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

June 8, 2012

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

8. Juni 2012

Diese Arbeit ist in englischer Sprache verfasst.

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

Abstract

This thesis presents new ways of solving Boolean relations. Modern approaches to this problem can be divided into those based on binary decision diagrams and those based on satisfiability solving. In this work, both models are explored and enhancements implemented. These enhancements aim at reducing the number of input variables which a function f, solving the relation, depends on. A lower amount of input variables serves the purpose of reducing the size of the circuit implementing f, which in prior solutions has not been satisfactory.

The first approach, based on binary decision diagrams, shows two ways for finding an exact and globally optimal solution for eliminating input variables. Previous methods have found locally optimal solutions only. The second approach is based on satisfiability solving and furthermore Craig interpolation. A particular, pre-existing interpolation system is implemented which provides an efficient way to find the minimum solution for a given resolution proof.

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

Kurzfassung

In dieser Arbeit werden neue Arten präsentiert um Boole'sche Relationen zu lösen. Moderne Ansätze können in jene unterteilt werden, die auf Binären Entscheidungsdiagrammen basieren und jene, die auf Satisfiability-Solvern beruhen. In dieser Arbeit werden beide Modelle untersucht und Erweiterungen implementiert. Diese Erweiterungen zielen darauf ab, die Anzahl der Variablen, von denen eine Funktion f, die die Relation löst, abhängt zu minimieren. Eine niedrigere Anzahl an Eingangsvariablen dient dazu die Größe der Schaltung, die f implementiert, zu reduzieren—eine Eigenschaft, die in vorhergehenden Lösungen nicht zufriedenstellend war.

Der erste Ansatz beruht auf Binären Entscheidungsdiagrammen: Es werden zwei Wege präsentiert um eine exakte und global optimale Lösung für die Minimierung der Eingangsvariablen zu finden. Der zweite Ansatz basiert auf Satisfiability-Solving und weiters Craig-Interpolation. Ein spezielles, existierendes Interpolationssystem wurde implementiert, welches die minimale Lösung für einen gegebenen Resolutionsbeweis effizient findet.

Wir beschreiben die beiden Ansätze im Detail und analysieren die Ergebnise 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

Contents ii

Acknowledgements iii

1 Introduction 1
- 1.1 Organization of this Thesis . . . 4

2 Preliminaries 7
- 2.1 Boolean Logic . . . 7
- 2.2 Boolean Function Representations . . . 11
- 2.3 Determinization of Boolean Relations . . . 23
- 2.4 Satisfiability Solving and Interpolation . . . 25

3 Related Work 35
- 3.1 Combinational Logic Minimization . . . 36
- 3.2 Building Circuits from Relations . . . 40
- 3.3 Extracting Circuits from Relations . . . 41
- 3.4 Interpolating Functions from Large Boolean Relations . . . 42
- 3.5 ABC . . . 46

4 Determinization of Boolean Relations Using BDDs 47
- 4.1 Problem Statement . . . 48
- 4.2 Cofactor Optimization is Sequence-Dependent . . . 49
- 4.3 Explicit Solution . . . 53
- 4.4 Logically Encoded Solution . . . 54
- 4.5 Experimental Results . . . 66

5 Determinization of Boolean Relations Using Interpolants **67**
5.1 Minimum-variable Labelling 68
5.2 Implementation 69
5.3 Experimental Results 71

6 Outlook **75**
6.1 General Trends 75
6.2 Ideas for Future Work 76

7 Concluding Remarks **79**

A Generalized Reactivity(1) Synthesis **81**

Bibliography **87**

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 not only ending my time in Princeton, but also close to the end of my studies towards the Master's degree 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 far 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 making it possible to focus on my studies over the course of the last five years.

Chapter 1

Introduction

" Begin at the beginning and go on till you come to the end; then stop. "

[Lewis Carroll, Alice in Wonderland]

Over the last decade, computers have become increasingly ubiquitous. Every day, we all are in contact with embedded computers in phones, cars, household appliances, etc. Computers have enabled new venues for science and new business opportunities, but also changed our leisure activities and the way we communicate. Programming computers correctly is not an easy task; it 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 most of the faults. By testing, one is unable to say whether the software follows the specification perfectly or not. Formal methods [Flo67, Hoa69, 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] 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 Boolean relations. This 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 property synthesis: The relationship between the two is along the lines of logic synthesis providing solutions to a sub-problem of property synthesis. Preliminary research has shown that the standard solutions from logic synthesis, however, do not perform well in a property synthesis setting. An additional shortcoming of existing techniques is that the produced systems are orders of magnitude larger than manual implementations. As described below, novel techniques are applied to these problems, in the hope of achieving efficient and more concise solutions.

One way of modelling the synthesis process, for example, is based on game theory [PP06]. This model will help to point out where and why relation determinization is necessary. The approach is outlined taking the game-theoretic model as an example. A hardware controller receives inputs and generates outputs. Furthermore, such a system has an internal state (represented by memory registers) which is taken into account when computing the output for a given input (through a combinational circuit).

The game-theoretic model serving as an example is defined as a two-player game between the environment (which provides the inputs) and the system (which computes the outputs). This approach tries to find a strategy for the system that follows the specification and fails for no possible input. In terms of the game that means that for each move of the environment the system can make an advantageous move in order to “win” eventually. This strategy is represented by a Boolean relation allowing multiple choices for an output (a system move),

when given the same state of—and inputs to the system (the same game situation). It is non-deterministic. The reason for this is that a specification can, and most often will, allow multiple solutions for the eventual implementation. In order to build hardware however, one solution has to be picked. In other words, the relation has to be determinized. The difficulty here is that there is a huge number of possibilities and that it is crucial to find a mapping which keeps the produced circuit small in the end. Current approaches do not scale well to bigger problem instances as they are slow and yield systems which are orders of magnitude larger than manual implementations.

Therefore the ideas pursued in this thesis target creating small circuits. The heuristic employed to achieve this is as follows: Given a non-deterministic Boolean input-output relation, it is solved such that its determinization depends on as few input variables as possible. The hope is that a circuit depending on fewer inputs is also smaller. Two different approaches have been tried with this heuristic in mind:

1. New ideas on top of an existing determinization algorithm, have been implemented. This approach is based on binary decision diagrams (BDDs), The existing technique already does some work to reduce the number of input variables. However, it ends up finding a local optimum. Our new approaches allo usw to get an exact and globally optimal solution. To achieve this we present two different approaches:

   (a) An explicit search enumerating combinations of variables one by one.

   (b) An implicit search, for which circuitry is added to the logic representing the relation, in order to enumerate variable combinations.

   We describe these approaches. Our experimental results show however that they are practically infeasible. For the benchmarks which do not timeout, our solution finding a global optimum furthermore seems to provide no improvements over the pre-existing approach which finds a local optimum.

2. The relations are determinized via Craig interpolation [Cra57]. Craig's interpolation theorem states the following:

   Given two Boolean formulas f and g, with $f \wedge g$ unsatisfiable, there exists a Boolean formula i referring only to the common variables of f and g such that $f \rightarrow i$, and $i \wedge g$

is unsatisfiable.

Here, i is the interpolant of f and g. How to apply interpolation to the problem of solving a relation has been shown by Jiang, Lin and Hung [JLH09]. Interpolants can be obtained by annotating resolution refutations produced by Boolean satisfiability (SAT) solvers. Therefore, advantage of the progress made in the development of SAT solvers over the course of the last two decades can be taken.

While interpolation inherently only talks about the shared alphabet of f and g it is possible to tweak the computation of the interpolant in such a way that it depends on a minimum amount of variables for a given resolution refutation. In order to achieve this a certain technique, described in [D'S10], was employed.

An additional upside of the interpolation approach is that it allows synthesis from more expressive specifications. It is possible to use higher-order logics (for example the theory of equality with uninterpreted functions) in the specification. Such satisfiability instances can be solved by a satisfiability modulo theories (SMT) solver. SMT subsumes propositional SAT. Our implementation was done within the OpenSMT [BPST10] solver. Although the approach currently is specific to propositional logic this gives more flexibility in future work.

We describe the idea behind the approach and how it was implemented within OpenSMT. Finally we check how it compares to existing interpolation systems in practice.

1.1 Organization of this Thesis

The thesis is split into four major parts:

1. The theoretical foundations and general terminology is provided in Chapter 2. Topics are Boolean functions and relations, logic represenations such as BDDs and normal forms as well as satisfiability solving and interpolation.

2. Previous work concerned with logic minimization and determinization of Boolean relations which are related to this thesis are discussed in Chapter 3.

3. Two of these works serve as the basis for the experimental contributions of this thesis. The BDD-based solutions are described in Chapter 4 while the approach based on interpolation is explained in Chapter 5.

4. In Chapter 6, general developments in the field are analyzed and ways to improve on the work presented in this thesis are presented. Finally, Chapter 7 concludes the thesis.

Chapter 2

Preliminaries

> *" We are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a great distance, not by virtue of any sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size. "*
>
> [Bernard of Chartres]

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 $\mathbb{B} = \{0, 1\}$. 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.

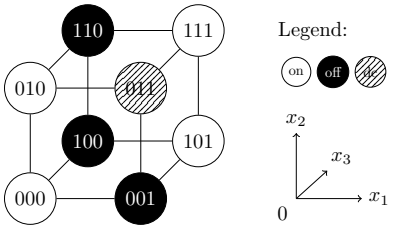
2.1.1 Boolean Functions

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 co-domain as the **output space**, respectively.

Sometimes it is not necessary to completely specify a Boolean function. That is, 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 co-domain. The unmapped minterms of a partial Boolean functions are treated as being mapped to −. Let $\mathbb{B}_+$ be the union of $\mathbb{B}$ and −. 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

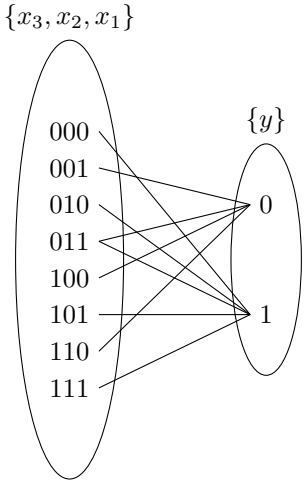
A more expressive way to describe Boolean mappings are **Boolean relations**. A relation can be seen as a set of ordered pairs (x, y), where x is a member of the domain and y is a member of the co-domain. A Boolean relation $R \subseteq X \times Y$ (also written as $R(X, Y)$) is represented by its **characteristic 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 $m > 1$. Most of the time the relations handled in this thesis have $m = 1$, though, as reasoning about such single-output relations makes life easier. Section 2.3.1 presents a scheme for handling multiple-output relations by breaking them down to single-output relations.



(a) Coloring of minterms.

