



## Assignment of bachelor's thesis

<b>Title:</b>	Exploiting Logic Synthesis in SAT-solving
<b>Student:</b>	Jan Kimr
<b>Supervisor:</b>	doc. Ing. Petr Fišer, Ph.D.
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Artificial Intelligence 2021
<b>Department:</b>	Department of Applied Mathematics
<b>Validity:</b>	until the end of summer semester 2024/2025

### Instructions

This work aims to explore the influence of logic synthesis on SAT solving. This is, to experimentally find out what simplifications (transformations) of logic expressions (CNF) lead to a SAT solver speed-up. Both positive and negative influences are reported in literature. However, no thorough exploration has been done yet.

Particular goals of the Thesis are:

- Apply different logic synthesis techniques to different SAT problem instances.
- Perform respective experiments and determine for what types of instances the optimization reduces the overall SAT-solving time and which optimization process is the most efficient.
- Perform the experiments using several available (open-source) SAT solvers and compare the results.
- In the case of positive findings (i.e., the pre-optimization reduces the overall run time), use the obtained knowledge to optimize some existing algorithms repeatedly calling the SAT-solver (e.g., SAT-based ATPG).

Bachelor's thesis

# **EXPLOITING LOGIC SYNTHESIS IN SAT-SOLVING**

**Jan Kimr**

Faculty of Information Technology  
Department of Applied Mathematics  
Supervisor: doc. Ing. Petr Fišer, Ph.D.  
May 15, 2024

Czech Technical University in Prague  
Faculty of Information Technology

© 2024 Jan Kimr. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

Citation of this thesis: Kimr Jan. *Exploiting Logic Synthesis in SAT-solving*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

## Contents

<b>Acknowledgments</b>	<b>vi</b>
<b>Declaration</b>	<b>vii</b>
<b>Abstract</b>	<b>viii</b>
<b>List of abbreviations</b>	<b>ix</b>
<b>Introduction</b>	<b>1</b>
Objectives . . . . .	1
<b>1 Theory</b>	<b>3</b>
1.1 Boolean Satisfiability Problem (SAT) . . . . .	3
1.1.1 Tseitin transformation . . . . .	4
1.1.2 Encoding problems and practical use of SAT . . . . .	6
1.2 SAT solving algorithms . . . . .	6
1.2.1 DPLL algorithm . . . . .	7
1.2.2 Conflict-driven clause learning (CDCL) . . . . .	8
1.3 Solvers used in the experimental part . . . . .	8
1.3.1 zChaff . . . . .	8
1.3.2 MiniSat . . . . .	9
1.3.3 Glucose . . . . .	9
1.4 Logic synthesis . . . . .	10
1.5 Optimum circuit generation . . . . .	10
1.6 Automatic Test Pattern Generation (ATPG) . . . . .	10
1.7 Machine learning models and related topics used in evaluating practical use of synthesis . . . . .	11
1.7.1 Model training and evaluation . . . . .	11
1.7.2 Linear regression . . . . .	12
1.7.3 Logistic regression . . . . .	12
1.7.4 Naive Bayes . . . . .	12
1.7.5 K-nearest neighbours . . . . .	12
1.7.6 Decision tree . . . . .	13
1.7.7 Random forest . . . . .	13
1.7.8 Gradient boosting decision trees . . . . .	13
1.7.9 Data preprocessing (feature normalization) . . . . .	14
1.8 Usage of logic synthesis in SAT-solving . . . . .	14

<b>2 Experiments</b>	<b>15</b>
2.1 Experimental Setup . . . . .	15
2.2 SATLIB benchmark instances . . . . .	16
2.2.1 Synthesis influence on CNF properties . . . . .	16
2.2.2 Synthesis influence on solving time . . . . .	17
2.3 MinCirc instances . . . . .	26
2.3.1 Synthesis influence on CNF properties . . . . .	26
2.3.2 Synthesis influence on solving time . . . . .	26
2.4 ATPG instances . . . . .	36
2.4.1 Synthesis influences on solving time . . . . .	36
<b>3 Using synthesis in SAT-solving</b>	<b>39</b>
3.1 Explored approaches . . . . .	39
3.1.1 Restarting . . . . .	39
3.1.2 Selecting instances to run with synthesis . . . . .	39
3.1.3 Selecting instances for restarting . . . . .	40
3.2 Model training and evaluation . . . . .	40
3.3 MinCirc instances . . . . .	41
3.3.1 Data . . . . .	41
3.3.2 Results . . . . .	42
3.4 ATPG instances . . . . .	42
3.4.1 Data . . . . .	43
3.4.2 Results . . . . .	43
<b>4 Discussion</b>	<b>46</b>
<b>5 Conclusion</b>	<b>48</b>
<b>Contents of the attachment</b>	<b>54</b>

