Determinization of Boolean Relations Using Interpolants

Master's Thesis
in Computer Science

Matthias Schlaipfer

Determinization of Boolean Relations Using Interpolants

Master's Thesis

at

Graz University of Technology

submitted by

Matthias Schlaipfer

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

April 23, 2014

Advisor: Univ.-Prof. Roderick Bloem
Co-Advisor: Univ.-Prof. Sharad Malik

Determinisierung Boole'scher Relationen Mittels Interpolanten

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Matthias Schlaipfer

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie (IAIK),
Technische Universität Graz
A-8010 Graz

23. April 2014

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Univ.-Prof. Roderick Bloem
Mitbetreuer: Univ.-Prof. Sharad Malik

Abstract

In this thesis we present novel ways for solving Boolean relations. Solving a relation means computing a deterministic function which characterizes a subset of the mapping described by the relation. The goal is that for every input only a single (deterministic) output is possible.

Modern approaches for solving this problem can be divided into those based on binary decision diagrams and those based on satisfiability solving. In this work, we explore both methods and implement improvements. These improvements aim at reducing the number of input variables a function f, which is a solution for the relation, depends on. A lower number of input variables means a smaller size of the combinational circuit implementing f, which in prior solutions has not been satisfactory.

In the first approach, based on binary decision diagrams, we present two ways for finding an exact and globally optimal solution for eliminating input variables. Previous methods have found locally optimal solutions only. In the second approach we use satisfiability solving for obtaining a resolution proof and furthermore Craig interpolation to compute a circuit for f. As the interpolant is computed from the resolution proof, the number of variables can be reduced by reducing this proof. We improve and generalize existing proof reduction techniques.

We describe the two approaches in detail and analyze our experimental results.

Kurzfassung

In dieser Arbeit präsentieren wir neue Wege um Boole'sche Relationen zu lösen. Das bedeutet, eine Funktion zu berechnen, die einen Teil der nicht-deterministischen Relation charakterisiert, sodass es für jede Eingabe nur eine mögliche Ausgabe gibt.

Moderne Ansätze können in jene unterteilt werden, die auf Binären Entscheidungsdiagrammen basieren und in jene, die auf Satisfiability-Solvern beruhen. In dieser Arbeit untersuchen wir beide Methoden und implementieren Verbesserungen. Diese Verbesserungen zielen darauf ab, die Anzahl der Variablen von denen eine Funktion f, welche eine Lösung für die Relation darstellt, zu minimieren. Eine niedrigere Anzahl an Eingangsvariablen bedeutet eine kleinere kombinatorische Schaltung, die f implementiert. Besonders die Schaltungsgröße ist in den bisherigen Methoden nicht zufriedenstellend.

Der erste Ansatz beruht auf Binären Entscheidungsdiagrammen: Wir präsentieren zwei Methoden, um eine exakte und global optimale Lösung für die Minimierung der Eingangsvariablen zu finden. Der zweite Ansatz nutzt Satisfiability-Solving um einen Resolutionsbeweis und in weiterer Folge Craig-Interpolation um eine Schaltung für f zu erlangen. Die Interpolante wird anhand des Resolutionsbeweises berechnet. Daher ist es möglich die Anzahl der Variablen zu minimieren, indem man den Beweis minimiert. Wir verbessern und generalisieren existierende Beweisminimierungstechniken.

Wir beschreiben die beiden Ansätze im Detail und analysieren die Ergebnisse unserer Experimente.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

Acknowledgements

I am foremost dearly indebted to both Prof. Roderick Bloem and Prof. Sharad Malik, who have made a dream come true for me. For a long time I had wanted to study abroad—and do so in the United States. However, I had never thought that it would be possible to do so at a history-charged university such as Princeton. I could not have hoped for a better opportunity.

In this respect I want to continue and acknowledge the great help provided by Princeton University's staff in all organizational matters. The cultural diversity at Princeton has enriched my understanding of academia, but also my personal development in general. A major reason for this has been Sharad Malik's research group, who has welcomed me heartily. I especially want to thank Daniel Schwartz-Narbonne, who has been a great companion on various occasions and, who is an overall inspiring person. Furthermore I am indebted to Georg Weissenbacher, who patiently explained matters and provided guidance for my research. Georg has been nothing but helpful throughout this whole experience. I also want to thank my landlady Felice Weiner for being such a good hostess. Finally, I am grateful to the Austrian Marshall Plan Foundation for having generously supported my stay at Princeton.

This thesis is ending my time at TU Graz. Thanks are in order for the people, who have affected me here and have made the time so enjoyable. I especially want to thank Stefan Kölbl, who is on the same page as me on way too many matters. I am grateful for having had him as a companion throughout these years in Graz. Furthermore I want to thank the people at IAIK for sparking my interest in research and for providing a great environment within TU Graz. I would like to point out Georg Hofferek for his explanations and co-supervision in various instances over the last two years.

Last but not least, I want to thank my family for their loving support and for making it possible to focus on my studies over the course of the last seven years.

Chapter 1

Introduction

Over the last decade, computers have become increasingly ubiquitous. Every day, we are in contact with embedded computers in phones, cars, household appliances, etc. Computers have enabled new venues for science and new business opportunities. They have changed our leisure activities and the way we communicate. Programming computers correctly is not an easy task and has become harder because of increasing concurrency and more important because of increasing ubiquity of computer systems. Bugs plague almost every implementation and the goal of computer science to become a well-founded engineering discipline is still far from being reached.

The classical approach to correctness consists of massive testing. However, testing often misses faults. By testing, one is unable to say whether the software follows its specification perfectly or not. Therefore, formal methods [Flo67, Hoa69, QS82, CES86, BCM+92, BCCZ99, CGJ+00, VHB+03] are gaining importance, as evidenced by the 2007 ACM Turing Award for Model Checking. In recent years there has been great progress towards practical usability of software verification in particular. Microsoft, for example, uses a "push-button tool" [BR02] to find bugs in hardware drivers. Using this approach, one can be sure about the correctness (respectively faultiness) of certain aspects of the software.

As of late, there has been a push away from seeing formal verification purely as a method to validate programs after they have been written, including faults. A new paradigm is slowly emerging that uses the techniques pioneered in the formal verification world to the problem of a-priori assistance of the programmer in writing correct programs. Automatic synthesis, or property synthesis [Chu62, PP06, SGF10, KMPS10, HB11, HGK+13] is a typical example of this approach: it uses techniques from the model checking world to automatically construct correct systems from their specifications. Synthesis, however, still has significant problems, preventing it from being used in realistic situations.

One of them is solving large Boolean relations, which we will attack in this thesis. It is a classical problem that has been addressed in the logic synthesis community [VOQ52, Mcc56, Law64, BS89, WB91, DM94, HS96]. Logic synthesis should not be confused with the property synthesis paradigm: Logic synthesis provides solutions to a sub-problem of property synthesis. Preliminary research has shown that the standard solutions from logic synthesis do not perform well in a property synthesis setting. Only small specifications can be synthesized, and the resulting systems are orders of magnitude larger than manual implementations [BGJ+07]. We apply novel techniques to achieve more efficient and concise solutions.

One way to model the synthesis process is game theory [PP06]. We describe where solving relations is necessary in this approach: The "game" is played between the environment and the system, which is synthesized. The environment moves by sending arbitrary Boolean inputs to the system. The system receives them and has to counter the environment by sending outputs back, while adhering to the rule set laid out by the specification. The synthesis problem is to find a strategy for the system which allows to counter any move by the environment, in order for the system to win eventually. A specification typically allows for multiple system moves in a given game state. In other words: the strategy is non-deterministic. The system is supposed to map Boolean inputs to Boolean outputs in a deterministic way, however. Such a mapping is called a combinational circuit. In order to compute this circuit, we must pick which move to make in a given state. The non-deterministic strategy is represented by a Boolean relation. Choosing the moves means solving, or determinizing, this relation. The challenge is twofold: On the one hand, we have to deal with large problem instances. On the other hand, we want to solve the relation in such a way that the circuit is small in the end, where size is typically measured in the number of logic gates needed. Current approaches do not scale well to larger problem instances, as they are slow and yield systems which are orders of magnitude larger than manual implementations.

Therefore, we pursue ideas which target the creation of small circuits. We describe both exact solutions, which however turned out to be infeasible in practice, as well as an approximative one, which improves over existing approaches without incurring additional cost. The heuristic we employed in both cases was to find solutions which depend on few input variables.

The exact approach is built on top of an existing determinization algorithm [BGJ+07] based on binary decision diagrams (BDDs) [Ake78]. The previous technique computed a solution based on local optima, in terms of the number of input variables. In our approach we search for a globally optimal solution in two different ways:

1. By explicitly enumerating combinations of variables one by one.

2. By adding circuitry to the combinational logic representing the relation, which implicitly enumerates variable combinations.

We implemented both approaches in RATSY [BCG+10], but our experimental evaluation revealed that these exact approaches are practically infeasible.

We then turned our attention to an approximative approach. We base this work on a result by Jiang, Lin and Hung [JLH09], which shows that a relation can be solved using Craig interpolation [Cra57]. Craig's interpolation theorem states the following:

Theorem. *Given two Boolean formulas A and B, with $A \wedge B$ unsatisfiable, there exists a Boolean formula I referring only to the common variables of A and B such that $A \rightarrow I$, and $I \wedge B$ is unsatisfiable.*

I is called the interpolant of A and B. Interpolants can be obtained by annotating resolution refutation proofs obtained by Boolean satisfiability (SAT) solvers. Therefore, we can take advantage of the progress made in the development of SAT solvers over the course of the last two decades (e.g. [MMZ+01, SS96]).

While interpolation inherently only talks about the shared alphabet of A and B it is possible to minimize the amount of variables in the interpolant by using a certain interpolation system as described in [D'S10].

Our approach to reduce the size of the interpolant further, is to minimize the resolution refutation obtained by a SAT solver. Recent approaches [BIFH+09, FMP11, Gup12] achieve up to 22.54% reduction of the number of proof vertices, by transforming the proof graph after solve-time. We show that reduction of the proof graph does not necessarily reduce the amount of variables in the interpolant, but in fact can increase it. Existing techniques don't take this into account, as their focus lies on proof reduction, rather than on interpolant reduction. We present a way to prevent certain unfavorable transformations, which lead to an increase in interpolant size. Our algorithms also improve the existing techniques in terms of proof reduction. Furthermore, our approach is more dynamic, in the sense that it can target either proof or interpolant reduction, based on parametrization.

We implemented our techniques in a stand-alone tool written in Scala and provide experimental evaluation comparing it to the best-performing algorithm from [Gup12], which shows that our technique achieves better proof reduction. We furthermore evaluate how many variables are removed from the final interpolant in both approaches. A metric, which to our knowledge, has not been looked at so far.

1.1 Organization of this Thesis

The thesis is split into the following chapters:

1. We provide the theoretical foundations and general terminology in Chapter 2. Topics are Boolean functions and relations, logic represention such as BDDs and normal forms as well as satisfiability solving and interpolation. We also give a brief introduction to proof reduction, but in-depth treatment is provided only in Chapter 5.
2. We discuss previous work concerned with logic minimization in Chapter 3.
3. We describe our BDD-based approaches in Chapter 4 and the approach based on interpolation in Chapter 5.
4. We conclude in Chapter 6, by looking back at our work and by providing an outlook on possible future improvements.

A draft version of this thesis, which was submitted to the Austrian Marshall Plan Foundation, is available at [Sch].

Chapter 2

Background

This chapter introduces the necessary preliminaries and establishes notation to understand the relation determinization problem and the presented solutions. This thesis cannot be a complete treatise of all the subjects involved. The interested reader can find further and more detailed information in the referenced works.

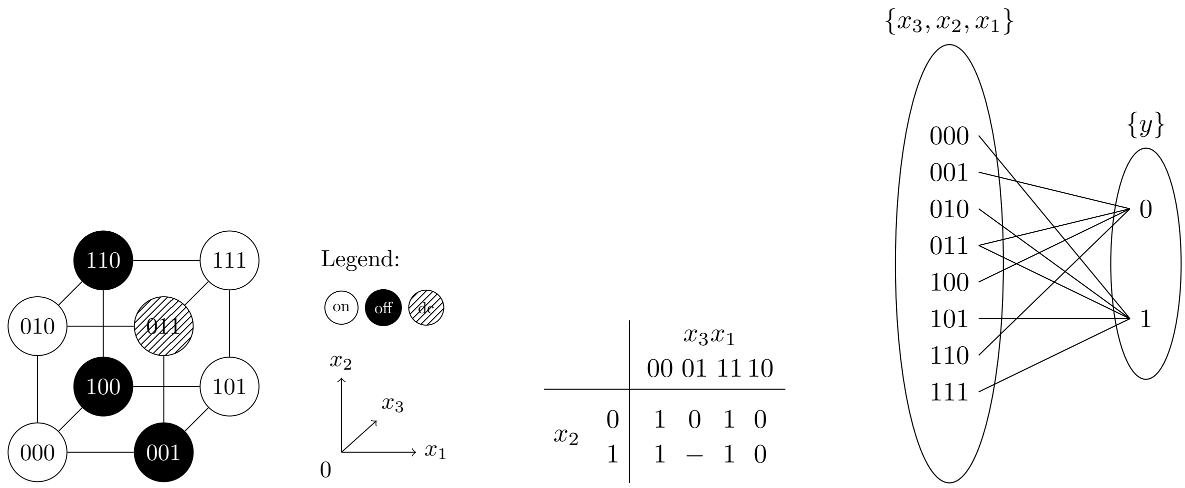
2.1 Boolean Logic

Boolean logic lies at the heart of computing as we know it. Digital circuits implement Boolean functions referred to as combinational logic. Boolean logic is two-valued: These two truth values are **false** and **true**, represented by the set $\mathbb{B} = \{0, 1\}$ or sometimes also $\{\mathsf{T}, \mathsf{F}\}$. A Boolean variable can be assigned either value of $\mathbb{B}$. The Boolean space is spanned by n Boolean variables $\vec{x} = (x_1, \ldots, x_n)$ and written as $\mathbb{B}^n$. The 2^n members (vertices) of $\mathbb{B}^n$ are called **minterms**. A minterm, in other words, is a total assignment of truth values to the n Boolean variables.

2.1.1 Boolean Functions

A **completely specified Boolean single-output function** $f : \mathbb{B}^n \mapsto \mathbb{B}$ maps the minterms of the **Boolean space** to either 0 or 1. The domain $\mathbb{B}^n$ is referred to as the **input space** and the codomain as the **output space**, respectively.

In some applications it is not necessary to completely specify a Boolean function–it doesn't matter for some minterms whether they are mapped to 0 or 1. This condition is called **don't care** and represented by a dash "$-$". A partial function is a function which does not define a mapping for each member of the domain into the codomain. The unmapped minterms of a partial Boolean function are treated as being mapped to $-$. Let



(a) Coloring of minterms. **(b)** Karnaugh map. **(c)** Input-output map.

Figure 2.1: Three different ways of illustrating the same incompletely specified Boolean function $f(x_1, x_2, x_3) = y$.

$\mathbb{B}_+ = \mathbb{B} \cup \{-\}$. An **incompletely specified Boolean single-output function** is then denoted as $f : \mathbb{B}^n \mapsto \mathbb{B}_+$. A simple such function, in three input variables and an output variable, is depicted as a coloring of minterm vertices in Figure 2.1a. Another way of representing such a function is as a Karnaugh map [Kar53] as can be seen in Figure 2.1b.

2.1.2 Boolean Relations

A more expressive way to describe Boolean mappings are **Boolean relations**. A relation is a set of ordered pairs (x, y), where x is a member of the domain and y is a member of the codomain. A Boolean relation $R \subseteq X \times Y$ (also written as $R(X, Y)$) is represented by its **characteristic function** $R : X \times Y \mapsto \mathbb{B}$, with $X = \mathbb{B}^n$ and $Y = \mathbb{B}^m$. The input space X is spanned by variables $\vec{x} = (x_1, \ldots, x_n)$ and the output space Y by $\vec{y} = (y_1, \ldots, y_m)$. The characteristic function is defined, such that $(x, y) \in R$ if and only if $R(x, y) = 1$ for $x \in X$ and $y \in Y$. Notice that in general, the output space can be of dimension $m > 1$. The relations handled in this thesis typically have $m = 1$, as reasoning about such single-output relations is easier. Section 2.4.1 presents a scheme for handling multiple-output relations by breaking them down to single-output relations.

Notice also that with Boolean relations there is no need for the augmented set $\mathbb{B}_+$, since relations—in contrast to functions—allow one-to-many mappings. A relation is said to be **total** (in the input space), if and only if the set $\{x \mid \exists y.\ (x, y) \in R\}$ is equal to $\mathbb{B}^n$. Otherwise it is a **partial** relation.

A typical way of representing a Boolean relation graphically is shown in Figure 2.1c. The set of input space minterms is on the left-hand-side and the output space on the right-hand-side. If $(x, y) \in R$ then $x \in X$ and $y \in Y$ are connected by an edge.

2.1.3 Terminology

Let $f(x_1, \ldots, x_n)$ be a completely specified Boolean single-output function and $R(x_1, \ldots, x_n, y)$ a Boolean single-output relation. Then the set of minterms mapped to 0 is called the **off-set** of f (and R respectively). The **on-set** is the set of minterms mapped to 1. The formal definitions are as follows.

$$f^0 = \{x \in \mathbb{B}^n \mid f(x) = 0\}, f^1 = \{x \in \mathbb{B}^n \mid f(x) = 1\}\}$$
$$R^0 = \{x \in \mathbb{B}^n \mid R(x, 0) = 1\}, R^1 = \{x \in \mathbb{B}^n \mid R(x, 1) = 1\}$$

For relations, there might be an overlap of the on-set and the off-set. Therefore, there is another set defined which represents the minterms mapping to both 0 and 1. This set is the **dc-set** and defined as $R^0 \cap R^1$. If $f^1 = \mathbb{B}^n$ then f is said to be a **tautology** or **valid**. If $f^0 = \mathbb{B}^n$ then f is **unsatisfiable**, otherwise $f^1 \neq \emptyset$ and f is **satisfiable**. A **literal** is a variable or its complement, written as x or $\overline{x}$, respectively. $\texttt{Lit}_{\vec{x}} = \{x, \overline{x} \mid x \in \vec{x}\}$ is the set of literals over X. The **negative** and **positive cofactors** of f with respect to x_i are defined as

$$f_{x_i=0} = f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n),$$
$$f_{x_i=1} = f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n).$$

A **cube** is a subset of the Boolean space, spanned by $k \leq n$ variables. If $k = n$, the cube is a minterm. A **substitution** of a variable x_i in f by a function g is written as $f|_{x_i=g}$.

2.2 Boolean Function Representations

There are many ways to represent Boolean functions. Common ones are truth tables, propositional logic, disjunctive normal form, conjunctive normal form, circuit graphs, or binary decision diagrams, to name just a few. All representations have certain benefits and drawbacks and their applicability depends on the particular use case. The representations can of course be converted between each other, but this might come at the cost of a jump in representation size. We will describe the representations important to our applications.

Name	Notation	Alternative	Read as
Negation	$\overline{x}$	$\neg x$	not x
Conjunction	$x \cdot y$	$x \wedge y$	x and y
Disjunction	$x + y$	$x \vee y$	x or y
Implication	$x \rightarrow y$		x implies y
Bi-implication	$x \leftrightarrow y$	$x \equiv y$	x bi-implies y

Table 2.1: Name and notation of the logic connectives.

2.2.1 Propositional Logic

Propositional logic is a formal system that lets us express propositions. A proposition is a statement which is either false or true, such as "the streets are wet". Propositional logic allows to formalize every Boolean function (and therefore every Boolean relation, since relations are represented by their characteristic functions).

2.2.1.1 Syntax and Notation

Propositional statements are constructed from a set of propositional symbols (variables) $X = \{x, x_1, x_2, \ldots, x_n, y, z\}$, the Boolean constants $\{0, 1\}$ and logic connectives $\{\overline{}, \cdot, +, \rightarrow, \leftrightarrow\}$. Sometimes the alternative connectives given in Table 2.1 are used. We refer to the variables occurring in a formula F as $\mathrm{Var}(F)$. The following grammar in Backus-Naur Form provides the rules for stating well-formed propositional logic formulas (wffs):

$$
\begin{aligned}
\langle \text{wff} \rangle &::= (\langle \text{wff} \rangle) \mid \overline{\langle \text{wff} \rangle} \mid \langle \text{wff} \rangle \cdot \langle \text{wff} \rangle \mid \\
&\quad\ \langle \text{wff} \rangle + \langle \text{wff} \rangle \mid \langle \text{wff} \rangle \rightarrow \langle \text{wff} \rangle \mid \\
&\quad\ \langle \text{wff} \rangle \leftrightarrow \langle \text{wff} \rangle \mid \langle \text{atom} \rangle \\
\langle \text{atom} \rangle &::= \langle \text{constant} \rangle \mid \langle \text{propositional symbol} \rangle \\
\langle \text{constant} \rangle &::= 0 \mid 1 \\
\langle \text{propositional symbol} \rangle &::= x \mid x_1 \mid \ldots \mid x_n \mid y \mid z
\end{aligned}
$$

The symbol $\equiv$ is used to denote logical equivalence. Following this definition, an example for a propositional formula f is $f \equiv ((x_1 + (\overline{x_1}) \cdot x_2) \rightarrow ((\overline{x_3}) \rightarrow x_4))$.

We use the following precedence rules of the connectives for evaluating formulas.

$$\text{Negation} \gtrdot \text{Conjunction} \gtrdot \text{Disjunction} \gtrdot \text{Implication} \gtrdot \text{Bi-implication}$$

The rule $a \gtrdot b$ is read as "a has precedence over b". Moreover, the binary connectives $\cdot, +, \leftrightarrow$ are left-associative, while $\rightarrow$ is right-associative.

For brevity, we sometimes drop either (consistently, such that there are no confuscions)

Op	Function	On-set	Off-set
$-$	$f \equiv \overline{g}$	$f^1 = g^0$	$f^0 = g^1$
$\cdot$	$f \equiv g \cdot h$	$f^1 = g^1 \cap h^1$	$f^0 = g^0 \cup h^0$
$+$	$f \equiv g + h$	$f^1 = g^1 \cup h^1$	$f^0 = g^0 \cap h^0$
$\rightarrow$	$f \equiv g \rightarrow h$	$f^1 = g^0 \cup h^1$	$f^0 = g^1 \cap h^0$
$\leftrightarrow$	$f \equiv g \leftrightarrow h$	$f^1 = (g^0 \cup h^1) \cap (g^1 \cup h^0)$	$f^0 = (g^1 \cap h^0) \cup (g^0 \cap h^1)$

(a) Set representation.

x	y	$\overline{x}$	$x \cdot y$	$x + y$	$x \rightarrow y$	$x \leftrightarrow y$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

(b) Truth table representation.

Table 2.2: Semantics of the logic connectives.

the conjunction or disjunction connective in between propositional symbols.

2.2.1.2 Semantics

To interpret a propositional statement, the semantics of the formalism must be defined. The truth value of a formula f depends on its interpretation under some environment. An environment is an assignment $\mathcal{A} : \mathcal{V} \mapsto \mathbb{B}$ to the propositional symbols in f. The meaning of the logic connectives can either be defined by operations on the on and off-sets of the functions (Table 2.2a), or by the more typical means of a truth table (Table 2.2b), enumerating the possible assignments.

2.2.1.3 Quantified Boolean Formulas

Quantified Boolean formulas (QBFs) provide syntactic additions to propositional logic. We use them to formalize and solve certain problems arising with Boolean functions and relations. The syntax of QBF is propositional logic, augmented with the **for all** ($\forall$) and the **exists** ($\exists$) **quantifiers**.

