolios, “Quantifier elimination via clause redundancy,” in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 85–92, 2013.
- [28] D. Bañeres, J. Cortadella, and M. Kishinevsky, “A Recursive Paradigm to Solve Boolean Relations,” *IEEE Trans. Computers*, vol. 58, no. 4, pp. 512–527, 2009.
- [29] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu, “Programming by Sketching for Bit-Streaming Programs,” in *Proc. of the ACM Conference on Programming Language Design and Implementation*, pp. 281–294, ACM, 2005.

- [30] D. Fried, L. M. Tabajara, and M. Y. Vardi, “BDD-Based Boolean Functional Synthesis,” in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pp. 402–421, 2016.
- [31] D. Geist and I. Beer, “Efficient Model Checking by Automated Ordering of Transition Relation Partitions,” in *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, pp. 299–310, 1994.
- [32] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi, “Symbolic LTL<sub>f</sub> Synthesis,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017 (to appear).
- [33] S. Akshay, S. Chakraborty, A. K. John, and S. Shah, “Towards Parallel Boolean Functional Synthesis,” in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pp. 337–353, 2017.
- [34] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley, “Efficient bdd algorithms for fsm synthesis and verification,” in *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV, 1995*.
- [35] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “NuSMV 2: An OpenSource Tool for Symbolic Model Checking,” in *Computer Aided Verification, 14th International Confer-*

- ence, *CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pp. 359–364, 2002.
- [36] R. E. Tarjan and M. Yannakakis, “Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs,” *SIAM J. Comput.*, vol. 13, no. 3, pp. 566–579, 1984.
- [37] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella, “Quantified Boolean Formulas satisfiability library (QBFLIB),” 2005. [www.qbflib.org](http://www.qbflib.org).
- [38] U. Kohlenbach, *Applied Proof Theory - Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics, Springer, 2008.
- [39] A. Biere, F. Lonsing, and M. Seidl, “Blocked Clause Elimination for QBF,” in *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pp. 101–115, 2011.
- [40] M. Couceiro, S. Foldes, and E. Lehtonen, “Composition of Post Classes and Normal Forms of Boolean Functions,” *Discrete Mathematics*, vol. 306, no. 24, pp. 3223–3243, 2006.