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

(b) Karnaugh map.



(c) Input-output map.

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

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 $\{x \mid \exists y.\ (x, y) \in R\} = \mathbb{B}^n$. 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. The positive literal x represents a completely specificed logic function f, where $f^1 = \{x \mid x = 1\}$. The negative literal $\overline{x}$ represents a function g, where $g^1 = \{x \mid x = 0\}$.

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 function is **positive unate** in x_i if $f_{x_i} \supseteq f_{\overline{x_i}}$ and **negative unate** in x_i if $f_{\overline{x_i}} \supseteq f_{x_i}$.

A function is said to be **unate** if it is unate in all of its variables. A **cube** is a Boolean sub-space with dimension $k \leq n$. 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: For example truth tables, propositional logic, disjunctive normal form, conjunctive normal form, circuit graphs, or binary decision diagrams, to name just some. All representations have certain benefits and drawbacks and their applicability depends on the particular use case. The representations can be converted between each other. This might come at the cost of a jump in representation size, though. The representations most interesting in the course of this thesis are propositional logic, binary decision diagrams (BDDs) and conjunctive, as well as disjunctive normal form. They will be described in this section.

2.2.1 Propositional Logic

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

2.2.1.1 Syntax and Notation

Propositional statements are constructed from a set of propositional symbols (variables) $\mathcal{V} = \{x, x_1, x_2, \ldots, x_n, y, z\}$, the Boolean constants $\{0, 1\}$ and logic connectives $\{\overline{\ }, \cdot, +, \rightarrow, \leftrightarrow\}$. Sometimes, when it brings along better readability, the alternative connectives given in Table 2.1 might be used. The following grammar in Backus-Naur Form provides the rules

Name	Notation	Alternative Notation	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.

for stating well-formed propositional logic formulas (wffs):

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

The symbol $\equiv$ is used to denote logical equivalence. Following this definition, a propositional formula f might, for example, be $f \equiv ((x_1 + (\overline{x_1}) \cdot x_2) \rightarrow ((\overline{x_3}) \rightarrow x_4))$. For brevity the "$\cdot$" connective in between propositional symbols is sometimes dropped. The expression $\overline{x_1} \cdot x_2$ would be shortened to $\overline{x_1} x_2$.

However, the syntactic definition provided might be ambiguous when trying to evaluate a formula. In the example it is unclear in which order $(x_1 + (\overline{x_1}) \cdot x_2)$ is to be evaluated. As usual, parentheses may determine the evaluation order. Parentheses therefore would be sufficient to overcome the problem. However, this approach reduces the readability of propositional formulas immensely. Therefore the binding strength of the connectives is defined. The following precedence rules for the logic connectives allow to keep the number of parentheses low, while maintaining unambiguity:

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

The rule $a \succcurlyeq b$ is read as "a has precedence over b". Moreover, the binary connectives

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

(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: Semantic of the logic connectives.

$\cdot, +, \leftrightarrow$ are left-associative, while $\rightarrow$ is right-associative. The example can now be written, for example, as $f \equiv x_1 + (\overline{x_1} \cdot x_2) \rightarrow (\overline{x_3} \rightarrow x_4)$. Notice, that parentheses might be kept, in order to make formulas even more readable.

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).

2.2.1.3 Quantified Boolean Formulas

Quantified Boolean formulas (QBFs) provide syntactic additions to propositional logic. They are used to formalize and solve certain problems arising with Boolean functions and relations. QBF is furthermore a central topic of computational complexity theory. Whereas Boolean satisfiability (SAT) is the canonical problem of NP, QBF is the canonical problem

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.

of PSPACE. 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 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 is to be represented by a decision tree.

Definition 2 (Decision Tree)**.** *A decision tree 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 $var(v) \in \mathcal{V}$ 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.*

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 $v_1 \in \{v \in V \mid var(v) = x_1\}$ and $v_2 \in \{v \in V \mid var(v) = x_2\}$, the condition $index(v_1) < index(v_2)$ has to hold.

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

Theorem 1 ([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 subfunctions by cofactoring the function. For a non-terminal vertex v, with $var(v) = x_i$ follows from the theorem that the subtree rooted in $low(v)$ represents the function $f_{\overline{x_i}}$ and the subtree rooted in $high(v)$

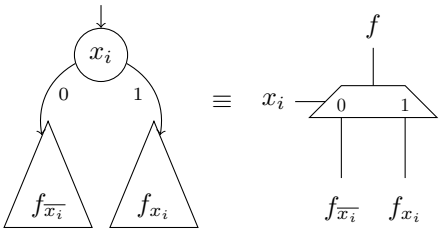


Figure 2.2: Equivalence of a BDD vertex and a 2-to-1 multiplexer.

represents f_{x_i}. The tree rooted in v, therefore, represents f. This is written as a triple $f = (var(v), high(v), low(v)) = (x_i, f_{x_i}, f_{\overline{x_i}})$. 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 node of the tree) can be converted easily into a logically equivalent 2-to-1 multiplexer as is depicted in Figure 2.2.

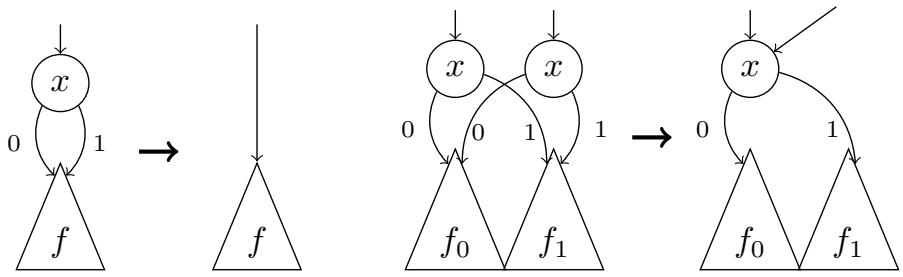
The variable x_i is the decision variable, hence the name of the representation. The tree is constructed by recursive application of Theorem 1, until there are no more variables to cofactor the function with. This procedure inherently leads to $2^{|\mathcal{V}|}$ paths starting in the root node and ending 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.

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



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

Figure 2.3: The two BDD reduction rules.

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 have been 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.

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

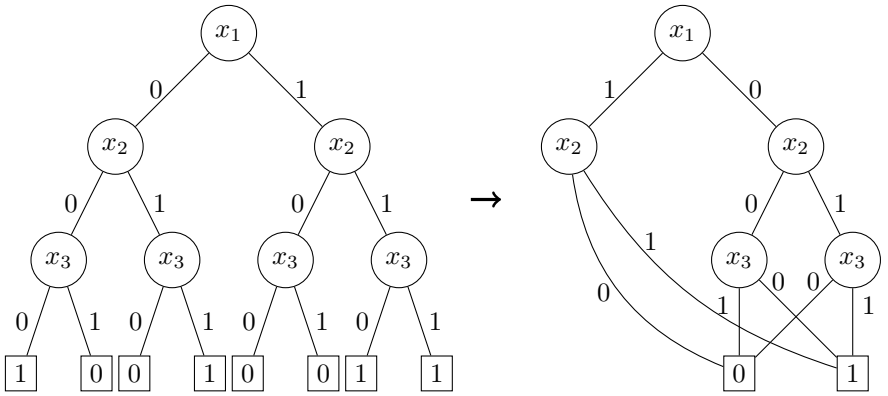


Figure 2.4: Ordered binary decision tree and BDD for the function $f \equiv \overline{x_1}\ \overline{x_2}\ \overline{x_3} + x_1x_2 + x_2x_3$, with variable order $x_1 < x_2 < x_3$.

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 goal is to use the Apply algorithm which is able 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 $var(v) = x$. The children of v are $high(v) = f_x\langle op\rangle g_x$ and $low(v) = f_{\overline{x}}\langle op\rangle g_{\overline{x}}$.

	Operation	*ite* form
0000	0	0
0001	$f \cdot g$	$ite(f, g, 0)$
0010	$f \cdot \overline{g}$	$ite(f, \overline{g}, 0)$
0011	f	f
0100	$\overline{f} \cdot g$	$ite(f, 0, g)$
0101	g	g
0110	$\overline{f} \leftrightarrow g$	$ite(f, \overline{g}, g)$
0111	$f + g$	$ite(f, 1, g)$
1000	$\overline{f} \cdot \overline{g}$	$ite(f, 0, \overline{g})$
1001	$f \leftrightarrow g$	$ite(f, g, \overline{g})$
1010	$\overline{g}$	$ite(g, 0, 1)$
1011	$g \rightarrow f$	$ite(f, 1, \overline{g})$
1100	$\overline{f}$	$ite(f, 0, 1)$
1101	$f \rightarrow g$	$ite(f, g, 1)$
1110	$\overline{f} + \overline{g}$	$ite(f, \overline{g}, 1)$
1111	1	1

Table 2.4: The ite operator.

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.

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

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

In order to compute the combination of two functions, the *ite* operator is used. The lattice

of all Boolean two-argument operators 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 1 provides pseudo-code for Apply, without elaborating on FindOrAddUniqueTable and InsertComputedTable.

Algorithm 1 Apply implementing the construction of a BDD from two BDDs for any two-argument Boolean operator.

```
procedure Apply((f, g, h))
    if Terminal case then
        return result
    else if Computed table has entry (f, g, h) then
        return result
    else
        x ← top_variable((f, g, h))
        f' ← ite(f_x, g_x, h_x)
        g' ← ite(f_x̄, g_x̄, h_x̄)
        if f' = g' then
            return g'
        end if
        R ← FindOrAddUniqueTable(x, f', g')
        InsertComputedTable((f, g, h), R)
        return R
    end if
end procedure
```

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 described how to compute the cofactor with respect to the top variable. If a BDD is cofactored with a cube, the procedure is to compute the cofactor recursively starting in the root node. Then a case distinction is made and if the function is cofactored with a node's variable the incoming edges are rewired to the children of that node. In case there is no need to cofactor, the recursion proceeds along towards the leaves.

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. In [HB11], for example, the authors, 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 with the useful property of being susceptible to the resolution calculus. 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.

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

The disjunctions are referred to as **clauses**. Clauses might also be referred to as sets of literals. Let us assume that clauses are non-tautological—that is they do not contain a literal in both of its phases.

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. Usually (i.e. when applying a SAT or QBF solver) it is sufficient to have an equi-satisfiable CNF formula. The transformation from an arbitrary propositional formula to an equi-satisfiable one can be achieved by Tseitin's transformation [Tse68]. The advantage of this method is that the formula size only grows 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 34). 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 Determinization of Boolean Relations

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—in turn the relation is determinized.

To compute each f_i the multiple-output case is reduced to the single-output case. One scheme accomplishing this is presented in the next section.