## List of Figures

1.1	Example of decision tree for binary classification . . . . .	13
-----	--	----

## List of Tables

2.1	SATLIB benchmark: influence of logic synthesis on the number of variables	17
2.2	SATLIB benchmark: influence of logic synthesis on the number of clauses	18
2.3	SATLIB benchmark: influence of logic synthesis on the clause-variable ratio	18
2.4	SATLIB benchmark: numbers of unsatisfiable instances which were transformed by synthesis into trivial ones (i.e., for variable $a$ , the CNF would be $a \wedge \neg a$ ) . . . . .	19
2.5	SATLIB benchmarks: separated by slashes are the numbers of instances (1) which were solved in 2000 seconds (real-time), both with and without synthesis, (2) which were solved in 2000 seconds (real-time) with synthesis but not without it, (3) which were solved in 2000 seconds (real-time) without synthesis but not with it, (4) which were solved in 2000 seconds (real-time) neither with nor without synthesis. In case values (2), (3), and (4) are all zeroes, they are omitted. . . . .	22
2.6	SATLIB benchmarks: the ratio of instances whose time of solving decreased after using the synthesis preprocessing. Instances that were solved neither with nor without synthesis are left out. . . . .	23
2.7	SATLIB benchmarks: potential influence of synthesis on the solving time (i.e., using synthesis only when it decreases the solving time of an instance). Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit. . . . .	24
2.8	SATLIB benchmarks: the influence of synthesis on the solving time if used always; values larger than or equal to +1000% are replaced by “-”. Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit. . . . .	25
2.9	MinCirc: influence of logic synthesis on the number of variables . . . . .	27
2.10	MinCirc: influence of logic synthesis on the number of clauses . . . . .	27
2.11	MinCirc: influence of logic synthesis on the clause-variable ratio . . . . .	28

2.12	MinCirc: numbers of unsatisfiable instances which were transformed by synthesis into trivial ones (i.e., for variable $a$ , the CNF would be $a \wedge \neg a$ ) .	28
2.13	MinCirc: separated by slashes are the numbers of instances (1) which were solved in 2000 seconds (real-time) both with and without synthesis, (2) which were solved in 2000 seconds (real-time) with synthesis but not without it, (3) which were solved in 2000 seconds (real-time) without synthesis but not with it, (4) which were solved in 2000 seconds (real-time) neither with nor without synthesis. In case values (2), (3), and (4) are all zeroes, they are omitted. . . . .	32
2.14	MinCirc: ratio of instances whose time of solving decreased after using the synthesis preprocessing. Instances that were solved neither with nor without synthesis are left out. . . . .	33
2.15	MinCirc: potential influence of synthesis on the solving time (i.e., using synthesis only when decreases the solving time of an instance). Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit. . . . .	34
2.16	MinCirc: the influence of synthesis on the solving time if used always; values larger than or equal to +1000% are replaced by “-”. Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit.	35
2.17	ATPG circuits: ratio of circuits whose time of solving decreased after using logic synthesis . . . . .	37
2.18	ATPG circuits: potential influence of synthesis on solving time (i.e., using synthesis only when decreases solving time of an instance) . . . . .	37
2.19	ATPG circuits: influence of synthesis on solving time if used always . . . . .	38
3.1	Practical use of synthesis: results of ten best models by a decrease in solving time on MinCirc instances using the MINISAT solver with <i>st</i> synthesis	43
3.2	Practical use of synthesis: results of ten best models by a decrease in solving time on MinCirc instances using the MINISAT solver with <i>6-LUT-1x</i> synthesis . . . . .	43
3.3	Practical use of synthesis: results of ten best models by a decrease in solving time on ATPG instances with <i>st</i> synthesis . . . . .	45
3.4	Practical use of synthesis: results of ten best models by a decrease in solving time on ATPG instances with <i>st-re2-1x</i> synthesis . . . . .	45

## List of code listings

1.1	DPLL algorithm pseudocode [16] . . . . .	7
-----	--	---

*I would like to thank my supervisor, doc. Ing. Petr Fišer, Ph.D., for all the help and valuable comments he provided me during the writing of this thesis.*



## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 15, 2024

## Abstract

This thesis analyzes the possibility of using logic synthesis and optimization to speed up solving satisfiability problem (SAT) instances coming from both the standard benchmarks and different practical applications.