Definition 1. *Let $f(x, y)$ be a Boolean function, then the quantifiers are defined as*

$$\forall y.\ f(x, y) \equiv f(x, 0) \cdot f(x, 1),$$
$$\exists y.\ f(x, y) \equiv f(x, 0) + f(x, 1).$$

Quantified variables are called **bound** variables and unquantified variables are called **free** variables. In both cases of Definition 1 y is bound, whereas x is free. It can be

Name	Notation	read as
Universal quantification	$\forall x.\ f(x)$	True if $f(x)$ is true for all choices of x
Existential quantification	$\exists x.\ f(x)$	True if $f(x)$ is true for at least one choice of x

Table 2.3: Name and notation of quantifiers.

seen that every QBF can be rewritten to an equivalent propositional formula by formula expansion.

2.2.2 Reduced Ordered Binary Decision Diagrams

An important data structure for representing Boolean formulas is the reduced ordered binary decision diagram. It is a graph-based data structure and allows representation as well as manipulation of Boolean functions. Typically its name is shortened to just binary decision diagram, or BDD. The BDD data structure has been around since 1978 [Ake78], but gained traction in 1986 when Bryant's seminal paper "Graph-based algorithms for Boolean function manipulation" [Bry86] was published. BDD-based approaches have been very successful, especially in the field of logic synthesis and symbolic model checking. A section dedicated to BDDs in Knuth's "The Art Of Computer Programming" [Knu09] hints at the importance and powerfulness of the data structure for combinatorial problems. It is easiest to think of BDDs as a more compact representation of **ordered binary decision trees**.

In the following, ordered binary decision trees are defined and notation is established. Subsequently, two reduction rules on these trees are presented, whose application leads directly to the DAG-structure of BDDs. Subsection 2.2.2.3 shows how BDDs in practice are built in a more efficient manner. Subsection 2.2.2.4 provides information on how the logic operations are implemented on the data structure.

2.2.2.1 Ordered binary decision trees

Let $\mathcal{V}$ be the set of propositional variables in the function that we want to encode as a decision tree.

Definition 2 (Decision Tree)**.** *A decision tree (V, E) is a rooted, directed graph with a set of vertices V and a set of edges E. There are two different types of vertices in V.*

1. *A **non-terminal** vertex v is labelled with a propositional variable $\mathrm{var}(v) \in \mathcal{V}$ and possesses a corresponding index argument $index(v) \in \{1, \ldots, |\mathcal{V}|\}$. Moreover, every non-terminal vertex has two children $low(v)$ and $high(v) \in V$. The edge from v to $low(v)$ is labelled 0 and the edge to $high(v)$ is labelled 1.*

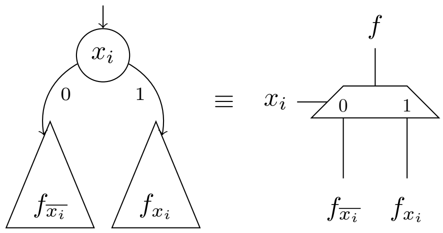


Figure 2.2: Equivalence of a BDD vertex and a 2-to-1 multiplexer.

2. *The second type of vertices are* ***terminal*** *vertices. A terminal vertex v is labelled with a constant value $val(v) \in \mathbb{B}$ and given the index $(|V| + 1)$.*

An ordering is imposed on the tree by the conditions $index(v) < index(low(v))$ and $index(v) < index(high(v))$. Every path starting in the root and ending in a terminal vertex must adhere to the same ordering. A variable order relation typically is written as $x_1 < x_2$, meaning that for all $v_1 \in \{v \in V \mid \mathrm{var}(v) = x_1\}$ and $v_2 \in \{v \in V \mid \mathrm{var}(v) = x_2\}$, the condition $index(v_1) < index(v_2)$ has to hold.

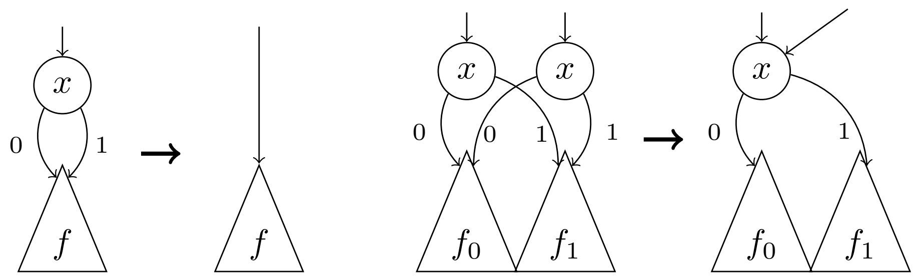
The semantics associated with this tree structure follows from Boole's expansion [Boo54] Theorem (also known as Shannon's expansion [Sha49]).

Theorem 1 (Expansion Theorem [Boo54])**.** *Let $f(x_1, \ldots, x_n)$ be a Boolean function, then*

$$f(x_1, \ldots, x_n) = (\overline{x_i} \cdot f_{\overline{x_i}}) + (x_i \cdot f_{x_i}).$$

The theorem allows to partition a function f into its sub-functions by cofactoring the function. For a non-terminal vertex v, with $\mathrm{var}(v) = x_i$, it follows from the theorem that the subtree rooted in $low(v)$ represents the function $f_{\overline{x_i}}$ and the subtree rooted in $high(v)$ represents f_{x_i}. The tree rooted in v, therefore, represents f. This is written as a triple $f = (\mathrm{var}(v), high(v), low(v)) = (x_i, f_{x_i}, f_{\overline{x_i}})$. The triple is read as "if x_i then f_{x_i} else $f_{\overline{x_i}}$", or $ite(x_i, f_{x_i}, f_{\overline{x_i}}) = \overline{x_i} f_{\overline{x_i}} + x_i f_{x_i}$. Every such if-then-else triple (or vertex of the tree) can be converted easily into a logically equivalent 2-to-1 multiplexer as is depicted in Figure 2.2.

The variable x_i is the decision variable, hence the name of the representation. The tree is constructed by recursive application of Theorem 1, until there are no more variables to cofactor the function with. This procedure inherently leads to $2^{|\mathcal{V}|}$ paths starting in the root node and ending in the terminal vertices. The value of a terminal vertex is determined by cofactoring f with the cube of the decisions made along the corresponding path.



(a) Node deletion. **(b)** Node merging.

Figure 2.3: The two BDD reduction rules.

2.2.2.2 Reduced Ordered Binary Decision Diagrams

The compactness of BDDs comes from two reduction rules on ordered decision trees. They allow for an efficient representation of Boolean functions and make it possible to cope with the inherent exponential size. The tree becomes a directed acyclic graph due to these rules:

1. Node deletion: Nodes which don't influence the outcome of the function are deleted. These are nodes for which both outgoing edges point to the same subgraph. An application of the rule can be seen in Figure 2.3a.

2. Node merging: Isomorphic subgraphs only need to appear once in the data structure. The edges are "rewired" and may point to the same subgraph. The dangling node causing the isomorphism finally gets removed. An application of the rule is depicted in Figure 2.3b.

BDDs are the result of maximally (i.e. until rule application is no longer possible) reducing an ordered binary decision tree. An ordered decision tree and the corresponding BDD, after maximal rule application, can be seen in Figure 2.4. BDDs are canonical due to the two reduction rules. This means that for a fixed variable order, two BDDs representing the same Boolean function are isomorphic. In an implementation this means that every function needs to be in memory only once and logical equivalence checks are reduced to checking the equivalence of two pointers.

An important addition to BDDs are complement edges. The representation of a BDD f and its complement $\neg f$ are very similar. Therefore if f has been computed, but $\neg f$ is needed, the edge pointing to f gets the complement property. The benefits are less memory consumption, constant time complementation (and check for complementation) and uncomplicated application of De Morgan's laws. These benefits outweigh the drawbacks of more complicated case analyses when operating on BDDs, appearing due to the complement property. For canonicity, complemented edges only occur on *low* edges.

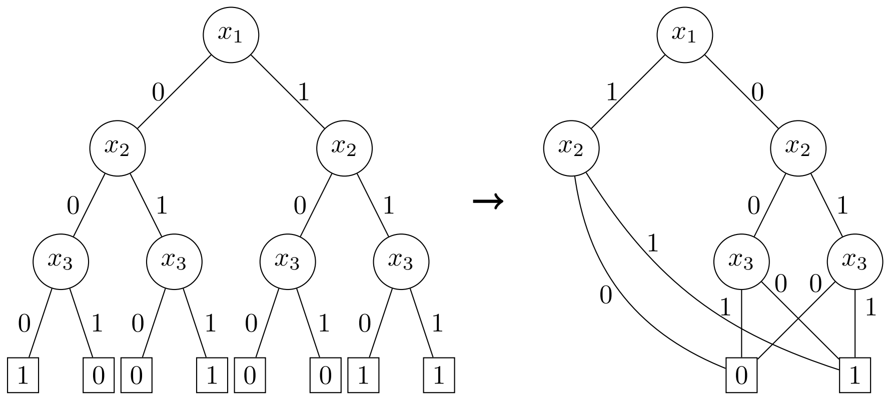


Figure 2.4: Ordered binary decision tree and BDD for the function $f \equiv \overline{x_1}\ \overline{x_2}\ \overline{x_3} + x_1x_2 + x_2x_3$, with variable order $x_1 < x_2 < x_3$.

It should be noted, however, that in practice BDDs are not generated by reducing ordered decision trees. They are rather built by combining smaller BDDs, starting from the basic BDDs $f_i = x_i$ for all variables $x_i \in \mathcal{V}$. The combination of two BDDs, let us say f and g, can be through any of the binary Boolean operations. Therefore, an algorithm able to compute $f\langle op\rangle g$ for any $\langle op\rangle$ is sought. Such an algorithm is APPLY [Bry86] which is described in the next section.

2.2.2.3 Construction of Binary Decision Diagrams

This section adheres to the descriptions in [Som99]. As stated, BDDs are constructed via combination of smaller BDDs through some Boolean operation $\langle op\rangle$. The APPLY algorithm, as depicted in Algorithm 2.5, can be used to compute this combination for every Boolean operation. It recursively forms the combination of two BDDs with the same variable order. This construction follows directly from Theorem 1:

$$f\langle op\rangle g = (x \cdot (f_x\langle op\rangle g_x)) + (\overline{x} \cdot (f_{\overline{x}}\langle op\rangle g_{\overline{x}})). \tag{2.1}$$

Both f and g must adhere to the same variable ordering, with x being the top variable. The functions f and g are cofactored with respect to x and the two simpler problems are then solved recursively. In each recursion step, a vertex v is created with $\text{var}(v) = x$. The children of v are $high(v) = f_x\langle op\rangle g_x$ and $low(v) = f_{\overline{x}}\langle op\rangle g_{\overline{x}}$.

The cofactor of a BDD with respect to the top variable x is the high child when computing the positive cofactor and the low child when computing the negative cofactor.

APPLY is a prime example of *dynamic programming*. In order to achieve efficient computation, it uses two data structures:

1. **Unique table**: This data structure is a dictionary of all BDD nodes of the program.

Operation	*ite* form
0	0
$f \cdot g$	$ite(f, g, 0)$
$f \cdot \overline{g}$	$ite(f, \overline{g}, 0)$
f	f
$\overline{f} \cdot g$	$ite(f, 0, g)$
g	g
$\overline{f \leftrightarrow g}$	$ite(f, \overline{g}, g)$
$f + g$	$ite(f, 1, g)$
$\overline{f} \cdot \overline{g}$	$ite(f, 0, \overline{g})$
$f \leftrightarrow g$	$ite(f, g, \overline{g})$
$\overline{g}$	$ite(g, 0, 1)$
$g \rightarrow f$	$ite(f, 1, \overline{g})$
$\overline{f}$	$ite(f, 0, 1)$
$f \rightarrow g$	$ite(f, g, 1)$
$\overline{f} + \overline{g}$	$ite(f, \overline{g}, 1)$
1	1

Table 2.4: The *ite* operator.

Two equivalent functions are represented by the same BDD node. Therefore, using the unique table, equivalence checks are constant time operations. The table helps to establish the canonicity of BDDs. It prevents nodes which would be deleted by the merging rule from being created.

2. **Computed table**: The computed table is used to make the computation of Apply more efficient. It is used as a cache of already computed functions and employed to prevent repeated computations of the same function. Before each complex computation, the table is queried to check whether the needed result has already been stored.

The lattice of all Boolean two-argument operations expressed in their respective *ite* form is depicted in Table 2.4. In the following, a recursion step of Equation 2.1, using the *ite* operator, is illustrated. Again, x is the top-most variable.

$$\begin{aligned} ite(f, g, h) &= f \cdot g + \overline{f} \cdot h \\ &= x \cdot (f \cdot g + \overline{f} \cdot h)_x + \overline{x} \cdot (f \cdot g + \overline{f} \cdot h)_{\overline{x}} \\ &= x \cdot (f_x \cdot g_x + \overline{f}_x \cdot h_x) + \overline{x} \cdot (f_{\overline{x}} \cdot g_{\overline{x}} + \overline{f}_{\overline{x}} \cdot h_{\overline{x}}) \\ &= (x, ite(f_x, g_x, h_x), ite(f_{\overline{x}}, g_{\overline{x}}, h_{\overline{x}})) \end{aligned}$$

The recursion terminates in the cases $ite(1, f, g) = ite(0, g, f) = f$ and $ite(f, g, g) = g$. Algorithm 2.5 provides pseudo-code for Apply, without elaborating on FindOrAddUniqueTable and InsertComputedTable.

```
proc APPLY (f, g, h)
  if terminal case
    return result
  elif computed table has entry (f, g, h)
    return result
  else
    x ← top_var(f, g, h)
    f' ← APPLY(f_x, g_x, h_x)
    g' ← APPLY(f_x̄, g_x̄, h_x̄)
    if f' = g'
      return g'
    R ← FINDORADDUNIQUETABLE(x, f', g')
    INSERTCOMPUTEDTABLE((f, g, h), R)
    return R
```

Figure 2.5: Apply implementing the construction of a BDD from two BDDs for any two-argument Boolean operator.

2.2.2.4 Operations on Binary Decision Diagrams

In the previous subsection we showed how to combine two BDDs via the Apply algorithm for any Boolean operator. In order to describe the algorithm, we showed how to compute the cofactor with respect to the top variable. If a BDD is cofactored with multiple variables (a cube), the procedure is to compute the cofactor recursively starting with the root node. Then a case distinction is made and if the BDD is to be cofactored with the current node's variable the incoming edges are reconnected to the children of that node. In case there is no need to cofactor the current node, the recursion proceeds along towards the leaves without changing the node.

Apply and cofactoring can directly be used to compute the existential and universal quantifications of a BDD with respect to a single variable, by formula expansion (cf. Section 2.2.1.3):

$$\forall y.\ f \equiv f_{\overline{y}} \cdot f_y,$$
$$\exists y.\ f \equiv f_{\overline{y}} + f_y.$$

Another important operation is functional composition. The goal is to compute

$$f|_{x_i=g} = f(x_1, \ldots, x_{i-1}, g, x_{i+1}, \ldots, x_n),$$

with g being a function. This can be done by applying Theorem 1 and subsequent substitution of x_i by g, resulting eventually in the computation of $ite(g, f_{x_i}, f_{\overline{x_i}})$. The literature describes an optimized algorithm for this computation of the functional composition of f and g named COMPOSE [Bry86, Som99]. A single satisfying assignment for the BDD can be found with the GETSATASSIGNMENT algorithm. Finding a satisfying assignment is equivalent to finding a path from root to the 1-sink with an even number of complemented edges.

2.2.2.5 Variable Ordering

The variable order has a major influence on the size (the number of vertices) of a BDD. The problem of finding an ordering such that the number of BDD vertices is bounded, was proven to be NP-hard [BW96]. In practice, the problem is tackled by applying heuristics such as presented in [Rud93, FMK91, ISY91, PS95, PSP96].

BDD reordering can either be applied at fixed positions in the program or *dynamically*. In dynamic reordering, a reordering algorithm is applied as soon as the size of a BDD exceeds a certain threshold.

Even though there are many ways to decrease the memory consumption of BDDs, excessive memory consumption is the primary problem when dealing with BDDs. Reordering algorithms may have trouble dealing with large instances. The authors of [HB11], for example, describe how finding a good variable order takes up the major amount of work in their computations.

2.2.3 Conjunctive Normal Form

Conjunctive normal form, or CNF, is a syntactic restriction of propositional logic. It has the useful property that the resolution calculus can be applied to it. CNF, therefore, is the representation used by SAT and QBF solvers.

The syntax of conjunctive normal form is a restriction of propositional logic to a conjunction of disjunctions ("and of ors") of literals. The following BNF defines it.

$$\begin{aligned}
\langle\text{cnf}\rangle &::= (\langle\text{clause}\rangle) \cdot \langle\text{cnf}\rangle \mid (\langle\text{clause}\rangle) \\
\langle\text{clause}\rangle &::= \langle\text{literal}\rangle + \langle\text{clause}\rangle \mid \langle\text{literal}\rangle \\
\langle\text{literal}\rangle &::= \overline{\langle\text{propositional symbol}\rangle} \mid \langle\text{propositional symbol}\rangle \\
\langle\text{propositional symbol}\rangle &::= x_1 \mid \dots \mid x_n \mid \dots
\end{aligned}$$

The disjunctions are referred to as **clauses**. Clauses might also be referred to as sets of literals. If it is clear that literals belong to a particular clause, the + connective is dropped. We say that a clause C_1 *subsumes* a clause C_2, if $C_1 \subseteq C_2$.

2.2.3.1 Tseitin's Transformation

Every arbitrary propositional formula can be transformed into an equivalent CNF formula, purely by application of syntactical rewrite rules. This might however lead to an exponential blowup of the size of the formula. When applying a SAT or QBF solver it is sufficient to have an equi-satisfiable CNF formula, however. Such a formula may contain additional fresh variables, which do not affect the satisfiability of the original formula. An equi-satisfiable formula in CNF can be obtained from an arbitrary propositional formula by application of Tseitin's transformation [Tse68]. The advantage of this method is that the formula grows only polynomially. The transformation of an arbitrary propositional formula F proceeds in two steps:

1. Every sub-formula $F_1 \diamond F_2$, with $\diamond \in \{\cdot, +, \rightarrow, \leftrightarrow\}$, of F (sub-formula $\overline{F_1}$ in the unary case) is recursively replaced by a fresh variable x. For every such replacement, a conjunct $(x \leftrightarrow F_1 \diamond F_2)$ ($(x \leftrightarrow \overline{F_1})$ in the unary case) is added to the new formula F'.

2. Every conjunct of F' can be rewritten into CNF using a set of rules. These rules are provided in Table 2.5 (Page 30). The final formula is a conjunction of CNF-sub-formulas and therefore also in CNF.

2.2.4 Disjunctive Normal Form

Disjunctive normal form (DNF) is similar to CNF. It is the "or of ands" of literals. Its BNF is the same as the one for CNF, but with all appearances of $\cdot$ and $+$ swapped. Cubes therefore take the place of clauses. A formula in DNF can be considered a set of cubes.

One reason for the usefulness of DNF is that it provides a straight-forward way to represent the cover of a Boolean function. Covering a function is the problem of finding cubes, such that the on-set minterms of a Boolean function are covered by the cubes. Solving this problem is an essential task in logic minimization and has been extensively studied throughout the years. An algorithm for finding a cover from a DNF will be presented in the related work (Chapter 3).

2.3 Satisfiability Solving and Interpolation

Boolean satisfiability, or short SAT, is the problem of determining if there is a satisfying assignment to the variables in a propositional logic formula F, which make it true.

The SAT problem for general propositional logic is usually reduced to the problem of determining whether a CNF formula is satisfiable or not. An equi-satisfiable CNF formula is the result of applying Tseitin's transformation (cf. Section 2.2.3.1). The reduction to CNF allows for the application of the resolution calculus.

2.3.1 Resolution Calculus

The resolution rule is an inference rule deriving a new clause from two clauses containing a complementary literal. The clauses $C + x$ and $D + \overline{x}$ are the **antecedents**, x is the **pivot**, and $C + D$ is the **resolvent**. $\textsc{Res}(C, D, x)$ denotes the resolvent of C and D with the pivot x. The pivot variable must be the only variable appearing in opposed phases between the two antecedents.

The resolution rule $\textsc{Res}(C + x, D + \overline{x}, x)$ is written as

$$\frac{C + x \qquad D + \overline{x}}{C + D} \qquad \text{or} \qquad \begin{tikzcd}[column sep=small] C + x \arrow[dr] & & D + \overline{x} \arrow[dl] \\ & C + D & \end{tikzcd}$$

Resolution can be regarded as existential quantification of the pivot variable in the conjunction of the antecedents.

$$\begin{aligned} \exists x.\ ((C + x) \cdot (D + \overline{x})) &\equiv ((C + x) \cdot (D + \overline{x}))_x + ((C + x) \cdot (D + \overline{x}))_{\overline{x}} \\ &\equiv (1 \cdot D) + (C \cdot 1) \\ &\equiv C + D \end{aligned}$$

For a formula F in CNF, repeated application of the resolution rule, starting with the clauses in F, yields a resolution proof for F.

Definition 3 (Resolution proof)**.** *A resolution proof R is a DAG $(V_R, E_R, cla_R, piv_R, s_R)$. V_R is the set of proof vertices. $E_R \subseteq V_R \times V_R$ is the set of edges. The proof consists of initial vertices, which have in-degree 0 and internal vertices, which have in-degree 2. The sink vertex $s_R \in V_R$ is the only vertex of the proof with out-degree 0. Let $v, v_1, v_2 \in V_R$, then the edges $(v_1, v) \in E_R$ and $(v_2, v) \in E_R$ represent the resolution*

$$cla_R(v) = \textsc{Res}(cla_R(v_1), cla_R(v_2), piv_R(v)).$$

For all initial vertices $v \in V_R$, $cla_R(v)$ is a clause from the CNF formula.

The subscripts are dropped if clear from the context. For a vertex $v_1 \in V_R$ with an edge (v_1, v), we write v^+ if v_1 contains the pivot in positive phase and v^- if it contains the pivot in negative phase. We say that $v_i \in V_R$ is a **parent** of $v_j \in V_R$, if $(v_i, v_j) \in E_R$.

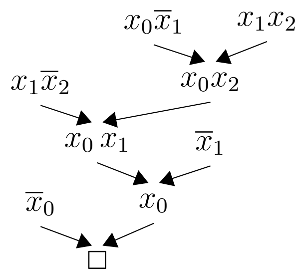


Figure 2.6: Resolution refutation of Equation 2.2.

Conversely, we call v_j a **child** of v_i. We say that v_i is an **ancestor** of v_j, if there is a path from v_i to v_j. We denote the set of all paths from v_i to v_j by $\text{Paths}(v_i, v_j)$ and a path as a set of vertices. We say that v_i dominates v_j, if all paths from v_j to the sink go through v_i.

A **refutation** is a resolution proof with $cla(s) \equiv 0$. This is usually expressed by the $\square$ symbol representing the empty clause. If every initial vertex of a proof is labelled with a clause of F, and it is a refutation, then the proof is said to be a refutation of F. Resolution is refutation-complete which means that the empty clause can always be derived, if the formula is unsatisfiable. Example 1 shows a resolution refutation. Note, that the + connectives are dropped in the figure.

Example 1. *Figure 2.6 shows an example of the refutation of*

$$F \equiv (\overline{x}_0)\,(x_1\overline{x}_2)\,(x_0\overline{x}_1)\,(x_1x_2)\,(\overline{x}_1) \tag{2.2}$$

2.3.2 Satisfiability Solving

SAT solvers use a complete search algorithm to establish the satisfiability of a formula F. The search space is a decision tree spanned by the variables in F. By assigning truth values to the variables, the tree is explored. If for no leaf of the tree, representing full assignments, it is possible to make the formula sat, the instance is unsatisfiable (unsat). If for at least a single one there is a valid assignment, it is said to be satisfiable (sat).