2.3.1 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 y. Ways to compute $FI(y, R)$ will be the topic of Chapter 3 (presenting existing work) as well as Chapters 4 and 5 (presenting new work).

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:

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

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

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

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 follows ([JLH09, Formula 2]):

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

2.4 Satisfiability Solving and Interpolation

Boolean satisfiability, or short SAT, is the problem of determining if there is a satisfying assignement to the variables making a propositional logic formula F true. It is the canonical problem of NP and therefore also interesting from a more theoretical standpoint. The focus here will be on practical applications of SAT.

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 form

is the result of applying Tseitin's transformation (cf. Section 2.2.3.1). The reason for the reduction is that CNF allows to apply the resolution calculus to the problem.

2.4.1 Resolution Calculus

In the following, let $\mathcal{C}$ be a set of clauses and $(C + x)$ and $(D + \overline{x})$ be members of that set. The resolution calculus or resolution principle consists of a simple rule which states that if $(C + x)$ and $(D + \overline{x})$ have a satisfying assignment, then so does $(C + D)$. $(C + x)$ and $(D + \overline{x})$ are called **antecedents**. Furthermore, x is called **pivot** variable and $(C + D)$ is called **resolvent**. The pivot variable must be the only variable appearing in opposed phases between the two antecedents.

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

$$\frac{C + x \qquad D + \overline{x}}{C + D} \quad [\textsc{Res}]$$

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

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

Repeated application of the resolution rule yields a resolution proof.

Definition 3 (Resolution proof)**.** *A resolution proof is a directed acyclic graph* (V_R, E_R, cla, piv, s)*.* V_R *represents the proof vertices.* $E_R \subseteq V_R \times V_R$ *is the set of edges. The clause function* $cla : V_R \mapsto \mathcal{C}$ *maps vertices to clauses, and the pivot function* $piv : V_R \mapsto \mathcal{V}$ *maps vertices to variables.* $s \in V_R$ *is the single node of the proof with out-degree 0 and is called the sink. Furthermore, an initial vertex is a vertex with in-degree 0, every other vertex is internal.*

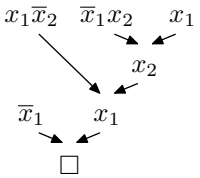


Figure 2.5: Resolution refutation of Equation 2.2.

Let $v, v_1, v_2 \in V_R$, then the edges (v_1, v) and (v_2, v) represent

$$cla(v) = \text{RES}(cla(v_1), cla(v_2), piv(v)).$$

For a node $v_1 \in V_R$ with an edge (v_1, v), let us write v^+ if v_1 contains the pivot in positive phase and v^- if it contains the pivot in negative phase.

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. *Figure 2.5 shows an example of the refutation of*

$$F \equiv (\overline{x_1})(x_1 + \overline{x_2})(\overline{x_1} + x_2)(x_1) \tag{2.2}$$

This example and the accompanying figure are taken from [DKPW10].

2.4.2 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 every leaf of the tree there it is impossible to find a valid assignment, 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 appearing in many domains, much focus has been on 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 important 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.

The following is a list of the developments. These have become standard features implemented in contemporary SAT solvers.

- DP:
  - 1-literal rule: If F contains the unit clause (x), then x is set to 1. All occurrences of $\overline{x}$ are removed from the formula and all clauses containing x are discarded. The rule is commonly called *unit propagation*.
  - Affirmative-negative rule: If x occurs only complemented or uncomplemented all clauses containing x are removed.
- DPLL:
  - Breadth-first search with backtracking: In DPLL the case split of Shannon's expansion was utilized. The search proceeds recursively. A recursion step is the assignment of a truth value to a decision variable x. In case that a *guessed* decision makes the formula evaluate to false the procedure backtracks and the other value is tried in a *forced* (or implied) decision. In order to establish unsat the instance has to be searched exhaustively, whereas for sat the solver can stop after finding a single satisfying assignment.
- Grasp:
  - Implication graph and conflict-driven clause learning: An implication graph stores all the guessed assignments to variables and the implied decisions. The assignments and implied decisions are nodes of the graph. The edges are directed from

the node representing the reason of the implication to the node representing the forced decision. It might happen that the forced decisions lead to a conflict such as a variable y should be assigned to 0 as well as to 1. Such a conflict results in the generation of a conflict clause which is added to the original formula F. The conflict clause is the negation of the conjunction of the assignments leading to the conflict. The purpose of such a clause is to suppress repeating guessed decisions leading to the same conflict. This is an effective way of pruning the search space.

- Non-chronological backtracking: For the application of conflict clauses to make sense it is necessary to backtrack further than just a single variable assignment. By analyzing the implication graph it is possible to determine how far the backtracking should be. This is done by cutting the implication graph and thereby separating the implied and guessed decisions.
- Restarts: Restarts allow to abandon a search tree which requires too much work to be explored. All the conflict clauses remain in the instance, thus leading to more available information after the restart of the procedure.

- Chaff:
  - Unique implication point: This is a strategy for learning conflict clauses which are succinct enough to represent a conflict while making the computation more efficient. A unique implication point is a node in the implication graph which lies on all paths from the decision node to the conflict. The novelty in Chaff is that a conflict clause is only learned at the closest unique implication point to the conflict rather than at every unique implication point.
  - 2-literal-watching: A major contribution of Chaff was the improvement in handling the 1-literal rule. In order for a clause of n literals to become unit, $n-1$ literals must be assigned 0. It is therefore possible to ignore the first $n-2$ assignments completely, and just "watch" any 2 literals of the clause. When a watched literal is assigned 0 one of the remaining unwatched literals becomes watched in that clause, If there is no such literal remaining, then the second watched literal is implied and a forced decision is made. This is implemented by pointers from the literals to the clauses in which they are watched.

  - Activity-based decision heuristics: The most critical impact on the search tree probably comes from the decisions when guessing variable assignments. Therefore, having good heuristics matters. One such heuristic is Variable State Independent Decaying Sum (VSIDS). Variables are ranked by their number of appearances in the formula. High ranking variables are chosen first for guessed decisions. Whenever a conflict clause is added, the variable count is incremented for all variables in the conflict clause. Periodically the counts get divided by a constant. This decay causes a bias of the rank of a variable towards recently added clauses. The effect is that more variables are common between the clauses. This in turn makes it possible to cover the search space more efficiently.

- Further improvements stem from clause subsumption and clause substitution. In [EB05] it is shown that subsumed clauses (that is a clause implied by another one) can be discarded in a preprocessing step. Furthermore, after a resolution step one of the antecedents might be implied by the resolvent and can be strengthened by so-called self-subsumption. Another preprocessing step involves Tseitin's transformation. When applying the transformation, additional variables are introduced together with their respective definitions. The preprocessing step substitutes the variables with their respective definitions, hence eliminating the variable.

The tree resulting from the exploration of the state space can be seen as a dual to a resolution proof. Every node corresponds to a resolution step. The children of the node become the antecedents.

2.4.3 Craig Interpolation

Modern SAT solvers can be used to compute a Craig interpolant. Such an interpolant is described by Craig's interpolation theorem.

Theorem 2 ([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.
2. An interpolation system is employed and each vertex is mapped to a propositional formula called a **partial interpolant**.
3. Next, these annotations of the initial interpolant are propagated towards the sink node (the empty clause) by employing rules defined by the interpolation system (see below).
4. Eventually, the annotation of the sink node is the final interpolant I.

As can be seen, interpolation depends on the respective interpolation system. Basically there exist three 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].

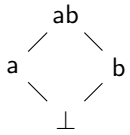
Since the latter system is a generalization subsuming the former two, it is the only system which is described here.

2.4.3.1 Labelled Interpolation System

The description of the labelled interpolation system follows the one in [DKPW10] closely. The authors 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 $\mathcal{L} : V_R \times Lit \mapsto \mathcal{S}$ maps all literals of a resolution proof R to a label from $\mathcal{S}$. For a literal $l \in Lit$ and a vertex $v \in V_R$, $\mathcal{L}$ must satisfy*

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



A variable $var(l)$ is called A-local if it appears only in $vars(A) \setminus vars(B)$, B-local if it appears only in $vars(B) \setminus vars(A)$ and shared otherwise. A labelling function is supposed to preserve locality, meaning that l has to be labelled a if $var(l)$ is A-local and l must be labelled b if $var(l)$ is B-local. For shared variables, any label is allowed.

Given a resolution proof and a labelling function, the labelled interpolation system is defined inductively. Remember that we use a set of literals to represent a disjunction of the literals (a clause).

Definition 5 (Labelled interpolation system)**.** *The following inference rules show how the labelling function is used to map vertices of the resolution proof to partial interpolants (in brackets).*

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

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

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

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

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

The partial interpolant at the sink node is called the final interpolant. Notice, that for internal nodes only the first and third case introduces a literal into the interpolant. This will be of particular interest in Chapter 5.

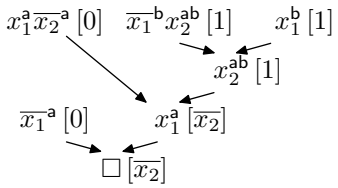


Figure 2.6: Labelled resolution proof annotated with partial interpolants.

Example 2. *Let us continue Example 1 by applying the labelled interpolation system to the resolution proof. F is split into $A \equiv (\overline{x_1})(x_1 + \overline{x_2})$ and $B \equiv (\overline{x_1} + x_2)(x_1)$. 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 2.6*[1]*. In the figure, the labels of the literals are their respective superscripts. The final interpolant I corresponds to $\overline{x_2}$. It can easily be seen that it is a valid interpolant by checking that $A \to I$ and $I \wedge B$ is unsatisfiable.*

[1]Figure adapted from a slide by Georg Weissenbacher.

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:

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

Implication:

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

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

> *" The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65000. I believe that such a large circuit can be built on a single wafer. "*
>
> [Gordon E. Moore]

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. The section presents classic minimization algorithms which tried to find an exact minimum implementation of an incomplete Boolean function. It furthermore gives a short insight into approaches to gain performance by neglecting exactness when applying heuristic algorithms. While Section 3.1 focuses on how to minimize incompletely specified Boolean functions, work extending the solutions to Boolean relations exists and is referenced.

Sections 3.2, 3.3 and 3.4 present more recent approaches for finding functional imple-

mentations from relations. The most obvious difference between the classic and modern approaches is the representation of the logic. While the classic algorithms are limited to more simple representations, the recent approaches can take advantage of developments such as binary decision diagrams, and satisfiability solving combined with interpolation.

The approaches as presented in the original publications take multiple output variables into account. The subsequent sections (with Section 3.2 being an exception) assume a single output variable—a condition which always can be achieved as was shown in Section 2.3.1.

3.1 Combinational Logic Minimization

This section briefly describes 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 product terms (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 difficult. 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.