Logic optimization, in principle, influences an instance in two main ways: (1) it reduces its size, and (2) structures making the instance difficult to solve could be dissolved. However, it is necessary to consider both times of synthesis and solving, as speed up in solving achieved by “stronger” synthesis could be outweighed by synthesis time, leading to an overall time increase.

The efficiency of logic synthesis and optimization was evaluated using different instances and syntheses. It can both positively and negatively influence the overall time of solving, which mostly depends on the instance size and used synthesis. Based on the results, recommendations regarding the practical use of synthesis are made.

**Keywords** SAT, Boolean satisfiability, logic synthesis, logic optimization, optimum circuit generation, ATPG

## Abstrakt

Tato práce se zabývá možností využití logické syntézy a optimalizace ke zrychlení řešení instancí problému splnitelnosti (SAT) pocházejících ze standardních benchmarků i praktických aplikací.

Logická syntéza může v principu ovlivnit instanci dvěma způsoby: (1) zmenšením velikosti a (2) rozptýlením struktur, které komplikovaly řešení původní instance. Nicméně při použití syntézy je nutné vzít v potaz nejen čas řešení upravené instance, ale i čas syntézy. Může se stát, že zrychlení dosažené při řešení SATu bude převáženo časem syntézy samotné.

Efektivita logické syntézy a optimalizace byla vyhodnocena na různých instancích a s různými syntézami. Syntéza ovlivňuje celkový čas řešení pozitivně i negativně v závislosti zejména na velikosti instance a použité syntéze. Na základě naměřených výsledků jsou popsány možnosti praktického využití syntézy.

**Klíčová slova** SAT, splnitelnost Booleovských formulí, logická syntéza, logická optimalizace, generování optimálních obvodů, ATPG

## List of abbreviations

AI	Artificial Intelligence
AIG	AND-inverter-graph
ATPG	Automatic Test Pattern Generation
BLIF	Berkeley Logic Interchange Format
CDCL	Conflict-Driven Clause Learning
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
CV	Cross-validation
DAG	Directed Acyclic Graph
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science
DPLL or DLL	Davis–(Putnam)–Logemann–Loveland algorithm
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
LBD	Literal Block Distance
LUT	LookUp Table
ML	Machine Learning
NB	Naive Bayes
PI	Primary Input
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
VSIDS	Variable State Independent Decaying Sum

# Introduction

The Boolean satisfiability problem (SAT) involves determining whether some variable valuation satisfies a given formula and, if so, finding such valuation. Although this problem is NP-complete, fast and efficient SAT solvers have been developed in the last two decades. This development has made it possible to use these solvers in practical areas such as artificial intelligence or circuit design.

Specifically, in circuit design, many Electronic Design Automation (EDA) tools use SAT or its derivatives, like pseudo-Boolean optimization or Satisfiability modulo theories (SMT). Furthermore, SAT is used for model-checking engines and the Automated Test Pattern Generation process (ATPG). Even though SAT solvers should be efficient enough to solve instances reasonably fast, large and difficult-to-solve instances still occur in areas such as optimum circuit generation or ATPG.

The SAT instances are most commonly represented in a Conjunctive Normal Form (CNF), which can be preprocessed and optimized by logic optimization tools, hoping that the resulting SAT problem will be solved faster. Nevertheless, both time of optimization and SAT-solving must be considered; thus, finding the optimal trade-off for minimizing the run time is necessary.

This approach is not new; multiple studies dealing with the application of logic synthesis in the SAT-solving process have been published [1, 2]. However, there is no definite conclusion, as the sets of tested instances were rather limited.


This thesis focuses on experimentally evaluating three case studies in which logic optimization could speed up SAT-solving. Based on the obtained results, we attempt to make a conclusive recommendation for using (or not using) logic optimization in the SAT-solving process.