Since SAT is a highly generic problem and therefore appears in many domains, much effort has been put into finding an efficient solving algorithm. A first step was made in 1960 with the Davis-Putnam [DP60] procedure (DP). Subsequently there have been various optimizations of the algorithm. The first improvement was the Davis-Putnam-Logeman-Loveland [DLL62] procedure (DPLL) in 1962. Further significant enhancements came only decades later in the late 1990s, resulting in the *GRASP* [SS96] and *Chaff* [MMZ+01] SAT solvers, which improved the size (usually measured by the number of variables) of the solvable instances by orders of magnitude.

Due to the recent improvements to SAT solvers[1], many new applications were made possible. Examples for the application of SAT solvers range from model-checking, over software package management, to cryptanalysis. For a more comprehensive overview, we refer the reader to [MS08].

2.3.3 Proof Extraction

Modern SAT solvers are not just decision procedures for propositional logic formulas. They are also able to produce resolution proofs (Definition 3). Initially resolution proofs were needed for checking the correctness of a SAT solver's implementation using an independent tool. The proofs act as a certificate when the result is unsat [ZM03] (it is trivial to check the sat case, by assigning the variables with the values from the model). Resolution proofs use excessive amounts of memory, however, and there has been a push away from using resolution proofs to using clausal proofs as certificates [GN03, WHH14].

The technique in [ZM03] stores a trace of clauses during the solver run. Construction of the resolution proof from this trace is straight-forward. The approach in [GN03] applies so-called reverse unit propagation, to limit the amount of clauses, which need to be computed during checking. While reverse unit propagation takes more time, it improves memory usage. Recent improvements [WHH14] lead to more efficient checking. A (trimmed) resolution proof can be emitted during checking of clausal proofs. A modern tool-chain for this task would be Glucose [AS09] followed by DRAT-trim [WHH14].

We use resolution proofs for computation of Craig Interpolants [Cra57], which we will describe in the next section.

2.3.4 Craig Interpolation

A Craig Interpolant is defined by Craig's interpolation theorem.

Theorem 2 (Craig Interpolant [Cra57])**.** *Given two Boolean formulas A and B, with $A \wedge B$ unsatisfiable, there exists a Boolean formula I referring only to the common variables of A and B such that $A \rightarrow I$, and $I \wedge B$ is unsatisfiable.*

Given a conjunction of A and B which is unsatisfiable, the steps involved in the computation of the interpolant are as follows.

1. The SAT instance $A \wedge B$ is solved. If necessary, it is transformed into CNF first. Since $A \wedge B$ is unsat, a resolution refutation is the result. The initial vertices of the resolution refutation are labelled with clauses from A and B.

[1] For example [ES03, EB05], but we refer the reader to `http://www.satcompetition.org/` for state-of-the-art solvers and their improvements.

2. An interpolation system is employed, annotating each vertex of the proof with a propositional formula, called a **partial interpolant**.

3. The annotation of the sink node is called the **final interpolant** I, which we are interested in.

There exist three different systems for computing the interpolant from a resolution refutation:

1. The symmetric system [Hua95, Kra97, Pud97],

2. McMillan's system (regular and inverse) [McM03], and

3. the labelled interpolation system [DKPW10].

The latter system is a generalization of the first two, and therefore we will only describe and work with the labelled system.

2.3.4.1 Labelled Interpolation System

This description of the labelled interpolation system follows the one in [DKPW10] closely. We first define a labelling function, mapping the literals of the resolution proof, denoted by `Lit`, to labels.

Definition 4 (Labelling function)**.** *Let $\mathcal{S} = \{\mathsf{a}, \mathsf{b}, \mathsf{ab}, \bot\}$ be a set of labels, partially ordered as defined by the Hasse diagram $(\mathcal{S}, \sqsubset, \sqcup)$ depicted below. A labelling function $L : V \times$* `Lit` *$\mapsto \mathcal{S}$ maps all literals of a resolution proof R to a label from $\mathcal{S}$. For a literal $l \in$* `Lit` *and a vertex $v \in V$, L must satisfy*

$$L(v, l) = \begin{cases} \bot, \textit{ if } l \notin cla(v). \\ L(v^+, l) \sqcup L(v^-, l), \textit{ if } v \textit{ is internal and } l \in cla(v). \end{cases}$$

$$\begin{tikzcd}[column sep=small, row sep=small] & \mathsf{ab} \arrow[dl, dash] \arrow[dr, dash] & \\ \mathsf{a} \arrow[dr, dash] & & \mathsf{b} \arrow[dl, dash] \\ & \bot & \end{tikzcd}$$

A variable $\mathrm{var}(l)$ is called A-local if it appears only in $\mathrm{Var}(A) \setminus \mathrm{Var}(B)$, B-local if it appears only in $\mathrm{Var}(B) \setminus \mathrm{Var}(A)$ and shared otherwise. A labelling function is supposed to preserve locality, meaning that l has to be labelled a if $\mathrm{var}(l)$ is A-local and l must be labelled b if $\mathrm{var}(l)$ is B-local. Shared variables, occurring both in A and B, might be labelled a, b or ab.

Given a resolution proof and a labelling function, the labelled interpolation system is defined inductively. The following inference rules define the labelled interpolation system and show how resolution proofs can be used to compute partial interpolants.

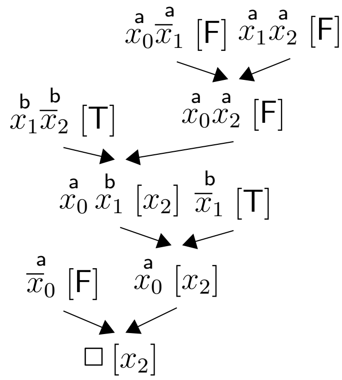


Figure 2.7: Labelled resolution proof annotated with partial interpolants.

Definition 5 (Labelled interpolation system)**.** *Let R be a resolution refutation and L be a locality-preserving labelling function L. The labelled interpolation system $\mathsf{Itp}(L, P)$ maps vertices to partial interpolants (in brackets) as defined below.*

Case 1. *Initial vertex v with $cla(v) = C$:*

$$\frac{}{C \quad [\{l \in C \mid L(v,l) = \mathsf{b}\}]} \; \textit{if } C \in A, \qquad \frac{}{C \quad [\neg\{l \in C \mid L(v,l) = \mathsf{a}\}]} \; \textit{if } C \in B$$

Case 2. *Internal vertex v with $cla(v) = C_1 + C_2$, $cla(v^+) = C_1 + x$ and $cla(v^-) = C_2 + \overline{x}$:*

$$\frac{C_1 + x \quad [I_1] \qquad C_2 + \overline{x} \quad [I_2]}{C_1 + C_2 \quad [I_3]}$$

$$\begin{aligned}
&\textit{if } L(v^+, x) \sqcup L(v^-, \overline{x}) = \mathsf{a}, && I_3 = I_1 + I_2,\\
&\textit{if } L(v^+, x) \sqcup L(v^-, \overline{x}) = \mathsf{ab}, && I_3 = (x + I_1) \cdot (\overline{x} + I_2),\\
&\textit{if } L(v^+, x) \sqcup L(v^-, \overline{x}) = \mathsf{b}, && I_3 = I_1 \cdot I_2
\end{aligned}$$

Example 2. *Let us continue Example 1 by applying the labelled interpolation system to the resolution proof. F is split into $A \equiv (\overline{x}_0)(x_0\overline{x}_1)(x_1x_2)$ and $B \equiv (\overline{x}_1)(x_1\overline{x}_2)$. According to this partitioning the literals are labelled by a locality-preserving labelling function. By annotation of the initial vertices with partial interpolants and propagation according to the rules from Definition 5 the result is the resolution proof depicted in Figure 2.7. The labels of the literals are shown as their respective superscripts. The final interpolant I corresponds to $\mathsf{Itp}(L)(\square) = x_2$. It can be seen that it is a valid interpolant by checking that $A \to I$ and $I \wedge B$ is unsatisfiable.*

Labelled interpolation systems support the elimination of *non-essential* (or *peripheral* [SDGC10]) variables from interpolants [D'S10], as stated by the following lemma.

Lemma 1. *Let L and L' be locality preserving labelling functions for an (A, B)-refutation R, where $L(v, t) = \mathsf{a}$ if $cla_R(v) \in A$ and $L(v, t) = \mathsf{b}$ if $cla_R(v) \in B$ for all initial vertices of R. Then* $\mathrm{Var}(\mathsf{Itp}(L)(v)) \subseteq \mathrm{Var}(\mathsf{Itp}(L')(v))$ *for all $v \in V_R$.*

For such labelling functions only the middle case (where the labels are merged to ab) introduces a variable into the interpolant. Therefore eliminating such cases would allow us to reduce the number of variables (our measure of size) of the interpolant. We will describe our approach at this problem in Chapter 5.

2.4 Determinization of Boolean Relations

After introducing the fundamental background, we now turn to introducing existing approaches to relation determinization, which we base our work on.

Determinization of Boolean relations is the problem of finding a functional implementation $\vec{f} = (f_1, \ldots, f_m)$ with $f_i : \mathbb{B}^n \mapsto \mathbb{B}$ of a Boolean relation $R \subseteq \mathbb{B}^n \times \mathbb{B}^m$. Every f_i is an unambiguous mapping from input variables $\vec{x} = (x_1, \ldots, x_n)$ to output variables $\vec{y} = (y_1, \ldots, y_m)$, such that

$$D = \bigwedge_{i=1}^{m} (y_i \equiv f_i(\vec{x}))$$

characterizes a subset of R (if $(x, y) \in D$ then $(x, y) \in R$, but not the other way round). Therefore, D implies R. D is a deterministic relation *compatible* with the non-deterministic relation R. In other words, this means that D is the relation after resolving all the ambiguity (i.e. one-to-many mappings) of R. For each one-to-many mapping one choice is picked—so to say—determinizing the relation.

To compute each f_i the multiple-output case is reduced to the single-output case. We present a scheme accomplishing this in the next subsection.

2.4.1 A Scheme for the Determinization of Multiple-output Relations

There exist multiple different schemes for the determinization of multiple-output relations $R \subseteq \mathbb{B}^n \times \mathbb{B}^m$. We will describe the method from [JLH09, Section 3.2.1]. The procedure is reminiscent of Gaussian elimination and similarly proceeds in two steps: Forward elimination and Back substitution. Let $FI(y, R)$ be a functional implementation of a single-output total relation R with output variable y. We present ways to compute $FI(y, R)$ next (existing work) and in Chapters 4 and 5 (our new approaches).

1. Forward elimination: Let $R^{(i)}$ stand for $\exists y_m \cdots \exists y_i.\ R$, for $2 \leq i \leq m$. The scheme first reduces the number of outputs by iterative existential quantification and saves all the intermediate results:

$$\begin{array}{rcl} R^{(m)} & = & \exists y_m.\ R \\ & \vdots & \\ R^{(i)} & = & \exists y_i.\ R^{(i+1)} \\ & \vdots & \\ R^{(2)} & = & \exists y_2.\ R^{(3)} \end{array}$$

2. Back substitution: Thereafter, for each output y_i, the functional implementation f_i is computed and the result substituted for y_i.

$$\begin{array}{rcl} f_1 & = & FI(y_1, R^{(2)}) \\ & \vdots & \\ f_i & = & FI(y_i, R|^{(i+1)}_{y_1=f_1,\ldots,y_{i-1}=f_{i-1}}) \\ & \vdots & \\ f_m & = & FI(y_m, R|_{y_1=f_1,\ldots,y_{m-1}=f_{m-1}}) \end{array}$$

Compared to the procedure used in [BGJ+07], which needs $\mathcal{O}(m^2)$ quantifications, this procedure gets by with $\mathcal{O}(m)$ quantifications by saving the intermediate results. Single-output relations are considered to be embedded in such a scheme throughout the thesis. The reduction from multiple outputs to a single one makes it easier to analyze the cases when trying to find a functional implementation.

The presented scheme takes care of reducing the number of outputs in order to compute $FI(y, R)$, but another precondition which says that R must be total is not necessarily given. However, a single-output partial relation $R(\vec{x}, y)$ can be totalized—by treating the unmapped inputs as don't cares—as presented in [JLH09, Formula 2]:

$$T(\vec{x}, y) = R(\vec{x}, y) + \forall y.\ \neg R(\vec{x}, y)$$

2.4.2 Extracting Circuits from Relations

We will now present ways for finding a functional implementation for a Boolean single-output total relation. This solves an essential problem within property synthesis (cf. Appendix A), that we are particularly interested in. Previous algorithms used BDDs for representing the relation, but more recently, an interpolation-based procedure has been proposed, taking advantage of the improvements made to SAT solvers.

2.4.2.1 BDD-based: Building Circuits from Relations

Kukula and Shiple [KS00] present a way of coping with the potential non-determinism of a multi-output relation $R(\vec{x}, \vec{y})$ and are able to construct a circuit from such a relation. They do so by adding parametric variables, which have the purpose of breaking up don't care conditions. The final result is a circuit representing $R_C(\vec{x}, \vec{p}, \vec{y}, v)$. They assume that the input relation is represented by a free BDD. A free BDD is a BDD which allows different variable orderings on different paths, thereby being more general than the definition of BDDs presented in Section 2.2.2. Let us from now on just refer to BDDs.

On a high level, their approach constructs a circuit which adheres to the structure of the BDD. For each BDD node representing an input variable x_i, an input module is built and for each node representing an output variable y_i, an output module is built, respectively. There is a 1-to-1 correspondence between BDD nodes and circuit modules. Every edge of the BDD corresponds to two wires in the circuit: one incoming and one outgoing signal connecting the modules corresponding to the nodes connected by the edge. On top of that, additional circuitry is added, but we will describe their solution without going into those details. The approach consists of three phases:

1. In the first phase, they gather information about whether there is a path to the 1 leaf for the current assignment to the input variables. They do so by propagating signals from the 1 sink towards the root of the DAG and added to their circuit as an auxiliary output.

2. In the second phase, they propagate the signals the other way (from the root towards the leafs) and activate a single path toward the 1 leaf, if possible. At an input module, the corresponding input variable is responsible for steering the path. At an output module, a parametric input p_i is responsible. At each module, they use the information from Phase 1 to make a valid decision. If an outgoing signal becomes active, the module connected to that signal becomes active as well.

3. In the third phase, they collect information along output modules corresponding to the output variable y_i. If any module chooses 1 for y_i, the final output should be 1 and 0 otherwise. If none of the modules representing y_i is active, they determine the value by the parametric input p_i. For this choice a 2-to-1 multiplexer (one per output variable) is used, with the activation signals from Phase 2 acting as selectors.

```
proc ExtractFunctionFromBDD (R, x⃗, y)
  R_1 ← R(x⃗, 1)
  R_0 ← R(x⃗, 0)
  R'_1 ← R_1 · ¬R_0
  R'_0 ← R_0 · ¬R_1
  foreach x ∈ x⃗ do
    R''_1 ← ∃x. R'_1
    R''_0 ← ∃x. R'_0
    if R''_1 · R''_0 = 0 then
      R'_1 ← R''_1
      R'_0 ← R''_0
  f ← R'_1
  return f
```

Figure 2.8: Extraction of a function from a relation.

The authors prove the correctness of their construction [KS00, Theorem 1], showing that

$$R(\vec{x}, \vec{y}) \leftrightarrow R_C(\vec{x}, \vec{p}, \vec{y}, 1).$$

This however is more general than is necessary for many applications, such as synthesis. As was described in Section 2.4, for a functional implementation it would be sufficient, if $R_C(\vec{x}, \vec{p}, \vec{y}, 1)$ would imply $R(\vec{x}, \vec{y})$.

2.4.2.2 BDD-based: Specify, Compile, Run: Hardware from PSL

The next approach is a simplified version from [BGJ+07, Figures 2 and 3]. The version we present assumes a single-output relation. Again, the relation is assumed to be in BDD form. The algorithm was proposed to find a circuit implementation of a strategy for a GR(1) game (cf. Appendix A) and is less general than the approach by Kukula and Shiple, presented in the previous subsection. The algorithm takes a relation $R \subseteq \mathbb{B}^n \times \mathbb{B}$, the set of input variables $\vec{x} = \{x_1, \ldots, x_n\}$ and the output variable y as arguments. ExtractFunctionFromBDD as presented in Figure 2.8 first computes both the positive and the negative cofactors of R with respect to y. It then computes the strict cofactors R'_1 and R'_0. If the relation is total in the input space, these expressions could be simplified to $R'_1 \leftarrow \overline{R_0}$ and $R'_0 \leftarrow \overline{R_1}$, respectively.

The `foreach` loop is an optional extension. This extension aims at simplifying the relation, by eliminating input variables. The inputs, which y does not depend on, do not influence the output and should therefore not appear in a functional implementation of R. To find these inputs, the algorithm iterates over the set of input variables and

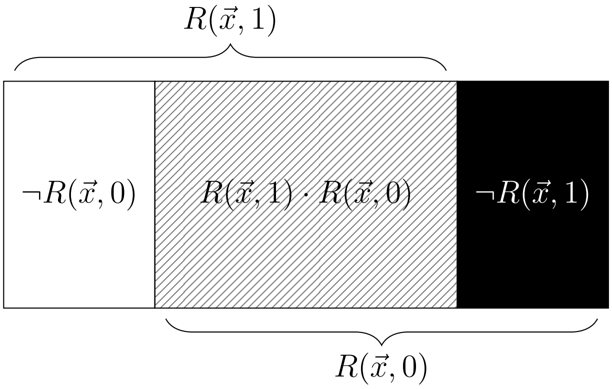


Figure 2.9: The set representation of a single-output total relation $R(\vec{x}, y)$ split into its cofactors.

existentially quantifies the input x of the current iteration in the strict cofactors R'_1 and R'_0. The resulting BDDs represent the sets where x is fully expanded. It then checks, if these sets intersect by computing the conjunction of the BDDs. If they do, the input x has influence on y and cannot be eliminated. Otherwise, the algorithm updates R'_1 and R'_0 and effectively eliminates x from the relation.

The functional implementation $FI(y, R)$ (see Section 2.4) is finally the strict positive cofactor of R with respect to y.

2.4.2.3 Interpolation-based: Interpolating functions from large Boolean relations

Jiang, Lin and Hung present a different approach [JLH09] to the determinization of Boolean relations, namely using interpolation (cf. Section 2.3.4). They also split the relation $R(\vec{x}, y)$ into parts ($\neg R(\vec{x}, 0)$ and $\neg R(\vec{x}, 1)$), in a similar way as ExtractFunctionFromBDD. They show that the conjunction of these parts is unsatisfiable. Therefore, it is possible to obtain a resolution refutation from a SAT solver and use it to compute a Craig interpolant. This interpolant turns out to be a functional implementation of the relation.

Figure 2.9 illustrates a single-output total relation as it appears throughout this section. When cofactoring R with respect to y three disjoint sets are distinguished:

1. S_A is the set characterized by $\neg R(\vec{x}, 0)$
2. S_B is the set characterized by $\neg R(\vec{x}, 1)$
3. The don't care set is the conjunction of $R(\vec{x}, 0)$ and $R(\vec{x}, 1)$.

The authors of [JLH09] make use of the following proposition.

Proposition 1. *A relation $R(\vec{x}, y)$ is total if and only if the conjunction of $\neg R(\vec{x}, 0)$ and $\neg R(\vec{x}, 1)$ is unsatisfiable.*

The main result of their work is the following theorem and proof thereof. The proof of the theorem immediately shows, how the interpolant maps the members of the three involved sets to either 0 or 1 and thereby determinizes the relation.

Theorem 3 ([JLH09, Theorem 2])**.** *Given a single-output total relation $R(\vec{x}, y)$, the interpolant I of the refutation of*

$$\neg R(\vec{x}, 0) \cdot \neg R(\vec{x}, 1) \tag{2.3}$$

with $A = \neg R(\vec{x}, 0)$ and $B = \neg R(\vec{x}, 1)$, corresponds to a functional implementation of R.

The interpolant maps every element of S_A to 1, every element of S_B to 0, and every other element to either 0 or 1. Furthermore, let f be $(y \equiv I)$. Then $f \rightarrow R$ and is a functional implementation $(FI(y, R))$.

The interpolant, and therefore the mapping of the elements not in $S_A \cup S_B$, depends on the the resolution proof found by the SAT solver on the one hand and on the interpolation system on the other hand. There are two trivial interpolants satisfying Theorem 3 which can be obtained without interpolation, however. These are $R(\vec{x}, 1)$ and $\neg R(\vec{x}, 0)$ (used by the algorithm presented in the previous section). The former is the weakest interpolant and characterizes the largest set. The latter is the strongest interpolant and characterizes the smallest set, as is depicted in Figure 2.9. The authors of [JLH09] claim that the trivial interpolants often lead to a more complex circuit than the functional implementations computed from a resolution refutation.

2.5 Resolution Proof Reduction

An optimization, similar as in Section 2.4.2.2 is made implicitly by Craig interpolation. The interpolant I only depends on variables in the shared alphabet of A and B.

Minimizing the number of variables in the interpolant further, requires modification of the resolution refutation. We present the following efficient proof post-processing techniques that we improve and generalize in Chapter 5:

1. RECYCLEUNITS: For every unit clause $cla(u)$ (that is a clause consisting of a single literal) the algorithm checks if a lause $cla(v)$ in the proof has the unit clause as a pivot. If it does, v gets replaced by u. Therefore, the resolutions of the literals in $cla(v)$ can potentially (barring merge literals) be spared.

2. RMPIVOTS: This algorithm is based on the observation [Tse68] that a minimal resolution proof has at most one resolution on the same pivot on every path from root to sink. Therefore RMPIVOTS analyzes the paths from sink to root and keeps

track of the encountered pivots in a set σ. As soon as a second resolution on a pivot p already in σ is found, the resolution step can be eliminated. The decision, which of the two clauses is removed depends on the path chosen from the first insertion of p. If the path containing the negative literal was chosen, the clause containing the positive literal is discarded, and vice-versa.

The simplest version of this algorithm assumes that the proof is a tree. In general however, a resolution proof can be a directed acyclic graph. The authors of [BIFH+09] propose two solutions: Firstly, stopping the algorithm as soon as a vertex with out-degree greater 1 is encountered. This is the approach the authors chose. Secondly, they propose a more complicated approach involving dominator analysis.

Fontaine et al. [FMP11] and Gupta [Gup12] independently published improvements to this version of the algorithm. Most importantly they generalize it to directed acyclic graphs, by computing the meet-over-all-paths for σ at vertices with out-degree greater 1. We will give an in-depth description of these techniques and present an optimization in Chapter 5.

Both RecycleUnits and RmPivots are part of a two-step process. First, the proof is reduced by applying the described techniques. This might leave the proof in an invalid state, containing incorrect resolutions. The proof has to be corrected, such that it is a valid resolution proof again. The algorithm performing these corrections is called ReconstructProof [BIFH+09].

All these modifications of resolution proofs do not consider interpolation systems at all. A smaller proof, that is one with less resolutions, is intuitively preferable for receiving a smaller interpolant and in turn a simpler functional implementation via [JLH09]. Our experiments (Section 5.3) show that less resolutions, lead to a smaller interpolant. We will show in Chapter 5 how to also keep track of labelling information in σ, in order to prevent certain unfavorable proof reductions.