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: How to perform exact minimization was shown in [BS89] and heuristic minimization was shown in [WB91]. In the following description, we consider minimizing 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*

$$f \equiv x_1x_2\overline{x_3}y + \overline{x_1}x_2x_3y + x_1x_2x_3y + x_1\overline{x_2}x_3y + x_1\overline{x_2}x_3\overline{y}.$$

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, it can be seen that there is an implicant covering the minterms $x_1x_2\overline{x_3}$ and $x_1x_2x_3$ which subsumes α. In Figure 3.1b α is then 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.

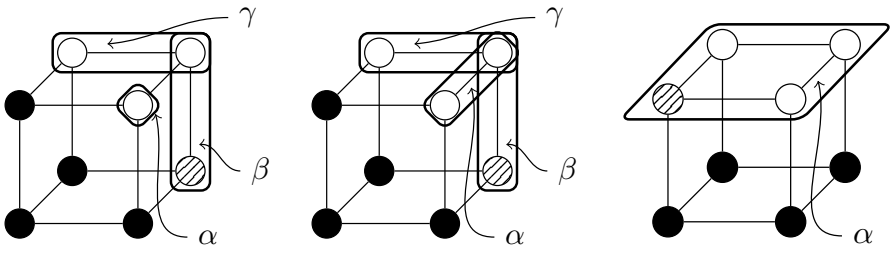


Figure 3.1: Example of covers and implicants. ($\rightarrow$: x_1, $\uparrow$: x_2, $\nearrow$: x_3)

3.1.2 Exact Minimization

The original algorithm for exact minimization of logic circuits is called Quine-McCluskey algorithm because of the concerted effort in finding the solution. The starting point was Quine's Theorem:

Theorem 3 ([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 reduces 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 will not be described here.

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

In practice, heuristic approaches are typically used. Those 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 used are

- EXPAND: This operation expands each implicant by replacing it with the prime implicant containing it.
- 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 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 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 $R_C(\vec{x}, \vec{p}, \vec{y},)$. 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 without going into too much construction detail, let us describe their solution. The approach consists of three phases.

1. In the first phase, information about whether there is a path to the 1 leaf for the current assignment to the input variables, is gathered. This information is represented by the auxiliary output v. This is done by propagating signals from the 1 sink towards the root of the DAG.

2. In the second phase, the signals propagate 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, the information from

Phase 1 is used 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, information along output modules corresponding to the same output variable y_i is collected. If any module chooses 1 for y_i, the final output should be 1—0 otherwise. If none of the modules representing y_i is active, the value is determined 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.

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 2.3, for a functional implementation it would be sufficient, if $R_C(\vec{x}, \vec{p}, \vec{y}, 1) \rightarrow R(\vec{x}, \vec{y})$.

3.3 Extracting Circuits from Relations

This section presents the algorithm given in Figures 2 and 3 of [BGJ+07] with the aforementioned change that it does only take into account a single-output relation. The determinization approach assumes that the relation is given as a BDD. The algorithm was proposed to find a circuit implementation of a strategy for a GR(1) game. 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. Algorithm 2 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 next step of the algorithm is an optional extension. This extension (the for-loop from Line 6 to 13) provides an optimization of the relation. Optimization in these cases means that output y might not depend on all the input variables and elimination of these inputs simplifies the relation. In other words, the inputs which y does not depend on, do not

Algorithm 2 Extraction of a function from a relation.

```
1:  procedure ExtractFunctionFromBDD(R, x⃗, y)
2:      R_1 ← R_y
3:      R_0 ← R_ȳ
4:      R'_1 ← R_1 · ¬R_0
5:      R'_0 ← R_0 · ¬R_1
6:      for all x ∈ x⃗ do
7:          R''_1 ← ∃x. R'_1
8:          R''_0 ← ∃x. R'_0
9:          if R''_1 · R''_0 = 0 then
10:             R'_1 ← R''_1
11:             R'_0 ← R''_0
12:         end if
13:     end for
14:     f ← R'_1
15:     return f
16: end procedure
```

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 existentially quantifies the input x of the current iteration in the strict cofactors R'_1 and R'_0. The resulting expressions represent the sets where x has full freedom. It then checks if these sets overlap. If they do, the input 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 2.3) is finally the strict positive cofactor of R with respect to y.

3.4 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 2.4.3). They show that a single-output total relation $R(\vec{x}, y)$ can be split into two parts ($\overline{R(\vec{x}, 0)}$ and $\overline{R(\vec{x}, 1)}$) and that the conjunction of these parts is unsatisfiable. An unsatisfiable formula in CNF is a necessary precondition for generating a resultation refutation using a SAT solver. Such a proof is then annotated

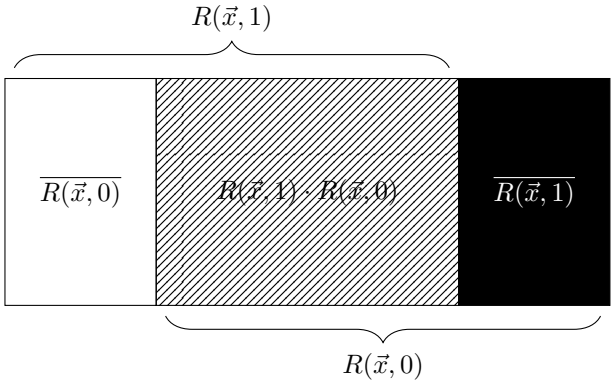


Figure 3.2: The set representation of a single-output total relation $R(\vec{x}, y)$ split into its cofactors.

and used to generate an interpolant (Section 2.4.3). It is shown that the interpolant is a functional implementation of the relation.

Figure 3.2 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 $\overline{R(\vec{x}, 0)}$
2. S_B is the set characterized by $\overline{R(\vec{x}, 1)}$
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 $\overline{R(\vec{x}, 0)}$ and $\overline{R(\vec{x}, 1)}$ 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 4 ([JLH09, Theorem 2])**.** *Given a single-output total relation $R(\vec{x}, y)$, the interpolant I of the refutation of*

$$\overline{R(\vec{x}, 0)} \cdot \overline{R(\vec{x}, 1)} \tag{3.1}$$

with $A = \overline{R(\vec{x}, 0)}$ and $B = \overline{R(\vec{x}, 1)}$, 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 used interpolation system on the other hand. There are two trivial interpolants satisfying Theorem 4 which can be obtained without interpolation, however. These are $R(\vec{x}, 1)$ and $\overline{R(\vec{x}, 0)}$ (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 3.2. The authors of [JLH09] claim that the trivial interpolants often lead to a more complex circuit than the functional implementations computed from a refutation proof.

3.4.1 Interpolant Optimization

An optimization, similar as in Section 3.3 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, combined with an interpolation system allowing for elimination of variables. There is previous work on proof post-processing. Bar-Ilan et al. show in [BIFH$^+$11] how resolution proofs can be reduced in time linear in the size of the proof. They present two approaches for rewriting a proof:

1. Recycle-Units: For every unit clause U (that is a clause consisting of a single literal) the algorithm checks if a clause C in the proof has the unit clause as a pivot. If it does, C gets replaced by U. Therefore, the resolutions for elimination of the literals in

C, besides the pivot, can be spared. The SAT solver may have learned the unit clause after making the "bad" decision, which can be corrected in the post-processing step, now that the unit clause is known.

2. Recycle-Pivots: This algorithm is based on the observation [Urq95] that in a resolution proof there has to be only one resolution on the same pivot on every path from root to sink. Therefore the paths are analyzed from sink to root and a list L of the pivots is kept. As soon as a second resolution on a pivot p already in L is found the resolution step is 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.

   This algorithm assumes that the proof is a tree. In general however, a resolution proof can be a DAG. In [BIFH+11] two solutions are proposed: Firstly, stopping the algorithm as soon as a node with two or more outgoing edges is found. This is the approach the authors chose. Secondly, they propose a more complicated approach which involves an analysis of all paths going through the branching node.

Both Recycle-Units and Recycle-Pivots are part of a two-step process. First the proof is reduced by applying the described techniques and then the proof has to be corrected, such that it is a valid resolution refutation again. Taking Recycle-Units as an example, the proof has to be corrected, since the resolutions on the eliminated literals coming from the subsumed clause C are still existent. After replacement of the subsumed clause these resolutions are superfluous and in fact cannot be made because the pivot is only available in one phase. All the clauses taking part in such resolutions are removed as well and the proof becomes valid again. The algorithm performing these corrections is called Reconstruct-Proof in [BIFH+11].

All these modifications of resolution proofs do not consider interpolation systems at all. A smaller proof, that is one with less resolutions, is still preferable for receiving a smaller interpolant and in turn a simpler functional implementation. The intuition is that less resolutions, lead to a smaller interpolant, since there are less opportunities to introduce variables into the interpolant.

3.5 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 checking and satisfiability solving are provided among others. For a complete list of all 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 minimization techniques.

[1] A sequential circuit is like a combinational circuit, but has memory. Typically these circuits are synchronous following a clock.

Chapter 4

Determinization of Boolean Relations Using BDDs

" After a good dinner one can forgive anybody, even one's own relations. "

[Oscar Wilde]

This chapter presents a contribution of the thesis to the problem of Boolean relation determinization. First, the problem is revisited and the main idea behind our approach is described in the following section. It is then shown that the size of the circuits (functions) computed by Algorithm 2 depends on the order in which the variables are picked by the "optimization loop" (Lines 6-13). Let us in the following refer to this order as variable sequence, not to be confused with the variable order of a BDD. A small example shows two different outcomes for two different variable sequences. As a result of that observation we present two solutions for finding the function depending on the minimum number of variables, independent of the variable sequence. Whereas Algorithm 2 computes a locally optimal solution, the new approaches search for the global optimum.

The first approach employs an explicit search (by enumerating variable subsets) and is described in Section 4.3. The second approach (Section 4.4) adds some logic to the circuit

representing the relation and thereby finds the solution by an implicit search.

Finally, the implementations are evaluated on benchmarks from the field of GR(1) synthesis (cf. Appendix A). The results show, surprisingly, that the local optimum equals the global optimum in these benchmarks. Moreover, as had to be expected, does the search result in a performance penalty compared to Algorithm 2. So far it was only possible to test our methods on small examples, as larger ones result in timeouts.

4.1 Problem Statement

As seen in the previous chapter, there exist various solutions 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.

The approach for attacking this problem assumes 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.

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 that is employed for measuring

the size of a function is the number of variables the function depends on. These variables are called support variables.

The following section illustrates that the extraction algorithm of [BGJ+07], which was presented in Chapter 3 does, in general, not reduce the number of support variables perfectly. This is demonstrated by providing a small example which shows that the result of two runs with different variable sequences differs, where one function is smaller than the other. Subsequently, two extraction algorithms are presented which find an optimal solution.

