Karlsruhe Institute of Technology

Recognition and Exploitation of Gate Structure in SAT Solving

to obtain the academic degree of a
Doktor der Naturwissenschaften

KIT Department of Informatics
Karlsruhe Institute of Technology (KIT)

Dissertation of

Markus Iser
from Kandel

Date of Oral Examination:	2020-02-07
First Reviewer:	Prof. Dr. Carsten Sinz Karlsruhe Institute of Technology (KIT)
Second Reviewer:	Prof. Dr. Laurent Simon Bordeaux Institute of Technology (INP)

Erkennung und Nutzung von Gate Struktur beim SAT Solving

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation von

Markus Iser

Tag der mündlichen Prüfung:	2020-02-07
Erster Gutachter:	Prof. Dr. Carsten Sinz Karlsruhe Institute of Technology (KIT)
Zweiter Gutachter:	Prof. Dr. Laurent Simon Bordeaux Institute of Technology (INP)

Abstract

In theoretical computer science, the SAT problem is the archetypal representative of the class of NP-complete problems for which the widely accepted conjecture holds, that there exists no poly-time algorithm for solving the problem, and thus SAT solving is generally considered intractable [41]. However, SAT algorithms are an active area of research, as exciting results can be found in practical SAT solving, where some applications generate problems with millions of variables, that can be solved in a reasonable time by recent solvers.

Background.

The success of practical SAT solving is due to state-of-the-art implementations of the Conflict Driven Clause-Learning (CDCL) algorithm. The performance of CDCL largely depends on the employed heuristics, e.g., for branching, learning and forgetting. Such heuristics implicitly exploit the structure of instances generated in industrial practice [137]. Formalizations of structure in SAT instances include the structure of their graph representations [29, 7, 124], properties of high-level constraints [26, 131] and gate structure in problem encodings [111, 59, *3].

The gate structure of combinational circuits is a wide-spread intermediate problem representation in many applications [126, 6, 1, 34]. In order to use a SAT solver, these combinational circuits are encoded in conjunctive normal form (CNF) using structural encodings like the well-known Tseitin encoding [128, 112].

The syntactic properties of structural encodings have been studied extensively [86, 15]. Järvisalo showed that structural CNF encodings can induce exponentially shorter proofs in CDCL for some unsatisfiable problems, as these encodings introduce encoding variables just like extended resolution, which is one of the strongest known proof systems [88, 94].

Contributions.

In this thesis, we present a new generic algorithm to efficiently *recognize* gate structure in CNF encodings and three approaches in which we *exploit* that structure. Our contributions also include the implementation of these approaches in the new SAT solver *Candy* and the development of a tool for distributed management of benchmark instances and their attributes, the *Global Benchmark Database* (GBD).

Gate Recognition. Based on a combined exploitation of syntactic and semantic properties of CNF formulas, we devised a generic gate recognition algorithm which is based on the detection of functional relations between variables [*3]. Our hierarchical approach allows for efficient tracking of monotonic arguments in the

detected functional relations and thus also enables the recognition of optimized circuit encodings.

Gate Exploitation. We use our algorithm for efficient *model minimization* via detection of “don’t cares” in the decoded gate structure [*2]. In another approach, we generate conjectures through *random simulation* on the extracted circuit structure [96]. The conjectures are then used in two approaches. In one approach, we perform *abstraction by under-approximation* and in another approach, we experimented with the modified conjecture-driven branching heuristic *implicit learning* [96, *5].

Candy. All the developed algorithms are implemented in our SAT solver Candy, which is publicly available on Github [*10]. Candy is forked from Glucose 3 [11] and underwent a complete refinement in order to provide a modular architecture that makes it specifically easy to implement new strategies for existing solver sub-systems. Candy also has a parallel mode with concurrent access to a shared clause database, which is described in [*8].

Global Benchmark Database. We collected benchmark meta-data in our public project Global Benchmark Database (GBD), and used the tool for evaluating our algorithms [*7]. GBD is available on Github [*12] and is aimed to support the research community in exchanging, organizing and analyzing benchmark attributes.

Structure is just lack of entropy.

Preface

Computer science includes the exploration of boundaries in the nature of computation and algorithms. In theoretical computer science, the SAT problem is the archetypal representative of the class of NP-complete problems, thus SAT solving is generally considered intractable. However, SAT algorithms are an active area of research, and exciting results can be found in practical SAT solving, where huge instances can be solved in a reasonable time by recent solvers.

In order to explain the success of state-of-the-art SAT solvers in industrial practice, SAT practitioners conjecture that the heuristics which are employed in these solvers implicitly exploit the structure of the highly structured industrial SAT instances.

Fundamental questions arise, which demonstrate the necessity to analyze the structure of SAT problems. Which formalism is most suitable to characterize the structure of SAT instances? How can we connect structure to complexity? Does the existence of structure impose a structure upon the space of unstructured instances?

In this thesis, we analyze *gate structure* in SAT instances and devise an efficient and generic algorithm for recognizing gates, i.e., functional relations of variables, in SAT instances. In three exemplary procedures, we show how these functional relations can be used to speed up SAT solving and SAT-based approaches. Our work includes the development of the new modular SAT solver "Candy" and the benchmark instance and attribute management system "Global Benchmark Database" (GBD).

Acknowledgments and Dedication

First of all, I want to thank Carsten Sinz for his support in exciting projects, for numerous discussions and fruitful questions, and for sharing his expertise whenever the situation required it. I also want to thank my second reviewer Laurent Simon for his effort and for inspirational results in SAT research.

I am grateful to my former colleagues Tomáš Balyo and Felix Kutzner for many discussions and exchange of ideas. Also, the energy and support of the present and former students in our group is highly appreciated.

Moreover, I would like to express my gratitude to my colleagues at the Institute of Theoretical Informatics for creating a fruitful and cooperative research environment, and especially to Peter Sanders for supporting research in massively parallel SAT solving and to Mana Taghdiri, who took a strong supportive role at the beginning of this project. Furthermore, I want to convey my appreciation to the international research community, especially to the SAT community.

This dissertation is dedicated to my daughter Rosalie. During the past 14 years, her curiosity and power have always been an inspiration.

Contents

If you really want to escape the things that harass you, what you're needing is not to be in a different place but to be a different person.

Seneca, Letters from a Stoic

Chapter 1 Introduction

SAT solving denotes the automatic determination of the satisfiability (SAT) of propositional formulas. In the past decades, SAT solvers have been applied in an increasing number of application domains, such as formal software verification [27, 57], hardware model checking [28, 24], electronic design automation [120, 107], bioinformatics [102] or applications in artificial intelligence (AI) such as automated planning [113] and scheduling [78, 5].

At the time of writing, state-of-the-art SAT solvers are based on the conflict-driven clause learning (CDCL) algorithm [121, 108] and capable of solving huge problem instances which contain millions of variables. The performance of CDCL SAT solvers largely depends on heuristics [119, 90], e.g., branching and forgetting heuristics [83], and the most successful heuristics implicitly exploit the structure of many instances which are generated in industrial practice [137].

Many recent solvers [*4] extract structural features in order to classify benchmark problems for subsequent automated algorithm selection or heuristic configuration by using methods of machine learning [136, 89, 7].

1.1 Structure of SAT Instances

Structure in SAT instances can be formalized in many ways, e.g., based on graph-based features such as *communities* in the variable incidence graph of a given problem instance [29, 124, 8] or *symmetries* [4].

Properties of SAT encodings of high-level *constraints* have been studied as well [26, 131] and Dixon et al. [50, 49, 48] generalize constraints with their definition of *augmented clauses*, where an augmented clause is the tuple of a clause and a permutation group over its literals. They show how to express high-level constraints, e.g., cardinality constraints, with a single augmented clause, and they even devise a resolution procedure for augmented clauses, which is, however, NP-complete.

Due to its ubiquity, the analysis of gate structure in *circuit* encodings has been of particular interest [111, 59, *3] and forms the basis of sophisticated preprocessing technology in state-of-the-art SAT solvers [85, 86]. In some experimental approaches, the preservation of information about the original circuit structure and its specialized exploitation was suggested [122, *1].

The size of *unsatisfiable cores* [103] and *backdoors* [133] has been related to problem hardness. Gaspers and Szeider use backdoors [61] and *tree-width* [62] to analyze SAT solving by methods of parameterized complexity.

Tractable Subclasses

Schaefer [117] identified 6 tractable subclasses of the SAT problem which are defined based on syntactic properties. *Positive (negative) formulas* are characterized such that their clauses only contain literals of positive (negative) polarity and are trivially satisfiable. Poly-time algorithms also exist for *horn formulas* (reverse horn formulas) which only consist of clauses which contain at most one positive (at most one negative) literal, and for *2-SAT formulas* where clause length is bound by 2. Affine formulas are equivalent to a conjunction of `XOR`-terms which can efficiently be solved using a system of linear equations.

Structure of Application Instances

In many decision procedures for NP-hard problems, the problem instances are represented as directed acyclic graphs (DAG) resembling the gate structure of combinational circuits [126, 6, 1, 34]. In order to reduce the problem to SAT, these problems are encoded in conjunctive normal form (CNF).

Traditional CNF encodings can lead to exponential increase in formula size due to nested applications of the distributive rule [36]. Thus, in practice, structural encodings like the well-known Tseitin encoding [128] and the optimized Plaisted Greenbaum encoding [112] are used, which only require linear overhead in formula size. Other researchers combine traditional and structural encodings, and mitigate the linear blow-up of structural encodings by using traditional encodings in special cases when they are smaller [127, 82].

The syntactic properties of structural encodings have been studied intensely [86, 75, 15]. Järvisalo showed in [88] that structural CNF encodings can induce exponentially shorter proofs in CDCL for some unsatisfiable problem instances, as these encodings introduce additional encoding variables, which results in proofs which are similar to those produced by extended resolution [128], which is one of the strongest known proof systems [68, 94].

Combinational gate structure depicts a set of functional relations of variables in SAT instances. Such functional relations might not always be explicitly encoded by using the aforementioned structural encodings. They can even be present in randomly generated SAT instances and may surface after further problem transformations [*3].

Can we efficiently recognize gate structure in CNF formulas? Does gate structure affect the tractability of SAT instances? How can we effectively exploit gate structure in SAT applications?

1.2 Contributions

In this thesis, we present a new generic algorithm to efficiently *recognize* gate structure in CNF encodings and three approaches in which we *exploit* that structure. Our contributions also include implementations of these approaches in the new modular SAT solver *Candy* and the distributed management of benchmark instance feature data in our project *Global Benchmark Database* (GBD).

Gate Recognition.

Based on the combined exploitation of syntactic and semantic properties of structural CNF encodings, we devise a generic gate recognition algorithm which is based on the detection of functional relations between variables [*3]. Our generic approach together with its hierarchical tracking of monotonicity also enables the recognition of optimized Plaisted Greenbaum encodings. We demonstrate the efficiency and effectiveness of our algorithm despite its greedy parts.

Gate Exploitation.

We used our algorithm for efficient *model minimization* via detection of "don't cares" in the decoded gate structure [*2]. In another approach, we generate conjectures about variable equivalences and constants using *random simulation* on the extracted circuit structure [96]. We further use these conjectures in two approaches. In one approach, we perform *abstraction by under-approximation* and in another approach, we employ the conjecture-driven branching heuristic *implicit learning* [96, *5].

Candy – A Modular SAT Solver.

We implemented our algorithms as part of our open-source SAT solver Candy which is available on Github [*10]. Candy is forked from Glucose 3 [11] and we refined it completely in order to provide a modular architecture that facilitates implementing new strategies for existing solver sub-systems. Candy is the first parallel *inprocessing* SAT solver which allows for concurrent access to a shared clause database [*8].

Global Benchmark Database.

In our public project Global Benchmark Database (GBD) we collect benchmark instance features and use the tool for evaluating our algorithms [*7]. GBD is available on Github [*12] and aims to assist in the exchange and organization of instance meta-data across the research community. GBD has a command-line tool and a web interface. However, the main contribution is the *GBD hash* function, which we use for identifying benchmark instances.

1.3 Structure of this Work

In Chapter 2, we introduce essential terminology and notations. Chapter 3 is dedicated to the theory and practice of gate recognition, where we formalize the procedure and present our algorithm in detail. Three exemplary approaches that exploit gate structure to speed up SAT solving and SAT-based applications are presented in Chapter 4. The tools which have been implemented in the course of this thesis are presented in Chapter 5. In Chapter 6, we present an extensive evaluation of our algorithmic approaches. Finally, our conclusions can be found in Chapter 7.

The first rule is to keep an untroubled spirit. The second is to look things in the face and know them for what they are.

Marcus Aurelius, Meditations

Chapter 2

Preliminaries

We denote the set of finite propositional formulas in the Boolean domain $\mathbb{B} = \{0, 1\}$ by Φ. Formulas in Φ are built from a countably infinite set of variable symbols $\mathbb{X}$, the unary negation symbol $\neg$, binary logical connectives $\kappa = \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$, and constants $\mathbb{B} = \{0, 1\}$.

Each formula $F \in \Phi$ is defined over a finite set of variables $\mathbb{X}_F \subseteq \mathbb{X}$. The function $\mathsf{vars}(F)$ denotes the set of variables which actually occur in F. $\mathbb{X}_F$ and $\mathsf{vars}(F)$ are not necessarily equal but it holds that $\mathsf{vars}(F) \subseteq \mathbb{X}_F$. Unless stated otherwise for a given formula F, we define that $\mathbb{X}_F := \mathsf{vars}(F)$. Variables in $\mathbb{X}_F \setminus \mathsf{vars}(F)$ are considered *unconstrained* in F.

Given a set of variables $X \subseteq \mathbb{X}$, the function $\mathsf{lits}(X)$ denotes the set of literals over X, i.e., $\mathsf{lits}(X) = X \cup \{\neg x \mid x \in X\}$. Given a formula $F \in \Phi$, the function $\mathsf{lits}(F)$ denotes the set of literals which actually occur in F.

The complement $\bar{l}$ of a literal l is defined such that $\bar{l} = \neg v$ if $l = v$, and $\bar{l} = v$ if $l = \neg v$. The variable of a literal l is denoted by $\mathsf{var}(l)$ which is v if $l = \neg v$ and l otherwise. The *polarity* of a literal l with $\mathsf{var}(l) = v$ is *negative* if $l = \neg v$ and *positive* otherwise.

In order to eliminate ambiguity, e.g., in argument lists of functions and in tuples, we impose a fixed ordering $<^{\mathbb{X}}$ over the variables in $\mathbb{X}$. Without loss of generality, whenever we use variable indices it holds that $v_i <^{\mathbb{X}} v_j$ iff $i < j$ for indices $i, j \in \mathbb{N}$.

Semantics

Given a formula $F \in \Phi$ over variables $\mathbb{X}_F$, a variable assignment for F is a function $\alpha : \mathbb{X}_F \rightarrow \mathbb{B}$. Given an assignment α the interpretation function $\mathcal{I}_\alpha : \Phi \rightarrow \{\bot, \top\}$ is defined as in Figure 2.1. An assignment α is represented by the set M_α of all literals which are satisfied under alpha, i.e., $M_\alpha := \{l \mid l \in \mathsf{lits}(\mathbb{X}_F) \wedge \mathcal{I}_\alpha(l) = \top\}$. Moreover, we use $\mathcal{A}(F)$ to denote the set of possible assignments for variables in $\mathbb{X}_F$, i.e., $\mathcal{A}(F) := \{P \cup \{\neg v \mid v \in \mathbb{X}_F \setminus P\} \mid P \in 2^{\mathbb{X}_F}\}$.

Definition 1 (Model)**.** *Given a formula F, an assignment α is a* model *for F denoted by $M_\alpha \models F$ iff $\mathcal{I}_\alpha(F) = \top$. The set of models of a formula F is specified by $\mathcal{M}(F) = \{M_\alpha \mid \mathcal{I}_\alpha(F) = \top\}$.*

Definition 2 (Logical Consequence)**.** *A formula F is a* logical consequence *of a formula G, denoted by $G \models F$, iff every model of G is also a model of F, i.e., $\mathcal{M}(G) \subseteq \mathcal{M}(F)$.*

Definition 3 (Logical Equivalence)**.** *Two formulas F and G are* logically equivalent, *denoted by $F \equiv G$, iff $\mathcal{M}(G) = \mathcal{M}(F)$, i.e., F and G have exactly the same models.*

$$\mathcal{I}_\alpha(0) = \bot \qquad \mathcal{I}_\alpha(1) = \top$$

$$\mathcal{I}_\alpha(F \to G) = \begin{cases} \top, & \text{if } \mathcal{I}_\alpha(F) = \bot \text{ or } \mathcal{I}_\alpha(G) = \top \\ \bot, & \text{if } \mathcal{I}_\alpha(F) = \top \text{ and } \mathcal{I}_\alpha(G) = \bot \end{cases}$$

$$\mathcal{I}_\alpha(v) = \begin{cases} \top, & \text{if } \alpha(v) = 1 \\ \bot, & \text{if } \alpha(v) = 0 \end{cases} \qquad \mathcal{I}_\alpha(F \oplus G) = \begin{cases} \top, & \text{if } \mathcal{I}_\alpha(F) \neq \mathcal{I}_\alpha(G) \\ \bot, & \text{if } \mathcal{I}_\alpha(F) = \mathcal{I}_\alpha(G) \end{cases}$$

$$\mathcal{I}_\alpha(\neg F) = \begin{cases} \top, & \text{if } \mathcal{I}_\alpha(F) = \bot \\ \bot, & \text{if } \mathcal{I}_\alpha(F) = \top \end{cases} \qquad \mathcal{I}_\alpha(F \wedge G) = \begin{cases} \top, & \text{if } \mathcal{I}_\alpha(F) = \top \text{ and } \mathcal{I}_\alpha(G) = \top \\ \bot, & \text{if } \mathcal{I}_\alpha(F) = \bot \text{ or } \mathcal{I}_\alpha(G) = \bot \end{cases}$$

$$\mathcal{I}_\alpha(F \leftrightarrow G) = \begin{cases} \top, & \text{if } \mathcal{I}_\alpha(F) = \mathcal{I}_\alpha(G) \\ \bot, & \text{if } \mathcal{I}_\alpha(F) \neq \mathcal{I}_\alpha(G) \end{cases} \qquad \mathcal{I}_\alpha(F \vee G) = \begin{cases} \top, & \text{if } \mathcal{I}_\alpha(F) = \top \text{ or } \mathcal{I}_\alpha(G) = \top \\ \bot, & \text{if } \mathcal{I}_\alpha(F) = \bot \text{ and } \mathcal{I}_\alpha(G) = \bot \end{cases}$$

Figure 2.1: Interpretation of a formula under an assignment α

Equivalence under Projection

In this work, we investigate transformations of formulas which introduce encoding variables. For the sake of brevity, we define *equivalence under projection.* Let F be a propositional formula over variables $\mathbb{X}_F$, and let X be a *non-empty* subset of variables $X \subseteq \mathbb{X}_F$. We define the projection of models $\mathcal{M}(G)$ to X as follows:

$$\mathcal{M}(F, X) := \{M \cap \mathsf{lits}(X) \mid M \in \mathcal{M}(F)\}$$

Based on this projection we extend the definition of equivalence.

Definition 4 (Equivalence under Projection)**.** *Given formulas F and G with a* non-empty *intersection of variables $I = \mathbb{X}_F \cap \mathbb{X}_G$. F and G are* equivalent under projection*, denoted by $F \cong G$, iff $\mathcal{M}(G, I) = \mathcal{M}(F, I)$.*

Boolean Functions

We denote the set of Boolean functions [132] by Ω. For every $k \in \mathbb{N}_0$ we denote the set of Boolean functions of arity k by Ω_k. For each formula $F \in \Phi$ we define its characteristic function $\mathcal{C}(F) \in \Omega_{|\mathbb{X}_F|}$ as follows.

Definition 5 (Characteristic Function)**.** *Given a formula F with models $\mathcal{M}(F)$ over variables $\mathbb{X}_F = \{v_1, \ldots, v_k\}$, we construct the set of tuples $\mathcal{T}(F) \subseteq \mathbb{B}^k$, such that $\mathcal{T}(F) := \{(t_1, \ldots, t_k) \mid \exists M \in \mathcal{M}(F) :$ if $v_i \in M$ then $t_i = 1$ else $t_i = 0\}$. We denote the* characteristic function *of the such constructed auxiliary set $\mathcal{T}(F)$ as the* characteristic function *$\mathcal{C}(F)$ of F.*

Note that formulas having the same characteristic function of fixed arity are equivalent. The set of Boolean formulas which have the same characteristic function $\omega \in \Omega$ is denoted by Φ_ω. We denote formulas $F \in \Phi_\omega$ as ω-encodings.

Examples. Boolean functions are, e.g., `AND`, `OR`, `EQIV`, `XOR`. The propositional formulas $(a \vee b) \wedge (\neg a \vee \neg b)$ and $a \oplus b$ are both `XOR`-encodings of arity 2. All unsatisfiable propositional formulas with n variables are encodings of one n-ary function that maps all n-tuples to zero.

Conjunctive Normal Form (CNF)

A formula F is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. A CNF formula F has the form $F = D_0 \wedge \cdots \wedge D_m$ and each D_i is of the form $(l_1 \vee \cdots \vee l_n)$ with $l_i \in \mathsf{lits}(\mathbb{X}_F)$. A CNF formula F is represented by a set of clauses, where a clause is a set of literals and represents one disjunction in the formula.

$$x \lor (y \lor z) \equiv (x \lor y) \lor z \qquad x \land (y \land z) \equiv (x \land y) \land z \tag{2.1}$$

$$x \lor y \equiv y \lor x \qquad x \land y \equiv y \land x \tag{2.2}$$

$$x \land (y \lor z) \equiv (x \land y) \lor (x \land z) \qquad x \lor (y \land z) \equiv (x \lor y) \land (x \lor z) \tag{2.3}$$

$$\neg(x \lor y) \equiv \neg x \land \neg y \qquad \neg(x \land y) \equiv \neg x \lor \neg y \tag{2.4}$$

$$x \lor (x \land y) \equiv x \qquad x \land (x \lor y) \equiv x \tag{2.5}$$

$$\neg(\neg x) \equiv x \tag{2.6}$$

$$x \lor x \equiv x \qquad x \land x \equiv x \tag{2.7}$$

$$x \lor 0 \equiv x \qquad x \land 1 \equiv x \tag{2.8}$$

$$x \land 0 \equiv 0 \qquad x \lor 1 \equiv 1 \tag{2.9}$$

$$x \land \neg x \equiv 0 \qquad x \lor \neg x \equiv 1 \tag{2.10}$$

$$\neg 0 \equiv 1 \qquad \neg 1 \equiv 0 \tag{2.11}$$

Figure 2.2: Some tautologies in propositional logic

Example 1 (CNF Notation)**.** *The set $\{\{a\}, \{b, c\}\}$ denotes the CNF formula $a \land (b \lor c)$. Important edge cases are the set containing the empty set $\{\emptyset\}$, which denotes false, and the empty set $\emptyset$, which denotes true.*

Definition 6 (Tautology)**.** *A propositional formula F is a* tautology*, denoted by $\models F$, iff every assignment in $\mathcal{A}(F)$ is a model of F.*

A clause C is tautologic if it contains a literal l as well as its complement $\bar{l}$. Figure 2.2 shows some well-known tautologies, subsets of which have also been used in axiomatic systems of Boolean algebra [125]. In Section 2.1 we use semantic tautologies to transform propositional formulas.

Resolution and Blocked Clauses

Resolution is a well-known inference rule in propositional logic. The resolution operator (Definition 7) can be applied to clauses which contain the same variable in complementary polarities.

Definition 7 (Resolution)**.** *Given two clauses C_1 and C_2 and a literal l with $l \in C_1$ and $\bar{l} \in C_2$, the* resolvent *$C_1 \otimes_l C_2$ is the clause $(C_1 \cup C_2) \setminus \{l, \bar{l}\}$. It holds that $\{C_1, C_2\} \models C_1 \otimes_l C_2$.*

Given a CNF formula F and a literal l, the set of literal occurrences $F[l]$ is defined to be the subset of clauses in F which contain the literal l, i.e.,

$$F[l] := \{C \in F \mid l \in C\}$$

The clauses which can be used to perform resolution with a variable v are denoted as the *resolution environment* of v.

Definition 8 (Resolution Environment)**.** *Given a formula F and variable v, the* resolution environment *of v is the set $F[v] \cup F[\neg v]$.*

Definition 9 (Pure Literal)**.** *Given a formula F, a literal $l \in \mathsf{lits}(F)$ is* pure *in F iff $F[\bar{l}] = \emptyset$.*

Given a formula F and a literal l, if l is either pure in F or if all possible resolvents of clauses in $F[l]$ and $F[\bar{l}]$ are tautologic, we denote l as *blocking literal* [94]. Note that given a formula F and a blocking literal $l \in \mathsf{lits}(\mathbb{X}_F)$, no resolvents on l can be used in a resolution proof of F. Clauses which contain a blocking literal are denoted as *blocked clauses*.