## Objectives

The primary goals of this thesis are the following:

- Apply different logic synthesis techniques to different SAT problem instances.
- Perform respective experiments and determine for what types of instances the optimization reduces the overall SAT-solving time and which optimization process is the most efficient.

- Perform the experiments using several available (open-source) SAT solvers and compare the results.
- In the case of positive findings (i.e., the pre-optimization reduces the overall run time), use the obtained knowledge to optimize some existing algorithms repeatedly calling the SAT-solver (e.g., SAT-based ATPG).



# Chapter 1

## Theory

In this chapter, topics and concepts used in Chapters 2 and 3 are introduced. We start by defining the SAT problem and describing its usefulness, followed by a brief description of algorithms for solving it. Specifics of three different implementations of these algorithms – SAT solvers used in the following chapters are then provided.

Logic synthesis can be used to preprocess SAT instances before solving them. It is briefly explained what it is and how it can be used specifically for SAT instance preprocessing.

Two algorithms, where the SAT is frequently used, are presented to give an idea of practical problems where it can be used and how the particular SAT instances are created. These are optimum circuit generation and (SAT-based) ATPG. The influence of logic synthesis on instances created while solving these problems is evaluated in the following chapters.

In most cases, logic synthesis speeds up the solving of only a small number of SAT instances (around 10–30%). One of the approaches to utilize it, despite the low rate of improved instances, is to find a method of selecting instances that should be preprocessed using synthesis to speed up as many instances as possible while not slowing down others. For this purpose, machine learning models are used. An overview of different models, as well as other aspects of machine learning, is provided.

Since the approach of using synthesis to speed up SAT-solving is not novel, the results of two papers on this topic are shortly discussed.

### 1.1 Boolean Satisfiability Problem (SAT)

A Boolean formula consists of variables that can be assigned a logic value, either 0 or 1, logical connectives ( $\neg$ ,  $\wedge$ ,  $\vee$ ), and parentheses determining the priority of operations. There are other logical connectives than the presented ones (i.e.,  $\implies$ ,  $\iff$ ); nevertheless, each formula can be rewritten to one using just  $\neg$ ,  $\wedge$ , and  $\vee$ . When all variables are assigned values, the formula evaluates to either 0 or 1.

The SAT is a problem of determining whether, for a given Boolean formula, there is such variable assignment that evaluates the formula to 1 (i.e., the formula is satisfiable) or no such assignment exists (i.e., the formula is unsatisfiable). For a satisfiable formula,

one of the satisfying variable valuations is usually reported.

The SAT problem was the first one to be proven to be NP-complete [3]. As a result (unless  $P=NP$ ), SAT solving has the worst-case exponential time complexity, as for  $n$  variables; there are  $2^n$  possible truth assignments to be checked. On the other hand, checking whether some variable valuation satisfies the formula can be done in polynomial time. In practice, however, instances are usually far from the worst-case, and solvers can solve large instances with many variables and clauses [4].

Each Boolean formula can be written in the conjunctive normal form (CNF), sometimes called the Product of Sums form. A literal is a variable or its negation; a clause is a literal or disjunction of multiple literals; and a formula is in CNF if it is a clause or conjunction of multiple clauses. A useful property of CNF is that in order for the formula to be satisfied, all clauses must be satisfied; conversely, if one clause is not satisfied, the whole formula can not be satisfied. An unsatisfied clause is called a conflicting clause.

There are usually many functionally equivalent CNF representations of a formula. Equation 1.1 shows an example Boolean formula in CNF. This formula has multiple satisfying assignments, one of them being  $a = 0$ ,  $b = 1$ , and  $c = 1$ .

$$(\neg a \vee b \vee c) \wedge (\neg a \vee c) \wedge (b \vee \neg c) \tag{1.1}$$

There are many solvers for the SAT problem (e.g., [4, 5, 6]), which can be divided into two main groups, incomplete and complete. An incomplete solver is not guaranteed to return an answer – it has some resource limit and either finds a solution or reports a failure. These solvers usually specialize in finding valid solution while not attempting to prove the formula unsatisfiable. Incomplete solvers are usually significantly faster compared to complete solvers which will always find a solution or prove unsatisfiability [7].

Most solvers accept formulas written in CNF, usually in DIMACS format [8]. The use of CNF is not limiting, as transforming a formula to CNF can be done in polynomial time using the Tseitin transformation [9].