4.2 Cofactor Optimization is Sequence-Dependent

This section presents an example that illustrates that the analysis of dependent variables (the loop from Line 6 to 13 in Algorithm 2) only finds a locally optimal solution. It applies 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*

$$\begin{aligned} R(\vec{x}, y) \equiv & \overline{x_1}\overline{x_2}\overline{x_3}\overline{y} + x_1\overline{x_2}\overline{x_3}\overline{y} + x_1\overline{x_2}\overline{x_3}y + x_1\overline{x_2}x_3\overline{y} + \\ & x_1\overline{x_2}x_3y + \overline{x_1}\overline{x_2}x_3\overline{y} + \overline{x_1}x_2\overline{x_3}\overline{y} + \overline{x_1}x_2\overline{x_3}y + \\ & x_1x_2\overline{x_3}y + x_1x_2x_3\overline{y} + x_1x_2x_3y + \overline{x_1}x_2x_3\overline{y} \end{aligned}$$

When applying Algorithm 2 on R, $\vec{x}$ and y, the positive cofactor of R with respect to y, R_1, is initialized to

$$R_1 \equiv x_1\overline{x_2}\overline{x_3} + x_1\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1x_2\overline{x_3} + x_1x_2x_3 \equiv x_1 + \overline{x_1}x_2\overline{x_3}.$$

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

$$\begin{aligned} R_0 \equiv & \overline{x_1}x_2\overline{x_3} + x_1\overline{x_2}\overline{x_3} + x_1\overline{x_2}x_3 + \overline{x_1}\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3} + x_1x_2x_3 + \overline{x_1}x_2x_3 \\ \equiv & \overline{x_1} + x_1\overline{x_2} + x_1x_2x_3. \end{aligned}$$

In subsequent steps R_1' is set to $x_1x_2\overline{x_3}$ and R_0' to $\overline{x_1x_2} + \overline{x_1}x_2x_3$.

The for-loop may iterate over the input variables in different sequences. Firstly the example proceeds with an assumed sequence x_1 before x_2 and finally x_3:

Iteration 1. *In the first loop iteration $x = x_1$. R_1'' is computed as $\exists x_1.\ R_1' \equiv x_2\overline{x_3}$ and R_0'' is set to $\exists x_1.\ R_0' \equiv \overline{x_2} + x_2x_3$. As can be seen, the conjunction of R_1'' and R_0'' is unsatisfiable and Line 9 of the algorithm evaluates to true. Therefore, R_1' and R_0' are updated and x_1 is effectively eliminated from the relation.*

Iteration 2. *In the second iteration $x = x_2$. R_1'' is assigned $\exists x_2.\ R_1' \equiv \overline{x_3}$ and R_0'' is assigned $\exists x_2.\ R_0' \equiv 1$. The conjunction yields $\overline{x_3}$ and is satisfiable, therefore Line 9 evaluates to false and no update is made.*

Iteration 3. *The final iteration has $x = x_3$ and R_1'' is $\exists x_3.\ R_1' \equiv x_1x_2$. R_0'' is set to $\exists x_3.\ R_0' \equiv 1$. The conjunction of R_1'' and R_0'' is x_1x_2 and therefore satisfiable. Again, no update is made and the loop is exited.*

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

This loop execution is now compared to an execution with reversed sequence of the input variables, that is x_3 first, followed by x_2 and x_1.

Iteration 1. *The first iteration has $x = x_3$. R_1'' is set to $\exists x_3.\ R_1' \equiv x_1x_2$. R_0'' is set to $\exists x_3\ R_0' \equiv \overline{x_1}$. As $R_0'' \cdot R_1'' \equiv 0$, the if statement is satisfied and the relations are updated. Input variable x_3 is eliminated.*

Iteration 2. *In the second iteration $x = x_2$. R_1'' is assigned $\exists x_2.\ R_1' \equiv x_1$. R_0'' is set to $\exists x_2.\ R_0' \equiv \overline{x_1}$. Again, the conjunction of R_1'' and R_0'' evaluates to 0 and Line 8 evaluates to true. Therefore x_2 is eliminated as well.*

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

(a) Relation R

	x_3	
	0	1
x_2 0	0	0
x_2 1	1	0

(b) $\forall x_1.\ R$

	x_1	
	0	1
x_2 0	0	–
x_2 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 3. *In the final iteration $x = x_1$. R_1'' is the result of $\exists x_1.\ R_1' \equiv 1$ and R_0'' of $\exists x_1.\ R_0' \equiv 1$. Therefore the if block is not entered and 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 Algorithm 2 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 might therefore be desirable.

4.2.1 Independency of Variables

An important notion used in the following approaches for determinizing the relation with the minimum number of support variables is the independency 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, was applied in [BGJ+07] (Algorithm 2).

Proposition 2. *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

The subsequent sections describe methods to find the maximum R-independent set $\vec{x}_{ind}$. When such a set is found, the relation can be determinized, such that it depends on the minimum number of support variables. The functional implementation of R, which depends on the minimum number of input variables can be found by first quantifying out the set of independent variables $\vec{x}_{ind} = \{x_1, \ldots, x_l\}$ universally from the cofactors R_y and $R_{\overline{y}}$. This yields 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 3.2 (Page 43).

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*

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

According to Section 4.2.1, $\{x_1\}$ is an R-independent subset of $\vec{x}$ ($\exists y \forall x_1.\ R(\vec{x}, y) \equiv 1$). It is maximum, since for the subset with higher cardinality (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 having determined that $\{x_1\}$ is the maximum R-independent subset, the relation can be determinized 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 can see that $f_{max} \equiv f_{min} \equiv x_2$ for this simple relation. The example is depicted in Figures 4.2a and 4.2b. Without analyzing for variable independency first,

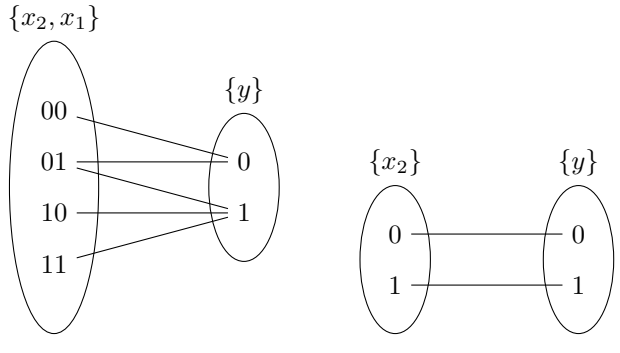
$$f_{max} \equiv R(\vec{x}, 1) \equiv x_1 + x_2$$

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, presented in this thesis, to find a maximum R-independent subset, is an explicit exhaustive search. The 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) $R(\vec{x}, y)$ **(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.

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. This algorithm is called Explicit and Algorithm 3 provides pseudo-code. 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 determinized such that its functional implementation depends on the minimum number of input variables as was described above.

4.4 Logically Encoded Solution

In this approach the goal was to encode the selection of the variable combinations as a combinational circuit. This circuit is capable of generating all combinations of its first k

Algorithm 3 Approach for the incremental computation of the maximum set of R-independent variables.

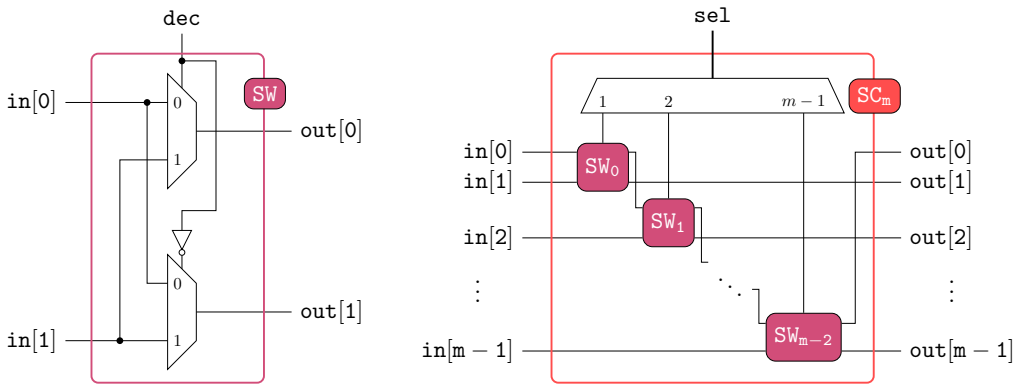
procedure EXPLICIT($R, \vec{x}$)
  $candidates = \vec{x}$
  $n \leftarrow |candidates|$
  $k = 1$
  while $k \leq n$ **do**
    $(candidates, \vec{x}_{ind}) \leftarrow$ INDEPENDENTCOMBINATIONS($candidates, k, R$)
    $k \leftarrow k + 1$
    $n \leftarrow |candidates|$
  end while
  return $\vec{x}_{ind}$
end procedure

inputs at its outputs. It is subsequently called a 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 is to be determinized. 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 independency, 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 $\texttt{CN.in}_1$ at its outputs. The network is comprised of several smaller building blocks, which are called selection cells. The selection cells, again, consist of a decoder and so-called swap cells. These blocks will be described in the subsequent sections.



(a) Swap Cell SW. **(b)** Selection Cell $\mathtt{SC}_m$.

Figure 4.3: 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. The notation $\mathtt{SC}_m$ is used for a selection cell with m inputs ($\mathtt{SC}_m.\mathtt{in}$) and outputs ($\mathtt{SC}_m.\mathtt{out}$). If unnecessary in the current context, the subscript may be dropped.

A selection cell furthermore utilizes $\lceil \log_2 m \rceil$ selector inputs $\mathtt{SC}_m.\mathtt{sel}$. 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 SC.out[0] ← SC.in[i]. The input with index 0 then takes i's place and gets propagated to output position i: SC.out[i] ← 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 ($\texttt{SW.dec}$).[1] The swap cells are connected in the following way:

Case 1. Swap cell $\texttt{SW}_0$ has inputs $\texttt{SC.in}[0]$ and $\texttt{SC.in}[1]$.