Definition 10 (Blocked Clause)**.** *Given a CNF formula F, a clause $C \in F$ is* blocked *in F if there exists a literal $l \in C$ such that l is either pure in F, or for every clause $D \in F[\bar{l}]$ the resolvent $C \otimes_l D$ is a tautology. In that case, the literal l is also called the* blocking literal *of C.*

Given a formula F, the set of *unit clauses* is the set of clauses of size 1 in F. The set of *facts* denotes the set of literals which appear in unit-clauses of F.

Given a formula F, a variable assignment $\alpha : \mathbb{X}_F \rightsquigarrow \mathbb{B}$ is *partial*, if some variables in $\mathbb{X}_F$ are not assigned by α. Given a partial assignment M_α, *unit-propagation* is the method to deduce an extended set of literals of $M^*_\alpha \supseteq M_\alpha$ which is implied by the clauses in F and the literals in M_α.

Definition 11 (Unit Propagation)**.** *Given a CNF formula F and a partial assignment M_α, the result of unit-propagation is the set of literals $M^*_\alpha \supseteq M_\alpha$ which is created as follows. A formula F' is created by removing all literals which are not satisfied under M_α from clauses in F. Then M^0_α is the union of M_α and facts in F'. The procedure is repeated with the extended set of literals M^0_α to obtain the set M^1_α and so on. The result of unit-propagation is the set M^*_α which is obtained when we repeat the procedure until fix point.*

Given a formula F, a partial assignment M_α is *conflicting* iff M^*_α, which is obtained by unit-propagation, contains two complementary literals l and $\bar{l}$.

2.1 CNF Encodings

Given a formula F, a CNF encoding usually refers to a set of rules that transforms the formula F to an equivalent formula $F' \equiv F$ such that F' is in CNF. Structural encodings do not maintain equivalence as they usually introduce new variable symbols. However, in structural encodings F' equivalence under projection is maintained, i.e., $F' \cong F$.

Direct CNF Encodings

One possibility to convert propositional formulas to CNF is the selective application of the tautologies shown in Figure 2.2. We denote this method as the *direct encoding* and use the symbol ${}^D\mathcal{E}$.

In order to reduce the set of operators, first, operators in $\{\rightarrow, \leftrightarrow, \oplus\}$ are replaced by equivalent representations using only operators in $\{\wedge, \vee, \neg\}$. This is denoted by the following rules 2.12, 2.13 and 2.14, where $A, B \in \Phi$.

$$A \rightarrow B \xrightarrow{{}^D\mathcal{E}} \neg A \vee B \tag{2.12}$$

$$A \leftrightarrow B \xrightarrow{{}^D\mathcal{E}} (\neg A \vee B) \wedge (\neg B \vee A) \tag{2.13}$$

$$A \oplus B \xrightarrow{{}^D\mathcal{E}} (\neg A \vee \neg B) \wedge (B \vee A) \tag{2.14}$$

In order to convert a formula to CNF, subsequently the following rules are applied.

$$\neg\neg A \xrightarrow{{}^{D}\mathcal{E}} A \tag{2.15}$$

$$\neg(A \lor B) \xrightarrow{{}^{D}\mathcal{E}} \neg A \land \neg B \tag{2.16}$$

$$\neg(A \land B) \xrightarrow{{}^{D}\mathcal{E}} \neg A \lor \neg B \tag{2.17}$$

$$(A \land B) \lor C \xrightarrow{{}^{D}\mathcal{E}} (A \lor C) \land (B \lor C) \tag{2.18}$$

$$C \lor (A \land B) \xrightarrow{{}^{D}\mathcal{E}} (A \lor C) \land (B \lor C) \tag{2.19}$$

Given a formula F, ${}^{D}\mathcal{E}(F)$ denotes the CNF formula which is obtained by application of the rules 2.12 to 2.19 until no more rule can be applied, and the application of rules is prioritized by the order in which they are given above. Note that by specifying rule prioritization we achieve confluence such that ${}^{D}\mathcal{E}(F)$ is well-defined. For the sake of termination, we introduced rule 2.19 and left out general commutativity.

In Example 2, a direct CNF encoding of the formula $\neg\big((a \lor \neg b) \land (\neg c \lor \neg d)\big)$ is deduced by subsequent rule application.

Example 2 (Direct Encoding of a Negated CNF)**.**

$$\begin{aligned}
& \neg\big((a \lor \neg b) \land (\neg c \lor \neg d)\big) \\
\equiv\ & \neg(a \lor \neg b) \lor \neg(\neg c \lor \neg d) \\
\equiv\ & (\neg a \land b) \lor (c \land d) \\
\equiv\ & \big((\neg a \land b) \lor c\big) \land \big((\neg a \land b) \lor d\big) \\
\equiv\ & (\neg a \lor c) \land (b \lor c) \land (\neg a \lor d) \land (b \lor d)
\end{aligned}$$

One disadvantage of the direct encoding is that it often obscures the structure of the original formula. Additionally, the size of the resulting CNF formula grows exponentially, as the distribution rule is required to be applied repeatedly [36].

Structural CNF Encodings

Based on Tseitin's extension rule of the resolution calculus [128], Plaisted and Greenbaum developed a structure-preserving method of converting propositional formulas to CNF [112]. An extensive set of references on CNF encodings and related work can be found in Chapter 3.

Given a formula F, structural encodings introduce new encoding variables $\mathcal{X}^*_F \notin \mathbb{X}_F$ (Equation 2.20) and define their equivalence to sub-formulas of F (Equation 2.21). We define structural encodings only for the reduced set of operators $\{\land, \lor, \neg\}$, assuming that other operators are first eliminated using rules 2.12, 2.13 and 2.14.

Let S be the formula to be encoded. In Equation 2.20, we define the operator $\mathcal{X}^*_F$. If F is a literal in $\mathsf{lits}(\mathbb{X}_S)$, then $\mathcal{X}^*_F = F$. Otherwise, $\mathcal{X}^*_F$ specifies a unique new variable d_F for the sub-formula F of S. Subsequently, e.g., if a sub-formula F occurs multiple times, $\mathcal{X}^*_F$ specifies the same variable such that at most one variable is obtained for F. Note that $\mathcal{X}^*_F$ returns a literal in order to inline negation operators.

$$\mathcal{X}^*_F = \begin{cases} d_F, & \text{if } F = G \circ H \text{ with } \circ \in \{\land, \lor\} \\ \neg d_G, & \text{if } F = \neg G \text{ and } F \notin \mathsf{lits}(\mathbb{X}_S) \\ F & \text{if } F \in \mathsf{lits}(\mathbb{X}_S) \end{cases} \tag{2.20}$$

Tseitin Encoding. The Tseitin encoding can be used to transform a structured propositional formula to CNF by introducing new variables and their definitions. In the following equations it holds that $\circ \in \{\wedge, \vee\}$.

$$\mathcal{E}'(F) = \begin{cases} {}^{D}\mathcal{E}(\mathcal{X}_F^* \leftrightarrow (\mathcal{X}_G^* \circ \mathcal{X}_H^*)) \wedge \mathcal{E}'(G) \wedge \mathcal{E}'(H), & \text{if } F = G \circ H \\ {}^{D}\mathcal{E}(\mathcal{X}_F^* \leftrightarrow \neg\mathcal{X}_G^*) \wedge \mathcal{E}'(G), & \text{if } F = \neg G \\ 1, & \text{if } F \in \mathsf{lits}(\mathbb{X}) \end{cases} \tag{2.21}$$

$${}^{T}\mathcal{E}(F) = \begin{cases} [\mathcal{X}_G^* \circ \mathcal{X}_H^*] \wedge \mathcal{E}'(G) \wedge \mathcal{E}'(H), & \text{if } F = G \circ H \\ \neg\mathcal{X}_G^* \wedge \mathcal{E}'(G), & \text{if } F = \neg G \\ \mathcal{X}_F^*, & \text{if } F \in \mathsf{lits}(\mathbb{X}) \end{cases} \tag{2.22}$$

Given a formula F, in its Tseitin encoding $F' = {}^{T}\mathcal{E}(F)$, new variable symbols $\mathcal{X}_G^*$ are introduced for sub-formulas G of F. Thus, the set of variables $\mathbb{X}_{F'}$ can be partitioned into *input variables* $\mathbb{X}_F$ that stem from the original formula F and *encoding variables* $\mathbb{X}_{F'} \setminus \mathbb{X}_F$. The formulas F' and F are not necessarily equivalent, as models M_α of F do not include assignments to encoding variables in $\mathbb{X}_{F'} \setminus \mathbb{X}_F$.

However, it holds that ${}^{T}\mathcal{E}(F) \cong F$, i.e., ${}^{T}\mathcal{E}(\mathrm{F})$ and F are equivalent under projection to input variables. Given an assignment α for variables $\mathbb{X}_F$, a complete assignment for $\mathbb{X}_{F'}$ can be deduced by unit-propagation. Note that it holds that ${}^{T}\mathcal{E}(F)$ and F have the same number of models.

The direct encoding of a negation of a set of clauses has been shown in Example 2. The Tseitin encoding of the same formula is shown in Example 3.

Example 3 (Tseitin Encoding of a Negated CNF)**.**

$$\begin{aligned} &\neg\big((a \vee \neg b) \wedge (\neg c \vee \neg d)\big) \\ \cong\ &\neg e_0 \wedge {}^{D}\mathcal{E}\big(e_0 \leftrightarrow (e_1 \wedge e_2)\big) \wedge {}^{D}\mathcal{E}\big(e_1 \leftrightarrow (a \vee \neg b)\big) \wedge {}^{D}\mathcal{E}\big(e_2 \leftrightarrow (\neg c \vee \neg d)\big) \end{aligned}$$

The number of clauses in the resulting formula is in $\mathcal{O}(n)$ where n is the number of sub-formulas of the formula to be encoded. The encoding of the equivalences contributes a constant factor to the number of generated clauses. Note that for readability, we did not apply the direct encoding of the equivalences.

Plaisted Greenbaum Encoding. Plaisted and Greenbaum [112] present an optimized Tseitin encoding, based on the observation that monotonic sub-formulas provoke the emergence of "don't care" values. They keep track of the polarity of the assignments under which the sub-formulas to be encoded are satisfied. In case of monotonic sub-formulas, there is one polarity under which the sub-formula is always satisfied and another under which that sub-formula is not satisfied. In this case, implications are used instead of equivalences to define the encoding variables and the direction of the implication depends on the polarity as can be seen in Equations 2.23 and 2.24.

$$\mathcal{E}^p(F) = \begin{cases} {}^{D}\mathcal{E}(\mathcal{X}_F^* \leftarrow (\mathcal{X}_G^* \circ \mathcal{X}_H^*)) \wedge \mathcal{E}^p(G) \wedge \mathcal{E}^p(H), & \text{if } F = G \circ H, p = 0 \\ {}^{D}\mathcal{E}(\mathcal{X}_F^* \rightarrow (\mathcal{X}_G^* \circ \mathcal{X}_H^*)) \wedge \mathcal{E}^p(G) \wedge \mathcal{E}^p(H), & \text{if } F = G \circ H, p = 1 \\ {}^{D}\mathcal{E}(\mathcal{X}_F^* \leftarrow \neg\mathcal{X}_G^*) \wedge \mathcal{E}^{p \oplus 1}(G), & \text{if } F = \neg G, p = 0 \\ {}^{D}\mathcal{E}(\mathcal{X}_F^* \rightarrow \neg\mathcal{X}_G^*) \wedge \mathcal{E}^{p \oplus 1}(G), & \text{if } F = \neg G, p = 1 \\ 1, & \text{if } F \in \mathsf{lits}(\mathbb{X}) \end{cases} \tag{2.23}$$

$${}^{PG}\mathcal{E}(F) = \begin{cases} [\mathcal{X}_G^* \circ \mathcal{X}_H^*] \wedge \mathcal{E}^1(G) \wedge \mathcal{E}^1(H), & \text{if } F = G \circ H \\ \neg\mathcal{X}_G^* \wedge \mathcal{E}^0(G), & \text{if } F = \neg G \\ \mathcal{X}_F^*, & \text{if } F \in \mathsf{lits}(\mathbb{X}) \end{cases} \tag{2.24}$$

Using implications instead of equivalences causes encoding variables in Plaisted Greenbaum encodings to be slightly under-constrained in comparison to Tseitin encodings, meaning that additional models emerge due to spurious assignments to "don't cares" in the circuit encoding. However, F' and F are equivalent under projection as no spurious assignments to input variables emerge.

As the encoded formula is normalized such that it only contains binary `AND` and `OR` operators, non-monotonic formulas are not recognizable locally. However, variables $\mathbb{X}_F$ of a non-monotonic formula F appear as literals of both polarities in the direct encoding of F (see Equations 2.14 and 2.13). In that case, both functions $\mathcal{E}^1$ and $\mathcal{E}^0$ are used to encode the respective sub-formula, thus producing a Tseitin encoding in the non-monotonic case.

Like in Examples 2 and 3 for direct and Tseitin encoding, we show in Example 4 the Plaisted Greenbaum encoding of the negation of a set of clauses.

Example 4 (Plaisted Greenbaum Encoding of a Negated CNF)**.**

$$\begin{aligned} & \neg\big((a \vee \neg b) \wedge (\neg c \vee \neg d)\big) \\ \cong\ & \neg e_0 \wedge {}^D\mathcal{E}\big(e_0 \leftarrow (e_1 \wedge e_2)\big) \wedge {}^D\mathcal{E}\big(e_1 \leftarrow (a \vee \neg b)\big) \wedge {}^D\mathcal{E}\big(e_2 \leftarrow (\neg c \vee \neg d)\big) \end{aligned}$$

The encoding of the implications contributes to a smaller constant factor to the number of generated clauses than the encoding of the equivalences in the Tseitin encoding. Note that for readability, we did not apply the direct encoding of the implications.

2.2 Conflict-Driven Clause Learning (CDCL)

The Conflict-Driven Clause Learning (CDCL) algorithm emerged as an extension to the previous systematic search algorithm by Davis, Putnam, Logemann and Loveland (DPLL) [45]. Clause learning has been introduced by Marques-Silva and Sakallah with the SAT solver GRASP [121].

Whenever search determines a conflicting assignment, clause learning deduces a conflict clause by a limited set of resolution operations on the reason clauses, i.e., clauses which contributed to the conflict via unit-propagation. Therefore, CDCL can be understood as search-directed resolution.

Data structures for efficient unit-propagation and the successful branching heuristic Variable-State Independent Decaying Sum (VSIDS) have originally been developed for the SAT solver Chaff [108]. Random restarts [66, 22, 65] and clause forgetting [64] became crucial for CDCL SAT solver efficiency. Furthermore, efficient clause learning data-structures [115, 98] are essential for the performance of state-of-the-art SAT solver.

Presently, some of the most successful CDCL SAT solvers are descendants of Minisat [52] such as the well-known solver Glucose [11, 10]. Further successful descendants of Minisat and Glucose include Maple, RISS and CryptoMinisat, but also competing systems such as Lingeling and CaDiCal are well-known for outstanding performance [71, 72]

The efficiency of SAT solvers is largely determined by heuristics [55]. Heuristics are used to determine branching order or restart intervals, and control strategies for clause learning and forgetting, among others. An empirical study of some heuristics used in CDCL has been conducted by Katebi and Sakallah [90].

Many heuristics can be explained by using connected components in graph representations of the SAT problem [29]. Recently, Jamali and Mitchel showed how branching and forgetting can be improved by taking into account the betweenness centrality of variables in the variable incidence graph of a SAT instance [83]. Ganesh et al. focus on statistical methods in order to weigh the success of heuristics based on successful clause learning [100, 56].

Instance-specific algorithm configuration is used in sophisticated portfolio approaches like SATZilla by Hoos et al. [136] and ISAC by Kadioglu et al. [89]. Ansotegui et al. empirically showed that only a few graph-based structural features of SAT instances can already be very effective in algorithm selection [7].

Recent work on CDCL includes satisfaction-driven clause learning by Heule at al. [74, 73] which they successfully applied to automatically generate cubic-sized proofs for pigeon hole formulas. Heule recently devised the largest known mathematical proof with Cube-and-Conquer and massively parallel SAT solving [70].

Pre- and Inprocessing

In the context of SAT solving, preprocessing denotes a family of various deductive formula simplification techniques which are aimed at reducing formula size and which are usually applied before CDCL search starts. Recent CDCL implementations regularly interrupt search in order to apply these preprocessing techniques in combination with learned clauses, which is commonly denoted as *inprocessing*.

Early work on SAT preprocessing includes the deduction of additional binary clauses via hyper-resolution [13] and an important landmark approach was presented by Eén and Biere who integrate variable and clause elimination strategies [51].

Järvisalo, Biere and Heule address blocked clause elimination [85, 84] and its power to simulate circuit level simplification techniques [86]. They present an implication graph based inprocessing procedure [76] and present a model that captures many recent inprocessing techniques [87].

Manthey et al. experimented with variable addition techniques [104]. Recently, elimination of redundant literals from clauses through vivification has been addressed in the context of inprocessing [99].

Incremental SAT Solving

Incremental SAT decision procedures are online algorithms that solve a sequence of SAT problems $S = (F_0, F_1, F_2, \dots)$ such that $F_i \subseteq F_{i+1}$ [81, 9]. Research on incremental SAT solving includes the applicability of known optimizations through pre- and inprocessing [110, 58].

In SAT modulo theory (SMT) solvers [118] and other approaches using counter-example guided abstraction refinement (CEGAR) [38], incremental SAT solving is applied for solving a sequence of related problems which are encoded to SAT. Incremental SAT solving has been used to increase the efficiency of SAT-based application, e.g., bounded model checking (BMC) [31, 95], hardware verification [109], AI planning [63] and optimization problems [105].

IPASIR [80] is an interface for incremental SAT solvers which has been developed for SAT Race 2015 [*4] in order to include the new incremental library track in SAT Race 2015 and the following SAT Competitions. In the encoding and solution of incremental SAT problems, special variables are used as *assumption* and *activation literals* [97].

Definition 12 (Assumption Literal)**.** *Given an incremental SAT solver S, an* assumption literal *l is used to define the assignment of the variable $v = \mathsf{var}(l)$, such that v is set to 0 iff $l = \neg v$ and to 1 otherwise. Assumption literals specify temporary facts which hold for a sequence of instances solved by S.*

In order to be able to remove clauses from the problem instance between subsequent incremental solver runs, clauses can be augmented by a so-called *activation literal*. A clause augmented in this way can be *activated* and *deactivated* by setting the value of its activation literal by using assumption literals.

Definition 13 (Activation Literal)**.** *Given an incremental SAT solver S, an* activation literal *l is a literal that is added to a clause or a set of clauses. Such* *clauses are activated (deactivated) when their activation literal is set to* 0 (1) *via an assumption literal* $\neg l$ (l)*.*

Given a sequence $S = (F_0, F_1, F_2, \dots)$ of incrementally solved SAT instances, a clause $C \in F_i$ which contains an activation literal l can be permanently deactivated by adding the unit-clause $\{l\}$ to subsequent SAT instances $F_j, j > i$. As $\{l\}$ subsumes C, incremental SAT solvers delete C during preprocessing.

CDCL Algorithm Outline

In Algorithm 1, we outline CDCL with input formula F and assumption literals K. CDCL accesses and modifies the current assignment A and the set of clauses DB, which is initialized with the clauses in F. If a satisfying assignment A is found, the algorithm returns SAT (line 23), otherwise, if we deduce the empty clause, the algorithm returns UNSAT (lines 3, 6 and 11)

If no conflict is found during preprocessing (lines 1 to 3), we initialize A with assumptions K and execute unit-propagation (line 4). If no conflict is detected (lines 5 and 6), search starts in line 7. During search, a partial assignment is determined by a sequence of branching decisions (line 22) with subsequent unit-propagation (line 8).

If a conflict occurs (line 9) and no branching decision is involved (line 10), the formula F is provably unsatisfiable and search stops (line 11). Otherwise, a conflict clause and backtracking level is derived via conflict analysis (line 13). Subsequently, the decisions having induced the conflict are undone in A (line 14) and the conflict clause is added to the clause database DB (line 15).

Several heuristics control the frequency of common extensions to CDCL such as search restarts (lines 16f.), inprocessing (lines 18f.) or clause forgetting (lines 20f.).

Algorithm 1: Conflict-driven Clause Learning (CDCL)

```
Input: CNF Formula $F$, Assumptions $K$
Output: SAT or UNSAT
Data: current-assignment A $\leftarrow \emptyset$
Data: clause-database DB $\leftarrow F$
1  PREPROCESSING
2  if empty clause in clause-database then
3      return UNSAT
4  Propagate Assumptions $K$
5  if $A$ falsifies a clause in DB then
6      return UNSAT
7  while $A$ is not complete do
8      PROPAGATION
9      if $A$ falsifies a clause in DB then
10         if branching is at level 0 then
11             return UNSAT
12         else
13             (conflict-clause, backtrack-level) $\leftarrow$ CONFLICT-ANALYSIS
14             backtrack to backtrack-level
15             add conflict-clause to clause-database
16     if RESTART is triggered then
17         backtrack to level 0
18     if INPROCESSING is triggered then
19         process clause database
20     if CLEANUP is triggered then
21         forget some learned clauses
22     BRANCHING
23 return SAT
```

The rules of logic are to mathematics what those of structure are to architecture.

Bertrand Russel, The Study of Mathematics

Chapter 3

Gate Recognition

Gate recognition denotes the problem of recovering structural information from CNF encodings of general Boolean formulas and combinational circuits. In this chapter, we devise an efficient and effective gate recognition algorithm which is superior to previous approaches as it is generic. We published an early version of this approach in [*3]. Gate recognition forms the basis for the approaches presented in Chapter 4.

In combinational logic, complex Boolean functions are represented by combinational Boolean circuits. The building blocks of Boolean circuits are Boolean gates representing elementary Boolean functions. In Table 3.1, we show an exemplary list of gate symbols as defined in the IEC 60617-12 standard. Most of these symbols can be extended to n-ary gates by adding more inputs (as in Example 5).

Internally, many SAT-driven applications use variants of Boolean circuit representations. In symbolic model checking [106] commonly used data structures are Reduced Boolean Circuits (RBC) [1] or Binary Decision Diagrams (BDD) [35]. Andersen et al. suggest Binary Expression Diagrams (BED) for a concise and efficient representation of Boolean functions [6]. And-Inverter Graphs (AIG) are applied, e.g., in SAT-based Bounded Model Checking (BMC) [34] or hardware equivalence checking [24]. In the relational model finder Kodkod, a data-structure called Compact Boolean Circuits (CBC) [126] is used. Boolean circuit representations have also been used in the development of one of the most effective CNF encodings of cardinality constraints [123].

The aforementioned data structures can be represented by directed acyclic graphs (DAG) with inner nodes representing operators and leaves that represent literals. The operators of the inner nodes represent different types of Boolean functions and each of the aforementioned data structure uses an individual subset of functions which are allowed for the inner nodes.

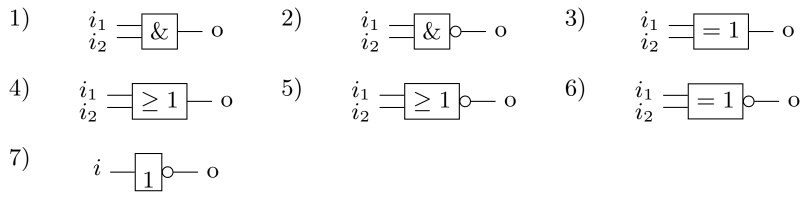


Table 3.1: Symbols of binary Boolean gates as defined in the IEC 60617-12 standard. The symbols represent (from 1 to 7) AND-, NAND-, XOR-, OR-, NOR-, XNOR-, and NOT-gates.

3.1 Related Work

Tseitin introduced the extension rule for the resolution calculus which is the basis for structural CNF encodings used in practice to convert Boolean circuit representations to CNF [128]. Greenbaum et al. later demonstrated its application to clause form translations and devised an optimized structural CNF encoding [67, 112]. Further optimized structural encodings have been developed by Boy de la Tour and Jackson and Sheridan [127, 82].

CNF encodings based on the extension rule, also called Tseitin encodings, have been analyzed with regard to proof complexity by Egly and others [53, 54]. Kullmann generalized the extension rule improving worst case lower bounds for SAT algorithms by introducing the notion of *blocked sets* [94].

Blocked sets are the basis for CNF formula simplification methods which have been shown to simulate circuit simplification methods [85, 86]. Blocked clause decomposition has been used in experimental methods to speed-up SAT solving [75, 15].

Using resolution graph representations of SAT instances, Ostrowski et al. extract `AND`, `OR` as well as binary `EQIV` gates based on sub-graph isomorphism [111]. In the resolution graph for each resolvent, they add an edge and an annotation which indicates whether that resolvent is tautological or not. The authors use the fact that the encodings of `AND`, `OR` and `EQIV`-gates appear as blocked sets and cliques in the resolution graph. However, they perform blocked clause elimination in order to simplify the remaining problem, which indicates they could have detected more gates.