Negation:

$$\begin{aligned} x \leftrightarrow \overline{y} &\equiv (x \rightarrow \overline{y}) \cdot (\overline{y} \rightarrow x) \\ &\equiv (\overline{x} + \overline{y}) \cdot (y + x) \end{aligned}$$

Disjunction:

$$\begin{aligned} x \leftrightarrow (y + z) &\equiv (y \rightarrow x) \cdot (z \rightarrow x) \cdot (x \rightarrow (y + z)) \\ &\equiv (\overline{y} + x) \cdot (\overline{z} + x) \cdot (\overline{x} + y + z) \end{aligned}$$

Conjunction:

$$\begin{aligned} x \leftrightarrow (y \cdot z) &\equiv (x \rightarrow y) \cdot (x \rightarrow z) \cdot ((y \cdot z) \rightarrow x) \\ &\equiv (\overline{x} + y) \cdot (\overline{x} + z) \cdot (\overline{(y \cdot z)} + x) \\ &\equiv (\overline{x} + y) \cdot (\overline{x} + z) \cdot (\overline{y} + \overline{z} + x) \end{aligned}$$

Implication:

$$\begin{aligned} x \leftrightarrow (y \rightarrow z) &\equiv (x \rightarrow (y \rightarrow z)) \cdot ((y \rightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + (y \rightarrow z) \cdot (\overline{(y \rightarrow z)} + x) \\ &\equiv (x + \overline{y} + z) \cdot (\overline{(\overline{y} + z)} + x) \\ &\equiv (x + \overline{y} + z) \cdot ((y \cdot \overline{z}) + x) \\ &\equiv (x + \overline{y} + z) \cdot (x + y) \cdot (x + \overline{z}) \end{aligned}$$

Bi-implication:

$$\begin{aligned} x \leftrightarrow (y \leftrightarrow z) &\equiv (x \rightarrow (y \leftrightarrow z)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (x \rightarrow ((y \rightarrow z) \cdot (z \rightarrow y)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (x \rightarrow (y \rightarrow z)) \cdot (x \rightarrow (z \rightarrow y)) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot ((y \leftrightarrow z) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot (((y \cdot z) + (\overline{y} \cdot \overline{z})) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot ((y \cdot z) \rightarrow x) \cdot ((\overline{y} \cdot \overline{z}) \rightarrow x) \\ &\equiv (\overline{x} + \overline{y} + z) \cdot (\overline{x} + \overline{z} + y) \cdot (\overline{y} + \overline{z} + x) \cdot (y + z + x) \end{aligned}$$

Table 2.5: Tseitin's transformation [Tse68] for each logic connective (table taken from [WM11])

Chapter 3

Related Work

This chapter aims at introducing existing techniques related to our work. We begin with the well-studied subject of combinational logic minimization in Section 3.1. We present classic minimization algorithms which tried to find an exact minimum implementation for an incomplete Boolean function. We also give a brief overview of heuristic minimization algorithms.

3.1 Combinational Logic Minimization

We briefly describe classic approaches to combinational logic minimization, following in part the presentation of [DM94, Chapter 7]. Without loss of generality it is assumed that the circuit is presented in disjunctive normal form. Due to the structure of DNF, logic minimization is commonly referred to as two-level logic minimization. An extension which generalizes the algorithms to more levels exists [Law64].

The goals of two-level logic minimization are to minimize the literals and conjunctions (the focus might be on one or the other) of the circuit and in turn to minimize circuit area. The first solutions [VOQ52, Mcc56] to the problem provided exact minimizations. These approaches have some success in practical scenarios. In general, though, finding an exact solution is computationally infeasible. Therefore the focus of later work changed to finding heuristic solutions which yield an approximate minimization. One standard tool implementing these minimization algorithms is the Espresso logic minimizer [BSVMH84].

The solutions initially only applied to incompletely specified Boolean functions. Interesting in the scope of this thesis is that very similar techniques can be applied to the more general case of Boolean relations as well: [BS89] shows how to perform exact minimization and [WB91] shows how to perform heuristic minimization. The following description targets minimization of incompletely specified Boolean single-output functions

$(f : \mathbb{B}^n \mapsto \mathbb{B}_+)$.

3.1.1 Definitions

Logic minimization revolves around covering the minterms of a Boolean function by implicants. Some definitions are in order:

Definition 6 (Implicant). *An implicant of f is a cube c contained in f.*

Definition 7 (Cover). *A cover of f is a set of cubes that represents f.*

Definition 8 (Minimum Cover). *A minimum cover is a cover with minimum cardinality.*

Definition 9 (Prime Implicant). *A prime implicant is an implicant which is not contained by another implicant of f.*

Definition 10 (Essential Implicant). *A prime implicant is essential if it is the only prime implicant covering a specific minterm.*

Definition 11 (Prime Cover). *A prime cover is a cover consisting only of prime implicants.*

Let us look at an example in order to make the definitions clearer.

Example 3. *Assume we are given an incompletely specified function*

$$f \equiv x_1 x_2 \overline{x_3} y + \overline{x_1} x_2 x_3 y + x_1 x_2 x_3 y + x_1 \overline{x_2} x_3 y + x_1 \overline{x_2} x_3 \overline{y}.$$

The function is depicted as a coloring of minterms in Figures 3.1a and 3.1b. In Figure 3.1a f is covered by three implicants α, β and γ where β and γ are prime since they are not contained by another implicant of f. Looking at α however, we see that there could be an implicant covering the minterms $x_1 x_2 \overline{x_3}$ and $x_1 x_2 x_3$ which would contain α. In Figure 3.1b α is now prime as well.

In the first two figures α and γ are essential, while β may be discarded because the only on-set minterm covered by β is already covered by γ (in Figure 3.1b also by α). Both these covers are therefore not minimum.

In Figure 3.1c the example is changed slightly. Including the don't care minterm in the cover allows to find a single implicant covering the depicted function. This cover is minimum.

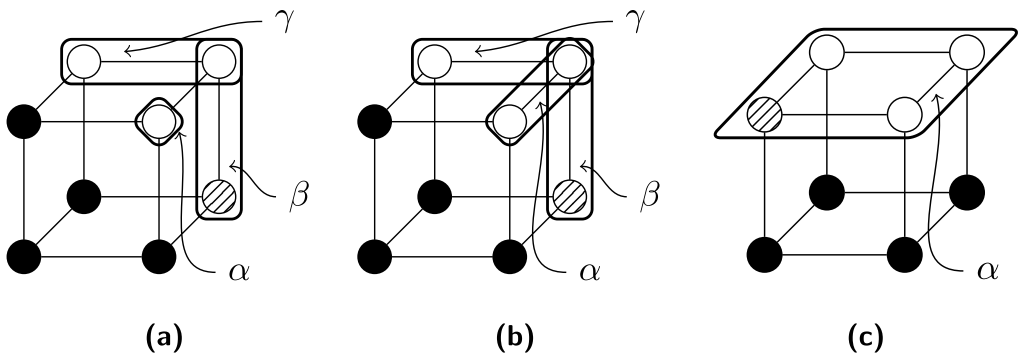


Figure 3.1: Example of covers and implicants. ($\rightarrow$: $x_1, \uparrow$: $x_2, \nearrow$: x_3)

3.1.2 Exact Minimization

The first algorithm for exact minimization of logic circuits is the Quine-McCluskey algorithm. The starting point was Quine's Theorem:

Theorem 4 ([VOQ52, Theorem 1])**.** *There exists a minimum cover for f that is prime.*

Proof. A minimum cover which is not prime contains non-prime implicants. All such implicants can be replaced by the prime implicants containing them without changing the minimality property of the cover. □

The benefit of Quine's theorem is that it limits the search for a minimum cover to the search for a minimum prime cover. Quine then proposed a prime implicant table to solve the covering problem. A means for computing all prime implicants is the ITERATEDCONSENSUS procedure (based on the *consensus* operation), which we will not describe here, however.

Definition 12 (Prime Implicant Table)**.** *A prime implicant table is a two-valued matrix whose columns represent the prime implicants of the function and the rows represent the minterms of the function. An entry a_{ij} of the matrix is 1 if the ith minterm is covered by the jth prime implicant.*

After setting up a prime implicant table it can be reduced by removing dominated rows and columns. Note that essential implicants must remain in the cover. The reduction may lead to a so-called *cyclic core*, which does not change by applying the reduction rules. In order to solve the cyclic core, a solution was proposed by McCluskey [Mcc56] which explores the cost of all possibilities. A better approach is to use *branch and bound* (Petrick's method) in order to prune some of the possibilities early by evaluating the cost of a subset of primes with a lower bound before computing the exact cost. If the evaluated cost is too high, the computation can be spared.

The major problem of this solution is the construction of the prime implicant table which might be exponential in size (both in the number of minterms and prime implicants). Therefore, it might be impossible to set up the table to begin with. Furthermore, the table covering problem is NP-complete.

By exploiting specific properties, such as unateness and complemented covers—moreover divide and conquer strategies and again smart pruning—it is possible to make the exact minimization approach practical to some extent.

3.1.3 Heuristic Minimization

Due to the described problems of exact approaches, heuristic approaches are favorable in practice. They provide a way to get close to the minimum cardinality cover, but with feasible computational effort. Heuristic approaches avoid computing the prime implicant table and start with a cover of the function as provided by the represented formula. This cover is then iteratively improved by applying operations on the cover. The common operators are:

- Expand replaces implicants with prime implicants containing them.
- Reduce replaces prime implicants with non-prime implicants. The update must result in a cover again.
- Reshape looks at pairs of implicants. One implicant is expanded and one is reduced in such a way that the updated cover is valid.
- Irredundant removes redundant implicants from the cover.

Different tools may use the operators in different orders or use only a subset of them. Espresso [BSVMH84] uses only Expand, Reduce and Irredundant (in that order). Furthermore, implementation details of the operators may differ since they are based on heuristics.

3.2 ABC

ABC [BM10] is a tool unifying synthesis and verification of combinational and sequential[1] circuits. ABC offers a broad set of functions: For sequential logic synthesis it is necessary to support functionality such as mapping of a circuit to standard cells, placement of these and retiming of the circuit. On the verification side, techniques such as bounded model

[1]A sequential circuit is like a combinational circuit, but has memory. Therefore such circuits have state, which changes either asynchronously, or synchronously following a clock.

checking and satisfiability solving are provided among others. For a complete list of the functionality we refer to `www.eecs.berkeley.edu/~alanmi/abc/`.

Internally ABC uses and-inverter-graphs graphs (AIGs) to represent circuits (combinational and with an extension also sequential ones) and implements various means for operating on the representation. Operations like reduction, rewriting, restructuring and balancing of the graph are available in ABC. In our case, these operations are helpful to estimate the gate count and area of the circuit in a more realistic way, when benchmarking the impact our minimization techniques have on circuit size.

Chapter 4

Determinization of Boolean Relations Using BDDs

In this chapter we present the first contribution of this thesis to the problem of Boolean relation determinization. First, we revisit the problem and describe our main idea behind our approach in the following section. We then show that the size of the circuits (functions) computed by ExtractFunctionFromBDD (Figure 2.8) depends on the order in which the variables are picked by the "optimization loop". We will refer to this order as variable sequence, not to be confused with the variable order of a BDD. We demonstrate the order dependency in a small example. As a result of this observation we present two solutions for finding the function depending on the minimum number of variables, independent of the variable sequence. Whereas ExtractFunctionFromBDD computes a locally optimal solution, our new approaches search for the global optimum.

In our first approach we employ an explicit search (by enumerating variable subsets). It is described in Section 4.3. In our second approach (Section 4.4) we add logic to the circuit representing the relation, which finds the solution in an implicit search.

Finally, we evaluate our implementation on benchmarks from GR(1) synthesis (cf. Appendix A). The performance impact, compared to ExtractFunctionFromBDD, was more significant than expected, and we were only able to complete the runs on small examples, with larger ones running into timeouts. The results for these small benchmarks show that the local optimum equals the global optimum.

4.1 Problem Statement

As seen in Section 2.4, there are various methods readily available for solving relations. A central problem remains however: Namely, the size of the resulting combinational circuit is unsatisfying. A metric for the circuit size is the number of logic gates. The goal set therefore was to find a determinization that minimizes the eventual number of gates.

In our approach for attacking this problem we assume that a relation $R \subseteq \mathbb{B}^n \times \mathbb{B}$ is given in BDD form. The sought-after functional implementation $f : \mathbb{B}^m \mapsto \mathbb{B}$, with $m \leq n$, of R then is also in BDD form. Such a function can be converted to a circuit, by replacing each BDD vertex with a 2-to-1 multiplexer (cf. Figure 2.2).

The circuit size, thus, is directly connected to the BDD size. The BDD size depends on the following two parameters.

1. The variable order of the BDD, and
2. the represented function f, itself.

Finding a good variable order is a central problem for BDDs. It has been studied intensely and there exist various heuristics for finding a good variable order (cf. Section 2.2.2.5).

As the problem of finding a good order can be considered solved (heuristically) the focus of this thesis lies on the second parameter: the represented function. In many applications, the function is fixed and therefore this parameter cannot be tweaked. In the case of relation determinization, however, the freedom of relations can be exploited to extract a function that may have a smaller BDD representation. The metric we employ for measuring the size of a function is the number of variables the function depends on. We call them support variables.

We illustrate in the following section that the extraction algorithm of [BGJ+07], which we presented in Section 2.4.2.2 does, in general, not reduce the number of support variables perfectly. Subsequently, we present two extraction algorithms which find an optimal solution.

4.2 Cofactor Optimization is Sequence-Dependent

The following example show that the analysis of dependent variables (the loop in ExtractFunctionFromBDD) only finds a locally optimal solution. We apply the algorithm with two different variable sequences. The result are two different functional implementations of the relation. One depending on two variables and another depending on a single variable only. The example is depicted in Figure 4.1 with the variable elimination step simplified to universal quantification.

Example 4. *Let $R \subseteq \mathbb{B}^n \times \mathbb{B}$, with input variables $\vec{x} = \{x_1, x_2, x_3\}$ and output variable*

y, be a relation with characteristic function

$$\begin{aligned}R(\vec{x},y) \equiv& \overline{x_1 x_2 x_3 y} + x_1\overline{x_2 x_3 y} + x_1\overline{x_2 x_3}y + x_1\overline{x_2}x_3\overline{y} + \\ & x_1\overline{x_2}x_3 y + \overline{x_1 x_2}x_3\overline{y} + \overline{x_1}x_2\overline{x_3 y} + \overline{x_1}x_2\overline{x_3}y + \\ & x_1 x_2\overline{x_3}y + x_1 x_2 x_3\overline{y} + x_1 x_2 x_3 y + \overline{x_1}x_2 x_3\overline{y}\end{aligned}$$

When applying ExtractFunctionFromBDD *to R, $\vec{x}$ and y, the positive cofactor of R with respect to y, R_1, is initialized to*

$$R_1 \equiv x_1\overline{x_2 x_3} + x_1\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1 x_2\overline{x_3} + x_1 x_2 x_3 \equiv x_1 + \overline{x_1}x_2\overline{x_3}.$$

The negative cofactor of R with respect to y is R_0. It is initialized to

$$\begin{aligned}R_0 \equiv& \overline{x_1 x_2 x_3} + x_1\overline{x_2 x_3} + x_1\overline{x_2}x_3 + \overline{x_1 x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1 x_2 x_3 + \overline{x_1}x_2 x_3 \\ \equiv& \overline{x_1} + x_1\overline{x_2} + x_1 x_2 x_3.\end{aligned}$$

In subsequent steps R_1' is set to $x_1 x_2\overline{x_3}$ and R_0' to $\overline{x_1 x_2} + \overline{x_1}x_2 x_3$.

The `foreach` *loop may iterate over the input variables in different order. In this run let us assume the sequence x_1, then x_2 and finally x_3:*

Iteration 1. *In the first loop iteration $x = x_1$. R_1'' is set to $\exists x_1.\ R_1' \equiv x_2\overline{x_3}$ and R_0'' to $\exists x_1.\ R_0' \equiv \overline{x_2} + x_2 x_3$. The conjunction of R_1'' and R_0'' is unsatisfiable. Therefore* ExtractFunctionFromBDD *updates R_1' and R_0' and eliminates x_1 from the relation.*

Iteration 2. *In the second iteration $x = x_2$. The algorithm assigns $\exists x_2.\ R_1' \equiv \overline{x_3}$ to R_1'' and $\exists x_2.\ R_0' \equiv 1$ to R_0''. The conjunction yields $\overline{x_3}$, therefore no update is made.*

Iteration 3. *The final iteration has $x = x_3$ and R_1'' is $\exists x_3.\ R_1' \equiv x_1 x_2$. The algorithm sets R_0'' to $\exists x_3.\ R_0' \equiv 1$. The conjunction of R_1'' and R_0'' is $x_1 x_2$ and again no update is made and we exit the loop.*

To summarize, the execution of the loop managed to eliminate one input variable—that is x_1—from the relation. The procedure yields the function $f \equiv x_2\overline{x_3}$ implementing the relation. A corresponding circuit, if converted from a BDD, consists of two 2-to-1 multiplexers.

We now compute the loop with reversed order of the input variables: First x_3, then x_2 and x_1.

Iteration 1. *The first iteration has $x = x_3$. R_1'' is set to $\exists x_3.\ R_1' \equiv x_1 x_2$. R_0'' is set to $\exists x_3\ R_0' \equiv \overline{x_1}$. As $R_0'' \cdot R_1'' \equiv 0$, the relations are updated. Input variable x_3 is eliminated.*

		x_3x_1: 00	01	11	10
x_2	0	0	–	–	0
	1	–	1	–	0

(a) Relation R

		x_3: 0	1
x_2	0	0	0
	1	1	0

(b) $\forall x_1.\ R$

		x_1: 0	1
x_2	0	0	–
	1	0	1

(c) $\forall x_3.\ R$

x_1: 0	1
0	1

(d) $\forall x_2 \forall x_3.\ R$

Figure 4.1: Example 4 in pictures.

Iteration 2. *In the second iteration $x = x_2$.* ExtractFunctionFromBDD *assigns $\exists x_2.\ R_1' \equiv x_1$ to R_1'' and $\exists x_2.\ R_0' \equiv \overline{x_1}$ to R_0''. Again, the conjunction of R_1'' and R_0'' evaluates to 0 and x_2 gets eliminated as well.*

Iteration 3. *In the final iteration $x = x_1$. R_1'' gets assigned $\exists x_1.\ R_1' \equiv 1$ and R_0'' gets $\exists x_1.\ R_0' \equiv 1$. Therefore no further update is made.*

This run of the loop eliminated both x_2 and x_3 from the relation. Finally, extracting the function yields $f \equiv x_1$, which can be implemented as a single 2-to-1 multiplexer when converting the BDD.

In the second run, the two variables x_2 and x_3 have been eliminated from R, as opposed to just x_1 in the first run. This example illustrates that ExtractFunctionFromBDD depends on the sequence in which the input variables are quantified and eliminated from the relation. The example also shows that eliminating more variables might lead to a smaller circuit implementation of the extracted function f and is therefore desirable.

4.2.1 Independence of Variables

An important notion used in the following approaches for determinizing the relation with the minimum number of support variables is the independence of a relation from a certain set of variables. Proposition 2 states what it means for a relation to be independent of a set of input variables. A similar condition is applied in ExtractFunctionFromBDD [BGJ+07].

Proposition 2. *We say that a single-output total relation $R(\vec{x}, y)$ is independent of a set of variables $\{x_0, \ldots, x_l\} \subseteq \vec{x}$, if and only if $\exists y\ \forall x_0 \cdots \forall x_l.\ R(\vec{x}, y)$ is valid.*

The set of variables $\vec{x}_{ind} = \{x_0, \ldots, x_l\}$, for which Proposition 2 is valid, is said to be ***R*-independent**. Otherwise, it is ***R*-dependent**. The set representing the R-dependent variables is $\vec{x}_{dep} = \vec{x} \setminus \vec{x}_{ind}$.

4.2.2 Determinization

We will now describe our methods for finding the maximum R-independent set $\vec{x}_{ind}$. With such a set, we can determinize a relation, such that it depends on the minimum number of support variables. Such a functional implementation of R can be computed by removing the set of independent variables $\vec{x}_{ind} = \{x_1, \ldots, x_l\}$ using universal quantification of the cofactors R_y and $R_{\overline{y}}$. We get the following relations:

$$R_0 = \forall x_0 \cdots \forall x_l.\ R(\vec{x}, 0),$$
$$R_1 = \forall x_0 \cdots \forall x_l.\ R(\vec{x}, 1).$$

The functional implementations of R, with the minimum and maximum on-set, respectively, are $f_{max} \equiv R_1$ and $f_{min} \equiv \overline{R_0}$. The minimality and respectively maximality properties of these functions can be seen in Figure 2.9 (Page 27).

Example 5. *Let $R \subseteq \vec{x} \times y$ be a relation over a set of input variables $\vec{x} = \{x_1, x_2\}$ and a single output variable y with characteristic function*

$$R(\vec{x}, y) \equiv \overline{x_1 x_2 y} + x_1 \overline{x_2} y + x_1 \overline{x_2 y} + \overline{x_1} x_2 y + x_1 x_2 y$$

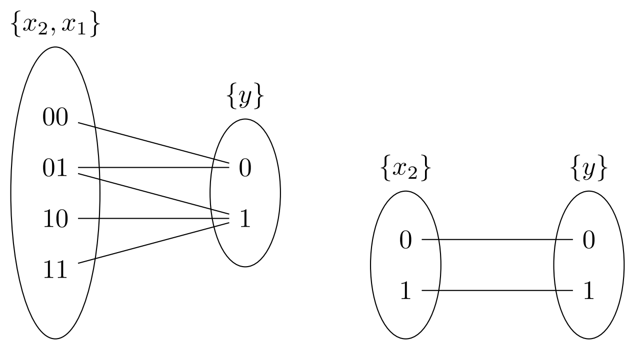
We first determine that $\{x_1\}$ is an R-independent subset of $\vec{x}$ according to Section 4.2.1 ($\exists y \forall x_1.\ R(\vec{x}, y) \equiv 1$). It is maximum, since for the only larger subset (i.e. $\{x_1, x_2\}$) Proposition 2 evaluates to false ($\forall x_1, x_2.\ R(\vec{x}, y) \equiv \overline{y} y \equiv 0$). In the second step, after determining that $\{x_1\}$ is the maximum R-independent subset, we can determinize the relation as described above. The computation of f_{min} proceeds as follows.

$$\begin{aligned} R_0 &\equiv \forall x_1.\ R(\vec{x}, 0) \\ &\equiv \forall x_1.\ (\overline{x_1 x_2} + x_1 \overline{x_2}) \\ &\equiv \overline{x_2} \end{aligned}$$

The resulting function $f_{min} \equiv \overline{R_0} \equiv x_2$. Let us now also compute f_{max}:

$$\begin{aligned} R_1 &\equiv \forall x_1.\ R(\vec{x}, 1) \\ &\equiv \forall x_1.\ (x_1 \overline{x_2} + \overline{x_1} x_2 + x_1 x_2) \\ &\equiv x_2 \end{aligned}$$

We see that $f_{max} \equiv f_{min} \equiv x_2$ for this simple relation. We depict the example in Fig-



(a) $R(\vec{x}, y)$ **(b)** $\forall x_1.\ R(\vec{x}, y) \equiv x_2$

Figure 4.2: The relation R of Example 5 and after universal quantification of x_1.

ures 4.2a and 4.2b. Without analyzing for variable independence first,

$$f_{max} \equiv R(\vec{x}, 1) \equiv x_1 + x_2$$

would be a functional implementation depending on more variables and demand a more complex circuit to implement it.

4.3 Explicit Solution

The first of two ways we present, to find a maximum R-independent subset, is an explicit exhaustive search. Our algorithm enumerates all the subsets of the set of input variables of R. For each subset it tests Proposition 2 until the maximum set, satisfying the condition, is found.