### 1.1.1 Tseitin transformation

Using naively de Morgan laws to transform a given Boolean formula to CNF can lead to an exponential blowup in the worst case. Fortunately, using the Tseitin transformation [9], we can get an equisatisfiable (i.e., satisfiable iff the original formula is satisfiable) formula in CNF of any Boolean formula. The resulting formula's size is linear in the original formula's size and can be found in polynomial time.

We can show the steps of the transformation on the formula  $F$ .

$$F := (a \wedge b) \implies \neg(c \vee \neg d) \tag{1.2}$$

We introduce a new auxiliary variable for each subformula (everything more than just a simple variable) and for the whole formula. We start with the simplest subformulas and use previously created auxiliary variables to ensure that each variable is assigned only a negation of another variable or a formula consisting of two variables joined by a binary operation.

$$\begin{aligned}
 x_1 &:= a \wedge b \\
 x_2 &:= \neg d \\
 x_3 &:= c \vee x_2 \\
 x_4 &:= \neg x_3 \\
 x_5 &:= x_1 \implies x_4
 \end{aligned}$$

Then we put each auxiliary variable into equivalence with its formula, forming its characteristic function.

$$\begin{aligned}
 x_1 &\iff (a \wedge b) \\
 x_2 &\iff \neg d \\
 x_3 &\iff (c \vee x_2) \\
 x_4 &\iff \neg x_3 \\
 x_5 &\iff (x_1 \implies x_4)
 \end{aligned}$$

We continue by modifying these equivalences (characteristic functions) using Boolean rules until they are in CNF. Since there is a finite number of binary operations, it is possible to prepare the CNF for each operation beforehand and then just swap the variables for the actual ones.

$$\begin{aligned}
 x_1 \iff (a \wedge b) &\equiv (x_1 \implies (a \wedge b)) \wedge ((a \wedge b) \implies x_1) \\
 &\equiv (\neg x_1 \vee (a \wedge b)) \wedge (\neg a \vee \neg b \vee x_1) \\
 &\equiv (\neg x_1 \vee a) \wedge (\neg x_1 \vee b) \wedge (\neg a \vee \neg b \vee x_1) \\
 x_2 \iff \neg d &\equiv (x_2 \implies \neg d) \wedge (\neg d \implies x_2) \\
 &\equiv (\neg x_2 \vee \neg d) \wedge (d \vee x_2) \\
 x_3 \iff (c \vee x_2) &\equiv (x_3 \implies (c \vee x_2)) \wedge ((c \vee x_2) \implies x_3) \\
 &\equiv (\neg x_3 \vee c \vee x_2) \wedge ((\neg c \wedge \neg x_2) \vee x_3) \\
 &\equiv (\neg x_3 \vee c \vee x_2) \wedge (\neg c \vee x_3) \wedge (\neg x_2 \vee x_3) \\
 x_4 \iff \neg x_3 &\equiv (x_4 \implies \neg x_3) \wedge (\neg x_3 \implies x_4) \\
 &\equiv (\neg x_4 \vee \neg x_3) \wedge (x_3 \vee x_4) \\
 x_5 \iff (x_1 \implies x_4) &\equiv (x_5 \implies (x_1 \implies x_4)) \wedge ((x_1 \implies x_4) \implies x_5) \\
 &\equiv (\neg x_5 \vee \neg x_1 \vee x_4) \wedge ((x_1 \wedge \neg x_3) \vee x_5) \\
 &\equiv (\neg x_5 \vee \neg x_1 \vee x_4) \wedge (x_1 \vee x_5) \wedge (\neg x_4 \vee x_5)
 \end{aligned}$$

As all the characteristic functions must be satisfied, they are concatenated by conjunction, forming the resulting formula  $F_T$  in CNF.



$$\begin{aligned}
F_T := & x_5 \wedge \\
& (\neg x_1 \vee a) \wedge (\neg x_1 \vee b) \wedge (\neg a \vee \neg b \vee x_1) \wedge \\
& (\neg x_2 \vee \neg d) \wedge (d \vee x_2) \wedge \\
& (\neg x_3 \vee c \vee x_2) \wedge (\neg c \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge \\
& (\neg x_4 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge \\
& (\neg x_5 \vee \neg x_1 \vee x_4) \wedge (x_1 \vee x_5) \wedge (\neg x_4 \vee x_5)
\end{aligned}$$

$F_T$  is not equivalent to the original formula  $F$ . It is only equisatisfiable. Nevertheless, if we have a variable assignment satisfying  $F_T$ , by dropping auxiliary variables, we get a variable assignment that satisfies the original formula  $F$  [10].