Roy et al. devise an algorithm that explicitly searches for clauses that exhibit the pattern of a specific gate type by looking at literal polarities and occurrences [114]. Their algorithm recognizes `AND`, `OR` and `EQIV`-gates as well as `NAND`, `NOR`, `NOT`, `XOR`, `XNOR`, and `MAJ3`-gates.

Balyo et al. presented a gate recognition algorithm which is based on blocked clause decomposition [15]. We could show in [*3] that their approach uses blocked sets in a less effective way than our gate recognition algorithm.

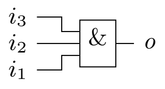
3.2 Gate Structure

In order to extract structural information from CNF formulas, we analyze the properties of structural encodings. The analysis is based on a formalism for gate structure. We start by the definition of a gate.

Definition 14 (Gate). *A gate G over a finite set of variables $\mathbb{X}_G \subseteq \mathbb{X}$ is a tuple (o, P, g) such that $\mathbb{X}_G = \{o\} \cup P$. We denote o as the* output variable *and P as the set of* input variables *of G. The third element is the $|P|$-ary Boolean function $g : \mathbb{B}^{|P|} \to \mathbb{B}$, denoted as the* characteristic function *of G.*

The *arity* n of a gate $G = (o, P, g)$ is determined by the number of input variables $n = |P|$. Furthermore, the *type* of a gate is determined by its characteristic Boolean function g. We emphasize the type of G by calling it a g-gate. A gate $G = (o, P, g)$ is a $|P|$-ary g-gate with output o and inputs P.

Example 5 (`AND`-Gate). *The gate $H = (o, \{i_1, i_2, i_3\}, \texttt{AND})$ is a ternary `AND`-gate with output o and input variables $\{i_1, i_2, i_3\}$. The symbol used in circuit representations after IEC 60617-12 looks as follows.*



The semantics of a gate $G = (o, P, g)$ is specified by its characteristic function g. The interpretation function is specified in Definition 15.

Definition 15 (Gate Semantics)**.** *Given a gate $G = (o, P, g)$, the assignment $\alpha : \mathbb{X}_G \to \mathbb{B}$ is a model of G iff the interpretation function $\mathring{\mathcal{I}}_\alpha(G)$ evaluates to $\top$. The interpretation function $\mathring{\mathcal{I}}_\alpha(G)$ is defined as follows.*[1]

$$\mathring{\mathcal{I}}_\alpha(G) = \begin{cases} \top, \textit{if } \alpha(o) = g\left(\alpha(p_1), \ldots, \alpha(p_n)\right) \\ \bot, \textit{otherwise} \end{cases}$$

Each n-ary gate $G = (o, P, g)$ imposes an $(n{+}1)$-ary functional relation on the variables $\mathbb{X}_G$. A relation is functional iff it is *left-total* and *right-unique* as specified in Definition 16.

Definition 16 (Functional Relation)**.** *A relation $R \subseteq \mathbb{B}^{n+1}$ is functional iff the following two formulas hold.*

$$\forall P \in \mathbb{B}^n, \exists o \in \mathbb{B} : (p_1, \ldots, p_n, o) \in R \qquad \text{(left-totality)}$$
$$\forall P \in \mathbb{B}^n, \exists o \in \mathbb{B} : (p_1, \ldots, p_n, o) \notin R \qquad \text{(right-uniqueness)}$$

Given a gate G, there exists a bijection between models of G and a functional relation R, such that for each tuple $(p_1, \ldots, p_{n+1}) \in R$ the corresponding model M_α of G is constructed such that $p_i \in M_\alpha$ iff $p_i = 1$ and $\neg p_i \in M_\alpha$ iff $p_i = 0$.

Structural Formulas

Our gate recognition algorithm starts with a given CNF formula F and an empty set of gates $\Gamma = \emptyset$. Whenever an encoding $E \subseteq F$ of a gate G is detected, we remove the clauses E from F and add G to Γ. In the following, we formalize this process starting with Definition 17.

Definition 17 (Structural Formula)**.** *A* structural formula *is a tuple (F, Γ) with a finite set of clauses F and a finite set of gates Γ.*

The semantics of structural formulas combines the semantics of propositional formulas and gate semantics.

Definition 18 (Structural Formula Semantics)**.** *Given a structural formula $S = (F, \Gamma)$ over variables $\mathbb{X}_S = \mathbb{X}_F \cup \{v \mid v \in \mathbb{X}_G, G \in \Gamma\}$, the assignment $\alpha : \mathbb{X}_S \to \mathbb{B}$ is a model of S iff $\mathcal{I}_\alpha(F) = \top$ and $\forall G \in \Gamma : \mathring{\mathcal{I}}_\alpha(G) = \top$.*

Given a structural formula $S = (F, \Gamma)$, its set of models $\mathcal{M}(S)$ is defined as follows.

$$\mathcal{M}(S) := \{M_\alpha \mid \mathcal{I}_\alpha(F) = \top, \forall G \in \Gamma : \mathring{\mathcal{I}}_\alpha(G) = \top\}$$

For establishing the notion of equivalence of structural formulas, we have to take projection to its input variables into account, as the Plaisted Greenbaum encoding is not preserving equivalence. The set of input variables $\mathsf{inp}(S)$ of a structural formula $S = (F, \Gamma)$ is defined as follows.

$$\mathsf{inp}(S) := \mathbb{X}_S \setminus \{o \mid (o, P, g) \in \Gamma\}$$

In Chapter 2, we defined model projection for propositional formulas. Similarly, we introduce model projection for structural formulas such that $\mathcal{M}(S, \mathsf{inp}(S))$ denotes the projection of models of S to its input variables.

[1]Note that $i < j \implies p_i <^{\mathbb{X}} p_j$ for all $p_i, p_j \in P$ as specified in Chapter 2.

Definition 19 (Structural Formula Equivalence)**.** *Given two structural formulas S_1, S_2 with a* non-empty *intersection of input variables $I = \mathsf{inp}(S_1) \cap \mathsf{inp}(S_2)$, then S_1 and S_2 are equivalent, denoted by $S_1 \cong S_2$, iff their sets of models projected to I are equivalent, i.e., $\mathcal{M}(S_1, I) = \mathcal{M}(S_2, I)$.*

Based on structural formula semantics, we now specify the coherence of structural encodings and gates in Definition 20.

Definition 20 (Gate Encodings)**.** *A CNF formula E encodes a gate G iff $\mathbb{X}_E = \mathbb{X}_G$ and $(E, \emptyset) \cong (\emptyset, \{G\})$.*

Given a structural formula $S = (F, \Gamma)$ over variables $\mathbb{X}_S$, a variable $v \in \mathbb{X}_S$ can be output of a gate $G \in \Gamma$ and at the same time input to several other gates $H \in \Gamma$. This induces the nesting relation $<$ which is a transitive, irreflexive and asymmetric relation of gates, thus, imposing a strict partial order on Γ.

Definition 21 (Nesting of Gates)**.** *Given two gates $G = (o_g, P_g, g)$ and $H = (o_h, P_h, h)$, G is* directly nested *in H, denoted by $G < H$, iff $o_g \in P_h$. Furthermore, G is* nested *in H, denoted by $G <^+ H$, iff it is in the transitive closure of $<$.*

Given a set of gates Γ, we call the maximal elements of its nesting relation the *output gates* of Γ. Similarly, we call the minimal elements of $<$ the *input gates* of Γ. Furthermore, we call the union of output variables of output gates the *output variables* of Γ and the union of input variables of input gates the *input variables* of Γ.

Definition 22 (Monotonicity)**.** *A function $g(p_1, \ldots, p_i, \ldots, p_n)$ is* monotonically increasing *in argument i iff $a \leq b$ implies $g(p_1, \ldots a, \ldots p_n) \leq g(p_1, \ldots b, \ldots p_n)$, and* monotonically decreasing *in argument i iff $a \leq b$ implies $g(p_1, \ldots a, \ldots p_n) \geq g(p_1, \ldots b, \ldots p_n)$. A Boolean function is monotonic iff it is monotonically increasing or decreasing in every argument.*

We define a gate to be monotonic iff its characteristic function is monotonic. Examples for monotonic functions are `AND` and `OR`, and examples for non-monotonic functions are `XOR` and `EQIV`. In Definition 23 we define the notion of monotonic nesting, allowing further insight into optimized gate encodings.

Definition 23 (Monotonic Nesting)**.** *Let Γ be a set of gates with $G, H \in \Gamma, G = (o_g, P_g, g), H = (o_h, P_h, h)$ such that $G < H$. G is monotonically nested in H iff o_g is a monotonic argument in h. G is monotonically nested in Γ iff $\forall I, J \in \Gamma$ with $I < J$ such that $G <^+ I$ it holds that I is monotonically nested in J.*

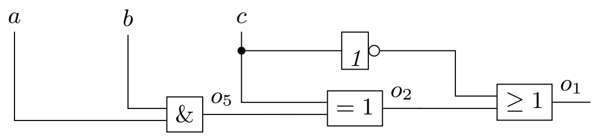
Plaisted and Greenbaum devised an optimized encoding for monotonically nested gates [112] as defined in Equations 2.23, where implications can be used whenever a gate is monotonically nested. Given a formula F, its Plaisted Greenbaum encoding ${}^{PG}\mathcal{E}(F)$ allows for additional models which are not models of the Tseitin encoding ${}^{T}\mathcal{E}(F)$ (assuming both use the same encoding variables, i.e., $\mathsf{vars}({}^{PG}\mathcal{E}(F)) = \mathsf{vars}({}^{T}\mathcal{E}(F))$).

Given such an additional model $M \in \mathcal{M}({}^{PG}\mathcal{E}(F)) \setminus \mathcal{M}({}^{T}\mathcal{E}(F))$, there exists a set of literals $L \subset M$ such that the set $M' := (M \setminus L) \cup \{\bar{l} \mid l \in L\}$ is a model of ${}^{PG}\mathcal{E}(F)$ and ${}^{T}\mathcal{E}(F)$, the variables in $\mathsf{vars}(L)$ are output variables of monotonically nested gates.

Example 6 (Combinational Gate Structure)**.** *In Equation 3.1, we show a propositional formula which is also illustrated by the combinational circuit below. The circuit in our example consists of a root-level* `OR`*-gate and subsequently nested* `XOR`*- and* `AND`*-gates which is also illustrated in the circuit below. In Equation 3.2 the* `XOR` *is replaced by an equivalent representation using* `AND` *and* `OR` *functions. The*

Plaisted Greenbaum encoding of the such obtained formula is successively derived in Equations 3.3 and 3.4.

$$
\begin{aligned}
&\neg c \lor (c \oplus (a \land b)) && (3.1)\\
\equiv\ &\neg c \lor \big((\neg c \lor \neg(a \land b)) \land (c \lor (a \land b))\big) && (3.2)\\
\cong\ &o_1 \land \big(o_1 \rightarrow (o_2 \lor \neg c)\big) \land (o_2 \rightarrow o_3 \land o_4)\\
&\land (o_3 \rightarrow \neg c \land \neg o_5) \land (o_4 \rightarrow c \land o_5) \land (o_5 \leftrightarrow a \land b) && (3.3)\\
\equiv\ &o_1 \land (\neg o_1 \lor o_2 \lor \neg c) \land (\neg o_2 \lor o_3) \land (\neg o_2 \lor o_4)\\
&\land (\neg o_3 \lor \neg c) \land (\neg o_3 \lor \neg o_5) \land (\neg o_4 \lor c) \land (\neg o_4 \lor o_5)\\
&\land (\neg o_5 \lor a) \land (\neg o_5 \lor b) \land (o_5 \lor \neg a \lor \neg b) && (3.4)
\end{aligned}
$$



Left-Totality of Gate Encodings

From the functional relation which is imposed by a gate and which is reflected in gate encodings, we deduce additional properties of CNF encodings of gates. Proposition 1 shows that the semantics of blocked sets captures left-totality of variable relations.

Proposition 1 (Left-Totality of Gate Encodings)**.** *Given a gate G with output variable o and its encoding E, it holds that for every clause $C \in E$ either $o \in E$ or $\overline{o} \in E$ (part 1) and all resolvents in $E[o] \otimes_o E[\overline{o}]$ are tautologic (part 2).*

Proof of Part 1. Let E be the CNF encoding of a gate $G = (o, P, g)$. Assume that there is a clause $C \in E$ such that $o \notin \mathsf{vars}(C)$. It follows that there exists an assignment to input variables P which falsifies C for any assignment to o. This contradicts left-totality. Thus, it follows that $\forall C \in E : o \in \mathsf{vars}(C)$. □

Proof of Part 2. Let R be a non-tautological resolvent in $E[o] \otimes_o E[\overline{o}]$. By Definition of resolution, it holds that $o \notin \mathsf{vars}(R)$ and $E \models R$. It follows that there exists an assignment to input variables P which falsifies R for any assignment to o. This contradicts left-totality. Therefore, each resolvent in $E[o] \otimes_o E[\overline{o}]$ is tautologic. □

3.3 Gate Recognition

Given a structural formula (F, Γ), gate recognition is a set of rules which we use to subsequently identify encodings $E \subseteq F$ of gates $G = (o, P, g)$, and create the new structural formula $(F \setminus E, \Gamma \cup \{G\}) \cong (F, \Gamma)$.

Given a set of gate recognition rules GR, we denote the application of a rule in GR by $(F, \Gamma) \xrightarrow{\text{GR}} (F', \Gamma')$. For rules which should only be applied if a condition K holds, we use the notation $K \mid (F, \Gamma) \xrightarrow{\text{GR}} (F', \Gamma')$ [12, 46].

The reflexive transitive *closure* of $\xrightarrow{\text{GR}}$ is denoted by $\xrightarrow{\text{GR}^*}$. A structural formula (F, Γ) is *irreducible* regarding GR if no rule in GR can be applied. If (F, Γ) is irreducible in GR, F is called the *remainder* of $\xrightarrow{\text{GR}^*}$. If Γ contains exactly one output gate with output literal o such that $F = \{\{o\}\}$, then (F, Γ) is called *generally irreducible.*

Gate recognition rules must be constructed such that $\sum_{C \in F'} |C| < \sum_{C \in F} |C|$ and $\Gamma' \supset \Gamma$, i.e., the size of the conjunctive formula strictly decreases and the number of gates strictly increases under repeated application of the rules such that the procedure is terminating.

However, a set of gate recognition rules is not necessarily confluent. The set of rules determines the power of gate recognition. Some recognition rules can only be used to recognize a limited set of gate types.

Definition 24 (Validity of Gate Recognition)**.** *A set of gate recognition rules* GR *is valid iff for every rule application* $A \xrightarrow{GR} B$ *it holds that* $A \cong B$*.*

Example 7 (`AND` Gate Recognition)**.** *Given a CNF formula* F *with a set of clauses* $E \subseteq F$ *such that* $E = \{\{\neg o, p_1\}, \{\neg o, p_2\}, \{o, \neg p_1, \neg p_2\}\}$*. Clearly,* E *is a binary* `AND`*-encoding with output* o*, i.e.,* $E \equiv (o \leftrightarrow (p_1 \wedge p_2))$*.*

Let $\xrightarrow{GR}$ *be a gate recognition procedure which recognizes* `AND`*-encodings, then*

$$\big(F, \{\}\big) \xrightarrow{GR} \big(F \setminus C, \{(o, \{p_1, p_2\}, \texttt{AND})\}\big)$$

is a valid transformation in the sense of Definition 24.

Trivial Gate Recognition

A CNF Formula is a conjunction of disjunctions such that we can trivially convert a given CNF formula F to a set of `OR`-gates whose outputs are then combined in one `AND`-gate. The trivial gate recognition rules TGR are defined as follows.

For each clause C, we introduce a fresh output variable $\mathcal{X}^*_C$ in order to create an `OR`-gate with output $\mathcal{X}^*_C$. Similarly, we introduce an output variable $\mathcal{X}^*_F$ in order to create an `AND`-gate with output $\mathcal{X}^*_F$.

$$\begin{aligned}
\exists C \in F, |C| > 1 \;\big|\; (F, \Gamma) &\xrightarrow{\text{TGR}} \Big(F \setminus C \cup \{\{\mathcal{X}^*_C\}\}, \Gamma \cup \{(\mathcal{X}^*_C, \mathsf{vars}(C), \texttt{OR})\}\Big) \\
\forall C \in F, |C| = 1, |F| > 1 \;\big|\; (F, \Gamma) &\xrightarrow{\text{TGR}} \Big(\{\{\mathcal{X}^*_F\}\}, \Gamma \cup \{(\mathcal{X}^*_F, \mathsf{vars}(F), \texttt{AND})\}\Big)
\end{aligned}$$

Note that the TGR rules are conditional, ensuring that we first rewrite all clauses C with $|C| > 1$ before we apply the second rule once. The resulting structural formula is generally irreducible. In the following, given a set of gate recognition rules GR and a structural formula S which is irreducible in GR, we can subsequently apply TGR rules in order to achieve general irreducibility. Algorithms which process gates, such as those presented in Chapter 4, can often be simplified when we assert generally irreducibility of the processed input formula.

Hierarchical Gate Recognition

Given a CNF formula F that has been created using a structural encoding, there exists a hidden gate structure and a nesting hierarchy. Starting with the output gates of the gate structure which is encoded in F, we hierarchically search for gate encodings by considering the resolution environments $F[o] \cup F[\overline{o}]$ of candidate output variables o.

For each recognized gate (o, P, g), we add new input variables P to the gate structure and remove its output variable o from the input variables of the gate structure. For each candidate gate encoding E, we apply the following general gate recognition rule.

$$\exists E \subseteq F : \big(E, \emptyset\big) \cong \big(\emptyset, \{(o, I, g)\}\big) \;\big|\; (F, \Gamma) \xrightarrow{\text{GR}} (F \setminus E, \Gamma \cup \{(o, I, g)\})$$

Algorithm 2: Root Selection

```
   Data: $(F, \Gamma) \in \Psi$
1  $S \leftarrow \emptyset$
2  $R \leftarrow \{C \in F \mid |C| = 1\}$
3  $(F, \Gamma) \leftarrow$ hierarchicalGateRecognition($(F \setminus R, \Gamma)$, lits($R$))
4  $S \leftarrow S \cup R$
5  $R \leftarrow$ selectUnblockingClauses($F$)
6  while $R \neq \emptyset$ do
7      $(F, \Gamma) \leftarrow$ hierarchicalGateRecognition($(F \setminus R, \Gamma)$, lits($R$))
8      $S \leftarrow S \cup R$
9      $R \leftarrow$ selectUnblockingClauses($F$)
10 return $(F \cup S, \Gamma)$
```

Root Selection

As the maximal gates of the gate structure to be decoded are unknown in advance, we use heuristics to determine candidate output variables of possible output gates. In Algorithm 2, we heuristically select clauses and treat their literals as candidate gate outputs in our hierarchical procedure. In the following we denote those clauses as *root clauses*.

In non-simplified CNF formulas F generated by gate encoders, the output variable of the maximal gate is a unit clause F. Therefore, unit clauses are generally promising candidates for root selection (line 2).

In absence of unit clauses, we continuously select candidate root clauses in F in such a way that at least one blocked set emerges in F (lines 5 and 9). Further details and an efficient data structure for this root selection heuristic is explained at the end of this section.

We keep the root clauses in a separate list (lines 1, 4 and 8) in order to add them to the remainder of the gate recognition procedure (line 10).

Hierarchical gate recognition is performed with the literals of the selected root clauses (lines 3 and 7). Note that hierarchical gate recognition modifies and returns the structural formula (F, Γ).

Hierarchical Gate Recognition

Hierarchical gate recognition is implemented as a breadth first search. The procedure is outlined in algorithm 3. In the inner loop (line 4), we iterate the root literals in order to perform gate recognition for each root literal o. First (in line 5), we determine whether o is nested monotonically in the gate structure decoded so far. In line 6, we use decodeGate to test if the resolution environment of o in F is a gate encoding with output o (see Algorithm 4 for details).

If decodeGate successfully recognizes a gate encoding (line 7), we add (o, P, g) to the set of gates Γ (line 8) and remove the resolution environment of o from the set of clause F (line 9). New input literals are appended to list L (lines 10 and 11) to serve as new root literals in the next iteration of the outer loop (lines 16, 2 and 3).

In lines 1 and 13, we keep track of the input literal polarities of the previously recognized gates. This is an efficient method for tracking non-monotonic arguments of the decoded gates, as the presence of input literals in both polarities is a necessary criterion for non-monotonicity in direct gate encodings (line 5). Note that in line 15, we transitively pass on non-monotonicity to nested gates.

Algorithm 3: Hierarchical Gate Recognition

```
Input: $(F, \Gamma) \in \Psi$, roots $\subseteq$ lits$(F)$
Output: $(F, \Gamma) \in \Psi$
1  for $l \in$ roots do setAsInput($l$)
2  while roots $\neq \emptyset$ do
3      $L \leftarrow \emptyset$
4      for $o \in$ roots do
5          monotonic $\leftarrow \neg$isSetAsInput($o$) $\vee$ $\neg$isSetAsInput($\overline{o}$)
6          $(o, P, g) \leftarrow$ decodeGate($F$, $o$, monotonic)
7          if $(o, P, g) \neq \bot$ then
8              $\Gamma \leftarrow \Gamma \cup (o, P, g)$
9              $F \leftarrow F \setminus (F[o] \cup F[\overline{o}])$
10             $L' \leftarrow$ lits$(F[\overline{o}]) \setminus \{o, \overline{o}\}$
11             $L \leftarrow L \cup L'$
12             for $l \in L'$ do
13                 setAsInput($l$)
14                 if $\neg$monotonic then
15                     setAsInput($\overline{l}$)
16     roots $\leftarrow L$
17 return $(F, \Gamma)$
```

Decode Gate

As we have seen in Proposition 1, blockedness is a necessary criterion for left-totality of gate encodings. Using the argumentation in Proposition 1, it is easy to see that blockedness is also a sufficient criterion for left-totality of gate encodings.

In the following, we use this fact to discover candidate gate encodings. Furthermore, we relate monotonic nesting and the possibility to skip right-uniqueness proofs with the optimized Plaisted-Greenbaum encoding ${}^{PG}\mathcal{E}$. This leads to an incremental algorithm that subsequently discovers partial candidate gate-encodings by keeping track of monotonicity and using right-uniqueness proofs when necessary.

Given a formula F and a candidate output o, we distinguish *forward clauses* fwd (line 1) and *backward clauses* bwd (line 2) in the resolution environment of

Algorithm 4: Decode Gate

```
Input: $F \in \Phi'$, $o \in$ lits$(F)$, monotonic
Output: Gate $(o, P, g)$
1  fwd $\leftarrow F[\overline{o}]$
2  bwd $\leftarrow F[o]$
3  inputs $\leftarrow$ vars(fwd) $\setminus \{o\}$
4  if fwd *and* bwd *are blocked on* $o$ then
5      if monotonic *is true* then
6          return ($o$, inputs, anon(fwd $\cup$ bwd))
7      else if isRightUnique($F$, $o$, fwd $\cup$ bwd) then
8          return ($o$, inputs, anon(fwd $\cup$ bwd))
9  return $\bot$
```

Algorithm 5: isRightUnique

```
Input: Formula $F$, Output $o$, Clauses $E$
Output: true if right-unique, false otherwise
Configuration : Strategy $\in$ {Patterns, Semantic, Holistic}
1 switch Strategy do
2     case Patterns do
3         return $E$ matches AND, OR or full pattern with output $o$
4     case Semantic do
5         $\mathcal{C} = \{C \setminus \{o, \overline{o}\} \mid C \in E\}$
6         return $\mathcal{C}$ is UNSAT
7     case Holistic do
8         $\mathcal{C} = \{C \setminus \{o, \overline{o}\} \mid C \in E\} \cup \{C \in F \mid C \cap \mathsf{lits}(E) \neq \emptyset\}$
9         return $\mathcal{C}$ is UNSAT
```

o in F. Literals in fwd other than $\overline{o}$ are the input literals of the candidate gate (line 3). If forward and backward clauses are blocked on o, we detected a left-total relationship between o and the inputs.

If the nesting is monotonic (as indicated by the flag calculated in hierarchical gate recognition), an *anonymous* gate with output o and inputs inputs is returned (line 6). An *anonymous gate* is a gate of which we have not decoded the encoded function. However, the implementation keeps references to the clauses in `fwd` $\cup$ `bwd`. In our algorithm, we use the expression `anon(fwd` $\cup$ `bwd)` to comply with the formalism, which requires a function as the third element of the triple which defines the gate.

If the nesting is non-monotonic, we test the encoding for right-uniqueness (line 7) by using one of the methods described in Algorithm 5.