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

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

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 $\texttt{SW}_\texttt{i}$ 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 $\texttt{SC.sel}$ signals. A 2-to-4 decoder with inputs $\texttt{SC.sel}[0]$ and $\texttt{SC.sel}[1]$ and outputs $\texttt{SW}_1\texttt{.dec}, \ldots, \texttt{SW}_3\texttt{.dec}$, for example, has the following minterms:

$$\begin{aligned}
\texttt{SW}_1\texttt{.dec} &\equiv \overline{\texttt{SC.sel}[1]} \cdot \texttt{SC.sel}[0], \\
\texttt{SW}_2\texttt{.dec} &\equiv \texttt{SC.sel}[1] \cdot \overline{\texttt{SC.sel}[0]}, \\
\texttt{SW}_3\texttt{.dec} &\equiv \texttt{SC.sel}[1] \cdot \texttt{SC.sel}[0].
\end{aligned}$$

A selection cell as described, comprised of a decoder and swap cells, is depicted in Figure 4.3b. 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* $\texttt{SC}_4$ *is 2 bits wide and allows the choices 0, 1, 2 and 3. Case 0 maps input* $\texttt{SC.in}[0]$ *(abbreviated as* $0_\texttt{i}$*) to output* $\texttt{SC.out}[0]$ *(abbreviated as* $0_\texttt{o}$*). Case 1 maps* $0_\texttt{i}$ *to* $1_\texttt{o}$ *(and* $1_\texttt{i}$ *to* $0_\texttt{o}$*), and so on. There is a choice for the selection signal to map the first input* $0_\texttt{i}$ *to any of the outputs. Table 4.1 shows the input-output mappings for all assignments of* $\texttt{SC}_4\texttt{.sel}$.

	$SC_4.sel$			
	0	1	2	3
0_o	0_i	1_i	2_i	3_i
1_o	1_i	0_i	1_i	1_i
2_o	2_i	2_i	0_i	2_i
3_o	3_i	3_i	3_i	0_i

Table 4.1: Example 6.

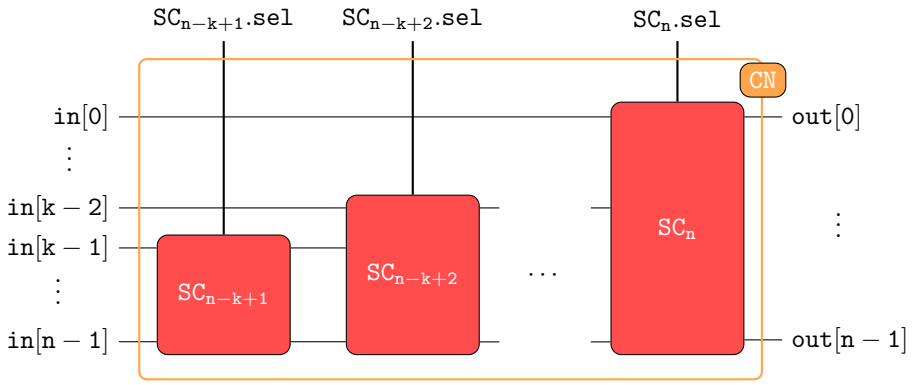


Figure 4.4: A combination network CN.

4.4.1.2 Construction of the Combination Network

Now, that the components of the combination network are defined, they will be used in the construction of the network.

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

Case 1. The first selection cell (in direction of the information flow) is $\texttt{SC}_{\texttt{n-k+1}}$. This cell gets inputs

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

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

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

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

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

[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 4.3a

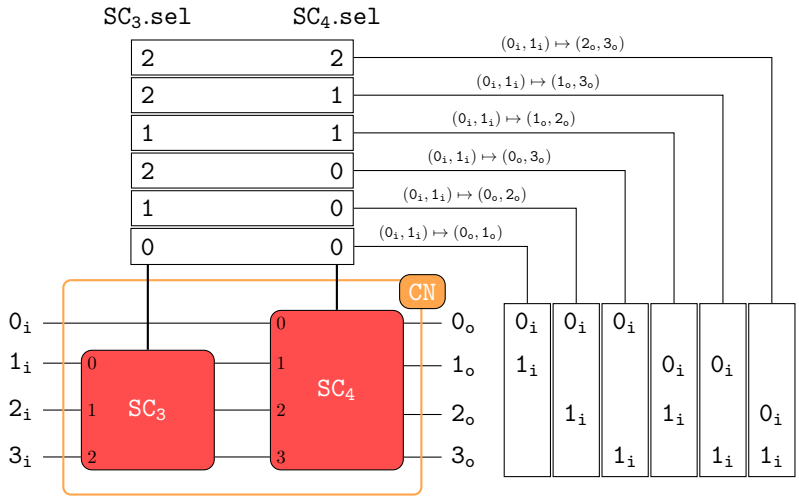


Figure 4.5: Example 7.

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 $\texttt{CN.out[i]} \leftarrow \texttt{SC}_\texttt{n}\texttt{.out[i]}$, for $0 \leq i \leq n-1$. Every output signal CN.out[i] is a Boolean function and the whole combination network is defined by a vector of functions $(\texttt{CN.out[0]}, \ldots, \texttt{CN.out[n} - \texttt{1]})$. A combination network as described above is depicted in Figure 4.4.

4.4.1.3 Mechanics of a Combination Network

The mechanics of a combination network are as follows. A selection cell $\texttt{SC}_\texttt{i}$ is responsible for the output position of input $\texttt{CN.in[n} - \texttt{i]}$. 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:* $\texttt{SC}_3$ *and* $\texttt{SC}_4$*. The functionality of the network is to map the first pair (due to $k = 2$) of inputs (i.e.* $(\texttt{CN.in[0]}, \texttt{CN.in[1]})$*, which is abbreviated as* $(\texttt{0}_\texttt{i}, \texttt{1}_\texttt{i})$*) to any pair of outputs. Analogous*

to the inputs written with a subscript $\mathtt{i}$*, the outputs are sub-scripted with* $\mathtt{o}$*. The* $\binom{4}{2} = 6$ *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 4.5 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.*

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. This observation will find application in the following section, when using a combination network for cofactor optimization. Figure 4.6 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

This subsection explains, how a described 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 $\vec{x} = \{x_0, \ldots, x_{n-1}\}$, first a combination network $\mathtt{CN}$ with $k = \lfloor n/2 \rfloor$ stages is constructed. When taking into consideration the remark of the previous subsection, such a network is generic and capable of producing all combinations of size smaller n. $\mathtt{CN}$ is then connected to R via functional composition. Therefore, first the characteristic func-

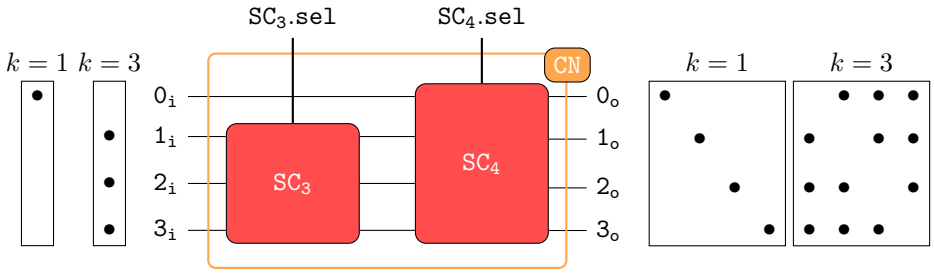


Figure 4.6: Depiction of input and output positions in a combination network with 4 inputs and $k = 1$ and $k = 3$.

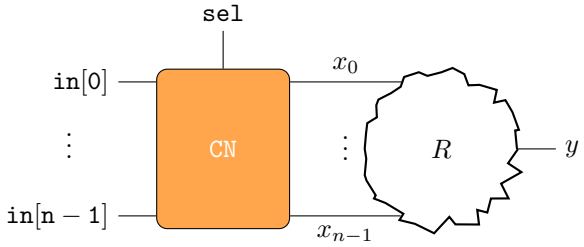


Figure 4.7: Relation $R' = \exists x_0 \cdots \exists x_{n-1}.(\chi_{\texttt{CN}} \cdot R)$.

tion $\chi_{\texttt{CN}} = \bigwedge_{i=0}^{n-1} \texttt{CN.out[i]} \leftrightarrow x_i$ of CN is computed. 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 in input variables CN.in, selector variables CN.sel and an output variable y. Figure 4.7 shows the construction of R'. After constructing R', the approach for finding the maximum R'-independent set of input variables consists of two steps:

1. A criterion, similar to Proposition 2, which says whether a subset of CN.in of size k is R'-independent, is defined.

2. The maximum subset, satisfying the independency criterion, is searched for with binary search.

The criterion for R'-independency 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. $k \leq \lfloor n/2 \rfloor$

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

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

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

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 depends 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 independency.

Now that the criterion is defined, the goal is to find the maximum k for which Proposition 3 is valid. This is done by 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 Algorithm 4 finds the maximum k and a satisfying assignment $\texttt{CN.sel}_0$ 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}, Algorithm 4 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 in $\texttt{CN.sel}_0$ 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 k_{max} is known, it is also known which inputs R' does (or does not) depend on. The R'-independent inputs can be universally quantified out, as in Section 4.2.2.

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

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

$$R_0 = \forall \texttt{CN.in}[0] \cdots \forall \texttt{CN.in}[\texttt{k}_{\texttt{max}} - 1].R'(\texttt{CN.in}, \texttt{CN.sel}_0, 0)$$
$$R_1 = \forall \texttt{CN.in}[0] \cdots \forall \texttt{CN.in}[\texttt{k}_{\texttt{max}} - 1].R'(\texttt{CN.in}, \texttt{CN.sel}_0, 1)$$

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

$$R_0 = \forall \texttt{CN.in}[\texttt{k}_{\texttt{max}}] \cdots \forall \texttt{CN.in}[\texttt{n} - 1].R'(\texttt{CN.in}, \texttt{CN.sel}_0, 0)$$
$$R_1 = \forall \texttt{CN.in}[\texttt{k}_{\texttt{max}}] \cdots \forall \texttt{CN.in}[\texttt{n} - 1].R'(\texttt{CN.in}, \texttt{CN.sel}_0, 1)$$

Both $f_{min} \equiv \overline{R_0}$ and $f_{max} \equiv R_1$ are functional implementations, with the minimum and

Algorithm 4 Binary search for the maximal k and a satisfying assignment to `CN.sel`.

procedure BINARYSEARCH(R', `CN.in`, `CN.sel`, y)
  $k_{max} \leftarrow 0$
  $upper \leftarrow n$
  $lower \leftarrow 0$
  while $lower \leq upper$ **do**
    $k \leftarrow \lfloor (upper + lower)/2 \rfloor$
    $qbf \leftarrow R'$
    $I_1 \leftarrow I_2 \leftarrow \{\}$
    if $k \leq \lfloor n/2 \rfloor$ **then**
      $I_1 \leftarrow \{\mathsf{CN.in[0]}, \ldots, \mathsf{CN.in[k-1]}\}$
      $I_2 \leftarrow \{\mathsf{CN.in[k]}, \ldots, \mathsf{CN.in[n-1]}\}$
    else
      $I_1 \leftarrow \{\mathsf{CN.in[k]}, \ldots, \mathsf{CN.in[n-1]}\}$
      $I_2 \leftarrow \{\mathsf{CN.in[0]}, \ldots, \mathsf{CN.in[k-1]}\}$
    end if

    for all $x \in I_1$ **do**
      $qbf \leftarrow \forall x.qbf$
    end for
    $qbf \leftarrow \exists y.qbf$
    for all $x \in I_2$ **do**
      $qbf \leftarrow \forall x.qbf$
    end for
    $qbf' \leftarrow qbf$
    for all $x \in$ `CN.sel` **do**
      $qbf \leftarrow \exists x.qbf$
    end for

    if $qbf = 1$ **then**
      $\mathtt{CN.sel}_0 \leftarrow$ GETSATASSIGNMENT(qbf')
      $k_{max} \leftarrow k$
      $lower \leftarrow k + 1$
    else
      $upper \leftarrow k - 1$
    end if
  end while
  return $(k_{max}, \mathtt{CN.sel}_0)$
end procedure

maximum on-sets, respectively, of R.