The feasibility of this approach heavily depends on the nature of the relation, as a set of size n has $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ subsets of size k (called k-combinations). Therefore, there are $\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$ subsets in total.

A relatively straight-forward approach is to incrementally increase the size k of the subsets starting with $k = 1$ and decrease the input space whenever a variable is determined to be R-dependent. That is, a variable which is in none of the R-independent subsets of a particular size. The hope for this approach is that there are many R-dependent variables. This would lead to the number of candidate variables n decreasing and approaching the size k of the subsets which are checked. In turn this would result in a pruning of the search space. As soon as $k \geq n$ the algorithm has found at least one maximum subset. We present pseudo-code for this algorithm in Figure 4.3. IndependentCombinations prunes the R-dependent variables and also returns a maximum R-independent combination for the current k.

With the information of the maximum R-independent set, the relation R can be

```
proc EXPLICIT(R, x⃗)
  candidates ← x⃗
  n ← |candidates|
  k ← 1
  while k ≤ n
    (candidates, x⃗_ind) ← INDEPENDENTCOMBINATIONS(candidates, k, R)
    n ← |candidates|
    k ← k + 1
  return x⃗_ind
```

Figure 4.3: Approach for the incremental computation of the maximum set of R-independent variables.

determinized such that its functional implementation depends on the minimum number of input variables, as described above.

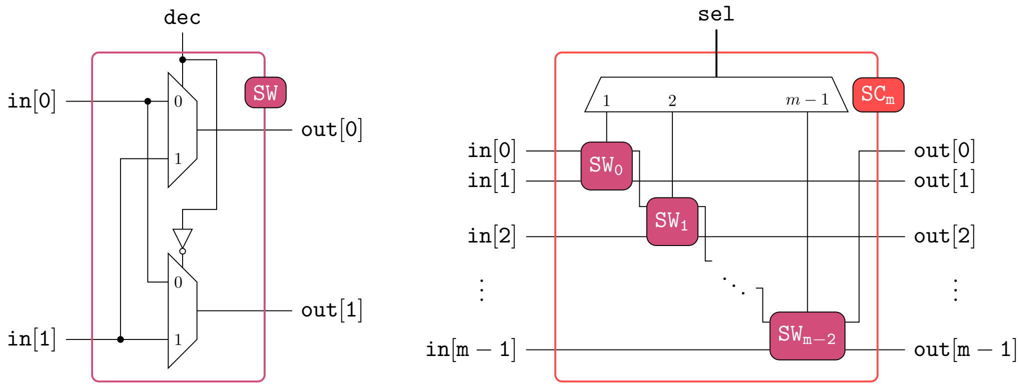
4.4 Logically Encoded Solution

Our second approach is to encode the selection of the variable combinations as a combinational circuit. This circuit is capable of generating all combinations of its first k inputs at its outputs. Such a circuit is called *combination network*.

The purpose of the combination network is to act as a proxy between the inputs and the combinational circuit representing the characteristic function of the relation which we want to determinize. The combination network and the relation circuit are connected via functional composition and this new logic circuit can be embedded in an argument for variable independence, similar to the one presented in Proposition 2.

4.4.1 Combination Network

A combination network `CN` is a circuit with n primary inputs $\texttt{CN.in}[0], \ldots, \texttt{CN.in}[\texttt{n}-1]$, and n primary outputs $\texttt{CN.out}[0], \ldots, \texttt{CN.out}[\texttt{n}-1]$. Furthermore it employs selection inputs `CN.sel` which encode a particular mapping from inputs to outputs. There is enough freedom in the network in order to generate all combinations of its first k inputs at its outputs. The network is comprised of several smaller building blocks, which we call *selection cells*. The selection cells again consist of a *decoder* and *swap cells*. We will now describe these blocks.



(a) Swap Cell `SW`. **(b)** Selection Cell $\mathtt{SC_m}$.

Figure 4.4: Components of a combination network (flow from `in` to `out`).

4.4.1.1 Selection Cell

A selection cell is a circuit with an equal number of inputs and outputs. We call a selection cell with m inputs ($\mathtt{SC_m.in}$) and m outputs ($\mathtt{SC_m.out}$) $\mathtt{SC_m}$. We may drop the subscripts if clear from the context.

A selection cell utilizes $\lceil \log_2 m \rceil$ selector inputs $\mathtt{SC_m.sel}$. These selector bits are interpreted as the binary representation of an index $0 \leq i \leq m-1$. The functionality of a selection cell can be split into two cases:

Case 1. The input with index i, selected by `SC.sel`, is propagated to the output with index 0, that is $\mathtt{SC.out[0]} \leftarrow \mathtt{SC.in[i]}$. The input with index 0 then takes i's place and gets propagated to output position i: $\mathtt{SC.out[i]} \leftarrow \mathtt{SC.in[0]}$.

Case 2. All other inputs with indices $j \neq i$ and $j \neq 0$ are propagated from `SC.in[j]` to `SC.out[j]`.

The inner workings of `SC` are as follows. The circuit uses $m-1$ swap cells $\mathtt{SW_0}, \ldots, \mathtt{SW_{m-2}}$, each with two inputs (`SW.in[0]` and `SW.in[1]`) and two outputs (`SW.out[0]` and `SW.out[1]`) and a decision input (`SW.dec`).[1] The swap cells are connected in the following way:

Case 1. Swap cell $\mathtt{SW_0}$ has inputs `SC.in[0]` and `SC.in[1]`.

Case 2. The inputs to the ith swap cell $\mathtt{SW_i}$, for $1 \leq i \leq m-2$, are $\mathtt{SW_i.in[0]} \leftarrow \mathtt{SW_{i-1}.out[0]}$ and $\mathtt{SW_i.in[1]} \leftarrow \mathtt{SC.in[i+1]}$.

The outputs of the selection circuit are defined as $\mathtt{SC.out[i+1]} \leftarrow \mathtt{SW_i.out[1]}$ for $0 \leq i \leq m-2$ and $\mathtt{SC.out[0]} \leftarrow \mathtt{SW_{m-2}.out[0]}$.

[1] A swap cell can simply be constructed from two 2-to-1 multiplexers and an inverter. Such a circuit can be seen in Figure 4.4a

	$\mathtt{SC_4.sel}$			
	0	1	2	3
$0_\mathtt{o}$	$0_\mathtt{i}$	$1_\mathtt{i}$	$2_\mathtt{i}$	$3_\mathtt{i}$
$1_\mathtt{o}$	$1_\mathtt{i}$	$0_\mathtt{i}$	$1_\mathtt{i}$	$1_\mathtt{i}$
$2_\mathtt{o}$	$2_\mathtt{i}$	$2_\mathtt{i}$	$0_\mathtt{i}$	$2_\mathtt{i}$
$3_\mathtt{o}$	$3_\mathtt{i}$	$3_\mathtt{i}$	$3_\mathtt{i}$	$0_\mathtt{i}$

Table 4.1: Example 6.

Finally, the $m-1$ decision signals—that is one per swap cell—are outputs of a $\lceil \log_2 m \rceil$-to-m decoder. These signals, therefore, are one-hot encoded: Swap cell $\mathtt{SW_i}$ is activated if the $(i+1)$st output of the decoder is active (index 0 is left unused, as no swap has to be performed, when the input with index 0 is selected). The input to the decoder are the $\mathtt{SC.sel}$ signals. A 2-to-4 decoder with inputs $\mathtt{SC.sel}[0]$ and $\mathtt{SC.sel}[1]$ and outputs $\mathtt{SW_1.dec}, \ldots, \mathtt{SW_3.dec}$, for example, has the following minterms:

$$\begin{aligned}
\mathtt{SW_1.dec} &\equiv \overline{\mathtt{SC.sel}[1]} \cdot \mathtt{SC.sel}[0], \\
\mathtt{SW_2.dec} &\equiv \mathtt{SC.sel}[1] \cdot \overline{\mathtt{SC.sel}[0]}, \\
\mathtt{SW_3.dec} &\equiv \mathtt{SC.sel}[1] \cdot \mathtt{SC.sel}[0].
\end{aligned}$$

A selection cell as described, comprised of a decoder and swap cells, is depicted in Figure 4.4b. Example 6 is supposed to provide a feel for how a selection cell with 4 inputs and outputs operates.

Example 6. *The selection signal of* $\mathtt{SC_4}$ *is 2 bits wide and allows the choices 0, 1, 2 and 3. Case 0 maps input* $\mathtt{SC.in}[0]$ *(abbreviated as* $0_\mathtt{i}$*) to output* $\mathtt{SC.out}[0]$ *(abbreviated as* $0_\mathtt{o}$*). Case 1 maps* $0_\mathtt{i}$ *to* $1_\mathtt{o}$ *(and* $1_\mathtt{i}$ *to* $0_\mathtt{o}$*), and so on. There is a choice for the selection signal to map the first input* $0_\mathtt{i}$ *to any of the outputs. Table 4.1 shows the input-output mappings for all assignments of* $\mathtt{SC_4.sel}$*.*

4.4.1.2 Construction of the Combination Network

Now, that the components of the combination network are defined, we will use them to construct the network.

Every selection cell can, simply put, push the first input ($\mathtt{SC.in}[0]$) to either of its outputs. The basic idea is to employ k selection cells of increasing size connected in series, so that the first k inputs of $\mathtt{CN}$ can be shifted to either output of $\mathtt{CN}$. We will describe the specific way of connecting the selection cells and illustrate the circuit behavior in an example.

Case 1. The first selection cell (in direction of the information flow) is $\mathtt{SC_{n-k+1}}$. This cell

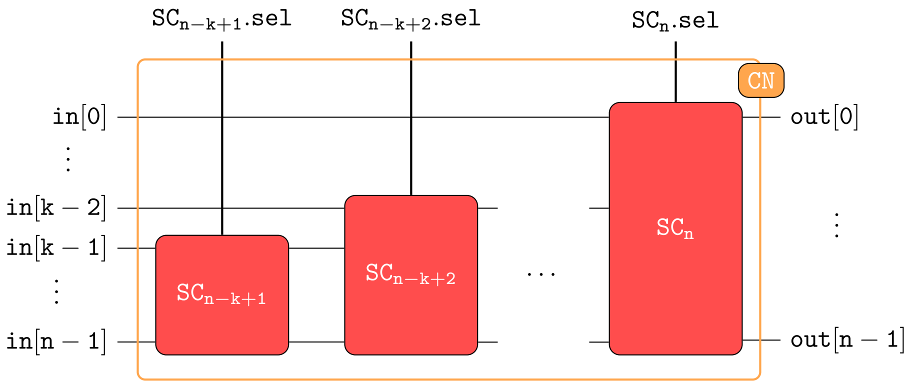


Figure 4.5: A combination network CN.

gets the inputs

$$\begin{aligned} \mathtt{SC}_{\mathtt{n-k+1}}.\mathtt{in}[0] &\leftarrow \mathtt{CN}.\mathtt{in}[\mathtt{k}-1], \\ &\vdots \\ \mathtt{SC}_{\mathtt{n-k+1}}.\mathtt{in}[\mathtt{n}-\mathtt{k}] &\leftarrow \mathtt{CN}.\mathtt{in}[\mathtt{n}-1]. \end{aligned}$$

Case 2. The inputs to the selection cells in the subsequent stages with $2 \leq i \leq k$ are defined as follows.

$$\begin{aligned} \mathtt{SC}_{\mathtt{n-k+i}}.\mathtt{in}[0] &\leftarrow \mathtt{CN}.\mathtt{in}[\mathtt{k}-\mathtt{i}], \\ \mathtt{SC}_{\mathtt{n-k+i}}.\mathtt{in}[1] &\leftarrow \mathtt{SC}_{\mathtt{n-k+i-1}}.\mathtt{out}[0], \\ &\vdots \\ \mathtt{SC}_{\mathtt{n-k+i}}.\mathtt{in}[\mathtt{n}-\mathtt{k}+\mathtt{i}-1] &\leftarrow \mathtt{SC}_{\mathtt{n-k+i-1}}.\mathtt{out}[\mathtt{n}-\mathtt{k}+\mathtt{i}-2]. \end{aligned}$$

Each selection cell has the necessary selection signals (cf. Section 4.4.1.1), which results in

$$M = \sum_{m=n-k+1}^{n} \lceil \log_2 m \rceil$$

selection signals in total for the combination network. M is of the order $\mathcal{O}(k \log n)$.

Finally, the outputs of CN are defined as $\mathtt{CN}.\mathtt{out}[\mathtt{i}] \leftarrow \mathtt{SC}_{\mathtt{n}}.\mathtt{out}[\mathtt{i}]$, for $0 \leq i \leq n-1$. Every output signal $\mathtt{CN}.\mathtt{out}[\mathtt{i}]$ is a Boolean function and the whole combination network is defined by a vector of functions $(\mathtt{CN}.\mathtt{out}[0], \ldots, \mathtt{CN}.\mathtt{out}[\mathtt{n}-1])$. A combination network as described above is depicted in Figure 4.5.

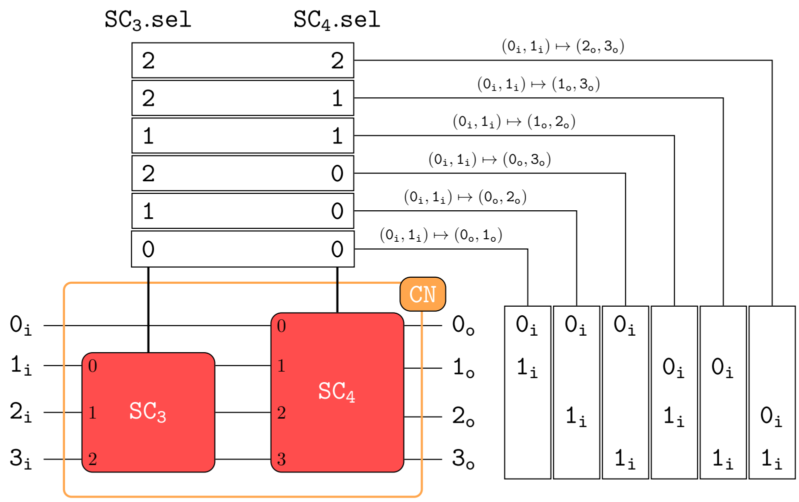


Figure 4.6: Example 7.

4.4.1.3 Mechanics of a Combination Network

The mechanics of a combination network are as follows. A selection cell $\mathtt{SC_i}$ is responsible for the output position of input $\mathtt{CN.in}[\mathtt{n}-\mathtt{i}]$. The values for the selection signals of the selection cells define a mapping from input indices to output indices of the combination network. The following example demonstrates how a combination network with $n = 4$ and $k = 2$ works.

Example 7. *The combination network with 4 inputs and $k = 2$ consists of 2 selection cells: $\mathtt{SC_3}$ and $\mathtt{SC_4}$. The functionality of the network is to map the first pair (due to $k = 2$) of inputs (i.e. $(\mathtt{CN.in}[0], \mathtt{CN.in}[1])$, which is abbreviated as $(\mathtt{0_i}, \mathtt{1_i})$) to any pair of outputs. Analogous to the inputs written with a subscript $\mathtt{i}$, the outputs are sub-scripted with $\mathtt{o}$. The $6(= \binom{4}{2})$ pairs, the network should be able to produce, are: $(\mathtt{0_o}, \mathtt{1_o}), (\mathtt{0_o}, \mathtt{2_o}), (\mathtt{0_o}, \mathtt{3_o}), (\mathtt{1_o}, \mathtt{2_o}), (\mathtt{1_o}, \mathtt{3_o})$ and $(\mathtt{2_o}, \mathtt{3_o})$. Figure 4.6 shows the respective choices for the selection signals $\mathtt{SC_3.sel}$ and $\mathtt{SC_4.sel}$ and the resulting positions of the input signals $\mathtt{0_i}$ and $\mathtt{1_i}$ after application of the combination network with two stages. Looking at the case with $\mathtt{SC_3.sel} = 2$ and $\mathtt{SC_4.sel} = 2$, first $\mathtt{SC_3}$ pushes $\mathtt{1_i}$ to $\mathtt{SC_4.in}[3]$. Then $\mathtt{SC_4}$ propagates $\mathtt{SC_4.in}[3]$ to $\mathtt{3_o}$ and input $\mathtt{0_i}$ to $\mathtt{2_o}$. This results in $(\mathtt{0_i}, \mathtt{1_i})$ ending up at positions $(\mathtt{2_o}, \mathtt{3_o})$.*

The network, however, provides more freedom than necessary: $\mathtt{SC_3.sel} = 0, \mathtt{SC_4.sel} = 1$ would, for example, generate $(\mathtt{0_o}, \mathtt{1_o})$ as well—if permuted. There are $3 \cdot 4 = 12$ possible assignments to $\mathtt{SC_3.sel}$ and $\mathtt{SC_4.sel}$ for just 6 pairs.

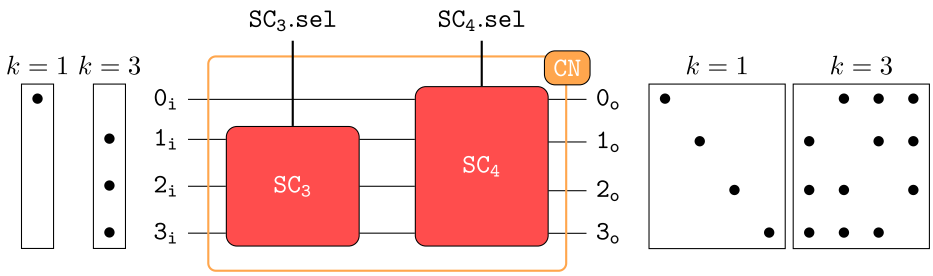


Figure 4.7: Depiction of input and output positions in a combination network with 4 inputs and $k = 1$ and $k = 3$.

4.4.1.4 Remark on the Symmetry of Choosing Elements

The combination network, as described in the previous sections, is not optimal because it does not take advantage of the symmetry of the binomial coefficient: "Choosing k of n elements" can also be regarded as "not choosing $(n-k)$ elements". Therefore, as soon as $k > \lfloor n/2 \rfloor$ elements are to be picked, that should be regarded as not picking $(n-k)$ elements. Effectively, this means a combination network should consist of no more than $\lfloor n/2 \rfloor$ selection cells. We will take advantage of this observation in the following section, when using a combination network for cofactor optimization. Figure 4.7 shows the symmetry in a combination network with $n = 4$ inputs when either picking $k = 1$ elements or not picking $(n-k) = 3$ elements.

4.4.1.5 Optimization of the Cofactors Using a Combination Network

We now explain how a combination network can be used to optimize the cofactors of a relation R.

Given a combinational circuit representing the characteristic function of a relation $R(\vec{x}, y)$ with $\vec{x} = \{x_0, \dots, x_{n-1}\}$, first we construct a combination network `CN` with $k = \lfloor n/2 \rfloor$ stages is. Such a network is capable of producing all combinations of size smaller n. We then connect `CN` to R via functional composition. Therefore, we compute the characteristic function $\chi_{\texttt{CN}} = \bigwedge_{i=0}^{n-1} \texttt{CN.out}[\texttt{i}] \leftrightarrow x_i$ of `CN`. One way of computing the functional composition of R and $\chi_{\texttt{CN}}$ is

$$R' = \exists x_0 \cdots \exists x_{n-1}.(\chi_{\texttt{CN}} \cdot R).$$

$R'(\texttt{CN.in}, \texttt{CN.sel}, y)$ is a relation of input variables `CN.in`, selector variables `CN.sel` and the output variable y. Figure 4.8 depicts R'. After constructing R', our approach for finding the maximum R'-independent set of input variables consists of two steps:

1. We define a criterion, similar to Proposition 2, which says whether a subset of `CN.in`

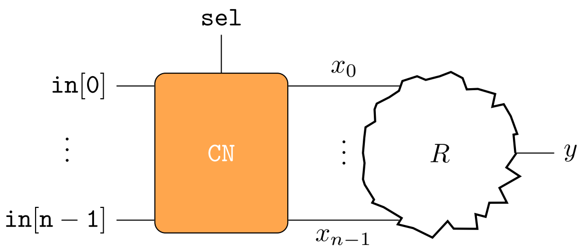


Figure 4.8: Relation $R' = \exists x_0 \cdots \exists x_{n-1}.(\chi_{\texttt{CN}} \cdot R)$.

of size k is R'-independent.

2. We search for the maximum subset, satisfying the independence criterion, with binary search.

The criterion for R'-independence of a subset of input variables is as follows.

Proposition 3. *A single-output total relation, augmented with a combination network, $R'(\texttt{CN.in}, \texttt{CN.sel}, y)$, is independent of a set of k input variables, if either formula of the following case distinction is valid.*

Case 1. $k \leq \lfloor n/2 \rfloor$

$$\begin{aligned}
&\exists \texttt{CN.sel}[0] \cdots \exists \texttt{CN.sel}[M-1] \\
&\forall \texttt{CN.in}[\texttt{k}] \cdots \forall \texttt{CN.in}[\texttt{n}-1] \\
&\exists y \\
&\forall \texttt{CN.in}[0] \cdots \forall \texttt{CN.in}[\texttt{k}-1].R'(\texttt{CN.in}, \texttt{CN.sel}, y)
\end{aligned}$$

Case 2. $k > \lfloor n/2 \rfloor$

$$\begin{aligned}
&\exists \texttt{CN.sel}[0] \cdots \exists \texttt{CN.sel}[M-1] \\
&\forall \texttt{CN.in}[0] \cdots \forall \texttt{CN.in}[\texttt{k}-1] \\
&\exists y \\
&\forall \texttt{CN.in}[\texttt{k}] \cdots \forall \texttt{CN.in}[\texttt{n}-1].R'(\texttt{CN.in}, \texttt{CN.sel}, y)
\end{aligned}$$

Either case of Proposition 3 is very similar to Proposition 2. The main difference is the inclusion of the selector signals in the criterion. The existential quantification of the `CN.sel` signals allows for the necessary freedom in the mapping from inputs to outputs in the combination network. Since all the adjustments in the combination network solely depend on the selection signals, the existential quantification implicitly generates all the k-combinations.

The first case corresponds to choosing k elements and the second case to not choosing $(n-k)$ elements. The case distinction is made on the value of k. The difference between the cases lies in the order of quantification over the `CN.in` signals. The innermost universal quantification is over the signals checked for independence.

Now that we have the criterion, our goal is to find the maximum k for which Proposition 3 is valid. To achieve this, we employ binary search. Let the function GetSatAssignment(f) (cf. Section 2.2.2.4) return an assignment to all the variables in f, such that the assignment makes f true. Then the algorithm in Figure 4.9 finds the maximum k and a satisfying assignment $\mathtt{CN.sel_0}$ to `CN.sel` which makes Proposition 3 valid.

4.4.1.6 Determinization of R

In addition to the maximum number of R'-independent variables k_{max}, The algorithm in Figure 4.9 yields a satisfying assignment to the selector variables of the combination network of R'. These pieces of information can be used to determinize R, such that R depends on the minimum number of input variables.

Plugging $\mathtt{CN.sel_0}$ into R' yields a circuit with inputs `CN.in` and output y. The mapping from `CN.in` to the inputs of R becomes fixed. As we know k_{max}, we also know which inputs R' does (or does not) depend on. The R'-independent inputs can be removed by universal quantification, as in Section 4.2.2.