The blockedness check in our implementation has quadratic time complexity in maximum clause lengths of any two clauses C_1 and C_2 occurring in the input. Depending on the number of tests this might induce large execution time costs for some problem instances and can be optimized by using blockedness checks of linear time complexity with sorted clauses based on a fixed literal ordering which also ensures that all literals v and $\neg v$ directly follow one another.[2]

In our implementation, we keep a literal occurrence list which is updated when the clauses of successfully decoded gates are removed from F. If a gate G is nested in many gates $H_i > G, i \in \mathbb{N}$, the same resolution environment can be tested multiple times for blockedness before G can be decoded. To limit the amount of redundant blockedness tests we cache the result of each such test and recalculate the result locally whenever the occurrence list is updated.

Right-Uniqueness

In order to check for right-uniqueness of variables occurring in the blocked sets $F[o], F[\overline{o}]$, we use known patterns of the Tseitin encodings of a set of common functions. This is similarly limited as previous approaches which search for such patterns globally [111, 114]. Due to its efficiency, we evaluate this approach and compare its performance to that of our generic approach which is described later in this Section.

Given a formula F and a candidate gate output o such that $F[o]$ and $F[\overline{o}]$ are blocked, we use `fwd`, `bwd` and `inputs` as defined in Algorithm 4. Under the precondition that `fwd` and `bwd` are blocked on o, we only need a few checks to test

[2]The approach is already very efficient, so we did not sort clauses in our implementation.

for some patterns. We also assume that neither redundant literals nor redundant clauses occur in `fwd` and `bwd`.

The `AND`/`OR` Pattern

Given that `fwd` and `bwd` are blocked on o, if $|\texttt{fwd}| = 1$ and $\forall c \in \texttt{bwd} : |c| = 2$, the clauses are an `OR`- or `NAND`-gate encoding, and if $|\texttt{bwd}| = 1$ and $\forall c \in \texttt{fwd} : |c| = 2$, the clauses are an `AND`- or `NOR`-gate encoding. Whether the clauses are an `OR`- or `NAND` gate encoding (or an `AND`- or `NOR`-gate encoding, respectively) depends on the polarity of o.

The Full Pattern

Let $n = |\mathsf{vars}(\texttt{inputs})|$. Given that `fwd` and `bwd` are blocked on o, if $|\texttt{fwd}|+|\texttt{bwd}| = 2^n$ and $\forall C \in \texttt{fwd} \cup \texttt{bwd}, |C| = n+1$, then from blockedness and absence of redundancy follows that for each of the 2^n possible assignments to $\mathsf{vars}(\texttt{inputs})$ there exists exactly one assignment to o and exactly one clause $C \in \texttt{fwd} \cup \texttt{bwd}$ such that C is falsified. The encodings of many gate-types satisfy this pattern, some of which are `NOT`, `XOR`, `XNOR` or `MAJ3`.

For some gate encodings which are detected by this pattern, it holds that literal could be the output literal due to the symmetry of their encoding. Previous approaches use special post-processing algorithms which consider the nesting structure to determine which literal is the output literal.

Due to hierarchical gate recognition, breadth first search and incremental blockedness checks we do not need this kind of post-processing. In our approach, as the output is determined by the so far decoded gate structure, the recognized variation can safely be integrated into the recognized gate structure.

Semantic Right-Uniqueness Check

We have proven that a set of clauses E is blocked on a variable $o \in \mathsf{vars}(E)$ iff it encodes a left-total relation between o and $P = \mathsf{vars}(E) \setminus \{o\}$. What remains in order to prove that E encodes a functional relation of o on P, is determination of right-uniqueness.

In the previously presented heuristics, we recurred on clausal patterns of known encodings. To overcome this, we devise a generic method, that is capable of detecting right-unique relations independently of their clausal patterns. In the following, we reformulate the criterion for right-uniqueness.

$$\begin{aligned} & \forall A \in \mathcal{A}(P), \exists l \in \{o, \bar{o}\} : A \cup \{l\} \models \neg E \\ \Longrightarrow\ & \neg\exists A \in \mathcal{A}(P), \forall l \in \{o, \bar{o}\} : A \cup \{l\} \models E \\ \Longrightarrow\ & \neg\exists A \in \mathcal{A}(P) : A \models E|_{o=0} \wedge E|_{o=1} \end{aligned}$$

This can be efficiently transformed to a SAT instance as shown in Equation 3.5. Obviously, the encoding time for such SAT instance is linear in the size of E.

$$\not\models E|_{o=0} \wedge E|_{o=1} \tag{3.5}$$

Similarly, we obtain a propositional formula which we can use to test for left-totality. As can be seen in Equation 3.6, the construction of the CNF formula involves the negation of a set of clauses. As shown in Examples 2, 3 and 4 in Section 2.1, the negation of a set of clauses can be achieved by methods of direct encoding or by using a structural encoding.

$$\not\models \neg E|_{o=0} \wedge \neg E|_{o=1} \tag{3.6}$$

Occurrence Lists with Blocking Counters

As indicated in Algorithm 2, in order to devise an effective repeated root selection heuristic, we use the new data structure *occurrence lists with blocking counters*. Given a CNF formula F, for each literal l in $\mathsf{lits}(F)$ we keep an occurrence list of clause $F[l]$. In addition, we keep track of a blocking counter $c[l]$, where we store the number of clauses in $F[l]$ which are blocked on l.

During hierarchical gate recognition, clauses recognized as gate encodings are repeatedly removed from F. Whenever a clause C is removed from F, the occurrence lists $F[l]$ for literals $l \in C$ are updated. Likewise, we update the blocking counters $c[l]$ and $c[\bar{l}]$ for each literal $l \in C$.

As a result of keeping track of blocking counters, we can efficiently test if $F[l]$ is blocked on l. If $c[l]$ equals the size of $F[l]$, each remaining clause in $F[l]$ is blocked by $F[\bar{l}]$ and vice versa.

Furthermore, this optimization allows for an efficient and effective root selection heuristic. During root selection, we select a literal l with minimal blocking counter $c[l]$. Then we select all clauses in $F[l]$ which are not blocked by $F[\bar{l}]$ and use them as root clauses for hierarchical gate recognition. This is how we ensure that at least $F[l]$ and $F[\bar{l}]$ are blocked when we start decoding gates with root literal l.

Sub-Gate Encodings

Some gate encodings are optimized, such that multiple gates with shared input variables might share a substructure of the encoding via additional encoding variables. This type of variable addition reduces the formula size and might result in smaller resolution proofs, but in the presence of such optimizations our method of local blockedness and right-uniqueness checks fails.

However, by using semantic checks for both *right-uniqueness* and *left-totality*, we could fix our method if we included the clauses in the resolution environments of the candidate inputs. This *single-level look-ahead* in the gate structure includes all clauses which in combination form a gate encoding.

However, the CNF formulas in this single-level look-ahead recognition method are likely to become infeasible for gate recognition where thousands of these checks are executed. Possible optimizations based on size-limits and other trade-offs have not been evaluated in this dissertation.

3.4 Conclusion

Hierarchical gate recognition allows for the recognition of monotonically nested gates in order to recognize both Tseitin and Plaisted Greenbaum optimized encodings. The beauty of gate recognition via blocked sets and semantic right-uniqueness checks lies in its genericity, as it decouples recognition of functional variable relations from the recognition of a specific gate type.

Note that each gate structure can have at most one monotonic root structure and if it is not fully monotonic, it must yet contain fully non-monotonic substructures. This follows from the transitivity of non-monotonic nesting. Moreover, we observed that Plaisted Greenbaum encodings of monotonic gates are isomorphic to horn problems. In Plaisted Greenbaum encodings of monotonic gates, the output literal of each gate encoding can be flipped such that it is the only positive literal in its encoding clauses.

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

Antoine de Saint-Exupéry, Terre des Hommes

Chapter **4**

Exploiting Gate Structure

In this chapter, we leverage gate structure extracted by the previously described algorithm in three exemplary approaches. In Section 4.1, we use gate structure in order to improve an algorithm that *minimizes models*. An early version of the minimization algorithm is published in [*2].[1]

The other two approaches, described in Sections 4.2 to 4.4, have been originally presented by Felix Kutzner in his diploma thesis in [96] and are published in [*5]. In Section 4.2 we use gate structure to perform *random simulation* in order to calculate conjectures about equivalent and constant gate output variables.

We effectively use these conjectures in two different ways. In Section 4.3, we use *abstraction via under-approximation* to solve SAT problems exhibiting gate structure. In a second approach, presented in Section 4.4, the conjectures are used to direct the search in a new branching heuristic based on *implicit learning*.

4.1 Model Minimization

Model minimization is the process of generating satisfying partial assignments from a satisfying full assignment. Minimization of a found model can be beneficial for many applications of SAT solvers. In software verification, minimization of a counter-example can be used to increase explainability by shifting the focus to the relevant parts of the complete assignment. In prominent approaches like Counter-example Guided Abstraction Refinement (CEGAR) [37, 130] and DPLL(T) [60, 33], model minimization can be employed to reduce the number of spurious counter-examples, thus, reducing the number of refinement loops.

In the following, we use the interpretation function depicted in Figure 4.1 in order to evaluate partial assignments $\alpha : \mathbb{X}_F \rightsquigarrow \mathbb{B}$ for a formula F. The universe of the modified interpretation is extended by the third value $\mathbf{U}$ signifying "unknown". Given a formula F, the evaluation $\widetilde{\mathcal{I}}_\alpha(F)$ of partial assignments is used in Definition 25 to classify the set of satisfying partial assignments that can be derived from a given full model.

Definition 25 (Model Minimization)**.** *Given a model $\alpha \models F$, a model $\beta \models F$ is called* α-minimized *if $M_\beta \subseteq M_\alpha$. The partial model β is* α-minimal *if no further subset $M_\gamma \subset M_\beta$ is a model of F. Furthermore, β is called* α-minimum *if there exists no other* α-minimal *partial assignment γ with $|M_\gamma| < |M_\beta|$. We omit α if α is clear from the context.*

The minimization of a model M_α for a formula F can be translated to a hitting set problem, in which we ask for a hitting set $M \subseteq M_\alpha$ for the clauses in F.

[1] Due to a bug in the first version of our gate recognition algorithm, the evaluation of our minimization approach described in [*2] contains erroneous results which has been fixed in the version used in the evaluation presented in this work.

$$\widetilde{\mathcal{I}}_\alpha(0) = \bot \qquad \widetilde{\mathcal{I}}_\alpha(F \to G) = \widetilde{\mathcal{I}}_\alpha(\neg F \vee G)$$

$$\widetilde{\mathcal{I}}_\alpha(1) = \top \qquad \widetilde{\mathcal{I}}_\alpha(F \leftrightarrow G) = \widetilde{\mathcal{I}}_\alpha((F \to G) \wedge (G \to F))$$

$$\widetilde{\mathcal{I}}_\alpha(v) = \begin{cases} \top, & \text{if } \alpha(v) = 1 \\ \bot, & \text{if } \alpha(v) = 0 \\ \mathbf{U}, & \text{otherwise} \end{cases} \qquad \widetilde{\mathcal{I}}_\alpha(F \wedge G) = \begin{cases} \top, & \text{if } \widetilde{\mathcal{I}}_\alpha(F) = \top \text{ and } \widetilde{\mathcal{I}}_\alpha(G) = \top \\ \bot, & \text{if } \widetilde{\mathcal{I}}_\alpha(F) = \bot \text{ or } \widetilde{\mathcal{I}}_\alpha(G) = \bot \\ \mathbf{U}, & \text{otherwise} \end{cases}$$

$$\widetilde{\mathcal{I}}_\alpha(\neg F) = \begin{cases} \top, & \text{if } \widetilde{\mathcal{I}}_\alpha(F) = \bot \\ \bot, & \text{if } \widetilde{\mathcal{I}}_\alpha(F) = \top \\ \mathbf{U}, & \text{otherwise} \end{cases} \qquad \widetilde{\mathcal{I}}_\alpha(F \vee G) = \begin{cases} \top, & \text{if } \widetilde{\mathcal{I}}_\alpha(F) = \top \text{ or } \widetilde{\mathcal{I}}_\alpha(G) = \top \\ \bot, & \text{if } \widetilde{\mathcal{I}}_\alpha(F) = \bot \text{ and } \widetilde{\mathcal{I}}_\alpha(G) = \bot \\ \mathbf{U}, & \text{otherwise} \end{cases}$$

Figure 4.1: Interpretation of a formula under a *partial* assignment α

Definition 26 (Hitting Set Problem). *Given a finite set of symbols Σ and a set $S \subseteq 2^\Sigma$ the hitting set problem is the problem of determining a minimal subset of Σ which contains an element of each set in S.*

Finding a hitting set $M \subseteq M_\alpha$ for F is equivalent to the problem of finding a hitting set for the purified formula $\mathsf{purify}_\alpha(F)$, which we obtain by removing all literals from F which are not satisfied by α.

Definition 27 (Purification). *Given a formula F and a model $M_\alpha \models F$, the purified formula $\mathsf{purify}_\alpha(F)$ is defined as follows.*

$$\mathsf{purify}_\alpha(F) = \{C \cap M_\alpha \mid C \in F\}$$

Given a formula F and a model $M_\alpha \models F$, for all models $M' \subseteq M_\alpha$ it holds that $M' \models F$ iff $M' \models \mathsf{purify}_\alpha(F)$.

Definition 28 (Flip). *Given a set of literals L and a model α, the method $\mathsf{flip}_\alpha(F)$ flips the polarity of all literals in L whose variable is negated under α.*

$$\mathsf{flip}_\alpha(L) = \{l \mid l \in L, \alpha(\mathsf{var}(l)) = 1\} \cup \{\bar{l} \mid l \in L, \alpha(\mathsf{var}(l)) = 0\}$$

We extend $\mathsf{flip}_\alpha()$ such that it can be applied to formulas as follows. Given a formula F and a model α then $\mathsf{flip}_\alpha(F) = \{\mathsf{flip}_\alpha(C) \mid C \in F\}$.

Proposition 2 (Model Preservation). *Given a formula F and a model M_α, it holds for all models M' that $M' \models F$ iff $\mathsf{flip}_\alpha(M') \models \mathsf{flip}_\alpha(F)$.*

$\Longrightarrow$. Let $C \in F$ and $\mathsf{flip}_\alpha(C) \in \mathsf{flip}_\alpha(F)$. Given a model M', such that $M' \models C$, i.e., $C \cap M' \neq \emptyset$, then $\mathsf{flip}_\alpha(C \cap M') \neq \emptyset$ such that $\mathsf{flip}_\alpha(C) \cap \mathsf{flip}_\alpha(M') \neq \emptyset$. It follows that $\mathsf{flip}_\alpha(M') \models \mathsf{flip}_\alpha(C)$. The argument for $M' \not\models C$ works analogously. ☐

$\Longleftarrow$. We can reuse the argumentation from above because $\mathsf{flip}_\alpha()$ is an involution, i.e., $\mathsf{flip}_\alpha(\mathsf{flip}_\alpha(F)) = F$. ☐

Iterative Minimization

Given a formula F and a model $M_\alpha \models F$, we first compute an α-minimum model M' of $F' = \mathsf{flip}_\alpha(\mathsf{purify}_\alpha(F))$. Then, from M' we construct the model $M = \mathsf{flip}_\alpha(M')$. M is an α-minimum model of F.

In order to minimize the number of variables assigned true, we encode a *cardinality constraint* $\mathtt{cc}(\mathsf{vars}(F'), k)$ and solve the problem $F' \cup \mathtt{cc}(F', k)$. We start with an upper bound $k = |\mathsf{vars}(F)|$ and decrement k iteratively as long as a solution with that bound exists by using incremental SAT solving. In order to encode cardinality constraints, we use the parallel encoding presented by Sinz in [123]

Algorithm 6: Iterative Minimization

```
   Input: Formula $F$, complete model $\alpha$ of $F$
   Output: Minimized model $\alpha_{\min}$ as a set $M_{\alpha_{\min}}$ of literals
1  $F' = \mathsf{flip}_\alpha(\mathsf{purify}_\alpha(F))$
2  $M' = \mathsf{flip}_\alpha(M)$
3  repeat
4      $M = M'$
5      $k = |M \cap \mathbb{X}|$
6      $C = \mathtt{cc}(\mathbb{X}, k)$
7      $(r, M') = \mathtt{solve}(F' \cup C)$
8  until $r = \bot$
9  return $\mathsf{flip}_\alpha(M)$
```

Definition 29 (Cardinality Constraint)**.** *Given a set of variables V and a bound $k \in \mathbb{N}$, the cardinality constraint $\mathtt{cc}(V, k)$ is a CNF formula such that for all models $M \in \mathcal{M}(\mathtt{cc}(V, k))$ it holds that $|M \cap V| < k$.*

Algorithm 6 outlines the procedure. Given a formula F and a model α, we first create the normalized formula F' (line 1). F' now contains only pure and positive literals, such that we start with the model $M' = \mathsf{vars}(F')$ (line 2). In the loop (lines 3 and 7), we calculate the number k of variables assigned true (line 5), after which we generate a cardinality constraint C that allows at most $k-1$ variables assigned true. Then we solve F' including the cardinality constraint C (line 7). If we find a solution, the model is updated (line 4), and the process is repeated. Otherwise, the model M is denormalized and returned (line 9).

Eager Iterative Minimization

Algorithm 6 computes an α-minimum model for F. By replacing line 6 with the statement $C = \big\{\{\neg v \mid v \in M \cap \mathbb{X}\}\big\} \cup \big\{\{l\} \mid l \in M \setminus \mathbb{X}\big\}$ we obtain a much simpler eager algorithm. In this statement, the full cardinality encoding is replaced by a single clause. The second part of the statement depicts the eager part of the solution, i.e., the addition of a unit clause for each negative literal in M. However, the such obtained algorithm computes an α-minimal and not necessarily an α-minimum model.

Structural Pruning

Structural pruning is a pre-processing step to model minimization, which exploits structural encodings to further minimize a model. Assuming that we can reconstruct gate structure from a structurally encoded formula F with the procedure described in Chapter 3, the idea of structural pruning is to exploit "don't cares" in the circuit structure in order to create the pruned formula $P \subseteq F$ by purging all clauses belonging to encodings of *monotonically nested*, unsatisfied sub-formulas. After pruning, we minimize α with respect to the pruned formula P as we have seen in Section 4.1.

Algorithm 7 outlines the procedure. Starting with root gates, in line 2, we traverse the gate structure Γ of the given structural formula (F, Γ) by breadth first search (lines 3 and 4). Whenever the traversed gate G is not satisfied under the given model α and G is not monotonically nested in Γ, the substructure is subject to pruning (line 5). Otherwise (lines 6 to 8), G is added to Γ' and its child-gates are enqueued in Q. The algorithm returns the pruned structural formula (F, Γ').

Note that gate recognition caches the encoding clauses for each gate $G \in \Gamma$, such that a CNF formula can be easily obtained from the pruned formula (F, Γ).

Algorithm 7: Structural Pruning

```
Input: Structural Formula $(F, \Gamma)$, Model $\alpha$
Output: Pruned formula $(F, \Gamma')$
1 $\Gamma' \leftarrow \emptyset$
2 Queue $Q \leftarrow$ root gates of $\Gamma$
3 while not empty $Q$ do
4     $G \leftarrow Q$.pop-front()
5     if ($\alpha$ satisfies $G$) or ($G$ is not monotonically nested in $\Gamma$) then
6         $\Gamma' \leftarrow \Gamma' \cup \{G\}$
7         for child gate $C$ of $G$ do
8             $Q$.push-back($C$)
9 return $(F, \Gamma')$
```

4.2 Random Simulation

Given a combinational circuit, random simulation is the process of repeatedly assigning random values to its input variables and propagating their values down to the root of the circuit in order to generate a complete assignment. The extraction of signal correlations via random simulation in combinational circuits has been studied by Feng Lu et al. in [101]. Random simulation has also been successfully applied in hardware verification with Binary Decision Diagrams (BDD) [92] and Bounded Model Checking (BMC) with And-Inverter Graphs (AIG) [93].

Heule and Biere use random simulation to apply SAT sweeping techniques to CNF formulas [75]. They use Blocked Clause Decomposition (BCD) to compute a large satisfiable subset of a given CNF formula, on which they apply a sweeping algorithm to generate a set of conjectures about literal equivalences which they subsequently verify. In their experiment, they selected 81 application instances from the benchmark set of SAT competition 2013, for which their tool was able to compute a large satisfiable subset within 100 seconds. With their approach, their solver Lingeling could solve 9 more instances than without it.

Signal correlations in terms of literal equivalences and constants can efficiently be recorded by keeping track of equivalence classes of literals during several steps of random simulation [101, 96]. Literals are kept in equivalence classes, which are split as soon as an assignment in one step of random simulation proves the opposite. Literals which have the same value under each complete assignment are kept separately. The procedure allows us to construct conjectures about literal equivalences and backbone variables.

Definition 30 (Backbone Variable)**.** *Given a formula F, a variable $v \in \mathsf{vars}(F)$ is a* backbone variable *iff $\exists l \in \{v, \neg v\}, \forall M \in \mathcal{M}(F) : l \in M$.*

Definition 31 (Conjecture)**.** *A conjecture K is a non-empty set of literals. We call K a* backbone conjecture *if its cardinality $|K| = 1$ and an* equivalence conjecture *otherwise.*

If a gate output literal has the same value in each simulation step, a backbone conjecture is generated. Equivalence conjectures are generated for those literals for which random simulation could not find a counter-example to prove their non-equivalence.

A backbone conjecture $\{l\}$ is satisfied under an assignment α iff $\alpha(l) = 1$. An equivalence conjecture $\{l_1, \ldots, l_n\}$ is satisfied under an assignment α iff it either holds that $\forall i, \alpha(l_i) = 1$ or that $\forall i, \alpha(l_i) = 0$.

Approach

We combine gate recognition and random simulation. For a given CNF formula F, we use gate recognition to construct a structural formula representing a combinational circuit. If the amount of gates surpasses a given threshold relative to the total number of variables, we perform random simulation on the extracted structure.

We generate pseudo-random bit-vectors containing one bit for each input variable, such that each bit holds the value of an input variable. Experiments suggest that the conjectures generated by random simulation are more likely to hold when using non-uniform probability distributions to randomize the circuit inputs [2]. For instance Knuth [91] (p.12) observed this by assigning the value 1 with a probability of 90% instead of 50%. We exploit this effect by generating bit-vectors with higher probability of assigning 1 than 0. The precise specification of bit-vector generation alongside an evaluation of several methods can be found in [96].

A complete assignment of gate outputs is determined by propagation within the gate structure. Our propagation algorithm is an optimized adaption of algorithm `GetMultipleSolutions` in [75]. During random simulation, conjectures are generated and maintained by a fast partitioning algorithm [96]. The process is repeated until a given maximum of simulation steps is reached, or if the amount of changes to conjectures falls below a given threshold.

The output of our algorithm is a set of conjectures which is generated from the results of random simulation. The generated set of conjectures C can be used to speedup the subsequent search. Kutzner presented this in his diploma thesis [96], parts of which we published in [*5]. His work encompasses the implementation and analysis of what is presented in the following two Sections. In Section 4.3, we introduce an approach where conjectures are used to under-approximate a formula in an incremental abstraction-refinement loop. In the approach described in Section 4.4, the same conjectures are input to a novel branching heuristic that stimulates implicit learning.

4.3 Abstraction Refinement

Abstraction is a method of problem simplification, e.g., problem size reduction, and has been used in many applications. Given a formula F, its abstraction F' can be created by omitting constraints or by adding more concise and at the same time stricter constraints. Solving the abstraction F' can produce results which do not hold for the original problem F. Consequently, it is necessary to maintain knowledge about the relationship between F and F' such that F' can be refined when necessary.

Counter-Example Guided Abstraction-Refinement (CEGAR) has been described by Clarke et al. [38, 39] and is the foundation of DPLL(T) and SMT-solvers [60]. Silva et al. present a formalization of abstraction-based approaches for CDCL [42]. They embed CDCL in a system in which they perform over- and under-approximating fix-point iteration. At the same time, they lift CDCL to other problem domains (similar to constructions like DPLL(T) or SMT). They even present a language for Abstract CDCL and combine it with other theories [43].

While in most approaches an abstraction refinement loop is built on top of an incremental SAT solver, only in a few approaches the abstraction refinement loop is integrated in the SAT solver itself. Dantsin and Wolpert [44] used clause-shortening-based abstraction in combination with local search and successive refinement to efficiently find models for SAT problems. In Cube and Conquer [77], Heule et al. use a look-ahead solver to examine a problem first and later propose unit-literals to the CDCL solver. Nadel et al. [109] observed that adding assumptions as unit-clauses to the problem increases the effectiveness of preprocessing.

Over- and Under-Approximation

Given a formula F, its abstraction F' can be created by over- or by under-approximation. The two types of abstraction specify the semantic relationship of the abstraction and the original problem.

Definition 32 (Over- and Under-Approximation). *Given a Boolean formula F, its abstraction F' over-approximates (under-approximates) F iff it holds that $F \models F'$ ($F' \models F$). This implies that the number of solutions of F' is larger (smaller) than or equal to that of F.*

In this approach, we create abstractions which are *under-approximations* of the given problem, i.e., they might have fewer solutions than the original problem and might even be unsatisfiable while the original problem is satisfiable.