4.5 Experimental Results

The algorithms were implemented 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 3.3. 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 happen as was hoped for.

Furthermore it was observed in our experiments that the heuristic search does not eliminate less variables than our exact methods; at least for the set of GR(1) benchmarks we used. This was not expected and is an interesting result. We do not have a thorough explanation why this happens at the moment.

[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

This chapter presents the second contribution of this thesis. In Section 3.4 it was presented how circuits can be built from relations via interpolation. Various SAT solvers are readily available for this application. Furthermore [D'S10] shows how, given a resolution refutation, the labelled interpolation system (cf. Section 2.4.3.1) can be used to compute an interpolant while introducing a minimum amount of variables into the interpolant. Therefore we implemented this system and used the particular labelling. This labelling is described in Section 5.1.

Many SAT solvers could be used for our implementation. As there is a trend towards the usage of more expressive logics in areas such as model checking or test case generation, but also synthesis, we decided to use a satisfiability modulo theories (SMT) solver. In particular the choice was OpenSMT [BPST10]. SMT solvers provide a superset of the functionality of SAT solvers (hence propositional instances are handled as well). These solvers usually employ a DPLL-style search, but with additional knowledge of the supported theories (this extended algorithm is called DPLL-$\mathcal{T}$). One example of a theory supported by most SMT solvers is the quantifier-free theory of uninterpreted functions and equality (QF_UF). In this

theory, functions are not evaluated. Functional consistency (i.e. if $x = y$ then $f(x) = f(y)$) is checked, though. A problem instance might look like

$$(x = y) \wedge \neg(y = z) \wedge (f(x) = f(z)).$$

In the example every conjunct is a literal. The example is unsat, since by transitivity and the functional consistency constraint $f(x)$ cannot equal $f(z)$. After this short detour into SMT solving, let us now return and only consider propositional resolution proofs again.

OpenSMT so far has the ability to compute a resolution proof and interpolate using the symmetric and McMillan's systems. Building on the existing implementation, the labelled interpolation system was added. The labelled interpolation system can be enabled by setting `proof_set_inter_algo` to 3 in OpenSMT's configuration file.

5.1 Minimum-variable Labelling

Previously a reference was made to a labelling resulting in an interpolant with minimum amount of variables. This particular labelling function was described in [D'S10, Section 5.2] and is based on the results of [DKPW10].

Let us consider a refutation of a CNF partitioned in $A \wedge B$. Remember from Section 2.4.3 that a labelling is completely defined by labelling the initial vertices of the proof. According to the labelled interpolation system, every literal l with $var(l)$ A-local (B-local) has to be labelled a (b). Therefore the only literals for which there is freedom of choice between a, b and ab are the ones where the corresponding variable appears in both partitions. The minimum-variable labelling results from the labelling function mapping a shared literal l to a for all occurrences of l in clauses of A and to b for all occurrences in clauses of B.

The proof of [D'S10, Lemma 4] shows that this labelling is indeed minimal. It is along the lines of showing that every variable added to the interpolant by this labelling would be added by any other labelling as well.

5.2 Implementation

This section documents the implementation of the labelled interpolation system within OpenSMT. As OpenSMT is a vast software, only a small portion of the code had to be touched. Subsequently the parts of OpenSMT relevant to our work such as proof generation and interpolation, are discussed. Afterwards our additions to the code are explained. It is hoped that this will make the entry hurdle for newbies, who want to make changes in this area, smaller.

So far, the previously explained labelling is hard-coded and it might be interesting to allow flexibility to support every valid labelling function. There are some difficulties, though. The final SAT instance is only known at runtime, since it might need to be transformed into CNF first. The best bet might be to apply Tseitin's transformation before starting the solver and define an additional input format which specifies the labelling function.

5.2.1 OpenSMT and Interpolant Generation

OpenSMT is a modern open source SMT solver, implementing DPLL-$\mathcal{T}$. It is written in C++ and based on miniSAT 2.0 [ES03]. More information can be found at `http://verify.inf.unisi.ch/opensmt` or in [BPST10]. Important for us was that OpenSMT supports proof generation and interpolation which gave us a foundation to build on. Furthermore SMT-LIB 2 support was a criterion for our choice. We are unaware of other open source SMT solvers satisfying these requirements.

The following describes briefly what happens internally when generating an interpolant. After parsing the instance and building the internal Egraph data structure representing it, the proof is generated by executing the DPLL-$\mathcal{T}$ procedure. Two classes are involved primarily in proof construction:

- `CoreSMTSolver` implements the DPLL-$\mathcal{T}$ procedure.
- `Proof` provides methods and a data structure for storing the proof.

This data structure is called `ProofDer`. It is a compacted representation of the proof which

does not store intermediate resolvents, but stores a chain of pivots and the initial clauses instead. This information suffices to infer an internal clause.

From this data structure the proof graph is generated. It is stored in the class `ProofGraph` as a vector of `ProofNodes`. `ProofGraph` provides a variety of functionality related to proof computation and interpolation. It is constructed in `ProofGraph::buildProofGraph`. OpenSMT differentiates between four types of clauses when building the proof graph:

1. `CLA_ORIG` are initial clauses of the proof in $A \wedge B$.
2. Deduced clauses are internal clauses of $A \wedge B$.
3. `CLA_LEARNT` are clauses which are derived from an implication graph during the execution of DPLL-$\mathcal{T}$.
4. `CLA_THEORY` are clauses which provide an explanation from a theory solver.

The former three types are of interest to us. For a node of one of these types, clause, pivot and (if applicable) antecedents are saved in a `ProofNode` object. For `CLA_ORIG` clauses, information of partition affiliation is stored as well. In the next step the proof graph is topologically sorted such that antecedents appear before resolvents, in order to facilitate iterative computation of the interpolant.

The existing interpolation systems differ from the labelled system: The partial interpolant at initial vertices (base case) depends only on the partition affiliation of the clause (and of course the clause itself). No information of the partition affiliation of variables is necessary.

For internal vertices, the essential information is global as well, as opposed to the labelled system: The case distinction is made on the partition affiliation of the pivot variable, whereas in the labelled system the computation depends on the label resulting from the join of the pivot's labels. This global information can be retrieved from the Egraph. Central procedures involved in the generation of interpolants are `ProofGraph::produceSequenceInterpolants` and `ProofGraph::setPartialInterp`. The eventual computation of the partial interpolants is done in the `ProofGraph::setInterp(Pudlak | McMillan | McMillanPrime)(Leaf | NonLeaf)` methods.

5.2.2 Our Additions

In order to add the labelled interpolation system, changes were made in the `ProofGraph` and `ProofNode` code sections, primarily. The lattice of labels (cf. Section 2.4.3.1) was added, with a class `LabelColor` taking care of the lattice operations. Furthermore, as mentioned, previously no local label information was needed. A map from clause literals to members of the lattice was implemented in `ProofNode` to serve this purpose. In `ProofGraph::buildProofGraph` initial vertices are labelled. Currently, the labelling is hard-coded as to follow the minimum-variable labelling which was described earlier. After the complete proof graph is built, the labels are propagated down the proof in `ProofGraph::colorProofLiterals`. Finally, given the whole labelling, it is possible to compute the partial interpolants. Analogous to the existing interpolation systems this is done in `ProofGraph::setInterpLabelledLeaf` and `ProofGraph::setInterpLabelledNonLeaf`.

5.3 Experimental Results

To see how much better the minimum-variable labelling performs in practice compared to the other three interpolation systems (McMillan, McMillan inverse, Symmetric), in terms of number of variables in the final interpolant, some experiments were performed. The benchmarks were taken from `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`. The unsat instances which were used in the experiments are from the AIM and JNH benchmark sets. These are easy artificial benchmarks, which can be solved quickly with proofs of reasonable size. The partitioning into A and B was done such that no interpolant is 0: The size of the A partition was set to be 5% of the clauses for the AIM benchmarks and 20% of the clauses for the JNH benchmarks.

The results are provided numerically in Table 5.2 and graphically in Figure 5.1. The numeric values from top to bottom are represented from left to right in the figure. It is observed that in the JNH benchmark set, the McMillan interpolation system performs comparably to the labelled system. In the AIM benchmarks, it is the inverse McMillan system which introduces a similar number of variables. The symmetric system performs considerably worse in both cases.

	mcmillan_inv	mcmillan	symmetric
JNH	0.59	0.97	0.60
AIM	0.85	0.28	0.33

Table 5.1: Average percentage of variables introduced into the final interpolant by the labelled system comparing it to the other interpolation systems for both benchmarks.

Table 5.1 provides a percentage comparison of the labelled interpolation system to all the other systems. The figures in the table show the average percentage of variables introduced into the final interpolant by the labelled system with minimum-variable labelling compared to the other systems for each set of benchmarks.