We can finally compute the functional implementations of R. The optimized cofactors, depending on the value of k_{max}, are as follows:

Case 1. $k_{max} \leq \lfloor n/2 \rfloor$

$$R_0 = \forall \mathtt{CN.in}[0] \cdots \forall \mathtt{CN.in}[\mathtt{k_{max}}-1].R'(\mathtt{CN.in}, \mathtt{CN.sel_0}, 0)$$
$$R_1 = \forall \mathtt{CN.in}[0] \cdots \forall \mathtt{CN.in}[\mathtt{k_{max}}-1].R'(\mathtt{CN.in}, \mathtt{CN.sel_0}, 1)$$

Case 2. $k_{max} > \lfloor n/2 \rfloor$

$$R_0 = \forall \mathtt{CN.in}[\mathtt{k_{max}}] \cdots \forall \mathtt{CN.in}[\mathtt{n}-1].R'(\mathtt{CN.in}, \mathtt{CN.sel_0}, 0)$$
$$R_1 = \forall \mathtt{CN.in}[\mathtt{k_{max}}] \cdots \forall \mathtt{CN.in}[\mathtt{n}-1].R'(\mathtt{CN.in}, \mathtt{CN.sel_0}, 1)$$

Both $f_{min} \equiv \overline{R_0}$ and $f_{max} \equiv R_1$ are functional implementations, with the minimum and maximum on-sets, respectively, of R.

```
proc BinarySearch(R', CN.in, CN.sel, y)
  k_max ← 0
  upper ← n
  lower ← 0
  while lower ≤ upper
    k ← ⌊(upper + lower)/2⌋
    qbf ← R'
    I_1 ← I_2 ← {}
    if k ≤ ⌊n/2⌋
      I_1 ← {CN.in[0], ..., CN.in[k − 1]}
      I_2 ← {CN.in[k], ..., CN.in[n − 1]}
    else
      I_1 ← {CN.in[k], ..., CN.in[n − 1]}
      I_2 ← {CN.in[0], ..., CN.in[k − 1]}

    foreach x ∈ I_1
      qbf ← ∀x.qbf
    qbf ← ∃y.qbf
    foreach x ∈ I_2
      qbf ← ∀x.qbf
    qbf' ← qbf
    foreach x ∈ CN.sel
      qbf ← ∃x.qbf

    if qbf = 1
      CN.sel_0 ← GetSatAssignment(qbf')
      k_max ← k
      lower ← k + 1
    else
      upper ← k − 1
  return (k_max, CN.sel_0)
```

Figure 4.9: Binary search for the maximal k and a satisfying assignment to CN.sel.

4.5 Implementation and Experimental Results

We implemented these methods as additions to the Marduk synthesis tool, which is part of Ratsy [BCG+10]. The tool is able to synthesize combinational logic circuits from temporal logic specifications given as GR(1) formulas (cf. Appendix A). Marduk itself is written in Python and uses the CUDD library (for BDD operations) which is implemented efficiently in C.

Both methods seem to be infeasible in practice. Besides toy examples, only the explicit method is able to synthesize tiny industrial examples: It was possible to synthesize the Genbuf01, Genbuf02 and Genbuf03 benchmarks[2]. The implicit method timed out when computing the characteristic function of the combination network. Different reordering methods, with and without dynamic reordering enabled, have been tried to no avail.

The working Genbuf benchmarks have a significant time penalty compared to the greedy method which was described in Section 2.4.2.2. This additional runtime had to be expected to some extent. The early pruning of the search space by elimination of the dependent variables, however, did not seem to happen.

Furthermore we observed in our experiments that the heuristic search eliminates as many variables as our exact methods for the small benchmarks we were able to complete.

[2]Genbuf is a buffer connected to 2 receivers and a variable number of senders—1, 2 and 3 in this case. Certain constraints must be satisfied in order to adhere to the specification. An example is that every request must be granted eventually (liveness).

Chapter 5

Determinization of Boolean Relations Using Interpolants

We will now present our improvements to relation determinization via interpolation. Section 2.4.2.3 explains how circuits can be built from relations using interpolation.

Again, our goal is to minimize the size of the interpolant by reducing the number of variables it depends on. To achieve this, we try to reduce the resolution refutation used for the interpolant computation, similar as described in Subsection 2.5. We will generalize existing techniques using an approach based on clause subsumption. Furthermore, we will present an optimization to existing algorithms, leading to better proof reduction. Finally, we will look the effect of proof reduction on interpolant computation.

Lemma 2, Lemma 3 (and the corresponding corollaries), Theorem 5 and Theorem 6, as well as Examples 8, 12 and 13 have been contributed by Georg Weissenbacher. His proofs can be found in Appendix B.

5.1 Proof Reduction via Clause Subsumption

We present a framework for proof reduction consisting of two steps: clause substitution based on clause subsumption and proof correction. We will first define substitution for resolution proofs (cf. Section 2.3.1).

Definition 13 (Clause substitution)**.** *Let $R = (V_R, E_R, cla_R, piv_R, s_R)$ be a resolution proof and let $v_1, v_2 \in V_R$, such that v_1 is not an ancestor of v_2. The substitution of v_2 by v_1 in R, denoted by $R[v_1 \leftarrow v_2]$ is the directed acyclic graph $G = (V_G, E_G, cla_G, piv_G, s_G)$, with $V_G = V_R \setminus \{v_1\}$, $E_G = E_R \setminus \{(u, v) \mid u = v_1 \vee v = v_1\} \cup \{(v, v_2) \mid (v, v_1) \in E_R\}$, $cla_G(v) = cla_R(v)$ and $piv_G(v) = piv_R(v)$ for all $v \neq v_1$ and $s_G = s_R$ if $v_1 \neq s_R$ and v_2 otherwise.*

A DAG $R[v_1 \leftarrow v_2]$ might not be a valid resolution proof and might have to be reconstructed. The transformation $\textsc{RestoreRes}(G, v)$ can be used to restore the single resolution step at vertex v.

Definition 14 ($\textsc{RestoreRes}$)**.** *Let $G = (V_G, E_G, cla_G, piv_G, s_G)$. $\textsc{RestoreRes}(G, v)$ with $v \in V_G$ yields G if v is an initial vertex. For an internal vertex,*

- *if the resolution step is already valid in G, i.e. $\exists (v^+, v), (v^-, v) \in E_G$ with $piv_G(v) \in cla_G(v^+)$ and $\overline{piv_G(v)} \in cla_G(v^-)$, $\textsc{RestoreRes}(G, v)$ yields G' with*

$$cla_{G'}(u) \stackrel{\text{def}}{=} \begin{cases} \textsc{Res}(cla_G(v^+), cla_G(v^-), piv(v)) & \text{if } u = v \\ cla_G(u) & \text{otherwise} \end{cases}$$

- *otherwise the graph is corrected, by computing $G' = G[v \leftarrow u]$, where u is selected such that $(u, v) \in E_G$ and $\mathrm{var}(piv_G(v)) \notin cla_G(u)$ (there might be two choices for u).*

The procedure $\textsc{ReconstructProof}(G)$ [BIFH+09] applies $\textsc{RestoreRes}$ at each vertex of the proof in a post-order (parents first) traversal. The result is a correct resolution proof R, where $\forall (v_1, v), (v_2, v) \in E_R . cla_R(v) = \textsc{Res}(cla_R(v_1), cla_R(v_2), piv_R(v))$. Pseudo-code for $\textsc{ReconstructProof}$ is provided in Figure 5.1a.

The following lemma states that after a series of substitutions based on clause subsumption, followed by proof reconstruction, the sink clause might decrease in size.

Lemma 2. *Let R be a resolution proof, and let $\pi = \{v_1 \mapsto u_1, \ldots, v_k \mapsto u_k\}$ be a mapping such that v_i is not an ancestor of u_j for $1 \leq i, j \leq k$. If $cla_R(u_i) \subseteq cla_R(v_i)$ for $1 \leq i \leq k$, then the proof P obtained by applying* $\textsc{ReconstructProof}$ *to $R[v_1 \leftarrow u_1] \ldots [v_k \leftarrow u_k]$ has sink s_P with $cla_P(s_P) \subseteq cla_R(s_R)$.*

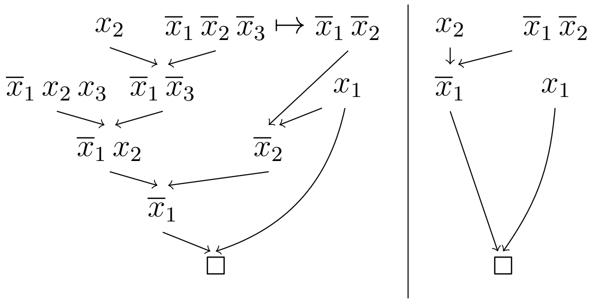
Let us look at Example 8 which demonstrates proof reduction via subsumption.

Example 8. *Consider the left proof in Figure 5.1b, in which the substitution is indicated by $\mapsto$. The refutation on the right of Figure 5.1b shows the result of* $\textsc{Reconstruct-Proof}$ *after substituting $\overline{x}_1\overline{x}_2$ for $\overline{x}_1\overline{x}_2\overline{x}_3$. Figure 5.2 shows the intermediate proofs after each application of* $\textsc{RestoreRes}$.

The algorithm $\textsc{RecycleUnits}$ presented in [BIFH+09] makes use of a special case of clause subsumption. Given a proof R, a subsuming clause $cla_R(w)$ with $|cla_R(w)| = 1$ and $w \in V_R$, at vertex $v \in V_R$ which is not an ancestor of w, is found by checking whether $piv_R(v)$ ($\overline{piv_R(w)}$, respectively) equals $cla_R(w)$. If that is the case, R is reduced by computing $R[v^+ \leftarrow w]$ ($R[v^- \leftarrow w]$, respectively).

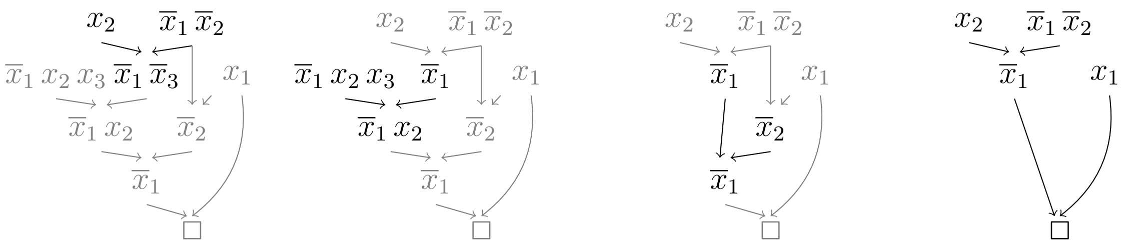
```
proc RECONSTRUCTPROOF (G)
  visited ← ∅
  Q ← {v | v ∈ V_G and v is initial}
  while Q is not empty
    v ← oldest element in Q
    Q ← Q \ {v}
    if (v ∉ visited
      and ∀(u,v) ∈ E_G.u ∈ visited)
    or v is initial
      visited ← visited ∪ {v}
      G ← RESTORERES(G, v)
      foreach (v,w) ∈ E_G
        Q ← Q ∪ {w}
```

(a) ReconstructProof



(b) Reducing proof size

Figure 5.1: Proof correction and effect of substitution



(a) **(b)** **(c)** **(d)**

Figure 5.2: Step by step application of ReconstructProof after the substitution in the left proof in Figure 5.1b, with invalid resolution steps highlighted.

5.1.1 Expansion Set

Tseitin [Tse68] first observed that a minimal tree-shaped proof is regular (Definition 15).

Definition 15 (Regular proof)**.** *A proof is regular, if on each path from sink to initial vertex, every literal is resolved at most once.*

Based on this observation, Bar-Ilan et al. presented the algorithm RmPivots in [BIFH+09] for regularization of parts of a resolution proof. Both Fontaine et al. [FMP11] and Gupta [Gup12] improve upon RmPivots, such that the algorithm works for proof DAGs instead of just the tree parts of a proof. We will show how these algorithms fit into our reduction framework using subsumption and present an optimization.

The way these algorithms work, is to compute an expansion set for each vertex v, containing the literals resolved along a path from the sink s to v. This set is used to detect

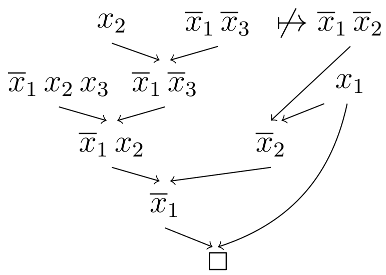


Figure 5.3: Invalid substitution according to Lemma 2

repeated resolutions on the same pivot along a path. In our framework, the expansion set allows the detection of certain substitutions, which are not found when checking with $cla(v)$. The following example demonstrates this by showing that the precondition of clause subsumption for substitutions in Lemma 2 is too strict. We will then proceed with formulating the expansion set.

Example 9. *Consider the proof in Figure 5.3. It is the same proof as in Figure 5.1b, with the only difference, that the clause* $cla(v_1) = \overline{x}_1\,\overline{x}_2\,\overline{x}_3$ *has been replaced by* $cla(v_1') = \overline{x}_1\,\overline{x}_3$*. Now, the substitution* $R[v_1' \leftarrow v_2]$ *with* $cla(v_2) = \overline{x}_1\,\overline{x}_2$ *is not valid according to Lemma 2 anymore, as* $cla(v_2) \not\subseteq cla(v_1')$*.* $R[v_1 \leftarrow v_2]$ *equals* $R[v_1' \leftarrow v_2]$ *and could be corrected in the same way as before, however. This shows that the precondition of Lemma 2 is too strict.*

Multiple formulations of the expansion set exist. The approach in [BIFH+09, RL in Algorithm 3] does not work on DAGs. The two improved approaches, working on DAGs as well, compute the *meet-over-all* paths at a vertex similar to the data flow equations known from compiler optimization [ALSU06]. The formulation in [Gup12, ρ in Section 5] constrains the set unnecessarily, by not allowing it to contain a literal in both its phases. We will restate the formulation of [FMP11, safeLiterals in Algorithm 6], which does not contain this restriction. The mapping $\sigma : V \mapsto 2^{\mathtt{Lit}}$, is the solution of

$$\sigma(v) = \begin{cases} cla(v) & \text{if } v \text{ has no children} \\ \bigcap_{(v,w)\in E}(\sigma(w) \cup \{\mathrm{rlit}(v,w)\}) & \text{otherwise} \end{cases} \tag{5.1}$$

where $\mathrm{rlit}(v, w)$ is defined for $(v, w) \in E$ as

$$\mathrm{rlit}(v,w) = piv(w),\ \text{if } piv(w) \in cla(v) \text{ and } \overline{piv(w)} \text{ otherwise.}$$

Let us revisit Example 9 taking into account expansion sets:

Example 10. *We get* $\sigma(v_1') = \{\overline{x}_1\,\overline{x}_2\,\overline{x}_3\}$ *and see that* $cla(v_2) \subseteq \sigma(v_1')$*. The expansion set allows us to detect that* $R[v_1' \leftarrow v_2]$ *is a valid substitution, which we did not find via plain clause subsumption.*

The following theorem improves Lemma 2 by using the expansion set for subsumption.

Theorem 5. *Let R be a resolution proof, let σ_R be a solution of Equation 5.1 for R, and let $\pi = \{v_1 \mapsto u_1, \ldots, v_k \mapsto u_k\}$ be a mapping such that for all $1 \leq i \leq j \leq k$ it holds that a) no vertex v_i is an ancestor of u_j, and b) if v_j is an ancestor of u_i then $\sigma_R(u_i) \subseteq \sigma_R(v_i)$. If $cla_R(u_i) \subseteq (cla_R(v_i) \cup \sigma_R(v_i))$ for $1 \leq i \leq k$, then applying* ReconstructProof *to $R[v_1 \leftarrow u_1] \ldots [v_k \leftarrow u_k]$ yields a proof P with sink s_P such that $cla_P(s_P) \subseteq cla_R(s_R)$.*

5.1.2 Algorithms for Proof Reduction

As for the formulation of the expansion set, there exist several similar versions of the algorithm for reducing (or partially regularizing) the proof. The different versions are called RmPivots [BIFH+09], RPI [FMP11] and AllRmPivots [Gup12] (among other minor variations).

In general they all work by building the expansion set iteratively in a bottom-up (children first) traversal of the proof R. When encountering a vertex v, such that $piv(v) \in \sigma(v)$ (or $\overline{piv(v)} \in \sigma(v)$), the sub-proof rooted at v^- (respectively v^+) is pruned. This is a valid transformation according to Theorem 5. The clause $cla_R(v^{\pm})$, with $v^{\pm}$ the remaining ancestor, subsumes $\sigma(v)$. Therefore, in our framework we can do $R[v \leftarrow v^{\pm}]$ collapsing the path. The precondition on the order of substitutions is fulfilled because of the children first traversal.

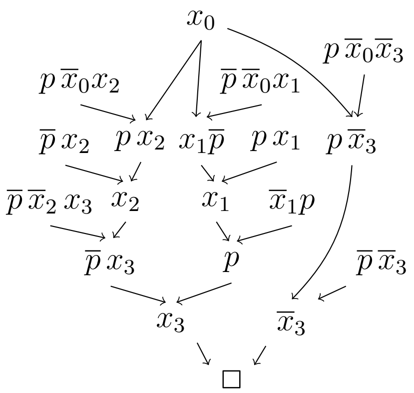
Note that RmPivots is very efficient, as it only needs two passes over the proof graph. One to reduce non-regular paths and another one to reconstruct the proof at the end. Its run-time is therefore linear in the size of the proof. Figure 5.5a shows pseudo-code for our version of the algorithm.

The following example shows the reduction of a redundant proof by applying RmPivots.

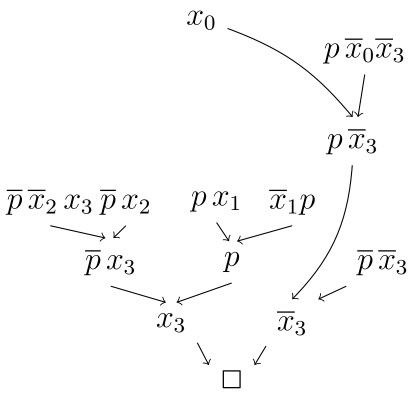
Example 11. *Consider the left refutation in Figure 5.4, containing redundant resolution steps.*

Let v_1, v_2 and v_3 be the vertices for which $cla(v_1) = p\,x_2$, $cla(v_2) = x_1\overline{p}$ and $cla(v_3) = p\overline{x}_3$, and let v_0 be such that $cla(v_0) = x_0$. We obtain $\sigma(v_1) = \{p, \overline{p}, x_2, x_3\}$, $\sigma(v_2) = \{p, \overline{p}, x_1, x_3\}$, and $\sigma(v_3) = \{p, \overline{x}_3\}$ thus $\sigma(v_0) = \{p, x_0\}$. RmPivots *detects repeated resolution on p, applies the substitutions and after proof reconstruction yields the proof on the right of Figure 5.4.*

Our, and the formulation of [FMP11], improves over [Gup12] in the following way: The expansion set in [Gup12] only contains a literal in the phase that was last encountered (top-most). For the previous example this would mean that the expansion set for v_0 only



(a) Non-reduced



(b) Reduced

Figure 5.4: Application of RmPivots

contains $\{x_0\}$ leading to fewer detections of possible substitutions in a potential sub-proof of v_0.

Looking at Example 11 we noticed a further optimization. Given that two children of v_0 get pruned, and only v_3 remains, $\sigma(v_0)$ should contain $\overline{x}_3$ as well. In general, whenever $\sigma(v)$ contains a literal in both of its phases (this can be interpreted as $\sigma(v)$ being subsumed by the tautological clause $\top$), we should propagate that information to all the first ancestors of v with out-degree greater than 1 (or to the initial vertices). This keeps us from taking paths, which get removed, into account when computing σ for ancestors of v. RmSubProof(G, v) takes care of this by transforming the graph G. It does so by removing the sub-proof rooted in $v \in V_G$ until either an initial vertex or an internal vertex with out-degree greater than 1 is encountered.

Definition 16 (RmSubProof)**.** *Let $G = (V_G, E_G, cla_G, piv_G, s_G)$.* RmSubProof$(G, v)$ *with $v \in V_G$ yields G' with $V_{G'} = \{w \mid w \in V_G \wedge (\exists P \in \mathrm{Paths}(w, s_G).v \notin P)\}$, $E_{G'} = \{(v_1, v_2) \mid v_1 \in V_{G'} \wedge (v_1, v_2) \in E_G\}$, $cla_{G'}(v) = cla_G(v)$ and $piv_{G'}(v) = piv_G(v)$ for all $v \in V_{G'}$ and $s_{G'} = s_G$.*

We call the algorithm implementing the RmSubProof optimization RmPivots$_\top$ and give pseudo-code in Figure 5.5b.

5.1.3 More General Clause Subsumption

RmPivots and similar algorithms are very efficient by exploiting proof regularization for reduction. Those algorithms may miss certain substitutions, leading to even smaller proofs, however. RecycleUnits, on the other hand, limits substitutions to unit clauses. Consider the following example where both algorithms miss valid subsumptions.

```
proc RmPivots (R)
  visited ← ∅
  Q ← {s_R}
  while Q is not empty
    v ← oldest element in Q
    Q ← Q \ {v}
    if (v ∉ visited
      and ∀(v,u) ∈ E_R.u ∈ visited)
      visited ← visited ∪ {v}
      σ(v) = ⋂_{(v,w)∈E}(σ(w) ∪ {rlit(v,w)})
      if piv_R(v) ∈ σ_R(v)
        R[v ← v⁻]
      if ¬piv_R(v) ∈ σ_R(v)
        R[v ← v⁺]
      foreach (w,v) ∈ E_R
        Q ← Q ∪ {w}
```

(a) RmPivots

```
proc RmPivots_⊤ (R)
  visited ← ∅
  Q ← {s_R}
  while Q is not empty
    v ← oldest element in Q
    Q ← Q \ {v}
    if (v ∉ visited
      and ∀(v,u) ∈ E_R.u ∈ visited)
      visited ← visited ∪ {v}
      σ(v) = ⋂_{(v,w)∈E}(σ(w) ∪ {rlit(v,w)})
      if piv_R(v) ∈ σ_R(v)
        R[v ← v⁻]
        R ← RmSubProof(R, v⁺)
      if ¬piv_R(v) ∈ σ_R(v)
        R[v ← v⁺]
        R ← RmSubProof(R, v⁻)
      foreach (w,v) ∈ E_R
        Q ← Q ∪ {w}
```

(b) RmPivots$_\top$

Figure 5.5: Single-pass reduction algorithms

Example 12. *Consider the refutation in Figure 5.6. Note that no pivots are eliminated more than once along any of the paths, and none of the unit clauses are valid candidates for substitutions, since their vertices violate the ancestor requirement of Definition 13. Let v be the vertex with $cla(v) = x_1x_3$. Since $\sigma(v) = \{x_1, x_2, x_3, \overline{x}_4\}$, v is subsumed by x_1x_2 (as indicated by $\mapsto$ in the figure).*

Searching for substitutions, which are not detected by RmPivots, is computationally expensive, though. The straight-forward approach leads to checking all pairs of clauses. RecycleUnits circumvents this by only checking unit clauses. The following result allows us to reduce run-time and memory usage in the general case.

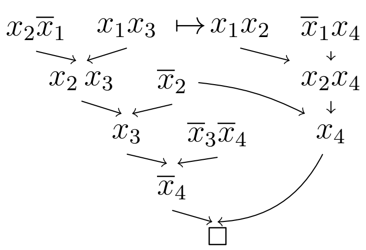


Figure 5.6: Proof, which cannot be reduced by RecycleUnits or RmPivots.

```
if σ(v) ≠ ⊤ and (v is initial or v has out-degree > 1)
   pick u ∈ {w | cla_R(w) ⊆ σ(v)} according to Theorem 5
   R ← R[v ← u]
```

Figure 5.7: General subsumption

5.1.3.1 Limiting the Candidates for Subsumption

We noticed that σ increases monotonically and only at vertices with out-degree greater than 1 might decrease in size (because of the set intersection). Given a chain of vertices with out-degree 1, it is sufficient to check for subsumption at the top-most vertex of this chain.[1] The formalization of our result follows.