Methods of deriving an under-approximation F' from a given formula F include the addition of a new clause or the removal of literals from an existing clause. In Example 8, we show possible CNF-based under-approximations.

Example 8 (Under-Approximation). *Given a formula $F = \big\{\{a,b,c\},\{a,\neg b,\neg c\}\big\}$, the following abstractions are under-approximations of F.*

$$\big\{\{a,b,c\},\{a,\neg b,\neg c\},\{\neg a,c\}\big\} \models F \qquad \text{(Clause Addition)}$$
$$\big\{\{a,b,c\},\{a,\neg b\}\big\} \models F \qquad \text{(Literal Removal)}$$

Approach

Given a formula F, we use random simulation as described in Section 4.2 to create a set of conjectures S. Subsequently, we encode each conjecture $K \in S$ to CNF as follows.

As can be seen in Definition 33, the encoding of a backbone conjecture is a unit-clause, and the encoding of a binary conjecture is the direct encoding of literal equivalence. For encoding n-ary conjectures we use the direct encoding of a circular implication chain.

Definition 33 (Conjecture Encoding). *Given a conjecture K, its CNF encoding ${}^{C}\mathcal{E}(K)$ is defined as follows.*

$${}^{C}\mathcal{E}(K) = \begin{cases} \{\{l\}\} & \text{if } K = \{l\} \\ {}^{D}\mathcal{E}(l_1 \leftrightarrow l_2) & \text{if } K = \{\{l_1, l_2\}\} \\ {}^{D}\mathcal{E}(l_1 \to l_2) \wedge {}^{D}\mathcal{E}(l_2 \to l_3) \wedge \dots {}^{D}\mathcal{E}(l_n \to l_1) & \text{if } K = \{\{l_1, l_2, \dots, l_n\}\} \end{cases}$$

In the following, the clauses in a conjecture encoding of a conjecture K are called conjecture clauses *of K. Given a set of conjectures S, its encoding is specified by the union of the conjecture clauses $K \in S$, such that ${}^{C}\mathcal{E}(S) = \{C \mid C \in {}^{C}\mathcal{E}(K), K \in S\}$.*

In Figure 4.2, we outline our approach. Given a formula F with conjectures S, we create the under-approximation F' of F such that $F' = F \cup {}^{C}\mathcal{E}(S)$. We then use CDCL and solve F'. It holds that $F' \models F$, and therefore, any model we find for F' is also a model of F. However, unsatisfiability of F' does not necessarily imply unsatisfiability of F.

If the approximation F' is unsatisfiable, we first check if a clause in ${}^{C}\mathcal{E}(S)$ has been used to deduce its unsatisfiability. If a conjecture $K \in S$ has been used in the proof, we refine K as described later. We then use the refined conjectures S to restart the search with the update formula $F' = F \cup {}^{C}\mathcal{E}(S \setminus K)$. This procedure is repeated until either a model for F' is found or no conjecture in S was used to derive unsatisfiability (with $S = \emptyset$ being the trivial case). Figure 4.2 includes an outline of the CDCL algorithm in order to highlight that branching with backtracking can be regarded as a specialized form of under-approximation with refinement.

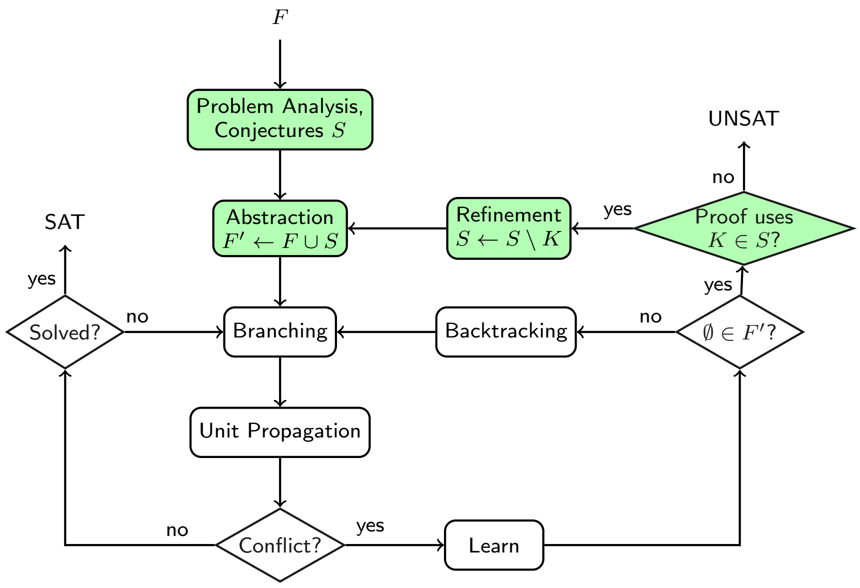


Figure 4.2: Model of an under-approximating SAT solver

Incremental Refinement

We add an activation literal to each clause in a conjecture encoding, in order to be able to *activate* and *deactivate* clauses during incremental solving. In the course of refinement, some conjectures are removed and their conjecture clauses are *deactivated* by satisfying their activation literals. We also use the activation literals to check if conjectures have been used in derivation of UNSAT by checking if they occur in the clauses that were reason to the final propagation [97].

While unary and binary conjectures can be easily removed by deactivating their clauses, n-ary conjectures have to be re-encoded when only a subset of their literals is to be removed. As an optimization, we identify the exact clauses which have to be removed from the implication chain, and deactivate only those clauses that include the literals to be removed from the chain. Then we add a few clauses to patch the implication chain.

Example 9 (N-Ary Conjecture Encoding)**.** *Given the conjecture* $K = \{v_1, v_2, v_3\}$ *and its encoding* $\{\{\neg v_1, v_2\}, \{\neg v_2, v_3\}, \{\neg v_3, v_1\}\}$*, we add a number of activation literals* $\{a_1, a_2, a_3\}$ *in order to gain control and we receive the set of clauses* $\{\{\neg v_1, v_2, a_1\}, \{\neg v_2, v_3, a_2\}, \{\neg v_3, v_1, a_3\}\}$*.*

Example 9 shows the encoding of the n-ary conjecture $K = \{v_1, v_2, v_3\}$. In order to activate the conjecture clauses, we run the solver with assumptions $A = \{\neg a_1, \neg a_2, \neg a_3\}$. Let $v_2 \leftrightarrow v_3$ be the conjecture to be deactivated during refinement. This means that now we are left with two conjectures $v_2 \leftrightarrow v_1$ and $v_1 \leftrightarrow v_3$. So we permanently deactivate $\{\neg v_2, v_3, a_2\}$ by adding the unit clause $\{a_2\}$. Furthermore, we encode two new implications $\{\neg v_2, v_1, a_4\}$ and $\{\neg v_3, v_1, a_5\}$ with fresh activation literals a_4 and a_5.

Conjecture Reliability

Subsequently, those conjectures are removed which are not implied by the original problem, i.e., which are false positives. In order to estimate the "reliability" of a

Algorithm 8: Random-Simulation based Abstraction Refinement (RSAR)

```
Input: $F$ : CNF formula, $[a_i]$ : list of filter arguments
Output: R : SAT or UNSAT
1  gates ← extractGates($F$)
2  conjectures ← randomSimulation(gates)
3  $i \leftarrow 0$
4  conjectures ← filter(conjectures, $a_0$)
5  $(E, A)$ ← encode(conjectures)
6  while cdcl($F \cup E, A$) = UNSAT do
7      $U$ ← calculateUsedConjectures()
8      if $U \neq \emptyset$ then
9          $i \leftarrow i + 1$
10         conjectures ← filter(conjectures $\setminus U, a_i$)
11         $(E, A)$ ← reencode(conjectures)
12     else
13         return UNSAT
14 return SAT
```

conjecture, i.e., its probability to hold, we define the gate-width of a variable in Definition 34.

Definition 34 (Gate Inputs and Gate Width)**.** *Given a gate structure (F, Γ) and a gate $G \in \Gamma$ with $G = (o, I, g)$, the gate width of G is given by the number of inputs $|\delta(o)|$ with the set of inputs $\delta(o)$ being recursively defined as follows.*

$$\delta(v) = \begin{cases} \bigcup\limits_{p \in P} \delta(p) & \textit{if } \exists (v, P, g) \in \Gamma \\ v & \textit{otherwise} \end{cases}$$

Given a gate structure (F, Γ) and a gate $G \in \Gamma$ with $G = (o, P, g)$, the number of possible evaluations for G and gates nested in G grows exponentially in the number of inputs $|\delta(o)|$. Thus, the likelihood for a conjecture $K = \{v, \dots\}$ to be false positive (after a fixed amount of random simulation steps) increases quickly with the number of inputs $|\delta(v)|$.

In order to respect conjecture reliability, we define the function filter which, given a set of conjectures S and natural number a, returns the set of conjectures with maximum number of inputs less than a. We use this filter to reduce the amount of possible false positive conjectures a priori.

$$\mathsf{filter}(S, a) = \big\{K \mid \forall l \in K : |\delta(l)| \leq a\big\}$$

Algorithm

In Algorithm 8, we summarize the approach. In lines 1 and 2, gate structure is extracted and a set of conjectures is determined by random simulation. A counter (line 3) is used in order to keep track of the number of refinement steps. In line 4, we filter the conjectures as described in Section 4.3. In line 5, we generate the conjecture encodings, which gives us a set of clauses E and a set of assumptions A. In line 6, the CDCL algorithm is executed on the formula $F \cup E$ with assumptions A. If the result is SAT, we are done (line 14).

If the result is UNSAT, we calculate the set of conjectures U which were used in the proof (line 7). If U is empty, we are done (line 13). Otherwise, we remove U

Algorithm 9: Random-Simulation based Implicit Learning (RSIL)

```
Data: CNF formula $F$, Conjectures $S$, Current Assignment $A$, Bound $b$
Result: l : Branching Literal
// $p$ is globally initialized with 1
1  if randomNumber$(0, 1) \leq p$ then
       // $i$ is globally initialized with 0
2      $i \leftarrow i + 1$
3      if $i > b$ then
4          $b \leftarrow b + b/2$
5          $p \leftarrow p/2$
6      for $K \in S$ do
7          $D \leftarrow \mathsf{vars}(K) \cap \mathsf{vars}(A)$
8          if $D \neq \emptyset \wedge |D| < |K|$ then
               // Violate conjecture $K$
9              $v \leftarrow$ pickOne$(\mathsf{vars}(K) \setminus D)$
10             $l \leftarrow$ pickOne$(D)$
11             if $l \in A$ then
12                 return $v \in K \ ? \ \neg v \ : \ v$
13             else
14                 return $v \in K \ ? \ v \ : \ \neg v$
15 else
16     return branchDefault
```

from the conjectures and filter again (line 10). The conjectures are re-encoded in line 11 and we continue with a new set of assumptions A and conjecture clauses E.

4.4 Implicit Learning

Implicit learning is a branching heuristic which takes into account a set of conjectures about variable equivalences [101]. Given an equivalence conjecture K, whenever a variable $\mathsf{var}(l_0), l_0 \in K$ is assigned, we heuristically prefer to branch on another variable $\mathsf{var}(l), l \in K$ and its value is picked such that K is violated by the solvers current assignment. Considering K holds, a conflict is provoked and thus implicit learning guides the solver to quickly learn clauses $C \equiv l_0 \leftrightarrow l$.

In Algorithm 9, we describe the branching algorithm of our implicit learning approach. We use implicit learning in combination with VSIDS in order to mitigate the possible runtime deterioration. Therefore, in line 1, we deterministically calculate a random number in order to determine if implicit learning is used or not. Otherwise, we fall back to the default branching strategy of the solver (line 16). We keep track of the number of calls (line 2), and if that number reaches the given bound b, the probability p is decreased and the bound b increased (lines 3 to 5).

If implicit learning is chosen by the algorithm (line 1), we iterate over the given equivalence conjectures (line 6) in order to find a conjecture with both assigned and unassigned variables (lines 7 and 8). We then select an unassigned variable v and an assigned variable r of the conjecture K (lines 9 and 10). In lines 11 to 14, we pick a value for v such that its assignment violates the conjecture as determined by r.

Optimizations

In the implementation, we optimize the overhead of the branching algorithm in two ways. First, we only use conjectures of a maximum size of 3. Furthermore, considering the current assignment A, we only use a bounded number of the most recent assignments [96].

Architektur ist Geiselnahme

Einstürzende Neubauten, Strategies Against Architecture III

Chapter **5**

Implementation

We implemented our algorithms in a fashion allowing for experimentation, analysis and systematic evaluation of the developed concepts. SAT-related algorithms have been implemented in the SAT solver Candy, which is presented in Section 5.1.

For the purpose of systematic evaluation, we developed the Global Benchmark Database (GBD). With GBD, we manage a collection of attributes of benchmark instances. The system is presented in Section 5.2.

5.1 Candy – A Modular SAT Solver

Candy [*10] is a fork of Glucose 3 [11] which is one of the most well-known descendants of Minisat [52]. Not only does Candy provide implementations of the previously presented approaches, also, its architecture facilitates building and maintaining a portfolio of competing solving strategies. Candy provides implementations of the IPASIR interface [80], as well as of the interface of the generic massively parallel SAT solver HordeSAT [21, 14]. Candy's sonification interface makes solver runs audible [*6].

The Architecture of Candy

With Candy, we provide a modular SAT solver architecture to efficiently experiment with many different strategies for branching, learning, restarting and so on. In order to increase *separation of concerns*, we orchestrate a set of loosely coupled systems by a close to bare-bone implementation of the CDCL algorithm. The authors of Minisat implemented simplification by using inheritance, which in our opinion is a design error which now persists in the Minisat family of solvers. By following the paradigm *composition over inheritance*, we eliminated that error in the code-base of Candy.

In Figure 5.1, we outline the architecture. At the time of writing, Candy is composed of the following five systems: branching, propagation, conflict-analysis, simplification and restart. Direct communication between systems is prohibited. All systems interact with two objects that manage the clause database and the current assignment, respectively.

The modular design of Candy allows for efficient and maintainable implementations of different strategies in a system. Among others, the branching system can resort to implementations of *gate recognition* and *random-simulation based implicit learning* (RSIL) which uses the algorithms presented in Chapter 4 [*5, *3].

Candy also has a parallel mode running selected combinations and configurations of different strategies. Candy is capable of maintaining clauses in a shared memory

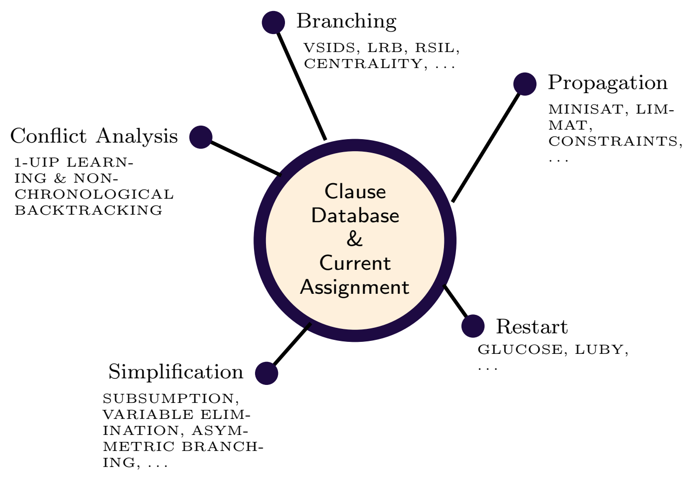

Branching
VSIDS, LRB, RSIL, CENTRALITY, ...
Propagation
MINISAT, LIMMAT, CONSTRAINTS, ...
Conflict Analysis
1-UIP LEARNING & NON-CHRONOLOGICAL BACKTRACKING
Clause Database & Current Assignment
Restart
GLUCOSE, LUBY, ...
Simplification
SUBSUMPTION, VARIABLE ELIMINATION, ASYMMETRIC BRANCHING, ...

Figure 5.1: The Compositional Architecture of Candy

region with sophisticated management of concurrent access, which makes Candy the first parallel *inprocessing* SAT solver with shared clause memory [*8].

Converting CNF instances to AIG: cnf2aig

And-Inverter Graph (AIG) is a format for circuit encodings that uses only binary AND-gates and negations, which has been used in several model checking competitions [25]. Part of Candy is our tool cnf2aig, in which we employ our hierarchical gate recognition algorithm with subsequent trivial gate recognition in order to generate a generally irreducible structural formula from a given CNF formula. Subsequently, we convert the structural formula to a circuit representation which uses only AND-gates and negations in three steps.

In the first step, we introduce an n-ary AND-gate for each gate in the structural formula. As input variables of these AND-gates we create new variables for each forward clause in the gate encoding. The forward clauses of the gate are modified such that they do not contain the output literal and an OR-gate is created for each such modified clause.

In the second step, all OR-gates are converted to AND-gates by double negation and by using their negated output variable as an input to their nesting AND-gates.

In a third step, we convert all n-ary AND-gates to binary AND-gates by introducing new output variables. We do this by subsequently splitting the inputs of each n-ary AND-gate into two halves until only binary AND-gates remain. The such transformed data structure is then output in the AIG format [25].

Sonification

Sonification of scientific data is the art of mapping aspects of a given dataset to sound. Executions of algorithms are suitable for sonification, as they produce time-series of internal states, and human auditory perception is particularly suitable for extracting time-series from auditory signals [69].

In order to sonify executions of the CDCL algorithm, we integrated the OSCPack library of Bencina [23] into Candy. Open Sound Control (OSC) is a network protocol based on UDP and was initially specified by Wright [134], which was well-received in the electronic music community, such that OSC has been implemented for most electronic sound generators [135].

We use OSC to transmit internal states of our SAT solver, namely the conflict-level, backtrack-level, restarts and conflict-clause size. Moreover, we use the programmable software synthesizer CSound [30] as a sound generator. We programmed CSound to receive and map OSC messages of a SAT solver to parameters used in sound synthesis.

We sonify changes in conflict-level and backtrack-level by scaling and mapping them to the frequency of a continuous sound. conflict-clause size is only sonified when clauses are sufficiently small and we emit special sounds for binary and ternary clauses. We map the size of longer learned clauses of up to size ten to the frequency of a low-pass filter over white noise. Restarts are sonified with an electronic drum sound.

Listening to sonified runs of a SAT solver gives a deep impression on the temporal relation of events in a SAT solver. Especially interesting was listening to the actual frequency of restarts and learned unit-clauses. Furthermore, listening to sonified runs of a SAT solver stimulates intellectual attempts to relate events perceived in in the soundscape to each other.

A presentation in the student cultural center "Arbeitskreis Kultur und Kommunikation" (AKK) on the KIT campus included visuals compiled by Jennifer McClelland which displayed SAT Problem visualizations of Carsten Sinz, the source of a CNF instance generated by Vegard Nossum and the source code of MiniSAT by Niklas Eén and Niklas Sörensson. Audible was a run of MiniSAT with an early implementation of our sonification interface.[1]

Future Work

In the future, using the observer pattern for system interaction could increase the modularity of Candy. The observer pattern is harder to optimize due to execution-time lookup of jump targets and conditional event consumption, but this does not affect the bottlenecks (namely *branching* and *propagation*) and hardware optimizations like optimized memory access or speculative execution could mitigate runtime overhead.

Possible future work on Candy's sonification interface includes an improved and more pleasing sonification interface as well as the sonification of further parameters, such as incremental calls and assumption usage, or clause forgetting.

5.2 Global Benchmark Database (GBD)

In our project "Global Benchmark Database" (GBD), we collect attributes of SAT instances and develop tools to organize, distribute and query that data. The core contribution of GBD is the definition of a hash function for SAT instances, which we use to identify benchmark problems in order to associate the collected benchmark attributes, such as solver runtimes or problem families. The main ideas of GBD have been presented and discussed at the Pragmatics of SAT Workshop 2018 (POS-2018) [*7].

GBD fills a gap in practical SAT research due to several reasons. Benchmark instance feature data which is crucial for a deep analysis of experimental results is not always available. Furthermore, association of instance feature data to actual instances based on filenames is unreliable. Compilations of SAT instances with specific attributes are hard to obtain and some existing compilations even contain duplicate instances.

[1]Recording: `https://www.youtube.com/watch?v=iupgZGlzMCQ`

Function GBD Hash(DIMACS File)

```
  Input: DIMACS File
  Output: GBD Hash
1 remove comments
2 remove header
3 replace each sequence of white-space characters (including line-breaks) by a
   single blank
4 append trailing 0 to last clause if missing
5 return md5sum of the remaining content
```

GBD Hash

SAT benchmark problems are used to compare and evaluate the performance of state-of-the-art SAT solvers, e.g., in international competitive events such as the SAT competition [116]. The widely used file format for CNF problems was specified by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) [47] and is known as the DIMACS format for CNF.

Due to its ubiquity, we use the `md5sum` hash function to associate instance features with DIMACS files. However, our definition of a benchmark hash includes a couple of normalization steps, such as removal of comments and normalization of white-space characters.

The DIMACS format requires clauses to be terminating with 0 which means that we can safely replace line-breaks by a single white-space character in order to harmonize line-breaks for the sake of portability. As the DIMACS header is redundant and sometimes incorrect, normalization also includes the removal of the DIMACS header. In Function GBD Hash, we summarize the GBD hash generation.

After discussions at the POS-2018 workshop about capturing different kinds of isomorphisms of CNF formulas by the hash function, such as clause order or variable names, we decided not to capture such isomorphisms in the GBD hash. A strong argument against clause and variable order invariance of the GBD hash is that the runtime of solvers can diverge tremendously for isomorphic formulas.

However, further equivalence classes can still be captured by using attributes such as representants (e.g. union-find data structure) or other hash functions which can be defined to include further normalization steps.

GBD Command Line Interface

We created our reference implementation using the programming language Python [129] and the database framework SQLite [3] to calculate the hashes and provide methods to maintain and query data [*12].

Initialization

In order to initialize GBD, we need a database which at the time of writing is an SQLite file. GBD uses the database path which is specified in the environment variable `GBD_DB`. In order to setup the database, GBD needs to register the paths to the locally available benchmark instances. For this purpose, the command `gbd init <path to benchmarks>` exists. When executed, GBD recursively scans the given directory and saves the association of found benchmark instances and their hash-values in the database.

GBD Queries

In Figure 5.2, we depict the query language of the GBD command line interface in Extended Backus-Naur-Form (EBNF) [79]. GBD allows querying for benchmark instances with specific attributes by automatic translation of the simplified GBD queries to SQL commands.

In Figure 5.3, we show two exemplary queries on a dataset which we collected for benchmark instances of SAT Competitions 2006 to 2018. With the first command, we query for benchmark instances with more than $5,000,000$ variables and display three attributes, their number of variables, clauses and the competition years in which they were used.

With the second command in Figure 5.3, we query for benchmark instances in which our gate recognition algorithm found more than 90% of the variables being output of the encoding of a monotonic function. In this example, we query for the benchmark family which the instances belong to and continue grouping them by their family and counting them.

In Figure 5.4, we show how we use GBD for targeted experimentation with specific instances. In the exemplary command, we query for the paths to instances of the `argumentation`-network family [18]. In an experiment, e.g., the thus obtained paths can be used as input to a SAT solver. Runtimes and other newly calculated instance attributes can then be stored in the database by using the corresponding GBD hash.

Collecting Data

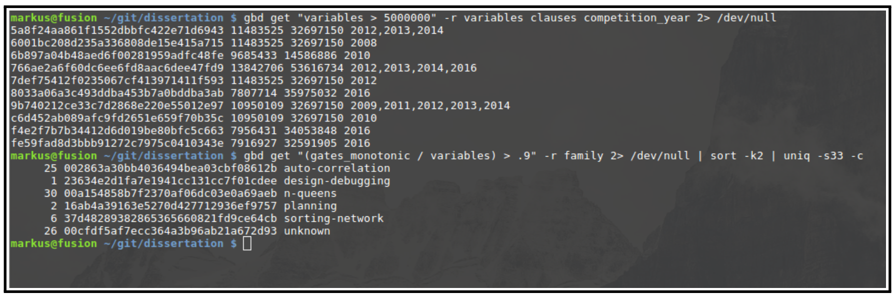
In order to store attributes in the database, GBD provides two commands: `gbd group` and `gbd set`. The `group` command is used to create a new attribute in the database by specifying a `name`, `type` and `default value`. The `set` command is used to store a value for a specific attribute and benchmark instance in the database by specifying the GBD `hash`, attribute `name` and `value`.

We maintain a collection of benchmark attributes for all instances available at SAT competition websites [116] in our database which we published for download [*11]. Some of the instance attributes we collected are listed in Table 5.1.