### 1.1.2 Encoding problems and practical use of SAT

Instances of many problems can be encoded in CNF and solved using a SAT solver. This can be beneficial since SAT solvers are fast, continuously developed, and optimized. Nevertheless, there are usually more strategies for encoding instances of a chosen problem, with encoding techniques having a significant impact on the efficiency of the solver [11].

We can use a simple problem called the Pigeon-hole problem to demonstrate encoding in CNF. This problem involves placing  $n + 1$  pigeons in  $n$  holes with, at most, one pigeon in each hole [12]. This obviously is not possible, but it shows how a simple problem can be transformed into the SAT.

For each pigeon  $i$  and each hole  $j$ , we create variable  $x_{i,j}$ , which is true if pigeon  $i$  is placed in hole  $j$ . For each of  $n + 1$  pigeons, we create a clause that ensures that each pigeon is placed in some hole, i.e.,  $\forall i \in \{1, \dots, n + 1\} : \bigvee_{j=1}^n x_{i,j}$ . Then, for each hole  $j$ , we create a set of clauses allowing at most one pigeon to be present, i.e.,  $\forall j \in \{1, \dots, n\} ; \forall i_1, i_2 \in \{1, \dots, n + 1\}, i_1 \neq i_2 : \neg x_{i_1,j} \vee \neg x_{i_2,j}$ . In the end, we have  $n \cdot (n + 1)$  variables and  $(n + 1) + n \cdot \frac{n \cdot (n + 1)}{2}$  clauses [12].

The development of efficient SAT solvers led to widespread use of SAT in many areas and problem domains: “*Examples include model-checking of finite-state systems, design debugging, AI planning, and haplotype inference in bioinformatics. Additional successful examples of practical applications of SAT include knowledge-compilation, software model checking, software testing, package management in software distributions, checking of pedigree consistency, test-pattern generation in digital systems, design debugging and diagnosis, identification of functional dependencies in Boolean functions, technology-mapping in logic synthesis, circuit delay computation, as well as the ones mentioned above. However, this list is incomplete as the number of applications of SAT has been on the rise in recent years.*” [13]

## 1.2 SAT solving algorithms

In this section, two main complete SAT-solving algorithms are described. The first one, DPLL, is not that fast and memory efficient on its own; however, it can be extended by clause learning and backjumping (CDCL), which is described the second part of this section. Many modern solvers use a combination of these two algorithms.

### 1.2.1 DPLL algorithm

In 1960, Davis and Putnam published a two-part algorithm for predicate logic theorem proving [14]. In the first part, a growing CNF formula is generated, while the second tests the formula's satisfiability. Two years later, the second part was improved by Davis, Logemann, and Loveland in [15]. In the improved version, while testing satisfiability, the formula is split into multiple parts, which are then evaluated separately as it significantly improves memory efficiency.

From this, a recursive algorithm called DPLL (or DLL) is built (Code listing 1.1). Both the pseudocode and the algorithm description are based on [16]. The DPLL function is called with the whole formula and an empty set of variable assignments.

The process starts by deducing assignments from the current one. This is usually done using *unit clause rule*<sup>1</sup> – if all but one literal of a clause were assigned value 0, the reminding one must be assigned value 1. In the deduction function, values of multiple variables can be deduced as assigning value to one can result in new unit clauses. During the deduction, if one variable should be assigned both 0 and 1 in order to satisfy different unit clauses, a conflict arises.

If the deduction finishes without a conflict, an unassigned variable is selected and assigned a value, and the algorithm calls itself recursively with the union of previous variable assignments and those assigned in the current step. If the deduction results in a conflict, the algorithm backtracks and tries different values for variables.

The algorithm ends if a satisfying variable assignment is found or if all possible assignments are checked (in this case, the formula is unsatisfiable). It can be extended to return a satisfying assignment as well.