Proposition 4. *If v_i dominates v_j then the following subset relations hold:*

$$a) \quad (cla(v_j) \setminus cla(v_i)) \subseteq \sigma(v_j) \qquad and \qquad b) \quad \sigma(v_i) \subseteq \sigma(v_j)$$

Corollary 1. *Let R be a resolution proof, then $cla(v) \subseteq \sigma(v)$ for all $v \in V_R$ that are ancestors of s_R.*

The following corollary is a consequence of Proposition 4 (monotonic growth of σ) and Corollary 1.

Corollary 2. *Let R be a resolution refutation, and let $u_i, v_i \in V_R$ be such that $cla(u_i) \subseteq (cla(v_i) \cup \sigma(v_i))$. Then $cla(u_i) \subseteq \sigma(v_j)$ for any $v_j \in V_R$ dominated by v_i.*

Lemma 3. *Let $v_j \to v_{j+1} \to \ldots \to v_k$ be a path in a refutation R such that all vertices v_i have out-degree 1 and $\mathrm{rlit}(v_i, v_{i+1}) \notin \sigma(v_k)$ (where $j \leq i < k$). Further, let u_k be such that $cla(u_k) \subseteq (cla(v_k) \cup \sigma(v_k))$ and v_j is not an ancestor of u_k. Then applying* ReconstructProof *to $R[v_k \leftarrow u_k]$ or $R[v_j \leftarrow u_k]$ yields the same refutation.*

We apply RmPivots together with the more general search, to establish $\mathrm{rlit}(v_i, v_{i+1}) \notin \sigma(v_k)$. The piece of code shown in Figure 5.7 can be added to RmPivots and RmPivots$_\top$ before the `foreach` loop to search for subsuming clauses.

5.2 Impact of Proof Reduction via Subsumption on Interpolation

Let us now look at the impact of proof reduction via subsumption on the computation of interpolants using the labelled interpolation system (cf. Section 2.3.4.1). In general, the

[1] Such a chain of vertices corresponds to the internal representation of learned clauses in MiniSAT [ES03].

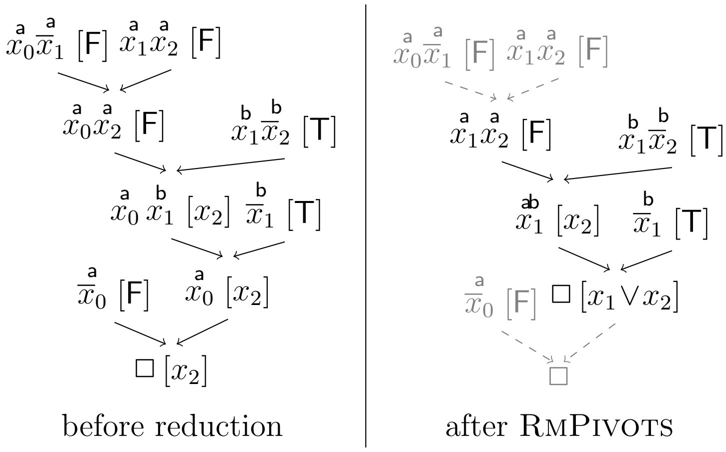


Figure 5.8: Reduced proof size may increase number of variables in interpolant

intuition is that eliminating resolution steps, results in less variables in the interpolant. There are certain cases, where proof reduction, might have a detrimental effect, however. Such cases happen, when local resolutions get eliminated, and introduce non-local ones. Consider the following example.

Example 13. *Consider the refutation R with $(\overline{x}_1), (x_0\,\overline{x}_1), (x_1\,x_2) \in A$ and $(x_1\,\overline{x}_2), (\overline{x}_1) \in B$ on the left of Figure 5.8. We use a labelled interpolation system (Definition 5) with the labelling function L (Definition 4) from Lemma 1. Each vertex is annotated with $cla(v)\,[\mathsf{Itp}(L, R)(v)]$ as described in Section 2.3.4.1, and the label $L(v, t)$ of each literal $t \in cla(v)$ is indicated using a superscript. The shared variable x_1 does not occur in $\mathsf{Itp}(L, R)(s)$, since the literals $\overset{\mathsf{a}}{x}_1$ and $\overset{\mathsf{a}}{\overline{x}}_1$ are* peripheral *(in other words, x_1 is eliminated locally within the A partition).*

We obtain the proof P on the right of Figure 5.8 by applying RmPivots *and* ReconstructProof *to R. P is smaller than R, but the substitution has eliminated a peripheral resolution step and $\mathsf{Itp}(L, P)$ is forced to introduce x_1 when we resolve on $\overset{\mathsf{ab}}{x}_1$ and $\overset{\mathsf{b}}{\overline{x}}_1$.*

Since the labelled interpolation system generalizes other systems, choosing another one would not make a difference. From Lemma 1 we know that the labelling function we chose results in the fewest variables in the interpolant. Therefore, any other labelling would also have to introduce x_1 into the interpolant.

We can address this issue by changing the subsumption condition. By propagating label information in addition to the pivot literals in σ, we can detect substitutions as in Example 13, which introduce variables, and refrain from executing them. We compute the mapping $\varsigma : V \times \mathtt{Lit} \mapsto \mathcal{S}$, containing the necessary information to make that decision, in a similar manner as σ.

$$\varsigma(v, t) = \begin{cases} \perp & \text{if } v = s_R \\ \prod_{(v,w)\in E} \text{litlab}(v, w, t) & \text{otherwise} \end{cases} \tag{5.2}$$

where

$$\text{litlab}(u, v, t) = \begin{cases} L(v^+, \text{var}(t)) \sqcup L(v^-, \overline{\text{var}(t)}) & \text{if } t = \text{rlit}(u, v) \\ \varsigma(v, t) & \text{otherwise} \end{cases}$$

Using this definition we can reformulate Theorem 5 with subsumption lifted to labels as follows:

$$\langle cla(u), L(u)\rangle \preceq \langle \sigma(v), \varsigma(v)\rangle \quad \stackrel{\text{def}}{=} \quad (cla(u) \subseteq \sigma(v)) \wedge (L(u) \sqsubseteq \varsigma(v))$$

Theorem 6. *Let R be an (A, B)-refutation and let σ_R, ς_R be solutions of the Equations 5.1 and 5.2 for R. Let $\pi = \{v_1 \mapsto u_1, \ldots, v_k \mapsto u_k\}$ be a mapping such that for all $1 \leq i \leq j \leq k$ it holds that a) no vertex v_i is an ancestor of u_j, and b) if v_j is an ancestor of u_i then $\langle \sigma_R(u_i), \varsigma_R(u_i)\rangle \preceq \langle \sigma_R(v_i), \varsigma_R(v_i)\rangle$. If $\langle cla_R(u_i), L(u_i)\rangle \preceq \langle \sigma_R(v_i), \varsigma_R(v_i)\rangle$ for $1 \leq i \leq k$, then applying* ReconstructProof *to $R[v_1 \leftarrow u_1] \ldots [v_k \leftarrow u_k]$ yields a proof P such that* $\text{Var}(\mathsf{Itp}(L, P)) \subseteq \text{Var}(\mathsf{Itp}(L, R))$.

We revisit Example 13 with the notion of labelled subsumption.

Example 14. *Let v_1 be the vertex with $cla(v_1) = x_0x_2$ and v_2 the vertex with $cla(v_2) = x_1x_2$ in Figure 5.8 and L as in Example 13. We get $\sigma(v_1) = \{x_0, x_1, x_2\}$ and $\varsigma(v_1) = \{x_0 \to \mathsf{a}, x_1 \to \mathsf{b}, x_2 \to \mathsf{b}\}$. The subsumption check of Theorem 6 now fails: $\langle cla(v_2), L(v_2)\rangle \not\preceq \langle \sigma(v_1), \varsigma(v_1)\rangle$ and the substitution is suppressed. The proof does not change and the interpolant does not increase in size.*

5.3 Implementation and Experimental Results

We implemented RmPivots$_\top$, and Gupta's AllRmPivots [Gup12] for comparison, in a stand-alone tool written in Scala. Unfortunately we were not aware of [FMP11] at the time and did not implement it. The results for that method should lie between Gupta's and our work, as their formulation of the extension set is less restrictive than Gupta's, and they do not seem to employ RmSubProof. The sources of our implementation are available at `https://bitbucket.org/mschlaipfer/proof-minimization` under an MIT license. For an efficient implementation, it is crucial to check the conditions of Theorem 5 and Theorem 6 efficiently. We describe our optimizations in the following:

- We use *watch literals* [MMZ+01] to search for subsumptions. The approach is very similar to the way Boolean constraint propagation is implemented in modern SAT solvers. As we are interested in subsumption, rather than clauses becoming unit, a single watch literal per clause is sufficient for our use case. The benefit of watch literals is that clauses, which do not share any literals with the clause we want to substitute, are not touched by the search at all. This outweighs the cost of additional book keeping. At first, a watch literal is selected in each clause. This literal acts as a pointer to the clause it is contained in. When looking for subsumptions for a clause $cla(v_1)$, the literals in $\sigma(v_1)$ get set to F one by one. When a watch literal gets assigned F, a new watch literal for all the clauses, which had the literal as a watcher need to be found. When for a clause $cla(v_2)$ all literals have been assigned, and it is not possible to find a fresh watch literal, v_2 is a valid substitute for v_1. Note that a subsuming clause contains at most the same amount of literals as σ. Thus at least one assignment must be to a watch literal resulting in detection of a subsumption (if applicable).

Example 15. *Consider the clause at vertex v in Example 12 and let the (partial) watch literal list be as follows.*

watch lit	watched clauses
x_1	x_1x_2, x_1x_3
x_2	x_2x_3
x_3	x_3

The watch literal approach works by setting each literal in $\sigma(v) = \{x_1, x_2, x_3, \overline{x}_4\}$ to F *one by one. After assigning $x_1 = \mathsf{F}$, the watch literal list is as follows (where bold font indicates assignment).*

watch lit	watched clauses
x_1	-
x_2	x_2x_3, $\mathbf{x_1}x_2$
x_3	x_3, $\mathbf{x_1}x_3$

After setting $x_2 = \mathsf{F}$ we end up with the following list.

watch lit	watched clauses
x_1	-
x_2	$\mathbf{x_1x_2}$
x_3	x_3, $\mathbf{x_1}x_3$, $\mathbf{x_2}x_3$

After this step, we detect that the clause x_1x_2 is a valid substitute for v. Notice, that the clause just containing x_3 never had to be touched by the search so far. By continuing the procedure (if we were interested in all the valid substitutes) the clauses watched by x_3 would be detected as a valid candidates as well. Both x_2x_3 and x_3 would be ruled out by the ancestor condition of Theorem 5, avoiding the introduction of a cycle.

- In order not to introduce cycles, we keep track of ancestor information for each vertex. However, we only need to store initial vertices and internal vertices with more than one children. Other clauses are not considered for substitution, and cannot introduce a cycle, as we know from Lemma 3.
- To fulfill the order restriction on substitutions (Theorem 5), we remove ancestor clauses from the watch literal list. That is, when vertex v_1 is substituted for vertex v_2, the ancestor clauses of v_1 get removed from the watch literal list.

5.3.1 Experiments

We used benchmarks from the plain MUS track of the SAT11 competition (58 passing) and single safety property examples from the 2013 Hardware Model Checking Competition (HWMCC) (83 passing), which we obtained by unrolling 10 times. We limited our experiments to resolution refutations with more than 100 vertices that we were able to construct within 1 minute. The largest proof comprised 290888 vertices. The experiments were run on an Intel Xeon E5645 2.40GHz with a 16GB JVM memory limit.

We used two different techniques to obtain resolution refutations from the benchmark SAT instances (cf. Section 2.3.3):

1. The reverse unit propagation approach presented in [GN03] (implemented in OCaml by Georg Weissenbacher and based on MiniSAT 2.2 [ES03])
2. Online proof-logging [ZM03] (implemented in MiniSAT 1.14p)

We present our main results in Table 5.1. We see that our approach yields slightly better proof reduction across the board, due to the less restrictive extension set and the optimization in RmPivots$_\top$. Searching for subsuming clauses does not yield noticeable improvements, however. We attribute this to the limitations encountered, when only doing a single pass over the proof: A single pass implies not knowing about the exact contents of σ after ReconstructProof. We need to be conservative and do not detect certain valid substitutions. Experiments with computing a fix-point (ReconstructProof and recomputation of σ after every substitution) yielded much better results, but this is not applicable to large proofs.

In a partial run of our experiments, we measured the impact of suppressing certain substitutions due to labelling information. For these benchmarks we also measured the size of the interpolant in terms of its AIG representation, using ABC [BM10][2]. We present

[2] We apply the following ABC commands before measuring AIG size, in order to get rid of some redundancy in the interpolant, for a more realistic approximation: `strash; balance; fraig; refactor -z; rewrite -z; fraig;`

	ProofMin			VarMin		
HWMCC	[Gup12]	T0	T10	[Gup12]	T0	T10
proof size (%)	17.66	18.44	18.44	17.66	18.02	18.02
vars (%)	-	-	-	3.44	4.38	4.38
time (s)	1.98	2.03	12.27	0.87	1.01	10.44
MUS	[Gup12]	T0	T10	[Gup12]	T0	T10
proof size (%)	9.57	10.14	10.15	9.57	9.74	9.75
vars (%)	-	-	-	0.73	0.97	0.97
time (s)	2.26	2.52	11.22	0.76	0.95	9.00

(a) Obtained through [GN03].

	ProofMin			VarMin		
HWMCC	[Gup12]	T0	T10	[Gup12]	T0	T10
proof size (%)	10.60	11.44	11.60	10.60	11.32	11.47
vars (%)	-	-	-	0.62	1.44	1.53
time (s)	3.91	4.28	11.21	0.74	0.87	10.10
MUS	[Gup12]	T0	T10	[Gup12]	T0	T10
proof size (%)	7.72	8.26	8.30	7.72	7.94	7.98
vars (%)	-	-	-	0.10	0.14	0.29
time (s)	0.84	1.49	8.12	0.46	0.72	6.22

(b) Obtained through [ZM03].

Table 5.1: We provide results for resolution refutations obtained through [GN03] and online proof-logging [ZM03]. ProofMin denotes experiments with labelling function $L(v,t) = \mathsf{ab}$ for all $v \in V$ (i.e. ς is ignored leading to maximum proof reduction, but an invalid interpolant). VarMin denotes experiments with locality-preserving labelling function (with a random partition (A, B) and averaged over 10 runs). We compare AllRmPivots [Gup12] to RmPivots$_\top$ without (T0) and with (T10) a search for subsumed clauses (limited to at most 10 minutes). Size (%) is the average reduction in proof vertices. Vars (%) is the average reduction in variables in the final interpolant. Time (s) is the average run time.

these results in Table 5.2. While Theorem 6 guarantees that the number of variables does not increase, we noticed an adverse effect of substitution suppression in our experiments. Both, the number of variables and the size of the AIG improve, when not suppressing substitutions because of ς. We ascribe this to not finding substitutions further up the DAG (due to the optimization in RmPivots$_\top$ when doing a substitution) which would be better.

	VarMin	
HWMCC	t0	t0ns
AIG size (%)	24.06	25.92
proof size (%)	18.01	18.44
vars (%)	4.70	5.13
time (s)	1.06	1.02
MUS	t0	t0ns
AIG size (%)	27.33	29.61
proof size (%)	9.75	10.14
vars (%)	1.09	1.14
time (s)	1.05	0.97

Table 5.2: Results for a partial run of our benchmarks, obtained through [GN03]. Displayed are improvements in percent for VarMin (cf. Table 5.1). AIG size improvement is computed from the `ands` value of the `print_stats` command of ABC. t0 denotes RmPivots$_{\top}$ without search for subsumptions and *with* suppression of substitutions due to Theorem 6. t0ns denotes RmPivots$_{\top}$ without search for subsumptions and *without* suppression of substitutions. Note that the results for t0 differ slightly from Table 5.1 because of randomized labelling.

Chapter 6

Conclusion

In this thesis we presented various different approaches for the determinization of Boolean relations. We started with revisiting the theoretical foundations, such as terminology of Boolean logic, BDD and normal form representations of Boolean functions. We described resolution proofs and how they can be used to compute Craig interpolants. We presented various existing techniques for logic minimization and relation determinization—classical as well as contemporary approaches. Building upon this work, we implemented three methods, with the goal to improve circuit size by minimizing the number of input variables the circuit depends on.

Two approaches are based on BDDs and compute the determinization with minimum amount of variables. Our experiments showed that these exact approaches are computationally infeasible. Furthermore, the benchmarks that did not time out did not provide better solutions than the existing approach.

The third approach we implemented, is based on determinization via interpolation. Our goal was to build upon existing resolution proof reduction techniques. On the one hand, we were able to improve the amount of proof reduction of existing techniques. On the other hand, we also looked at the impact of these techniques in terms of interpolant extraction.

6.1 Future Work

We list various ideas which could improve our results.

6.1.1 BDD-based approach

- The implicit search presented in Section 4.4 might be improved by modelling it as a QBF instance. There is a possibility that a QBF solver can solve such an instance more efficiently. There are a couple of steps needed for making such a solution work. The combination network and the relation, which are present as BDDs have to be converted into an appropriate format. Furthermore, the conversion should entail the transformation into CNF via Tseitin's transformation. It might be possible to use ABC in the process to some extent as it supports reading BLIF and writing DIMACS. Additionally, the necessary quantifications must be added.

6.1.2 Interpolation-based approach

- A first step would be to have more sophisticated benchmarks. This means that we would like to run experiments with larger proofs, but also to integrate our approaches with synthesis tools (or model checkers) in order to have the most realistic instances available. We will use Glucose [AS09] and DRAT-trim [WHH14] to produce benchmarks in the future. After learning from our prototype implementation in Scala, it would be good to rewrite the tool in a more performant language, like C or C++ for faster runs and handling of larger benchmarks.

- Both RestoreRes and the general subsumption approach in Figure 5.7 potentially allow for multiple valid substitutions for a vertex. We would like to find a good heuristic for picking one. Right now, we choose the (locally, depending on assignment order of the watch literals) smallest clause, which comes naturally with the watch literal based search. Alternatives would be, to choose the clause which has the smallest sub-proof or the smallest partial interpolant, among others. We have not done sophisticated analysis to decide which one is best, yet.

- We think that most improvement is possible, by propagating more information during analysis. Right now, changes due to ReconstructProof are not considered during RmPivots and the substitutions have to be chosen very carefully due to the conditions of Theorem 5. We would like to get closer to the information available due to fixpoint computation (ReconstructProof after every substitution), without actually performing it. A similar improvement should be made when considering labels (in ς). The information that is used to decide, whether to suppress a substitution or not, is local to the respective resolution step. We see in Table 5.2, that suppressing all resolution steps is not advisable, when trying to reduce the interpolant as much as possible. We do not have a good enough understanding of when to suppress and when to allow substitutions, yet.

- A further improvement to our technique could be to target the elimination of certain variables, without aiming at reduction of the interpolant as such. One approach could be to encode variable dependencies as a SAT instance, where dependency means that certain variables get introduced if another variable gets removed from the final interpolant. A straight-forward approach seems to result in a large instance, though.

- Targetting SMT problems (as arise for example in [HB11]) with our method of interpolant reduction is difficult because of non-uniform proofs, due to the different decision procedures. For QF_UF, however, a method exists to rewrite the proof of the decision procedure [FGG+09] into a propositional proof. This is described briefly in [Mcm08]. After rewriting, the labelling described in Lemma 1 can be applied to a larger portion of the proof. Theory proofs of QF_UF which would need to be considered separately for variable minimization (or not at all) could be brought into the framework of propositional interpolation and minimized using our techniques.

Appendix A

Generalized Reactivity(1) Synthesis

In the appendix we try to put relation determinization into context and introduce Generalized Reactivity(1) (GR(1) for short) synthesis. The benchmarks for the BDD based solutions in Chapter 4 come from GR(1) synthesis. This description is based on [PP06] and [SHB12].

Property synthesis, in general, is a paradigm for constructing correct systems. The idea is to synthesize a system's implementation directly from the specification, rather than to write a program that adheres to the specification separately and to later verify it against the specficiation. Synthesis allows the programmer to stop caring about implementation details, that is *how* a system satisfies the specification, and rather allows to just care about *what* a system's properties must be in the end. GR(1) synthesis is concerned with the synthesis of reactive systems. These systems can be seen as automata with Boolean input variables $\mathcal{I}$ and Boolean output variables $\mathcal{O}$. At every discrete time step an environment provides inputs (i.e. values for $\mathcal{I}$) and the system reacts by computing the output values.

Approaches to synthesizing reactive systems from temporal specifications have been discouraging at first, since LTL synthesis is 2EXPTIME-complete [PR90]. Therefore, in [PP06] the authors suggest to use only a subset of LTL—that is GR(1)—which can be solved in time cubic in the size of the state space. It is claimed that this syntactic restriction of LTL is sufficient to specify most systems (i.e. systems which are compassion-free [PP06]).

GR(1) specifications are of the form $\varphi \equiv \varphi_e \rightarrow \varphi_s$. Each φ_α, where $\alpha \in \{e, s\}$, is a conjunction of:

- φ_α^i: A propositional formula which represents the initial states of the system/environment.
- φ_α^t: A formula which represents the possible transitions of the system/environment.

It is of the form $\bigwedge_i \mathsf{G}(B_i)$, where each B_i is a Boolean combination of variables $(\mathcal{I} \cup \mathcal{O})$ and next state variables expressed as $\mathsf{X}(v)$. If $\alpha = e$, then $v \in \mathcal{I}$, otherwise $v \in (\mathcal{I} \cup \mathcal{O})$.

- φ^g_α: A formula which characterizes the winning condition for the system/environment. It is of the form $\bigwedge_i \mathsf{GF}(B_i)$, where each B_i is a Boolean combination of variables from $(\mathcal{I} \cup \mathcal{O})$.

Solving GR(1) is modelled as deciding the winner of a 2-player game. $\varphi^i_\alpha, \varphi^t_\alpha, \varphi^g_\alpha$ are used to construct a **game structure** (GS). The following definition of the GS sticks to the one provided in [PP06] closely.

Definition 17 (Game structure)**.** *A game structure is a 6-tuple $(\mathcal{I}, \mathcal{O}, \Theta, \rho_e, \rho_s, \varphi)$. $\mathcal{I}$ and $\mathcal{O}$ are sets of Boolean input and respectively output variables of the game structure. The input variables are controlled by the environment, whereas the output variables are controlled by the system. Every minterm of the space spanned by $(\mathcal{I} \cup \mathcal{O})$, is a state of the game structure. The set of all states is denoted by Q. A state is written as (i, o), where $i \in A_\mathcal{I}$ is an assignment to the input and $o \in A_\mathcal{O}$ is an assignment to the output variables. $A_\mathcal{I}$ and $A_\mathcal{O}$ are the sets representing all possible assignments to $\mathcal{I}$ and $\mathcal{O}$, respectively. The initial states of the game structure are characterized by $\Theta \equiv \varphi^i_e \wedge \varphi^i_s$. $\rho_e(\mathcal{I}, \mathcal{O}, \mathcal{I}')$ is the transition relation of the environment. It relates a state $q \in Q$ to possible next input values i'—that is an assignment $i' \in A_\mathcal{I}$. The primed variables are next state variables. Every occurrence of $\mathsf{X}(v)$ is replaced by v' for $v \in (\mathcal{I} \cup \mathcal{O})$. The sets representing these next state variables are $\mathcal{I}'$ and $\mathcal{O}'$, respectively. $\rho_s(\mathcal{I}, \mathcal{O}, \mathcal{I}', \mathcal{O}')$ is the transition relation of the system. It relates a state $q \in Q$ and a next input i' to all possible next outputs o', where $o' \in A_\mathcal{O}$. The transition relations for the environment and system are given by φ^t_α. The winning condition of the game structure is defined as $\varphi \equiv \varphi^g_e \rightarrow \varphi^g_s$.*

For such a game structure, a play σ is defined as a maximal sequence of states $q_0, q_1, \ldots$ such that q_0 satisfies Θ and each state q_k is a successor of q_{k-1} (for $k > 0$). For a pair of states (q_{k-1}, q_k), q_k is a successor of q_{k-1} if $(q_{k-1}, q_k) \in \rho_e \wedge \rho_s$ (that is, there is an edge from q_{k-1} to q_k in the joint transition relation). The game is played as follows: The game starts in an initial state. From there the environment moves by providing a next state input i'. The system reacts to the move by providing a next state output o'. Both moves are supposed to be according to the respective transition relations ρ_α. This procedure advances the play into the next state and the next round begins.