Our database can be downloaded and used on a local machine by initializing GBD hashes using the locally available benchmark instances. After calculation

$$
\begin{aligned}
\langle \mathit{start} \rangle &= \langle \mathit{query} \rangle \mid \epsilon \\
\langle \mathit{query} \rangle &= \text{'('}, \langle \mathit{query} \rangle, \text{')'} \mid \langle \mathit{query} \rangle, (\text{' and '} \mid \text{' or '}), \langle \mathit{query} \rangle \mid \langle \mathit{constraint} \rangle \\
\langle \mathit{constraint} \rangle &= \langle \mathit{name} \rangle, (\text{'='} \mid \text{'!='}), \langle \mathit{value} \rangle \mid \\
&\quad\; \langle \mathit{name} \rangle, \text{' like '}, [\text{'\%'}], \langle \mathit{value} \rangle, [\text{'\%'}] \mid \\
&\quad\; \text{'('}, \langle \mathit{term} \rangle, (\text{'='} \mid \text{'!='} \mid \text{'<'} \mid \text{'>'}), \langle \mathit{term} \rangle, \text{')'} \\
\langle \mathit{term} \rangle &= \langle \mathit{name} \rangle \mid \langle \mathit{number} \rangle \mid \\
&\quad\; \text{'('}, \langle \mathit{term} \rangle, (\text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'}), \langle \mathit{term} \rangle, \text{')'} \\
\langle \mathit{name} \rangle &= \langle \mathit{letter} \rangle, \{\langle \mathit{letter} \rangle \mid \langle \mathit{digit} \rangle \mid \text{'\_'}\} \\
\langle \mathit{number} \rangle &= [\text{'-'}]\langle \mathit{digit} \rangle\{\langle \mathit{digit} \rangle\}[\text{'.'}\langle \mathit{digit} \rangle\{\langle \mathit{digit} \rangle\}] \\
\langle \mathit{value} \rangle &= \{\langle \mathit{letter} \rangle \mid \langle \mathit{digit} \rangle \mid \text{'\_'} \mid \text{'.'} \mid \text{'-'} \mid \text{'/'}\} \\
\langle \mathit{letter} \rangle &= \text{'a'} \mid \text{'b'} \mid \cdots \mid \text{'z'} \mid \text{'A'} \mid \text{'B'} \mid \cdots \mid \text{'Z'} \\
\langle \mathit{digit} \rangle &= \text{'0'} \mid \text{'1'} \mid \cdots \mid \text{'9'}
\end{aligned}
$$

Figure 5.2: Query Language of GBD Command Line Interface



```
markus@fusion ~/git/dissertation $ gbd get "variables > 5000000" -r variables clauses competition_year 2> /dev/null
5a8f24aa861f1552dbbfc422e71d6943 11483525 32697150 2012,2013,2014
6001bc208d235a336808de15e415a715 11483525 32697150 2008
6b897a04b48aed6f00281959adfc48fe 9685433 14586886 2010
766ae2a6f60dc6ee6fd8aac6dee47fd9 13842706 53616734 2012,2013,2014,2016
7def75412f0235067cf413971411f593 11483525 32697150 2012
8033a06a3c493ddba453b7a0bddba3ab 7807714 35975032 2016
9b740212ce33c7d2868e220e55012e97 10950109 32697150 2009,2011,2012,2013,2014
c6d452ab089afc9fd2651e659f70b35c 10950109 32697150 2010
f4e2f7b7b34412d6d019be80bfc5c663 7956431 34053848 2016
fe59fad8d3bbb91272c7975c0410343e 7916927 32591905 2016
markus@fusion ~/git/dissertation $ gbd get "(gates_monotonic / variables) > .9" -r family 2> /dev/null | sort -k2 | uniq -s33 -c
     25 002863a30bb4036494bea03cbf08612b auto-correlation
      1 23634e2d1fa7e1941cc131cc7f01cdee design-debugging
     30 00a154858b7f2370af06dc03e0a69aeb n-queens
      2 16ab4a39163e5270d427712936ef9757 planning
      6 37d482893828653656608 21fd9ce64cb sorting-network
     26 00cfdf5af7ecc364a3b96ab21a672d93 unknown
markus@fusion ~/git/dissertation $ 
```

Figure 5.3: Screenshot of GBD Command Line Interface

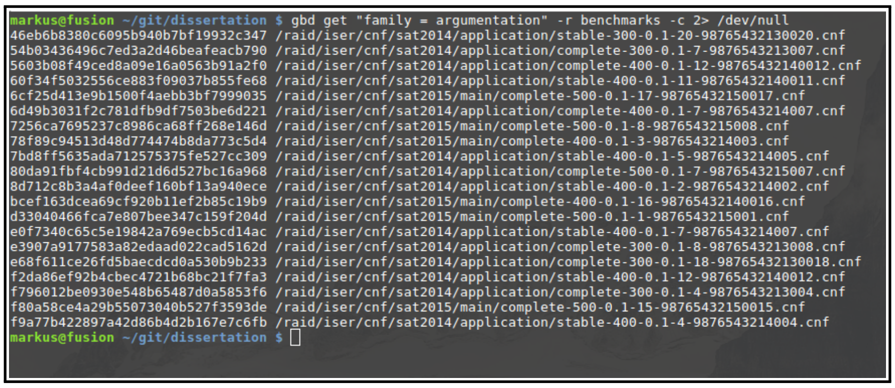
Local Path to Instance
Number of Variables / Clauses
Solution: SAT, UNSAT or UNKNOWN
Problem Family
Competition Year
Number of Horn, Positive and Negative Clauses
Number of Gates and Monotonic Gates
Runtimes of Diverse Configurations of Candy

Table 5.1: Exemplary Attributes in our GBD Database

of GBD hashes, GBD can associate the available attributes of the downloaded database and the corresponding instances on the local file-system.

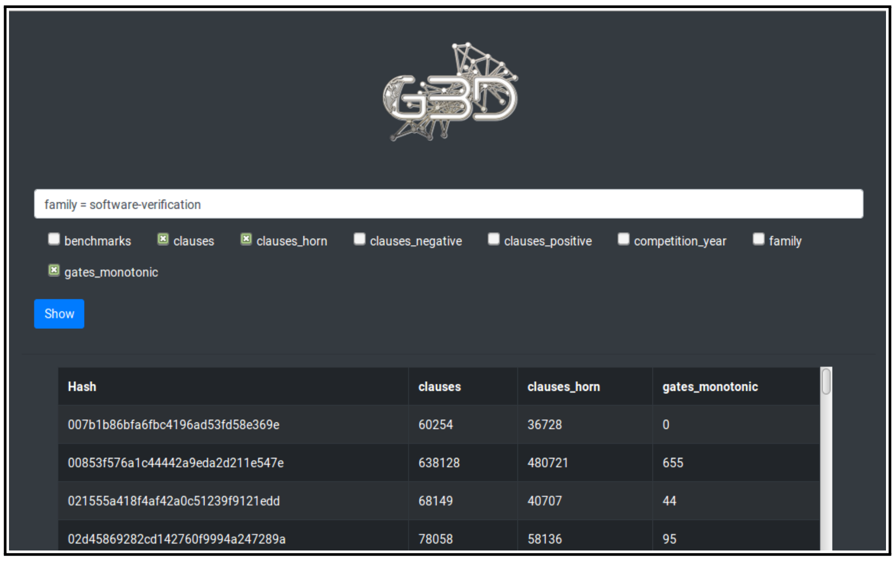
Duplicates

Note that no hash collisions have been recorded so far in hashing all the available competition instances. However, the data sets might contain duplicates. In past SAT competitions, benchmark sets have been compiled by using newly as well as previously submitted instances. By preferring GBD hashes over filenames as instance identifier, we can safely experiment with the complete instance set without wondering about which instances appear more than once in our data set.



```
markus@fusion ~/git/dissertation $ gbd get "family = argumentation" -r benchmarks -c 2> /dev/null
46eb6b8380c6095b940b7bf19932c347 /raid/iser/cnf/sat2014/application/stable-300-0.1-20-98765432130020.cnf
54b03436496c7ed3a2d46beafeacb790 /raid/iser/cnf/sat2014/application/complete-300-0.1-7-9876543213007.cnf
5603b08f49ced8a09e16a0563b91a2f0 /raid/iser/cnf/sat2014/application/complete-400-0.1-12-98765432140012.cnf
60f34f5032556ce883f09037b855fe68 /raid/iser/cnf/sat2014/application/stable-400-0.1-11-98765432140011.cnf
6cf25d413e9b1500f4aebb3bf7999035 /raid/iser/cnf/sat2015/main/complete-500-0.1-17-98765432150017.cnf
6d49b3031f2c781dfb9df7503be6d221 /raid/iser/cnf/sat2014/application/complete-400-0.1-7-9876543214007.cnf
7256ca7695237c8986ca68ff268e146d /raid/iser/cnf/sat2015/main/complete-500-0.1-8-9876543215008.cnf
78f89c94513d48d774474b8da773c5d4 /raid/iser/cnf/sat2015/main/complete-400-0.1-3-9876543214003.cnf
7bd8ff5635ada712575375fe527cc309 /raid/iser/cnf/sat2014/application/stable-400-0.1-5-9876543214005.cnf
80da91fbf4cb991d21d6d527bc16a968 /raid/iser/cnf/sat2014/application/complete-500-0.1-7-9876543215007.cnf
8d712c8b3a4af0deef160bf13a940ece /raid/iser/cnf/sat2014/application/stable-400-0.1-2-9876543214002.cnf
bcef163dcea69cf920b11ef2b85c19b9 /raid/iser/cnf/sat2015/main/complete-400-0.1-16-98765432140016.cnf
d33040466fca7e807bee347c159f204d /raid/iser/cnf/sat2015/main/complete-500-0.1-1-9876543215001.cnf
e0f7340c65c5e19842a769ecb5cd14ac /raid/iser/cnf/sat2014/application/stable-400-0.1-7-9876543214007.cnf
e3907a9177583a82edaad022cad5162d /raid/iser/cnf/sat2014/application/complete-300-0.1-8-9876543213008.cnf
e68f611ce26fd5baecdcd0a530b9b233 /raid/iser/cnf/sat2014/application/complete-300-0.1-18-98765432130018.cnf
f2da86ef92b4cbec4721b68bc21f7fa3 /raid/iser/cnf/sat2014/application/stable-400-0.1-12-98765432140012.cnf
f796012be0930e548b65487d0a5853f6 /raid/iser/cnf/sat2014/application/complete-300-0.1-4-9876543213004.cnf
f80a58ce4a29b55073040b527f3593de /raid/iser/cnf/sat2015/main/complete-500-0.1-15-98765432150015.cnf
f9a77b422897a42d86b4d2b167e7c6fb /raid/iser/cnf/sat2014/application/stable-400-0.1-4-9876543214004.cnf
markus@fusion ~/git/dissertation $ 
```

Figure 5.4: Screenshot of GBD Command Line Interface



family = software-verification

[ ] benchmarks [✓ clauses] [✓ clauses_horn] [ ] clauses_negative [ ] clauses_positive [ ] competition_year [ ] family [✓ gates_monotonic]

Hash	clauses	clauses_horn	gates_monotonic
007b1b86bfa6fbc4196ad53fd58e369e	60254	36728	0
00853f576a1c44442a9eda2d211e547e	638128	480721	655
021555a418f4af42a0c51239f9121edd	68149	40707	44
02d45869282cd142760f9994a247289a	78058	58136	95

Figure 5.5: Screenshot of GBD Web Interface

One surprising result is that the **agile** set of instances used in the Agile tracks of SAT competitions 2016 and 2017 contains many duplicates. For the Agile tracks of the two competitions, $2 \times 5,000$ instances have been automatically generated by recording the instances of incremental SAT solver calls of a regular SMT solver. Out of these $10,000$ instances, GBD found only $2,361$ instances to be unique. For only 713 instances, no duplicate exists in the dataset. For other instances, there exist up to 225 duplicate files in the dataset. The results were so surprising that we immediately checked for hash collisions, by checking the **diff** of all the files GBD recorded as duplicates. All diffs were empty.

Benchmark Classification

Benchmark features have been used for benchmark classification [136], a method which plays a major role in automatic algorithm configuration [89]. Some strategies in SAT solving work well on specific *types* of problems but not on others. Benchmark classification, as well as algorithm selection, are driven by experiments and statistical methods [7].

The data that is used for automatic classification of benchmarks consists of fast to compute features of SAT instances. Based on a set of training instances, a classifier selects the best algorithm or configuration for new instances. Since AI classification is tremendously successful, the question arises whether similar classifications can be achieved by a more analytic approach. While classifiers such as neural networks are used as black boxes, what is the exact relation between structural property and "optimal" configuration?

The characteristics of SAT instances which can be solved quickly by a new SAT algorithm might be unknown in advance, and attributes of SAT instances are very diverse and distributed. Maintaining a collection of metadata of SAT instances gives us the advantage of a more differentiated analysis of our algorithms. Thus, we can gain a better understanding of the method under analysis.

GBD Web Interface

A web interface for GBD [*11], of which we show a Screenshot in Figure 5.5, is currently under development. The GBD web interface allows to query GBD databases from the web browser. Thus, collections of attributes can be downloaded as CSV files. Furthermore, archives of instances with specific attributes can be downloaded. Due to the sheer size of the archives created as a result of most queries, future work should include sharing of dynamically built collections of instances using a peer-to-peer protocol such as BitTorrent [40].

Essentially, all models are wrong, but some are useful.

George Box, Empirical Model-Building and Response Surfaces

Chapter 6

Evaluation

We experimentally investigated the effectiveness and efficiency of our methods. Our experiments were executed on `Acamar`, a compute cluster hosted at our institution. `Acamar` consists of 20 compute nodes with each node equipped with 32 GiB of RAM and 2 *Intel Xeon E5430* CPUs running at 2.66 GHz. An additional node with the same specification is used as a master node to control job scheduling. `Acamar`'s operating system is *Ubuntu 16.04 Xenial Xerus (LTS)* which uses Linux kernel 4.4.0.

Each process ran with a CPU time limit T and a memory limit M. We used the limits $(T, M) = (5000\,\mathsf{s}, 16\,\mathsf{GiB})$. We executed at most two SAT solver processes per compute node.

6.1 Benchmark Instances

We experimented with a collection of SAT benchmark instances stemming from the annual SAT competitions of the years 2006 to 2018. We denote the set of all instances of the competition sets of benchmark instances by `Comp`. If we used a competition set from a specific year, we add the year as a suffix, e.g., `Comp2014`. Usually, we experimented with the most recent competition set, and if there were several tracks we used the instances of the main track. We indicate the restriction to a specific track of a competition by the addition of another suffix, e.g., `Comp2018-main`.

Other sources of benchmark instances have been used as well. For one experiment, we obtained a set of unsatisfiable circuit equivalence checking problems of Armin Biere [24] denoted by `Miter`.

Table 6.1 depicts the distribution of instances and their problem families over the competitions as we extracted them from GBD. The number of instances of important applications like *cryptographic problems* or instances from *software- and hardware-verification* domains are highlighted in bold.

The descriptions of the instance families submitted to SAT competitive events can be found in the proceedings of several SAT Competitions including the SAT Challenge [16, 17, 18, 19, 20, 71]. For many past competitive events, no proceedings have been published, which is why we augmented the information about problem families in GBD with information from additional sources in the Web and by contacting benchmark authors. However, some instances in GBD are still unclassified (depicted by *unknown* in Table 6.1).

Problem Family	Total	Year 2006 - 2018												
		06	07	08	09	10	11	12	13	14	15	16	17	18
agile	2361	-	-	-	-	-	-	-	-	-	-	1580	1981	-
argumentation	20	-	-	-	-	-	-	-	-	20	6	-	-	-
auto-correlation	47	-	-	-	-	-	-	-	30	24	-	-	-	-
automata-synchron.	12	-	-	-	-	-	12	8	-	1	-	-	-	-
biology	60	-	-	-	50	6	11	28	5	11	3	11	-	-
bitvector	115	-	-	-	-	-	26	21	33	12	-	40	-	20
cellular-automata	11	-	-	-	-	-	-	-	-	-	-	-	-	11
chromatic-numbers	20	-	-	-	-	-	-	-	-	-	-	-	-	20
clique-width	24	-	-	-	-	-	-	-	24	2	-	-	-	-
cryptography	**349**	**20**	**20**	**17**	**48**	**18**	**87**	**69**	**107**	**79**	**39**	**19**	**-**	**37**
design-debugging	120	-	-	-	-	-	120	-	-	-	-	-	-	-
diagnosis	74	-	13	-	29	4	25	37	14	19	12	9	-	-
edge-matching	32	-	-	-	30	-	5	32	8	6	-	-	-	-
ensemble-computation	13	-	-	-	-	-	-	12	7	5	-	-	-	-
erdos-discrepancy	20	-	-	-	-	-	-	-	-	20	-	-	-	-
factorization	5	-	-	-	-	-	-	-	-	-	-	-	-	5
fdmus	1000	-	-	-	-	-	1000	-	-	-	-	-	-	-
grandtour-games	19	-	-	-	-	-	-	-	-	-	-	-	-	19
graph-coloring	51	-	-	-	-	-	-	-	-	21	-	15	-	15
graph-isomorphism	60	-	-	-	-	-	-	-	30	30	-	-	-	-
gray-codes	34	-	-	-	-	-	-	-	-	-	34	-	-	-
hardware-verif.	**530**	**118**	**36**	**24**	**50**	**31**	**185**	**142**	**46**	**36**	**11**	**39**	**-**	**-**
hidoku	34	-	-	-	-	-	-	3	22	12	-	-	-	-
modulo-game	31	-	-	-	-	-	-	-	-	-	31	-	-	-
n-queens	31	-	-	-	-	-	-	-	-	30	-	-	-	1
ordering	8	-	7	-	3	-	1	8	2	-	-	-	-	-
parity-games	28	-	-	-	25	-	2	26	-	2	-	-	-	-
partial-ordering	4	-	-	-	-	-	-	-	-	-	-	4	-	-
pebbling-games	18	-	12	-	5	-	1	13	2	1	-	5	-	-
phnf	11	-	10	-	8	-	1	3	2	-	-	-	-	-
pigeon-hole	102	-	7	-	5	-	6	4	13	1	-	35	-	40
planning	**211**	**1**	**8**	**1**	**3**	**-**	**81**	**57**	**55**	**49**	**44**	**54**	**-**	**-**
polynomial-multipl.	19	-	-	-	-	-	-	-	-	-	-	-	-	19
prime-numbers	43	-	4	-	4	-	1	37	4	2	-	-	-	2
product-conf.	**22**	**-**	**-**	**-**	**-**	**-**	**22**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
protocol-verification	12	3	5	2	1	-	1	7	-	-	-	-	-	-
railway	9	-	-	-	-	-	-	-	-	-	-	9	-	-
ramsey-theory	91	-	1	-	10	-	1	8	1	1	-	80	-	-
random	399	-	10	-	23	-	27	66	27	6	26	89	-	165
scheduling	**111**	**-**	**-**	**-**	**-**	**-**	**9**	**9**	**62**	**28**	**13**	**5**	**-**	**12**
sgen	78	-	-	-	26	-	19	54	9	16	-	-	-	-
social-golfer	1	-	-	-	-	-	1	-	-	-	-	-	-	-
software-verif.	**392**	**22**	**58**	**30**	**66**	**30**	**100**	**138**	**26**	**36**	**16**	**19**	**-**	**34**
sorting-network	26	-	5	-	3	1	2	3	-	-	-	22	-	-
stone	13	-	-	-	-	-	-	-	-	-	-	13	-	-
strippacking	46	-	-	-	-	-	46	10	5	4	2	3	-	-
subgraph-isomorphism	80	-	2	-	22	-	52	46	-	-	-	-	-	-
subset-cardinality	10	-	-	-	-	-	-	-	-	-	-	10	-	-
sudoku	1	-	-	-	-	-	1	-	-	-	-	-	-	-
toughsat	8	-	-	-	-	-	-	-	8	4	-	-	-	-
tree-decompositions	22	-	-	-	-	-	-	-	-	-	-	-	-	22
tseitin-grid	38	-	-	-	-	-	-	-	-	-	-	38	-	-
uniform-random	3206	-	511	-	570	-	600	600	430	225	-	180	-	90
unknown	**977**	**32**	**197**	**26**	**162**	**10**	**330**	**315**	**49**	**90**	**54**	**40**	**-**	**143**
waerden-numbers	110	-	-	-	-	-	110	41	9	1	-	-	-	-
zero-one	30	-	-	-	-	-	-	-	-	30	-	-	-	-

Table 6.1: SAT Benchmark Instances of Competitive Events from 2006 to 2018 by Problem Family

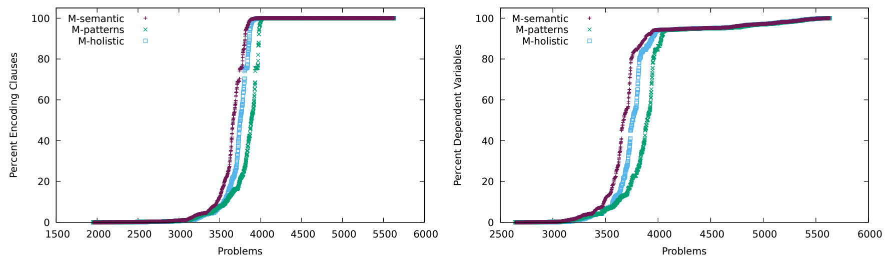


Figure 6.1: Effectiveness of Gate Recognition Methods: Percent Encoding Clauses (left) and Percent Dependent Variables (right)

6.2 Gate Recognition

We evaluated our gate recognition algorithms using $5,632$ benchmark instances in `Comp` excluding the `uniform-random` and `agile` problem families. In the following, we compare the effectiveness and efficiency of three methods for right-uniqueness checks: patterns, semantic and holistic.

In patterns, we use clausal patterns for right-uniqueness checks, while in semantic, we employ Candy for conducting semantic right-uniqueness checks. In holistic we include the resolution environment of candidate gate inputs in the semantic right-uniqueness checks. In this comparison, we employed only unit clauses for root selection *without* further iterations.

Successful Gate Recognition

On the right in Figure 6.1, we show the percentage of variables per instance which we recognized as being functionally dependent on other variables by using one of our methods to test for right-uniqueness. Clearly, our generic semantic method has a significant advantage over the method patterns which is limited to specific clausal patterns.

patterns	semantic	holistic	family-size	family
25	25	25	47	auto-correlation
41	53	53	115	bitvector
-	11	11	11	cellular-automata
87	96	96	349	cryptography
3	7	7	120	design-debugging
3	3	3	74	diagnosis
13	13	13	13	ensemble-computation
1000	1000	1000	1000	fdmus
149	244	181	530	hardware-verification
30	30	30	31	n-queens
-	-	13	18	pebbling-games
2	2	2	211	planning
36	41	41	43	prime-numbers
12	12	12	12	protocol-verification
112	115	114	392	software-verification
6	6	6	26	sorting-network
82	96	95	977	*unknown*

Table 6.2: Number of instances per problem family where more than 90% of variables were recognized to be functionally dependent on other variables.

patterns	semantic	holistic	family-size	family
24	24	24	47	auto-correlation
76	93	93	115	bitvector
-	11	11	11	cellular-automata
94	103	103	349	cryptography
4	9	9	120	design-debugging
3	3	3	74	diagnosis
13	13	13	13	ensemble-computation
1000	1000	1000	1000	fdmus
153	248	281	530	hardware-verification
13	13	13	18	pebbling-games
2	2	2	211	planning
36	41	41	43	prime-numbers
12	12	12	12	protocol-verification
119	121	121	392	software-verification
6	6	6	26	sorting-network
100	124	124	977	*unknown*

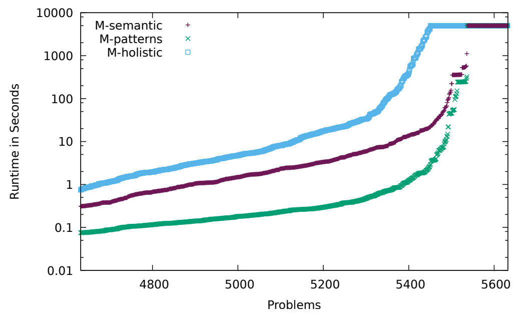
Table 6.3: Number of problems per problem family where pattern-, semantic- or holistic gate recognition found more than 90% of clauses to encode a gate.

Our holistic approach however, with which we should theoretically be capable of recognizing more gates, turned out to be infeasible in terms of runtime. As a result of several timeouts, the holistic recognition method recognizes fewer gates than our semantic method.

On the left in Figure 6.1, we depict the percentage of clauses per instance which we recognized as gate encodings. In many instances all clauses have been discovered to be part of a gate encoding, i.e., gate recognition left no remainder. While we recognized 1, 539 problems without remainder with the pattern method, we recognized 1, 597 instances without remainder with the semantic method. With the holistic approach, we only recognized 1, 586 instances without remainder. Again, we see in Figure 6.1 that the semantic approach has been most successful in terms of remainder sizes.

Our algorithm discovered 90 instances where more than 90% of all variables are output variables of the encoding of a monotonic gate. Monotonic root structures are particularly useful for structural pruning (see Section 6.5).