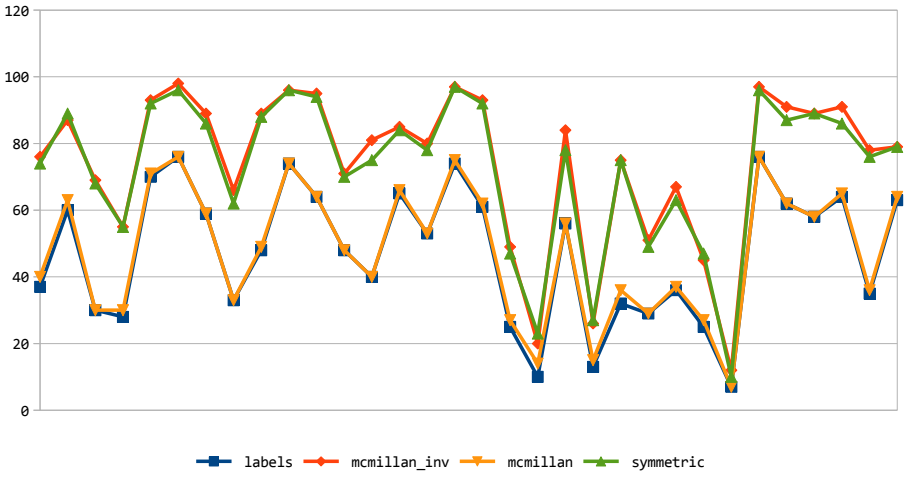
	S1	S2	S3	S4
jnh10	37	76	40	74
jnh11	60	87	63	89
jnh13	30	69	30	68
jnh14	28	55	30	55
jnh15	70	93	71	92
jnh18	76	98	76	96
jnh19	59	89	59	86
jnh202	33	66	33	62
jnh203	48	89	49	88
jnh206	74	96	74	96
jnh208	64	95	64	94
jnh20	48	71	48	70
jnh211	40	81	40	75
jnh214	65	85	66	84
jnh215	53	80	53	78
jnh216	74	97	75	97
jnh219	61	93	62	92
jnh2	25	49	27	47
jnh302	10	20	14	23
jnh303	56	84	56	78
jnh304	13	26	15	27
jnh305	32	75	36	75
jnh307	29	51	29	49
jnh308	36	67	37	63
jnh309	25	45	27	47
jnh310	7	12	7	10
jnh3	76	97	76	96
jnh4	62	91	62	87
jnh5	58	89	58	89
jnh6	64	91	65	86
jnh8	35	78	36	76
jnh9	63	79	64	79

(a) JNH

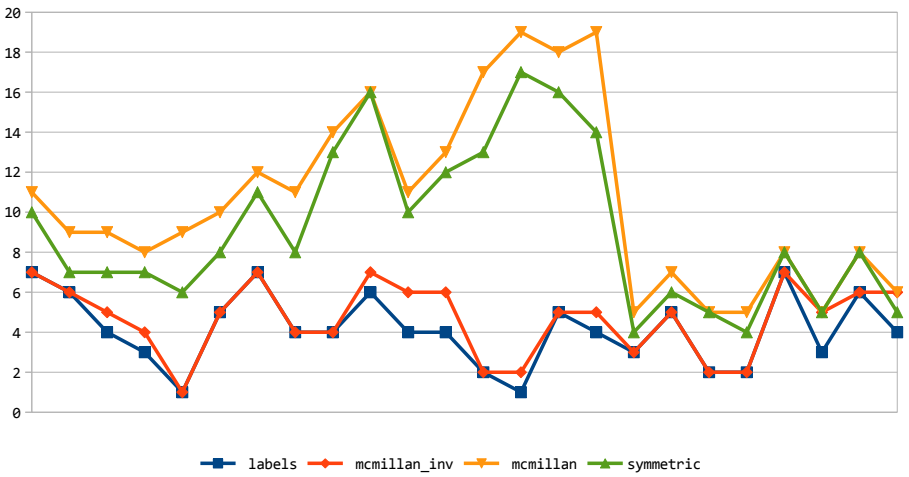
	S1	S2	S3	S4
aim-100-1_6-no-1	7	7	11	10
aim-100-1_6-no-2	6	6	9	7
aim-100-1_6-no-3	4	5	9	7
aim-100-1_6-no-4	3	4	8	7
aim-100-2_0-no-1	1	1	9	6
aim-100-2_0-no-2	5	5	10	8
aim-100-2_0-no-3	7	7	12	11
aim-100-2_0-no-4	4	4	11	8
aim-200-1_6-no-1	4	4	14	13
aim-200-1_6-no-2	6	7	16	16
aim-200-1_6-no-3	4	6	11	10
aim-200-1_6-no-4	4	6	13	12
aim-200-2_0-no-1	2	2	17	13
aim-200-2_0-no-2	1	2	19	17
aim-200-2_0-no-3	5	5	18	16
aim-200-2_0-no-4	4	5	19	14
aim-50-1_6-no-1	3	3	5	4
aim-50-1_6-no-2	5	5	7	6
aim-50-1_6-no-3	2	2	5	5
aim-50-1_6-no-4	2	2	5	4
aim-50-2_0-no-1	7	7	8	8
aim-50-2_0-no-2	3	5	5	5
aim-50-2_0-no-3	6	6	8	8
aim-50-2_0-no-4	4	6	6	5

(b) AIM

Table 5.2: # of variables in the final interpolant for each benchmark. S1=labels, S2=mcmillan_inv, S3=mcmillan, S4=symmetric



(a) JNH



(b) AIM

Figure 5.1: # of variables in the final interpolant (y) for each benchmark (x).

Chapter 6

Outlook

" Prediction is very difficult, especially if it's about the future. "

[Niels Bohr]

This chapter gives a brief overview of general trends in the field. Furthermore it highlights ways of improving upon the approaches presented in this thesis.

6.1 General Trends

The most obvious trend in the field is certainly the change from classical Boolean logic to higher-order logics. Beginning with the improvements in propositional SAT solving around the turn of the century [SS96, MMZ+01], as well as a rising number of decision procedures for higher-order logics, satisfiability modulo theories (SMT) has gained in importance. These expressive logics simplify the modeling of certain problems and advanced tools like z3 [dMB08] add to the usefulness of these logics. Therefore SMT has found entrance into practical tools concerned with software correctness. [BM09] provides an overview of the opportunities and future challenges for applications of SMT: The list of applications includes symbolic execution, model checking and static analysis of programs.

SMT also facilitates modelling synthesis problems. [HB11] presents an approach of synthesizing a pipeline controller from a specification in the quantifier-free theory of arrays. The approach eventually leads to a non-deterministic relation and in order to compute the system's implementation it has to be determinized. Applying [JLH09, Theorem 2] is also possible when the relation is in a more expressive logic. Therefore the determinization technique of Jiang, Lin and Hung can be used here as well.

6.2 Ideas for Future Work

There are various ways which might improve our results.

- 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.

- Regarding the interpolation-based approach, the ground work was done for future improvements in this area. It might be of use to support any labelling function instead of the current approach of fixing one particular labelling.

- In order to decrease the number of variables in the final interpolant further, the goal is to try rewriting the resolution proofs such that more local resolutions are possible (cases where no variable is introduced). The idea is to rewrite the proof similar as in [BIFH+11] (described in Section 3.4.1). Instead of decreasing the overall proof size the goal is to decrease the number of introduced variables when using the minimum-variable labelling.

- Furthermore, for QF_UF a method exists to rewrite the proof of the theory solver (the decision procedure is described in [FGG+09]) into a propositional proof. This is

described briefly in [Mcm08]. After rewriting, the minimum-variable labelling 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) can be brought into the framework of propositional interpolation.

Chapter 7

Concluding Remarks

" Finally, in conclusion, let me say just this. "

[Peter Sellers]

In this thesis various different approaches for the determinization of Boolean relations have been presented. The theoretical foundations, such as terminology of Boolean logic, BDD and normal form representations and SAT solving together with interpolation were discussed. Various different existing techniques for logic minimization and determinization have been introduced—classical as well as contemporary approaches. Building upon this work, three approaches were implemented, with the goal to improve circuit size. Two approaches are based on BDDs and are able to compute the determinization with minimum amount of variables. From our experiments it was observed that these exact approaches are computationally infeasible. Furthermore, the benchmarks that did not time out did not provide better solutions than the existing approach.

We also implemented an approach based on interpolation which uses a specific labelling function. This approach provides the minimum-variable solution for a given resolution proof. The interpolation system we implemented was compared to the existing interpolation systems.

The implementation lays groundwork for improvements that are to be implemented in the future. On the one hand, the goal is to include theory proofs into the propositional part of the proof. On the other hand, it might be possible to rewrite the proofs with the particular interpolation system in mind, such that even less variables are introduced into the interpolant and therefore into the functional implementation.

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.

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 to get rid of 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^i_α: A propositional formula which represents the initial states of the system/environment.
- φ^t_α: 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 13 (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.0.1 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

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

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.0.2 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.

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

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).

Bibliography

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978. (Cited on page 14.)

[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 page 66.)

[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 25, 41, 49 and 51.)

[BIFH+11] Omer Bar-Ilan, Oded Fuhrmann, Shlomo Hoory, Ohad Shacham, and Ofer Strichman. Reducing the size of resolution proofs in linear time. *International*

Journal on Software Tools for Technology Transfer (STTT), 13:263–272, 2011. 10.1007/s10009-010-0167-5. (Cited on pages 44, 45 and 76.)

[BM09] Nikolaj Bjørner and Leonardo De Moura. Z3 10: Applications, enablers, challenges and directions. 2009. (Cited on page 75.)

[BM10] Robert K. Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *CAV*, pages 24–40, 2010. (Cited on page 46.)

[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 15.)

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015, pages 150–153, Paphos, Cyprus, 2010. Springer, Springer. (Cited on pages 4, 67 and 69.)

[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 14, 18 and 21.)

[BS89] R K Brayton and F Somenzi. An exact minimizer for boolean relations, 1989. (Cited on pages 2 and 36.)

[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 21.)

[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 2.)

[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 30.)

[DKPW10] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *VMCAI*, pages 129–145, 2010. (Cited on pages 27, 31 and 68.)

[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 28.)

[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 36.)

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. (Cited on page 75.)

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960. (Cited on page 28.)

[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 4, 67 and 68.)

[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 30.)

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003. (Cited on page 69.)

[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 76.)

[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 22.)

[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 2, 22 and 76.)

[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 31.)

[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 22.)

[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 4, 24, 25, 42, 43, 44 and 76.)

[Kar53] M. Karnaugh. The map method for synthesis of combinational logic circuits. 1953. (Cited on page 8.)

[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 2.)

[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 14.)

[Koz83] D Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. (Cited on page 83.)

[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 31.)

[KS00] James H. Kukula and Thomas R. Shiple. Building circuits from relations. In *CAV*, pages 113–123, 2000. (Cited on pages 40 and 41.)

[Law64] El Lawler. An approach to multilevel boolean minimization. *Journal of the ACM JACM*, 11(3):283–295, 1964. (Cited on pages 2 and 36.)

[Mcc56] E. J. Mccluskey. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444, November 1956. (Cited on pages 2, 36 and 39.)

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003. (Cited on page 31.)

[Mcm08] K. L. Mcmillan. Quantified invariant generation using an interpolating saturation prover. In *In TACAS*, pages 413–427, 2008. (Cited on page 77.)

[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 28 and 75.)

[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 2, 81, 82, 83, 84 and 85.)

[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 81.)

[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 22.)

[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 22.)

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations, 1997. (Cited on page 31.)

[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 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 2.)

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949. (Cited on page 15.)

[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 18 and 21.)

[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 28 and 75.)

[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 23 and 34.)

[Urq95] Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1:425–467, 1995. (Cited on page 45.)

[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, 36 and 38.)

[WB91] Y Watanabe and R K Brayton. Heuristic minimization of multiple-valued relations, 1991. (Cited on pages 2 and 36.)

[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 34.)