A play σ is winning for the system if it is infinite and every state of σ satisfies the winning condition φ. Otherwise, a play is winning for the environment. The goal of the system is to choose outputs, such that a play is winning for the system. It does so by adhering to its **strategy**. The strategy is a partial Boolean function $f : Q^+ \times A_\mathcal{I} \mapsto$

$A_{\mathcal{O}}$, mapping a finite sequence of states $q_0, \ldots, q_k$, with $k \geq 0$, and an input, provided by the environment, to an output o'. For $(q_k, i') \in \rho_e$ the strategy provides o', where $f(q_0, \ldots, q_k, i') = o'$, such that $(q_k, i', o') \in \rho_s$.

If a strategy makes all the plays starting in initial states of the GS winning for the system, then it is called a **winning strategy**. If there exists a winning strategy, then the game is winning for the system and the system is realizable—the strategy is a working implementation of the system. Otherwise the environment is winning and the system is unrealizable.

A.1 μ-Calculus

The algorithm [PP06] for extracting a strategy from a game structure is given as a μ-calculus [Koz83] formula. The μ-calculus is employed to iteratively compute the set of states from which there exists a winning strategy. The intermediate values of this computation can be used to form a winning strategy.

The μ-calculus over game structures is defined as follows. Let $v \in (\mathcal{I} \cup \mathcal{O})$ be a Boolean variable and $V = \{X, Y, Z_1, Z_2, \ldots\}$ a set of relational variables. A relational variable $X \in V$ can be assigned a set of states $P \subseteq Q$. The BNF defining the syntax of μ-calculus formulas is as follows:

$$\langle\varphi\rangle ::= v \mid \neg v \mid \langle\varphi\rangle \vee \langle\varphi\rangle \mid \langle\varphi\rangle \wedge \langle\varphi\rangle \mid \mu X \langle\varphi\rangle \mid \nu X \langle\varphi\rangle \mid \mathsf{MX}\ \langle\varphi\rangle.$$

A μ-calculus formula φ is interpreted as the set of states, written as $[\![\varphi]\!] \subseteq Q$, where φ is true. Formally, the semantic of μ-calculus formulas is as follows:

$$\begin{aligned}
[\![v]\!] &= \{q \in Q \mid v \models q\} \\
[\![\neg v]\!] &= \{q \in Q \mid v \not\models q\} \\
[\![X]\!] &= X \subseteq Q \\
[\![\varphi \vee \psi]\!] &= [\![\varphi]\!] \cup [\![\psi]\!] \\
[\![\varphi \wedge \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!].
\end{aligned}$$

Let X be a free variable in φ. The notation for assigning a set of states P to X in φ is $[\![\varphi]\!]^{X \leftarrow P}$. Then the two fixpoint operators μ (least fixpoint) and ν (greatest fixpoint) are defined as

$$[\![\mu X \varphi]\!] = \bigcup_i Q_i, \text{ where } Q_0 = \emptyset \text{ and } Q_{i+1} = [\![\varphi]\!]^{X \leftarrow Q_i}$$
$$[\![\nu X \varphi]\!] = \bigcap_i Q_i, \text{ where } Q_0 = Q \text{ and } Q_{i+1} = [\![\varphi]\!]^{X \leftarrow Q_i}.$$

Finally, the authors of [PP06] add a non-standard operator for computation on game structures: The mixed-preimage operator MX. The formal definition of this operator is

$$[\![\mathsf{MX}\ \varphi]\!] = \{q \in Q \mid \forall i'.\ (q, i') \in \rho_e \rightarrow \exists o'.\ (q, i', o') \in \rho_s \text{ and } (i', o') \in [\![\varphi]\!]\}.$$

Informally, the interpretation of this operator is that all states q, for which the system can force the play into $[\![\varphi]\!]$ by choice of o' after the environment has moved by choosing i', are included in $[\![\mathsf{MX}\ \varphi]\!]$. Such states can be considered system-controlled.

A.2 Computation of the Strategy

A μ-calculus formula to solve GR(1) games, that is used compute a strategy, is given in [PP06]. The formula characterizes all states from which there exists a winning strategy for the system, when the winning condition is given as $\varphi \equiv \bigwedge_{i=1}^{m} \mathsf{GF} J_i^A \rightarrow \bigwedge_{j=1}^{n} \mathsf{GF} J_j^G$. Simplified, this condition means: "As long as the environment satisfies the environment assumptions (J_i^A), the system has to fulfill the system guarantees (J_j^G)". The set of states from which there exists a winning strategy is called the winning region, or short Win.

$$\mathsf{Win} = \nu Z \bigwedge_{j=1}^{n} \mu Y \left(\bigvee_{i=1}^{m} \nu X \left(\left(J_j^G \wedge \mathsf{MX}\ Z \right) \vee (\mathsf{MX}\ Y) \vee \left(\neg J_i^A \wedge \mathsf{MX}\ X \right) \right) \right)$$

Notice that the square brackets were dropped for better readability. When implemented, every fixpoint corresponds to a loop. All the intermediate values for X, Y, Z from the loop iterations, are saved and the information is used to construct the strategy.

- X: These are the states, where the environment violates an assumption and the play stays in an X state.
- Y: These are the states, where the system can get closer to satisfying a guarantee.
- Z: These are the states, where a guarantee approach is completed, and the next guarantee to approach is selected.

There are different ways to construct the strategy from these intermediate results: The original approach [PP06] suggests creating three sub-strategies ρ_3, ρ_2 and ρ_1, corresponding to X, Y and Z, respectively. Each sub-strategy is a transition relation containing the valid moves when in a particular state. However, multiple moves might be possible.

In order to compute the final implementation of the circuit the strategy has to be determinized at some point. Determinizing the strategy means that whenever multiple moves for the system are possible, one has to be picked. That is, computing the functional implementation of a Boolean relation, which then can be converted to a combinational circuit (usually a circuit of 2-to-1 multiplexers, see Figure 2.2).

Appendix B

Proofs

We present the proofs for Lemmas 2 and 3, as well as Theorems 5 and 6. The proofs are contributed by Georg Weissenbacher.

Lemma 2. *Let R be a resolution proof, and let $\pi = \{v_1 \mapsto u_1, \ldots, v_k \mapsto u_k\}$ be a mapping such that v_i is not an ancestor of u_j for $1 \leq i, j \leq k$. If $cla_R(u_i) \subseteq cla_R(v_i)$ for $1 \leq i \leq k$, then the proof P obtained by applying* ReconstructProof *to $R[v_1 \leftarrow u_1] \ldots [v_k \leftarrow u_k]$ has sink s_P with $cla_P(s_P) \subseteq cla_R(s_R)$.*

Proof. By induction on the number of ancestors of s_R (cf. the more general proof of Theorem 5) □

Lemma 3. *Let $v_j \to v_{j+1} \to \ldots \to v_k$ be a path in a refutation R such that all vertices v_i have out-degree 1 and $\mathrm{rlit}(v_i, v_{i+1}) \notin \sigma(v_k)$ (where $j \leq i < k$). Further, let u_k be such that $cla(u_k) \subseteq (cla(v_k) \cup \sigma(v_k))$ and v_j is not an ancestor of u_k. Then applying* ReconstructProof *to $R[v_k \leftarrow u_k]$ or $R[v_j \leftarrow u_k]$ yields the same refutation.*

Proof. We consider only the case that v_k is an ancestor of s_R, since v_j and v_k are otherwise not visited by ReconstructProof. Since R is a refutation, $cla(u_k) \subseteq \sigma(v_k)$ (Corollary 1), and therefore $cla(u_k) \subseteq \sigma(v_j)$ (Corollary 2). Since $\mathrm{rlit}(v_{i-1}, v_i) \notin \sigma(v_k)$ for $j < i \leq k$ and $cla(u_k) \subseteq \sigma(v_k)$, we have $\mathrm{rlit}(v_{i-1}, v_i) \notin cla(u_k)$ and $\overline{\mathrm{rlit}(w, v_i)} \in cla(w)$ for $w \neq v_{i-1}$ and $(w, v_i) \in E$. Therefore, RestoreRes propagates vertex u_k until v_k is reached (cf. Definition 14). □

Theorem 5. *Let R be a resolution proof, let σ_R be a solution of Equation 5.1 for R, and let $\pi = \{v_1 \mapsto u_1, \ldots, v_k \mapsto u_k\}$ be a mapping such that for all $1 \leq i \leq j \leq k$ it holds that a) no vertex v_i is an ancestor of u_j, and b) if v_j is an ancestor of u_i then $\sigma_R(u_i) \subseteq \sigma_R(v_i)$. If $cla_R(u_i) \subseteq (cla_R(v_i) \cup \sigma_R(v_i))$ for $1 \leq i \leq k$, then applying* ReconstructProof *to $R[v_1 \leftarrow u_1] \ldots [v_k \leftarrow u_k]$ yields a proof P with sink s_P such that $cla_P(s_P) \subseteq cla_R(s_R)$.*

Proof. Because of condition *a)*, $R[v_1 \leftarrow u_1] \dots [v_k \leftarrow u_k]$ is cycle-free. Otherwise, there must be a substitution $v_j \mapsto u_j$ introducing a cycle through u_j. Since v_j is not an ancestor of u_j in R, the cycle must visit an edge from u_i to a successor of v_i introduced by the substitution $v_i \mapsto u_i$. This is impossible, since v_i is not an ancestor of u_j. Condition *b)* prevents that the substitution $v_i \mapsto u_i$ introduces a path from v_j through a successor of v_i to s_R along which not all literals in $\sigma(v_j)$ are eliminated.

The core of the proof is led by nested structural induction on the number of substitutions and the number of ancestors of s_R:

Outer base case ($\pi = \emptyset$). Applying ReconstructProof to R trivially results in a proof P satisfying that $cla_P(s_P) \subseteq (cla_R(s_R) \cup \sigma_R(s_R))$.

Outer induction step. The outer induction hypothesis is that for every vertex v in $R[v_1 \leftarrow u_1] \dots [v_j \leftarrow u_j]$, the literals in $\sigma_R(v)$ are eliminated along every path from v to the sink, and ReconstructProof yields a proof P with $cla_P(s_P) \subseteq (cla_R(s_R) \cup \sigma_R(s_R))$ if applied to $R[v_1 \leftarrow u_1] \dots [v_j \leftarrow u_j]$.

Inner base case. Assume s_R has no ancestors. If $s_R \neq v_{j+1}$, then $s_P = s_R$ and $cla(s_P) = cla(s_R)$. Otherwise, $s_P = u_{j+1} = \pi(s_R)$, where u_{j+1} is the root of a sub-proof of R such that no v_i is an ancestor of u_{j+1} for $1 \leq i \leq j+1$. Therefore, ReconstructProof leaves u_{j+1} and $cla(u_{j+1})$ unmodified. The premise guarantees that $cla_R(u_{i+1}) \subseteq (cla_R(s_R) \cup \sigma_R(s_R))$, and therefore $cla_R(s_P) \subseteq (cla_R(s_R) \cup \sigma_R(s_R))$. Condition *b)* warrants that the substitutions $\{v_i \mapsto u_i \,|\, j+1 < i \leq k\}$ remain feasible.

Inner induction step. Consider the case that s_R has $n+1$ ancestors. The case where $s_R = v_{j+1}$ is equivalent to the base case above. Therefore, let $s_R \neq v_{j+1}$, and let R^+ and R^- be the sub-proofs rooted at s_R^+ and s_R^-, respectively. Each parent of s_R^+ has at most n ancestors, so by induction applying ReconstructProof to $R^+[v_1 \leftarrow u_1] \dots [v_{j+1} \leftarrow u_{j+1}]$ yields a proof P^+ with sink s_P^+ and $cla_{P^+}(s_P^+) \subseteq (cla_R(s_R^+) \cup \sigma(s_R^+))$, and similarly for R^-. Since RestoreRes(s_R) eliminates the literals $\text{rlit}(s_R^+, s_R)$ and $\text{rlit}(s_R^-, s_R)$, applying RestoreRes to s_R results in a proof P satisfying $cla_P(s_P) \subseteq (cla_R(s_R) \cup \sigma(s_R))$.

Finally, the fact that $\sigma(s_R) = \emptyset$ establishes $cla_P(s_P) \subseteq cla_R(s_R)$. □

Theorem 6. *Let R be an (A, B)-refutation and let σ_R, ς_R be solutions of the Equations 5.1 and 5.2 for R. Let $\pi = \{v_1 \mapsto u_1, \dots, v_k \mapsto u_k\}$ be a mapping such that for all $1 \leq i \leq j \leq k$ it holds that a) no vertex v_i is an ancestor of u_j, and b) if v_j is an ancestor of u_i then $\langle \sigma_R(u_i), \varsigma_R(u_i) \rangle \preceq \langle \sigma_R(v_i), \varsigma_R(v_i) \rangle$. If $\langle cla_R(u_i), L(u_i) \rangle \preceq \langle \sigma_R(v_i), \varsigma_R(v_i) \rangle$ for $1 \leq i \leq k$, then applying* ReconstructProof *to $R[v_1 \leftarrow u_1] \dots [v_k \leftarrow u_k]$ yields a proof P such that* $\mathrm{Var}(\mathsf{Itp}(L, P)) \subseteq \mathrm{Var}(\mathsf{Itp}(L, R))$.

Proof. Given a sub-proof rooted at s_R, applying ReconstructProof yields a sub-proof P such that $cla(s_P) \subseteq \sigma(s_R)$ (by Corollary 1 and the induction hypothesis of the proof in Theorem 5). By lifting the proof of Theorem 5 to $\preceq$, we derive $L(s_P) \sqsubseteq \varsigma(s_R)$. Let s_R be a vertex with ancestors s_R^+ and s_R^-. By induction, $\langle cla(s_P^+), L(s_P^+)\rangle \preceq \langle \sigma(s_R^+), \varsigma(s_R^+)\rangle$, and similarly for s_R^-. Since $\varsigma(s_P^+, piv(s_P)) \sqsubseteq \text{litlab}(s_P^+, s_P, piv(s_P))$ and similarly for s_P^- and $\overline{piv(s_P)}$ (by Equation 5.2), we have $(L(s_P^+, piv(s_P)) \sqcup L(s_P^-, \overline{piv(s_p)})) \sqsubseteq (L(s_R^+, piv(s_R)) \sqcup L(s_R^-, \overline{piv(s_R)}))$, and therefore $\text{Var}(\mathsf{Itp}(R, L)(s_P)) \subseteq \text{Var}(\mathsf{Itp}(R, L)(s_R))$ by Definition 5. $\square$

Bibliography

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978. (Cited on pages 2 and 10.)

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. (Cited on page 56.)

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. (Cited on pages 20 and 68.)

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS*, pages 193–207, 1999. (Cited on page 1.)

[BCG+10] Roderick Paul Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy - a new requirements analysis tool with synthesis. In Springer, editor, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 425 – 429, 2010. (Cited on pages 3 and 52.)

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992. (Cited on page 1.)

[BGJ+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electron. Notes Theor. Comput. Sci.*, 190(4):3–16, November 2007. (Cited on pages 2, 24, 26, 38 and 40.)

[BIFH+09] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Linear-time reductions of resolution proofs. In Hana Chockler and AlanJ. Hu, editors, *Hardware and Software: Verification and Testing*, volume 5394 of *Lecture Notes in Computer Science*, pages 114–128. Springer Berlin Heidelberg, 2009. (Cited on pages 3, 29, 54, 55, 56 and 57.)

[BM10] Robert K. Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *CAV*, pages 24–40, 2010. (Cited on pages 34 and 64.)

[Boo54] George Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. 1854. (Cited on page 11.)

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM. (Cited on page 1.)

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677 –691, aug. 1986. (Cited on pages 10, 13 and 16.)

[BS89] R K Brayton and F Somenzi. An exact minimizer for boolean relations, 1989. (Cited on pages 2 and 31.)

[BSVMH84] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984. (Cited on pages 31 and 34.)

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *Computers, IEEE Transactions on*, 45(9):993 –1002, sep 1996. (Cited on page 16.)

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. (Cited on page 1.)

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement, 2000. (Cited on page 1.)

[Chu62] A. Church. Logic, arithmetic, and automata. In *International Congress of Mathematicans*, 1962. (Cited on page 1.)

[Cra57] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic*, 22(3):269–285, September 1957. (Cited on pages 3 and 20.)

[DKPW10] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *VMCAI*, pages 129–145, 2010. (Cited on page 21.)

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. (Cited on page 19.)

[DM94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill series in electrical and computer engineering: Electronics and VLSI circuits. McGraw-Hill, 1994. (Cited on pages 2 and 31.)

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960. (Cited on page 19.)

[D'S10] Vijay D'Silva. Propositional interpolation and abstract interpretation. In *Proceedings of the 19th European conference on Programming Languages and Systems*, ESOP'10, pages 185–204, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on pages 3 and 22.)

[EB05] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. *Theory and Applications of Satisfiability Testing*, pages 102–104, 2005. (Cited on page 20.)

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003. (Cited on pages 20, 60 and 64.)

[FGG+09] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstić, and Cesare Tinelli. Ground interpolation for the theory of equality. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 413–427, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 69.)

[Flo67] R W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):19–32, 1967. (Cited on page 1.)

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Design Automation. EDAC., Proceedings of the European Conference on*, pages 50 –54, feb 1991. (Cited on page 16.)

[FMP11] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 237–251, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 3, 29, 55, 56, 57 and 62.)

[GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. pages 886–891, 2003. (Cited on pages 20, 64, 65 and 66.)

[Gup12] Ashutosh Gupta. Improved single pass algorithms for resolution proof reduction. In *Proceedings of the 10th International Conference on Automated Technology for Verification and Analysis*, ATVA'12, pages 107–121, Berlin, Heidelberg, 2012. Springer-Verlag. (Cited on pages 3, 29, 55, 56, 57, 62 and 65.)

[HB11] Georg Hofferek and Roderick Paul Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In IEEE, editor, *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MemoCODE 2011)*, pages 31 – 42. IEEE, 2011. (Cited on pages 1, 16 and 69.)

[HGK+13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *FMCAD*, pages 77–84. IEEE, 2013. (Cited on page 1.)

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. (Cited on page 1.)

[HS96] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, 1996. (Cited on page 2.)

[Hua95] Guoxiang Huang. Constructing craig interpolation formulas. In *Proceedings of the First Annual International Conference on Computing and Combinatorics*, COCOON '95, pages 181–190, London, UK, UK, 1995. Springer-Verlag. (Cited on page 21.)

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pages 472 –475, nov 1991. (Cited on page 16.)

[JLH09] Jie-Hong R. Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating functions from large boolean relations. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 779–784, New York, NY, USA, 2009. ACM. (Cited on pages 3, 23, 24, 27, 28 and 29.)

[Kar53] M. Karnaugh. The map method for synthesis of combinational logic circuits. 1953. (Cited on page 6.)

[KMPS10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM. (Cited on page 1.)

[Knu09] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, March 2009. (Cited on page 10.)

[Koz83] D Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. (Cited on page 73.)

[Kra97] Jan Krajícek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symb. Log.*, 62(2):457–486, 1997. (Cited on page 21.)

[KS00] James H. Kukula and Thomas R. Shiple. Building circuits from relations. In *CAV*, pages 113–123, 2000. (Cited on pages 25 and 26.)

[Law64] El Lawler. An approach to multilevel boolean minimization. *Journal of the ACM JACM*, 11(3):283–295, 1964. (Cited on pages 2 and 31.)

[Mcc56] E. J. Mccluskey. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444, November 1956. (Cited on pages 2, 31 and 33.)

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003. (Cited on page 21.)

[Mcm08] K. L. Mcmillan. Quantified invariant generation using an interpolating saturation prover. In *In TACAS*, pages 413–427, 2008. (Cited on page 69.)

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535, 2001. (Cited on pages 3, 19 and 62.)

[MS08] Joao Marques-Silva. Practical applications of boolean satisfiability. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 74–80. IEEE, 2008. (Cited on page 20.)

[PP06] Nir Piterman and Amir Pnueli. Synthesis of reactive(1) designs. In *In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06*, pages 364–380. Springer, 2006. (Cited on pages 1, 2, 71, 72, 73 and 74.)

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, SFCS '90, pages 746–757 vol.2, Washington, DC, USA, 1990. IEEE Computer Society. (Cited on page 71.)

[PS95] Shipra Panda and Fabio Somenzi. Who are the variables in your neighborhood. In *In Int'l Conf. on CAD*, pages 74–77, 1995. (Cited on page 16.)

[PSP96] Shipra Panda, Fabio Somenzi, and Bernard F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. pages 628–631, 1996. (Cited on page 16.)

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations, 1997. (Cited on page 21.)

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982. (Cited on page 1.)

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. (Cited on page 16.)

[Sch] Determinization of boolean relations (draft). `http://www.marshallplan.at/images/papers_scholarship/2012/Schlaipfer.pdf`. Accessed: 2014-04-20. (Cited on page 4.)

[SDGC10] Jocelyn Simmonds, Jessica Davies, Arie Gurfinkel, and Marsha Chechik. Exploiting resolution proofs to speed up LTL vacuity detection for BMC. 12(5):319–335, 2010. (Cited on page 22.)

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM. (Cited on page 1.)

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949. (Cited on page 11.)

[SHB12] Matthias Schlaipfer, Georg Hofferek, and Roderick Bloem. Generalized reactivity(1) synthesis without a monolithic strategy. In Kerstin Eder, JoÃ£o LourenÃ§o, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, volume 7261 of *Lecture Notes in Computer Science*, pages 20–34. Springer Berlin Heidelberg, 2012. (Cited on page 71.)

[Som99] Fabio Somenzi. Binary decision diagrams. In *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999. (Cited on pages 13 and 16.)

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on pages 3 and 19.)

[Tse68] G S Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 8(115-125):234–259, 1968. (Cited on pages 17, 28, 30 and 55.)

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003. 10.1023/A:1022920129859. (Cited on page 1.)

[VOQ52] W. Van Orman Quine. *The Problem of Simplifying Truth Functions*. Mathematical Association of America, 1952. (Cited on pages 2, 31 and 33.)

[WB91] Y Watanabe and R K Brayton. Heuristic minimization of multiple-valued relations, 1991. (Cited on pages 2 and 31.)

[WHH14] N. Wetzler, M.J.H. Heule, and W.A. Hunt, Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs, 2014. Accepted for SAT 2014. (Cited on pages 20 and 68.)

[WM11] Georg Weissenbacher and Sharad Malik. Boolean satisfiability solvers: Techniques and extensions. In T. Nipkow, O. Grumberg, B. Hauptmann, and G. Kalus, editors, *Tools for Analysis and Verification of Software Safety and Security*. IOS Press, 2011. (Cited on page 30.)

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. pages 10880–10885, 2003. (Cited on pages 20, 64 and 65.)