We investigated in which problem families our gate recognition methods are particularly successful. Table 6.2 shows the number of instances where more than 90% of variables were discovered to be output of a gate, and Table 6.3 shows the number of problems where more than 90% of clauses were discovered to be part of a gate encoding.



configuration	timeouts
`patterns`	95
`semantic`	95
`holistic`	181

Figure 6.2: Longest 1, 000 Runtimes of Gate Recognition Methods

#conflicts	#instances	#semantic checks	∅checks per problem
0	2,382	83,991,813	35,261
1	754	14,848,502	19,693
2	97	950,734	9,801
3	86	258,900	3,010
4	27	32,780	1,214
5	23	26,157	53
6	0	0	0
7	17	1,528	90

Table 6.4: Number of instances, total and average number of SAT-based right-uniqueness proofs according to their number of conflicts

The diverging numbers between the two tables show that both measures do not necessarily correlate, as an instance's gate structure might be fully exposed by our algorithm but at the same time, it might contain many variables which are not functionally dependent.

As shown in Table 6.2, for many hardware-verification instances we discovered less dependencies with our holistic method than with our semantic method. However, Table 6.3 shows that with the holistic method, we discovered more instances without remainder. Hardware verification instances seem to be generated with many optimizations which our holistic method seems to recognize, but due to its infeasibility regarding runtime, the approach is not reliable.

Runtime of Gate Recognition

In Figure 6.2, we plotted the $1,000$ longest runtimes (out of a total of $5,632$) of our gate recognition methods. We omitted the smaller runtimes, as most instances have been processed in a couple of seconds.

In this plot, we can see that the semantic approach is very competitive in terms of runtime as compared to the patterns approach. For most instances, gate

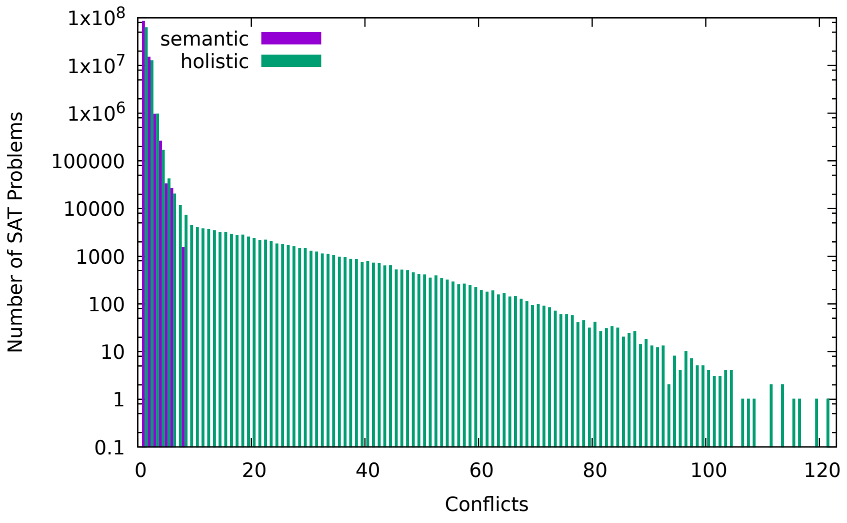


Figure 6.3: Number of SAT-based right-uniqueness checks grouped by their number of conflicts

recognition finishes within a minute. However, both the patterns as well as the semantic approach exhibit 95 timeouts.

The runtime curve of the holistic approach depicts runtimes of several minutes for many more instances, and does not finish recognition within the time-limit of $5,000s$ in 181 cases.

In our semantic approach, we generate a large number of small SAT problems to prove right-uniqueness. Running gate recognition on $2,382$ instances for which semantic right-uniqueness checks were produced, we counted a total of $100,110,414$ SAT solver calls. On average, each execution of gate recognition generated $42,028$ SAT solver calls per instance. In Table 6.4, we show the hardness of these problems in terms of the number of conflicts occurring in our solver while solving these problems.

Note that solver reinitialization times can become a bottleneck in such a scenario. Candy is optimized for this massively incremental application as we reduced the reinitialization overhead in our implementation to a minimum.

In each call to the SAT solver, we actually proved right-uniqueness. From a practical point of view, that means that left-totality (or blockedness, respectively) is a strong indicator for functionality of gate encodings. The evaluation underlines the feasibility of our SAT-based approach to prove right-uniqueness.

As shown in Figure 6.3, in contrast to the semantic approach, we can see that the problems generated by the holistic approach are much harder to solve in terms of conflicts. While for 17 instances the semantic approach employs $1,528$ SAT instances with a maximum number of 7 conflicts, the holistic approach generates instances inducing a significantly larger amount of conflicts (up to 123 conflicts).

6.3 Repeated Root Selection

In this section, we evaluate gate recognition with respect to repeated root selection based on occurrence lists with blocking counters. Again, we experimented with a total of $5,632$ instances in `Comp` under exclusion of the uniform-random and agile instance families.

In repeated root selection, we repeatedly select new candidate root literals. In a first iteration, we select roots via unit clauses as in the previous approach. Then, we continue selecting roots based on minimal blocking counters for a bounded number of iterations.

We evaluate four configurations of repeated root selection using different bounds for the maximum number of repeated root selections. In the first configuration, we unboundedly select roots (`T-unlimited`), in two further configurations, we select roots with number of repetitions bounded by 10 (`T-10`) and 100 (`T-100`), respectively. We added a fourth configuration (`T-1`) as a reference, where we only select unit clauses.

In the evaluation of the four configurations we used a combination of the previously evaluated `pattern`- and `semantic`-based approach, in which we first use the `pattern` method for right-uniqueness checks and fall-back to the `semantic` method if the `pattern` method does not succeed.

In Figure 6.4, we show that gate recognition with repeated root selection is significantly more effective. The percentage of dependent variables as well as the percentage of encoding clauses increases as we increase the repetition bound.

However, in Figure 6.5 we see that runtimes deteriorate dramatically with increasing bounds. Each configuration exhibits timeouts, even configuration `T-1`. In the approaches evaluated in Section 6.2, we lazily executed blockedness checks. In this approach we use occurrence lists with blocking counters not only for repeated root selection but also for blockedness checks. As blocking counters are calculated

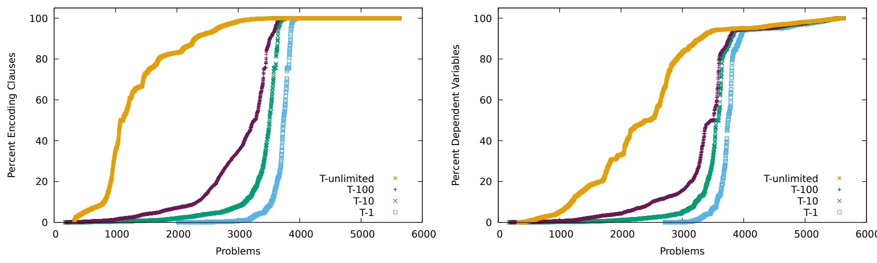
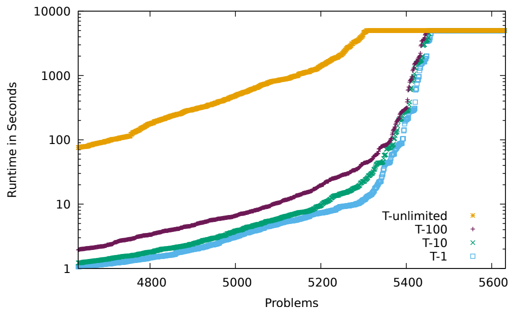


Figure 6.4: Effectiveness of Gate Recognition with Repeated Root Selection: Percent Encoding Clauses (left) and Percent Dependent Variables (right)



configuration	timeouts
`T-unlimited`	324
`T-100`	183
`T-10`	175
`T-1`	171

Figure 6.5: Longest 1,000 Runtimes of Gate Recognition with Repeated Root Selection

in advance, the approach turned out to be infeasible for large problems. This calls for optimizations which we address in Chapter 7.

6.4 Comparison to mvSAT

Balyo et al. developed the tool mvSAT which creates And-Inverter Graphs (AIG) [25] from a given CNF formula using Blocked Clause Decomposition (BCD) [15]. They simplified the such created AIGs using the tool abc [32] (with option -dc2) and investigated the efficiency of SAT solvers applied to the re-encoded CNF instances of the such simplified AIGs.

Using the application instances in Comp2013, we repeated the same experiment with mvSAT [15] and our tool cnf2aig [*3] which generates AIGs based on our gate recognition algorithm. Note that in this approach we only used unit clauses for root selection.

We show in Figure 6.6 that in most cases, our approach could discover more functional dependencies per CNF instance. We also measured the runtime of lingeling on the simplified and re-encoded formulas. The cactus plot in Figure 6.7 depicts a runtime comparison using lingeling on the re-encoded CNF instances by using mvSAT and by using cnf2aig. Lingeling could solve the re-encoded CNF instances which we generated based on our approach much faster than the original instances as well as those instances generated based on the competing approach of Balyo et al.

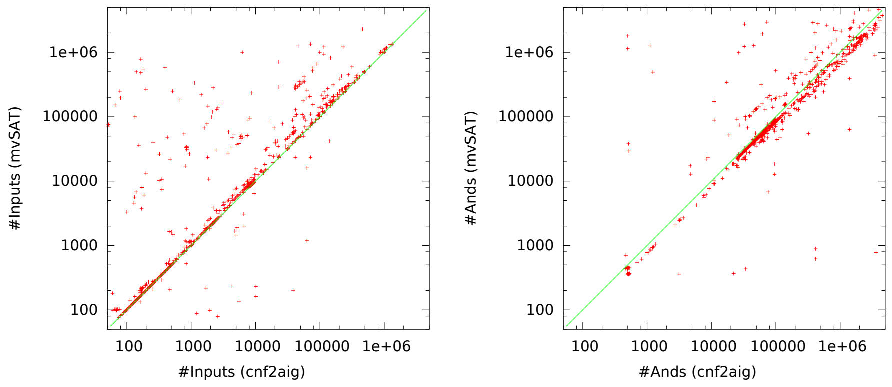


Figure 6.6: cnf2aig builds AIG with *fewer* input variables and *more* gates than mvSAT.

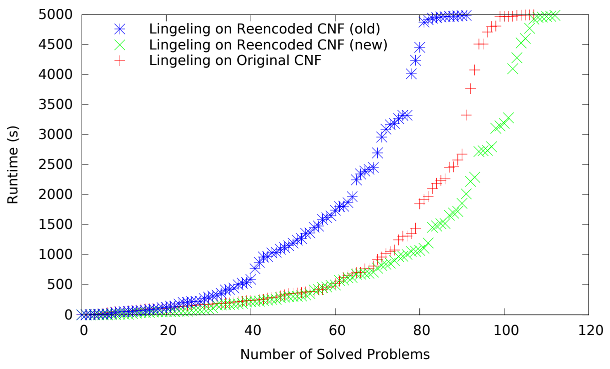


Figure 6.7: Comparison of the runtimes of lingeling on application instances in Comp2013 with the re-encoded CNF using mvSAT (old) vs. cnf2aig (new)

6.5 Model Minimization

We experimentally evaluated our model minimization algorithm on satisfiable instances in Comp. We evaluated the different minimization methods and their combinations with respect to their effective reduction of model size. Note that for formula pruning, we used a configuration of our gate recognition algorithm that used unit clauses for root selection.

Minimization

In Figure 6.8, we show two scatter plots which compare the sizes of the minimized models with their original sizes. Few models can be minimized with our eager iterative approach as can be seen in the left plot in Figure 6.8. The right plot shows the results of minimization by formula pruning based on "don't care" information and subsequent eager iterative minimization. Note that fewer instances are considered in the right plot, as only those instances can be considered for pruning where we recognized monotonic root gates.

We also experimented with full cardinality encodings in order to generate a minimum model with respect to the given model. However, no further minimization could be achieved than with the eager iterative minimization. While the runtime of greedy minimization for most instances is below one second and always below 10

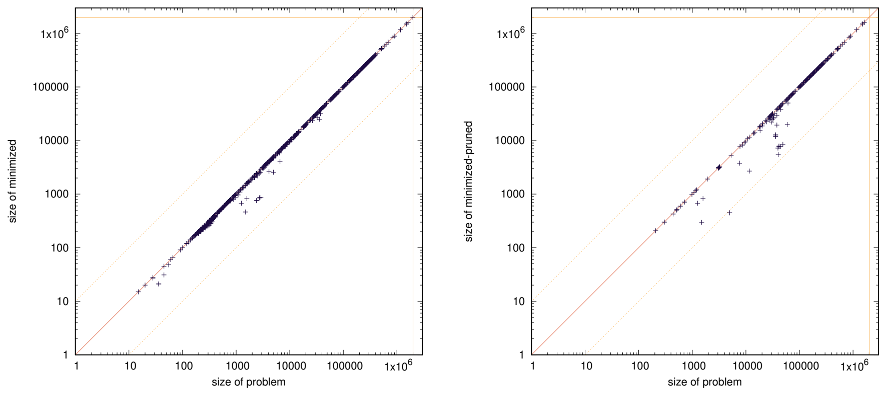


Figure 6.8: Model size reduction by greedy minimization (left) and greedy minimization with structural pruning (right)

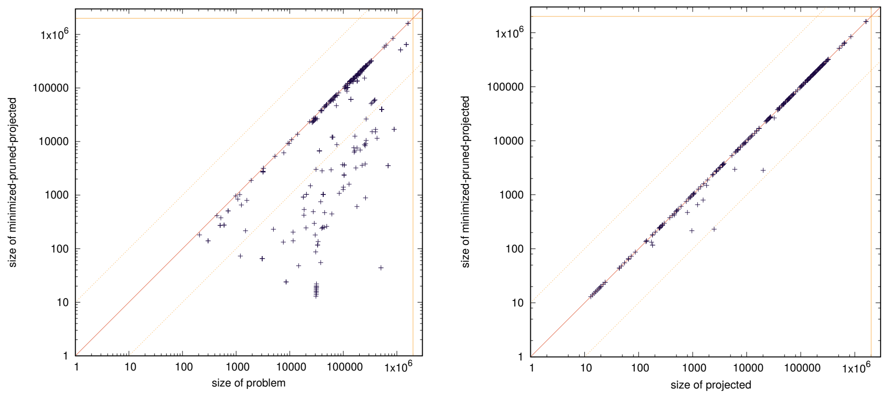


Figure 6.9: Model size reduction with structural pruning and projection compared to the total number of variables (left) and to the total number of input variables (right)

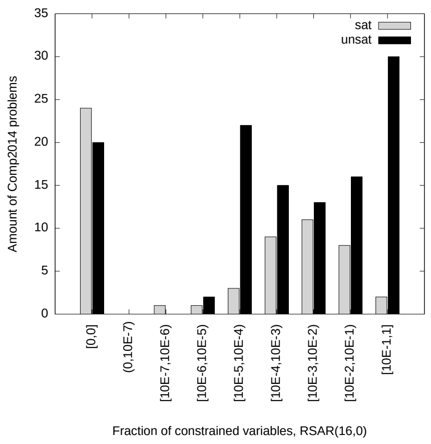
seconds, the runtimes of the full cardinality encodings were much higher (including 3 timeout above 5000s). In conclusion, the greedy approach was as effective as the non-greedy approach while being orders of magnitude more efficient.

Minimization with Projection

In Figure 6.9, we evaluate minimization with pruning and projection. After having run gate recognition, we know the input variables of the problem. Projection to input variables is very effective, as can be seen in the left part of Figure 6.9. However, in the right plot, we compare minimization with pruning and projection to the number of input variables, i.e., projection without minimization. Only for a few models, the number of input variables can be significantly reduced.

6.6 Abstraction Refinement

In our experiments with the RSAR algorithm, we used a maximum of 2^{20} random simulation rounds (as described in Chapter 4). Note that in this experiment we only used unit clauses for root selection in gate recognition. We ran our experiments



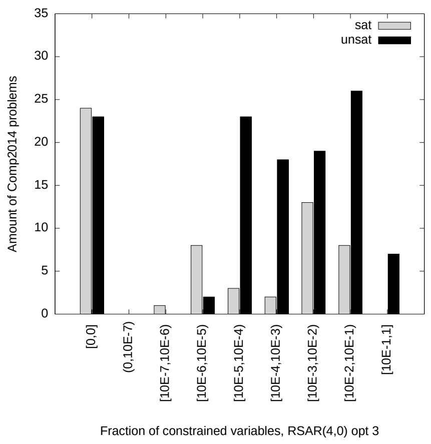


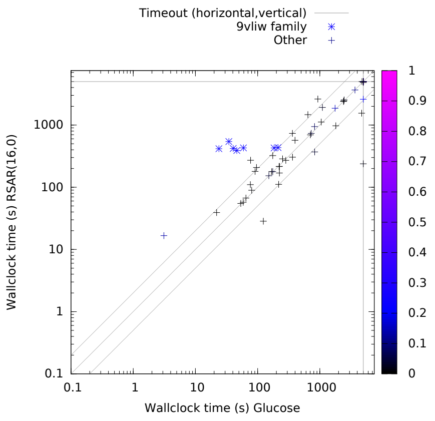
Figure 6.10: Distribution of fractions of numbers of variables in conjectures used in the initial approximation of problems in Comp2014 for unbounded conjecture sizes (left) and conjecture sizes bound by 3 (right)

with instances in Comp2014 and a timeout of 5000 seconds. Relative to the timeout, the total runtime of preprocessing, i.e., gate recognition and random simulation, was negligible, as it never exceeded 12 minutes and for the great majority completed within 2 minutes [96].

We used a harsh refinement heuristic removing all conjectures generated during preprocessing in the first refinement. In Variant A, we use all conjectures, and in variant B, we bound conjecture size by 3 (denoted by "opt 3" in Figures 6.11 and 6.10). The reason for the additional size constraint in variant B is our assumption that small conjectures are more likely to hold.

In Figure 6.10, we show two histograms depicting the distribution of the fractions of numbers of variables constrained by conjectures used to construct the first approximation. The data shows that by using harsher filtering, significantly fewer variables are constrained by conjectures.

In Table 6.5, we show the runtimes for instances in Comp2014. In the first



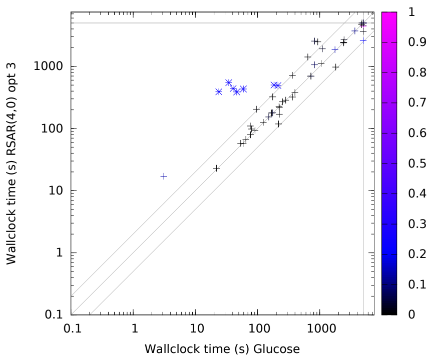


Figure 6.11: Runtimes of RSAR vs. Glucose on satisfiable instances in Comp2014 for filter configurations [16, 0] (left) and [4, 0] (right); colors indicate the fraction of gate outputs relative to the total number of variables

	Variant A		Variant B	
filter	SAT	UNSAT	SAT	UNSAT
$[2, 0]$	48 (+1)	86 (-6)	49 (+2)	87 (-5)
$[4, 0]$	48 (+1)	86 (-6)	**50** (+3)	86 (-6)
$[8, 0]$	48 (+1)	81 (-11)	49 (+2)	86 (-6)
$[16, 0]$	49 (+2)	79 (-13)	49 (+2)	88 (-4)
$[32, 0]$	49 (+2)	72 (-20)	49 (+2)	83 (-9)
$[64, 0]$	49 (+2)	74 (-18)	49 (+2)	76 (-16)

Table 6.5: Amount of problems solved by RSAR on instances in `Comp2014`

bound	`Comp2014/sat`	`Comp2014/unsat`	`Miter`
10^3	49 (0)	92 (-1)	316 (0)
10^4	49 (0)	93 (0)	316 (0)
10^5	49 (0)	93 (0)	320 (+4)
10^6	49 (0)	**94** (+1)	**323** (+7)
10^7	49 (0)	91 (-2)	**323** (+7)

Table 6.6: Amount of problems solved by RSIL on instances in `Comp2014` and `Miter`

column, the heuristic configuration of filter indicates the number of maximum input dependencies allowed in each refinement step (compare with Section 4.3).

In the second and third column, we specify the number of solved instances for each variant. While we solved 47 of the satisfiable instances using Glucose, our modifications using RSAR allowed us to solve up to 3 more satisfiable instances.

However, RSAR did not work particularly well on unsatisfiable instances in `Comp2014`. The bad runtimes that RSAR produced on these instances could not be mitigated by the harsh filter configuration. For example, in variant B with `filter` configuration $[4, 0]$, we solved only 86 unsatisfiable instances using RSAR, as opposed to 92 solved instances using Glucose.

In Figure 6.11, we show the runtime scatter plots for two heuristic configurations of the RSAR algorithm. Although we observed fewer timeouts with RSAR, the results are mostly inconclusive. We highlighted the deteriorating runtimes on the `9vliw` family of problems contained in `Comp2014`. We could observe that curiously, the pattern for `9vliw` repeats in all the experiments that we conducted, also in the following evaluation of RSIL (Section 6.7).

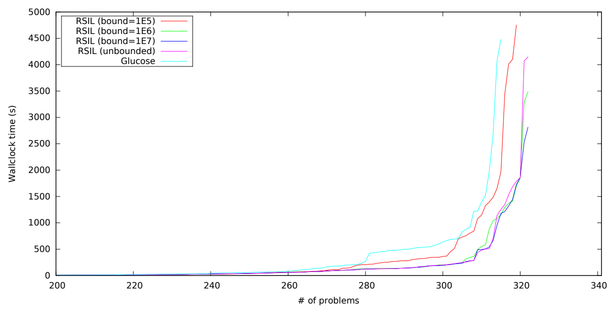


Figure 6.12: Runtimes of RSIL with several bounds vs. Glucose on instances in `Miter`

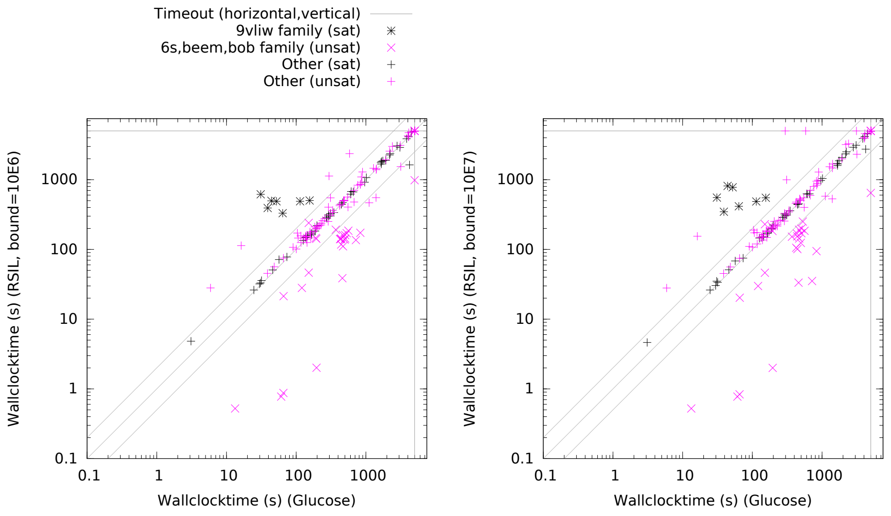


Figure 6.13: Runtimes of RSIL with bound 10^6 (left) and with bound 10^7 (right) vs. Glucose on instances in `Comp2014`

6.7 Implicit Learning

In our experiments with the RSIL algorithm, we used a maximum number of random simulation rounds of 2^{17} (in RSAR we used 2^{20}), as RSIL is more robust regarding false positive conjectures. We also only use conjectures with a maximum size of 3 in order to reduce the overhead in branching, where we iterate conjectures.

In Table 6.6, we show the number of solved instances for several bounds. We achieved the best results with a bound of 10^6, where we solved 1 more unsatisfiable instance in the `Comp2014` benchmark set and 7 more instances in the `Miter` benchmark set.

In Figure 6.13, we show two runtime scatter plots for bounds 10^6 and 10^7 on the `Comp2014` benchmark set. We achieved significant speedups on the circuit equivalence problems. However, the performance on the `9vliw` family of problems included in `Comp2014` deteriorated in any scenario we used in our experimentation (see also Section 6.6).

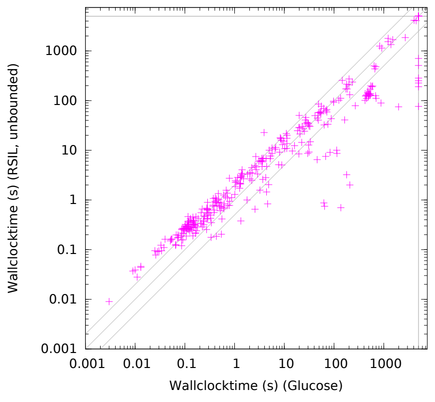


Figure 6.14: Runtime scatter plot of RSIL without bound vs. Glucose on instances in `Miter`

In the cactus plot in Figure 6.12, we summarize the runtimes of RSIL configurations using several bounds on instances in `Miter`. Clearly, no bounding of the method is needed for the `Miter` set of instances. Small bounds mitigate runtime deterioration for instances where the method is unsuccessful. However, the advantage of RSIL is noticeable for instances in `Miter`, even if the bounds are small.