■ **Code listing 1.1** DPLL algorithm pseudocode [16]

```
DPLL(formula, assignment) {
    necessary = deduction(formula, assignment);
    new_asgnmnt = union(necessary, assignment);
    if (is_satisfied(formula, new_asgnmnt))
        return SATISFIABLE;
    else if (is_conflicting(formula, new_asgnmnt))
        return CONFLICT;
    var = choose_free_variable(formula, new_asgnmnt);
    asgn1 = union(new_asgnmnt, assign(var, 1));
    if (DPLL(formula, asgn1) == SATISFIABLE)
        return SATISFIABLE;
    else {
        asgn2 = union(new_asgnmnt, assign(var, 0));
        return DPLL(formula, asgn2);
    }
}
```

If we store assigned variables in the order they were assigned value, we can divide them into groups based on the decision level. Each manual value assignment creates a

<sup>1</sup>This rule is based on “*Rule for the Elimination of One-Literal Clauses*” [14]. However, it differs slightly as solvers do not eliminate variables with an assigned value, so it is an *unsatisfied clause with one unassigned literal* instead of a *clause with only one literal*. This clause is usually called *unit clause*.

new decision level, and the manually assigned variable is the first variable on the new level. Variables assigned values before any decision are on level 0 [17].

The algorithm can be rewritten into an iterative form, using checkpoint symbols to keep track of decision levels and to simulate the stack used in the recursive version. The iterative version has two main benefits: (1) it avoids creating a new stack each time a variable is manually assigned value [17], and (2) it enables *non-chronological backtracking* (i.e., returns more than one decision level at once). *Non-chronological backtracking* is described in the following subsection.

### 1.2.2 Conflict-driven clause learning (CDCL)

*Conflict-driven clause learning* consists of two parts: *clause learning* and *non-chronological backtracking*. It extends and significantly improves the DPLL algorithm. It was introduced in 1996 in GRASP solver [18] and it is used by many modern solvers [17]. This description is based on [18].

The idea of *clause learning* is that conflicts are inevitable and that we can learn from the “mistakes” that led to them. If a conflict occurs, it is a result of some variable assignments the solver chose earlier (or that the formula is unsatisfiable). As variable assignments are either chosen by the solver or deduced from the choices (using the *unit clause rule*), we can find which choices of the solver lead to the conflict and learn a new clause from them. This clause consists of literals representing negations of variable assignments leading to the conflict. E.g., if assignments  $a = 0$ ,  $b = 1$ , and  $c = 1$  caused the conflict, the newly learned clause would be  $a \vee \neg b \vee \neg c$ .

The conflict can either result from the last chosen assignment (current decision level) or from the previous ones (lower decision levels). In the first case, it implies the opposite assignment of the last variable (if it has not been tried already). In the second case, the solver can backtrack to the highest decision level on which the conflict can yet be avoided. Multiple levels can be backtracked at once since conflict will always arise after assigning all variables leading to it. This second case is called *non-chronological backtracking* and can help to prune the search space significantly. If the solver backtracks before the first decision, the instance is proven unsatisfiable.

These methods speed up the solving of structured instances, i.e., the instances have some inner structure that does not depend on the instance size; industrial and real-world instances usually do have some structure. While solving random instances, they do not help [4].

## 1.3 Solvers used in the experimental part

In this section, solvers used in Chapter 2 are described. All three solvers use the CDCL algorithm, with the main differences being strategies for selecting the variable to assign a value to and for keeping/removing learned clauses and an approach to restarts.

The solvers are listed in order of their first appearance, from oldest to newest.

### 1.3.1 zChaff

zCHAFF was created in 2001, improved in 2004 [4], and then slightly modified in 2007 [19].

The search procedure follows the standard conflict-driven idea utilizing backjumping. It uses a heuristic variable selection method called Variable State Independent Sum (VSIDS). It keeps scores for each literal of a variable, which is incremented each time a newly learned clause contains this literal. After a fixed time period, all scores are divided by 2. The score of a variable is greater of two associated literal scores. The variable with the highest score is assigned value – true if a positive literal has higher VSIDS and false otherwise.

zCHAFF uses methods to learn shorter clauses, which lead to faster clause propagation and can significantly prune the search space. Usage statistics and clause length are used to estimate its usefulness. Less useful clauses are periodically deleted.

Lastly, frequent restarts are used. After restarting, all VSIDS scores are set to 0, and the first variable is chosen arbitrarily [4].