In Figure 6.14, we show a runtime scatter plot for unbounded RSIL on the `Miter` benchmark set, where the method exhibits significant speedups. For the lower runtimes (a few seconds and less) the overhead incurred through the problem analysis step is clearly noticeable in the plot. However, the advantage of RSIL becomes clearly visible in runtimes of long-running benchmark instances.

So Long, and Thanks for All the Fish

Douglas Adams, The Hitchhiker's Guide to the Galaxy

Chapter 7

Conclusion

We devised a generic algorithm for recognizing gate structure in CNF formulas. The algorithm is generic as it does not require knowledge about the clausal patterns of the encoded gates. We proved that the semantics of blocked sets captures left-totality and devised a semantic method for proving right-uniqueness by solving a small SAT problem. In our experiments, we could show that the algorithm is efficient, and that it effectively recognizes gate structure in many SAT instances which have been used as benchmark problems in past SAT competitions.

We showed that our tool `cnf2aig` is superior to previously presented gate recognition tools in terms of the number of recognized gates. We also showed that the recognized gate structure can be used to improve the runtime efficiency of SAT solvers on many application instances after application of circuit simplification techniques.

Furthermore, we exploited gate structure in an approach that uses "don't care" variables in the circuit encoding to minimize models and experimentally showed that this helps reducing the model sizes of some satisfiable problem instances.

Moreover, we used random simulation on the extracted circuit structure of a given CNF formula in order to generate conjectures about backbone literals and literal equivalences. We experimented with using those conjectures to perform under-approximation in incremental SAT solving, which was only successful on some satisfiable instances. In another approach, we used these conjectures to employ the branching heuristic "implicit learning" which performed particularly well on unsatisfiable circuit-equivalence checking problems. Both approaches are complementary as in the abstraction approach, we stimulate faster solution of some satisfiable problems and in the implicit learning approach, we target violations of conjectures to stimulate faster proof generation for unsatisfiable instances.

The SAT solver `Candy` is a result of implementing many new strategies for CDCL heuristics. Its compositional architecture facilitates the implementation of new strategies. Candy can also be used as a parallel portfolio solver, and as such Candy is the first inprocessing solver which is able to manage a shared clause database in the parallel scenario.

The Global Benchmark Database (GBD) is a tool for distributed management of benchmark instances and their attributes. Its key innovation is the definition of the GBD hash function, which is mandatory for reliable association of benchmark instances and their attributes. Analysis of large collections of benchmark instance features in our data-driven approach leads to more informed evaluations of experiments, and thus, it can lead to better hypotheses in the future.

Future Work

We have seen that instances of the circuit equivalence checking domain can be solved much faster with random-simulation-based implicit learning (RSIL). In a portfolio of strategies, properties of those instances where SAT solver performance benefits from RSIL can be used by methods of dynamic strategy selection in order to use less restrictive strategy configurations for those instances than the configurations which we used to obtain a better *overall* performance.

If the monotonic root structure in a CNF formula is large and many clauses can be deactivated for propagation, then specialized propagation algorithms and data-structures which exploit "don't care" information in the gate structure could be beneficial for SAT solver performance.

Gate recognition can also be the foundation to recognize more expressive constraints, e.g., adder circuits, parity constraints or circuit-based cardinality constraints [123, 26], which could then be exploited by specialized constraint propagation algorithms and data-structures [131]. Moreover, constraint-based abstraction, which is based on necessary but insufficient criteria to correctly classify constraints, should be examined.

GBD has turned out to be a powerful tool for analyzing SAT instances and algorithms, thus, suggesting the further development of the tool. More instance features should be collected such as the number of models of satisfiable instances, the size of the shortest known proof of unsatisfiable instances, the number of connected components, or the number of mincuts of a certain size in the variable incidence graph, to name a few. GBD should be used to run the evaluations of the annual SAT competitions.

The modular design of Candy improves the comparability of experiments with different strategies in CDCL. A thorough delta-analysis of recently devised competing strategies in SAT solving, including instance metadata available in GBD, should follow.

List of Algorithms

List of Figures

List of Tables

Publications

[*1] Markus Iser, Mana Taghdiri, and Carsten Sinz. "Optimizing MiniSAT Variable Orderings for the Relational Model Finder Kodkod - (Poster Presentation)". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pp. 483–484.

[*2] Markus Iser, Carsten Sinz, and Mana Taghdiri. "Minimizing Models for Tseitin-Encoded SAT Instances". In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pp. 224–232.

[*3] Markus Iser, Norbert Manthey, and Carsten Sinz. "Recognition of Nested Gates in CNF Formulas". In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pp. 255–271.

[*4] Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz. "SAT Race 2015". In: *Artif. Intell.* 241 (2016), pp. 45–65.

[*5] Markus Iser, Felix Kutzner, and Carsten Sinz. "Using Gate Recognition and Random Simulation for Under-Approximation and Optimized Branching in SAT Solvers". In: *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pp. 1029–1036.

[*6] Markus Iser and Jennifer McClelland. *MiniSAT Sonification - Public Presentation (Video)*. 2017. URL: `https://www.youtube.com/watch?v=iupgZGlzMCQ`.

[*7] Markus Iser and Carsten Sinz. "A Problem Meta-Data Library for Research in SAT". In: *Proceedings of Pragmatics of SAT 2018, Oxford, UK, July 7, 2018.* Pp. 144–152.

[*8] Markus Iser, Tomáš Balyo, and Carsten Sinz. "Memory Efficient Parallel SAT Solving with Inprocessing". In: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019.*

[*9] Matthias Kern, Ferhat Erata, Markus Iser, Carsten Sinz, Frédéric Loiret, Stefan Otten, and Eric Sax. "Integrating Static Code Analysis Toolchains". In: *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, pp. 523–528.

[*10] *Candy*. URL: `https://github.com/Udopia/candy-kingdom`.

[*11] Markus Iser, Carsten Sinz, Luca Springer, and Martin Heil. *GBD Server*. URL: gbd.iti.kit.edu.

[*12] Markus Iser and Luca Springer. *GBD*. URL: https://github.com/Udopia/gbd.

Bibliography

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. "Symbolic Reachability Analysis Based on SAT-Solvers". In: *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, pp. 411–425.

[2] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital systems testing and testable design.* Computer Science Press, 1990.

[3] Grant Allen and Mike Owens. *The Definitive Guide to SQLite.* 2nd. Berkely, CA, USA: Apress, 2010.

[4] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. "Solving difficult instances of Boolean satisfiability in the presence of symmetry". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 22.9 (2003), pp. 1117–1137.

[5] Rui Alves, Filipe Pereira Alvelos, and Sérgio Dinis T. Sousa. "Resource Constrained Project Scheduling with General Precedence Relations Optimized with SAT". In: *Progress in Artificial Intelligence - 16th Portuguese Conference on Artificial Intelligence, EPIA 2013, Angra do Heroísmo, Azores, Portugal, September 9-12, 2013. Proceedings*, pp. 199–210.

[6] Henrik Reif Andersen and Henrik Hulgaard. "Boolean Expression Diagrams". In: *Inf. Comput.* 179.2 (2002), pp. 194–212.

[7] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. "Structure features for SAT instances classification". In: *J. Applied Logic* 23 (2017), pp. 27–39.

[8] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. "Community Structure in Industrial SAT Instances". In: *J. Artif. Intell. Res.* 66 (2019), pp. 443–472.

[9] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. "Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction". In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pp. 309–317.

[10] Gilles Audemard and Laurent Simon. "On the Glucose SAT Solver". In: *International Journal on Artificial Intelligence Tools* 27.1 (2018), pp. 1–25.

[11] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pp. 399–404.

[12] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[13] Fahiem Bacchus and Jonathan Winter. “Effective Preprocessing with Hyper-Resolution and Equality Reduction”. In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pp. 341–355.

[14] Tomáš Balyo. *HordeSAT - Public GIT Repository*. URL: `https://github.com/biotomas/hordesat`.

[15] Tomáš Balyo, Andreas Fröhlich, Marijn Heule, and Armin Biere. “Everything You Always Wanted to Know about Blocked Sets (But Were Afraid to Ask)”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pp. 317–332.

[16] Tomáš Balyo and Marijn J.H. Heule. *Proceedings of SAT Competition 2012; Solver and Benchmark Descriptions*. 2012.

[17] Tomáš Balyo and Marijn J.H. Heule. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*. 2013.

[18] Tomáš Balyo and Marijn J.H. Heule. *Proceedings of SAT Competition 2014; Solver and Benchmark Descriptions*. 2014.

[19] Tomáš Balyo and Marijn J.H. Heule. *Proceedings of SAT Competition 2016; Solver and Benchmark Descriptions*. 2016.

[20] Tomáš Balyo, Marijn J.H. Heule, and Matti Järvisalo. *Proceedings of SAT Competition 2017; Solver and Benchmark Descriptions*. 2017.

[21] Tomáš Balyo, Peter Sanders, and Carsten Sinz. “HordeSat: A Massively Parallel Portfolio SAT Solver”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pp. 156–172.

[22] Luís Baptista and João P. Marques Silva. “Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability”. In: *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, pp. 489–494.

[23] Ross Bencina. *OSCPack*. URL: `https://github.com/RossBencina/oscpack`.

[24] A. Biere, M. Heule, M. Järvisalo, and N. Manthey. “Equivalence checking of HWMCC 2012 Circuits”. In: *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions. Department of Computer Science Series of Publications, vol. B-2013-1, University of Helsinki, Helsinki*, p. 104.

[25] Armin Biere. *The AIGER And-Inverter Graph (AIG) Format Version 20071012*. Tech. rep. Report 07/1. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, 2007.

[26] Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. “Detecting Cardinality Constraints in CNF”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pp. 285–301.

[27] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Advances in Computers* 58 (2003), pp. 117–148.

[28] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. "Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pp. 60–71.

[29] Armin Biere and Carsten Sinz. "Decomposing SAT Problems into Connected Components". In: *JSAT* 2.1-4 (2006), pp. 201–208. URL: `https://satassociation.org/jsat/index.php/jsat/article/view/26`.

[30] Richard Boulanger, ed. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming.* Cambridge, MA, USA: MIT Press, 2000.

[31] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pp. 70–87.

[32] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-strength Verification Tool". In: *Proceedings of the 22Nd International Conference on Computer Aided Verification.* CAV'10, pp. 24–40.

[33] Robert Brummayer and Armin Biere. "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays". In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pp. 174–177.

[34] Robert Brummayer and Armin Biere. "Local two-level And-Inverter Graph minimization without blowup". In: *Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2006.*

[35] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691.

[36] Chin-Liang Chang and Richard C. T. Lee. *Symbolic logic and mechanical theorem proving.* Computer science classics. Academic Press, 1973.

[37] Edmund M. Clarke. "SAT-Based Counterexample Guided Abstraction Refinement". In: *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings*, p. 1.

[38] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pp. 154–169.

[39] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794.

[40] Bram Cohen. *Incentives Build Robustness in BitTorrent.* 2003.

[41] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pp. 151–158.

[42] Vijay D'Silva, Leopold Haller, and Daniel Kroening. "Abstract conflict driven learning". In: (2013), pp. 143–154.

[43] Vijay D'Silva, Leopold Haller, and Daniel Kroening. "Abstract satisfaction". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pp. 139–150.

[44] Evgeny Dantsin and Alexander Wolpert. "A Faster Clause-Shortening Algorithm for SAT with No Restriction on Clause Length". In: *JSAT* 1.1 (2006), pp. 49–60. URL: `https://satassociation.org/jsat/index.php/jsat/article/view/7`.

[45] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (1960), pp. 201–215.

[46] Nachum Dershowitz and David A. Plaisted. "Rewriting". In: *Handbook of Automated Reasoning (in 2 volumes)*. 2001, pp. 535–610.

[47] DIMACS. *Satisfiability Suggested Format.* 1993. URL: `http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`.

[48] Heidi E. Dixon, Matthew L. Ginsberg, David K. Hofer, Eugene M. Luks, and Andrew J. Parkes. "Generalizing Boolean Satisfiability III: Implementation". In: *J. Artif. Intell. Res.* 23 (2005), pp. 441–531.

[49] Heidi E. Dixon, Matthew L. Ginsberg, Eugene M. Luks, and Andrew J. Parkes. "Generalizing Boolean Satisfiability II: Theory". In: *J. Artif. Intell. Res.* 22 (2004), pp. 481–534.

[50] Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. "Generalizing Boolean Satisfiability I: Background and Survey of Existing Work". In: *J. Artif. Intell. Res.* 21 (2004), pp. 193–243.

[51] Niklas Eén and Armin Biere. "Effective Preprocessing in SAT Through Variable and Clause Elimination". In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, pp. 61–75.

[52] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pp. 502–518.

[53] Uwe Egly. "On Different Structure-Preserving Translations to Normal Form". In: *J. Symb. Comput.* 22.2 (1996), pp. 121–142.

[54] Uwe Egly and Thomas Rath. "On the Practical Value of Different Definitional Translations to Normal Form". In: *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, pp. 403–417.

[55] Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström, and Laurent Simon. "Seeking Practical CDCL Insights from Theoretical SAT Benchmarks". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pp. 1300–1308.

[56] Matthew England and Vijay Ganesh. "Preface". In: *Proceedings of the 2nd International Workshop on Satisfiability Checking and Symbolic Computation co-located with the 42nd International Symposium on Symbolic and Algebraic Computation (ISSAC 2017), Kaiserslautern, Germany, July 29, 2017.*

[57] Stephan Falke, Florian Merz, and Carsten Sinz. "LLBMC: Improved Bounded Model Checking of C Programs Using LLVM - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pp. 623–626.

[58] Katalin Fazekas, Armin Biere, and Christoph Scholl. “Incremental Inprocessing in SAT Solving”. In: *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, pp. 136–154.

[59] Zhaohui Fu and Sharad Malik. “Extracting Logic Circuit Structure from Conjunctive Normal Form Descriptions”. In: *20th International Conference on VLSI Design (VLSI Design 2007), Sixth International Conference on Embedded Systems (ICES 2007), 6-10 January 2007, Bangalore, India*, pp. 37–42.

[60] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “DPLL(T): Fast Decision Procedures”. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pp. 175–188.

[61] Serge Gaspers and Stefan Szeider. “Backdoors to Satisfaction”. In: *The Multivariate Algorithmic Revolution and Beyond - Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday*, pp. 287–317.

[62] Serge Gaspers and Stefan Szeider. “Strong Backdoors to Bounded Treewidth SAT”. In: *CoRR* abs/1204.6233 (2012).

[63] Stephan Gocht and Tomás Balyo. “Accelerating SAT Based Planning with Incremental SAT Solving”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, pp. 135–139.

[64] Evguenii I. Goldberg and Yakov Novikov. “BerkMin: A Fast and Robust Sat-Solver”. In: *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pp. 142–149.

[65] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. “Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems”. In: *J. Autom. Reasoning* 24.1/2 (2000), pp. 67–100.

[66] Carla P. Gomes, Bart Selman, and Henry A. Kautz. “Boosting Combinatorial Search Through Randomization”. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. Pp. 431–437.

[67] Steven Greenbaum, A. Nagasaka, Paul O’Rorke, and David A. Plaisted. “Comparison of Natural Deduction and Locking Resolution Implementations”. In: *6th Conference on Automated Deduction, New York, USA, June 7-9, 1982, Proceedings*, pp. 159–171.

[68] Armin Haken. “The Intractability of Resolution”. In: *Theor. Comput. Sci.* 39 (1985), pp. 297–308.

[69] Thomas Hermann, Andy Hunt, and John Neuhoff. *The Sonification Handbook.* Jan. 2011.

[70] Marijn J. H. Heule. “Schur Number Five”. In: *CoRR* abs/1711.08076 (2017). URL: `http://arxiv.org/abs/1711.08076`.

[71] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. *Proceedings of SAT Competition 2018; Solver and Benchmark Descriptions.* 2018.

[72] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. *Proceedings of SAT Competition 2019; Solver and Benchmark Descriptions.* 2019.

[73] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. “Encoding Redundancy for Satisfaction-Driven Clause Learning”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, pp. 41–58.

[74] Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. “PRuning Through Satisfaction”. In: *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, pp. 179–194.

[75] Marijn Heule and Armin Biere. “Blocked Clause Decomposition”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pp. 423–438.

[76] Marijn Heule, Matti Järvisalo, and Armin Biere. “Efficient CNF Simplification Based on Binary Implication Graphs”. In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, pp. 201–215.

[77] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. “Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads”. In: *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pp. 50–65.

[78] Andrei Horbach. “A Boolean satisfiability approach to the resource-constrained project scheduling problem”. In: *Annals OR* 181.1 (2010), pp. 89–107.

[79] International Organization for Standardization, ed. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996.

[80] *Ipasir*. URL: `https://github.com/biotomas/ipasir`.

[81] J.N.Hooker. “Solving the incremental satisfiability problem”. In: *The Journal of Logic Programming* 15.1 (1993), pp. 177–186.

[82] Paul B. Jackson and Daniel Sheridan. “Clause Form Conversions for Boolean Circuits”. In: *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pp. 183–198.

[83] Sima Jamali and David Mitchell. “Centrality-Based Improvements to CDCL Heuristics”. In: *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pp. 122–131.

[84] Matti Järvisalo and Armin Biere. “Reconstructing Solutions after Blocked Clause Elimination”. In: *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pp. 340–345.

[85] Matti Järvisalo, Armin Biere, and Marijn Heule. “Blocked Clause Elimination”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pp. 129–144.

[86] Matti Järvisalo, Armin Biere, and Marijn Heule. “Simulating Circuit-Level Simplifications on CNF”. In: *J. Autom. Reasoning* 49.4 (2012), pp. 583–619.

[87] Matti Järvisalo, Marijn Heule, and Armin Biere. "Inprocessing Rules". In: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pp. 355–370.

[88] Matti Järvisalo and Tommi A. Junttila. "Limitations of restricted branching in clause learning". In: *Constraints* 14.3 (2009), pp. 325–356.

[89] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. "ISAC - Instance-Specific Algorithm Configuration". In: *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, pp. 751–756.

[90] Hadi Katebi, Karem A. Sakallah, and João P. Marques Silva. "Empirical Study of the Anatomy of Modern Sat Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, pp. 343–356.

[91] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. 2015.

[92] Florian Krohm, Andreas Kuehlmann, and Arjen Mets. "The use of random simulation in formal verification". In: *1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings*, pp. 371–376.

[93] Andreas Kuehlmann. "Dynamic transition relation simplification for bounded property checking". In: *2004 International Conference on Computer-Aided Design, ICCAD 2004, San Jose, CA, USA, November 7-11, 2004*, pp. 50–57.

[94] Oliver Kullmann. "On a Generalization of Extended Resolution". In: *Discrete Applied Mathematics* 96-97 (1999), pp. 149–176.

[95] Stefan Kupferschmid, Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. "Incremental preprocessing methods for use in BMC". In: *Formal Methods in System Design* 39.2 (2011), pp. 185–204.

[96] Felix Kutzner. "Exploiting Gate-Structure to Direct CDCL Search via Variable Selection and Approximation". Diplomarbeit. Karlsruhe Institute of Technology, 2016.

[97] Jean-Marie Lagniez and Armin Biere. "Factoring Out Assumptions to Speed Up MUS Extraction". In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pp. 276–292.

[98] Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. "Speedup Techniques Utilized in Modern SAT Solvers". In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, pp. 437–443.

[99] Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. "Clause Vivification by Unit Propagation in CDCL SAT Solvers". In: *CoRR* abs/1807.11061 (2018). URL: `http://arxiv.org/abs/1807.11061`.

[100] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. "Learning Rate Based Branching Heuristic for SAT Solvers". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pp. 123–140.

[101] Feng Lu, Li-C. Wang, Kwang-Ting (Tim) Cheng, John Moondanos, and Ziyad Hanna. "A Signal Correlation Guided Circuit-SAT Solver". In: *J. UCS* 10.12 (2004), pp. 1629–1654.

[102] Inês Lynce and João Marques-Silva. "SAT in Bioinformatics: Making the Case with Haplotype Inference". In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pp. 136–141.

[103] Inês Lynce and João P. Marques Silva. "Hidden Structure in Unsatisfiable Random 3-SAT: An Empirical Study". In: *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, pp. 246–251.

[104] Norbert Manthey, Marijn Heule, and Armin Biere. "Automated Reencoding of Boolean Formulas". In: *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, pp. 102–117.

[105] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. "On Using Incremental Encodings in Unsatisfiability-based MaxSAT Solving". In: *JSAT* 9 (2014), pp. 59–81. URL: `https://satassociation.org/jsat/index.php/jsat/article/view/126`.

[106] Kenneth L. McMillan. *Symbolic model checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[107] Andrew Mihal and Steve Teig. "A Constraint Satisfaction Approach for Programmable Logic Detailed Placement". In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, pp. 208–223.

[108] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pp. 530–535.

[109] Alexander Nadel and Vadim Ryvchin. "Efficient SAT Solving under Assumptions". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pp. 242–255.

[110] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. "Preprocessing in Incremental SAT". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pp. 256–269.

[111] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. "Recovering and Exploiting Structural Knowledge from CNF Formulas". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pp. 185–199.

[112] David A. Plaisted and Steven Greenbaum. "A Structure-Preserving Clause Form Translation". In: *J. Symb. Comput.* 2.3 (1986), pp. 293–304.

[113] Jussi Rintanen. "Engineering Efficient Planners with SAT". In: *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pp. 684–689.

[114] Jarrod A. Roy, Igor L. Markov, and Valeria Bertacco. "Restoring Circuit Structure from SAT Instances". In: *Proceedings of international workshop on Logic and synthesis*, pp. 663–678.

[115] Lawrence Ryan. "Efficient algorithms for clause-learning SAT solvers". Master thesis. Simon Fraser University, 2004.

[116] *SAT Competition Website*. URL: `http://www.satcompetition.org/`.

[117] Thomas J. Schaefer. "The Complexity of Satisfiability Problems". In: *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pp. 216–226.

[118] Roberto Sebastiani. "Lazy Satisability Modulo Theories". In: *JSAT* 3.3-4 (2007), pp. 141–224. URL: `https://satassociation.org/jsat/index.php/jsat/article/view/39`.

[119] João P. Marques Silva. "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms". In: *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, pp. 62–74.

[120] João P. Marques Silva and Karem A. Sakallah. "Boolean satisfiability in electronic design automation". In: *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*, pp. 675–680.

[121] 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 1996, San Jose, CA, USA, November 10-14, 1996*, pp. 220–227.

[122] João P. Marques Silva and Luís Guerra e Silva. "Solving Satisfiability in Combinational Circuits". In: *IEEE Design & Test of Computers* 20.4 (2003), pp. 16–21.

[123] Carsten Sinz. "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pp. 827–831.

[124] Carsten Sinz. "Visualizing SAT Instances and Runs of the DPLL Algorithm". In: *J. Autom. Reasoning* 39.2 (2007), pp. 219–243.

[125] M. H. Stone. "The Theory of Representation for Boolean Algebras". In: *Transactions of the American Mathematical Society* 40.1 (1936), pp. 37–111.

[126] Emina Torlak and Daniel Jackson. "Kodkod: A Relational Model Finder". In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pp. 632–647.

[127] Thierry Boy de la Tour. "An Optimality Result for Clause Form Translation". In: *J. Symb. Comput.* 14.4 (1992), pp. 283–302.

[128] G. S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pp. 115–125.

[129] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Paramount, CA: CreateSpace, 2009.

[130] András Vörös, Dániel Darvas, Vince Molnár, Attila Klenik, Ákos Hajdu, Attila Jámbor, Tamás Bartha, and István Majzik. "PetriDotNet 1.5: Extensible Petri Net Editor and Analyser for Education and Research". In: *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*, pp. 123–132.

[131] Toby Walsh. "SAT v CSP". In: *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, pp. 441–456.

[132] Ingo Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons, Inc., 1987.

[133] Ryan Williams, Carla P. Gomes, and Bart Selman. "Backdoors To Typical Case Complexity". In: *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pp. 1173–1178.

[134] Matthew Wright. *Open Sound Control 1.0 Specification*. 2002. URL: `http://opensoundcontrol.org/spec-1_0`.

[135] Matthew Wright, Adrian Freed, and Ali Momeni. "Open Sound Control: State of the Art 2003". In: *New Interfaces for Musical Expression - NIME 2003 - 2nd International Conference, Montreal, Quebec, Canada*, pp. 153–159.

[136] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "SATzilla: Portfolio-based Algorithm Selection for SAT". In: *CoRR* abs/1111.2249 (2011). URL: `http://arxiv.org/abs/1111.2249`.

[137] Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. "The Effect of Structural Measures and Merges on SAT Solver Performance". In: *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, pp. 436–452.