### 1.3.2 MiniSat

MINISAT is a small, efficient, and easily extendable SAT-solver. It was developed in 2003 and modified in the following years [20].

It uses a similar heuristic for selecting variables as zCHAFF. Each variable is assigned an *activity* value, which is increased each time it appears in a new conflict clause. After each conflict, all *activity* values are multiplied by a constant smaller than 1. The unassigned variable with the highest activity is assigned a value first.

The clauses have *activity* is well. When a learned clause is used in the analysis of conflict, its *activity* is increased. Learned clauses with lower activity are periodically deleted. By default, MINISAT starts with a small set of learned clauses and gradually increases its size [5].

It also utilizes restarts to escape from hopeless parts of the search tree. The number of conflicts leading to restart increases during solving [5].

### 1.3.3 Glucose

The GLUCOSE [6] SAT-solver is based on MINISAT [5]. It was introduced in 2009 and was significantly improved in the following years.

The main extension is a new quality measure of learned clauses called Literal Block Distance (LBD). LBD is calculated as: “*Given a clause  $c$ , and a partition of its literals into  $n$  subsets according to the current assignment, such that literals are partitioned w.r.t their decision level. The LBD of  $c$  is exactly  $n$ .*” Practical results show LBD is a very good indicator of learned clause usefulness [6].

To speed up clause propagation, the number of learned clauses is repeatedly halved so that their number slightly increases each time clauses are halved. Clauses with lower LBD are kept.

The restart policy is based on the quality of recently learned clauses, which has proven to be efficient in practice. When restarting, learned clauses remain. Moreover, if the solution seems to be near, the restart can be postponed.

An additional extension is parallel solving. Solvers solve the whole problem on each CPU separately and exchange the most useful learned clauses with each other [6].

## 1.4 Logic synthesis

“Logic synthesis transforms HDL code into a netlist describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required.” [21]

A SAT instance can be seen as a circuit with one input for each variable and one output for the whole formula. This circuit can then be optimized using logic synthesis and transformed back to CNF using the Tseitin transformation [9] (see Subsection 1.1.1), hoping that the resulting SAT instance will be “easier” to solve.

In the experimental part (Chapter 2), the ABC tool [22] is used for synthesis. ABC is a public-domain tool for logic synthesis and verification. It uses And-Inverter Graphs<sup>2</sup> (AIGs) for representing circuits. Using AIG-based optimization repeated multiple times can be more efficient than approaches used by earlier tools while achieving the same or better results and using less memory and runtime, thus enabling its use on much larger circuits [23].

## 1.5 Optimum circuit generation

An optimum circuit is a circuit representing a given  $k$ -variable Boolean function that is optimal in some aspect – usually has the minimum number of gates. Optimum circuits can be used in logic synthesis, e.g., the *rewriting* algorithm [24] repetitively rewrites subcircuits by their smaller, preferably optimum representations.

SAT-based method of generating the optimum circuit of a function is presented in [25]. The described procedure is as follows: for the  $k$ -variable function, start with  $n = 1$  and attempt to find optimum implementation using  $n$  gates by generating and solving a corresponding SAT instance; if the instance is unsatisfiable, such implementation does not exist, increase  $n$  by one and repeat the process; if the instance is satisfiable extract optimum circuit structure from it.

In Section 2.3, the use of logic synthesis on SAT instances created while generating optimum implementations of four- and five-variable functions is evaluated.

## 1.6 Automatic Test Pattern Generation (ATPG)

Automatic test pattern generation (ATPG) is a method of generating patterns (inputs) for testing the presence of defects in a physical circuit. It is possible to convert an ATPG instance into multiple SAT instances, which can then be solved by a SAT solver. If the SAT instance is satisfiable, a test pattern is extracted from the satisfying assignment. For unsatisfiable instances, no test pattern for the fault exists. This approach is called SAT-based ATPG [26].

In Section 2.4, the effectiveness of logic synthesis is evaluated on SAT instances generated by a simple SAT-based ATPG based on the original idea of Larrabee [26].

---

<sup>2</sup>“An And-Inverter Graph is a directed acyclic graph (DAG), in which a node has either 0 or 2 incoming edges. A node with no incoming edges is a primary input (PI). A node with 2 incoming edges is a two-input AND gate. An edge is either complemented or not. A complemented edge indicates the inversion of the signal.” [23]

This approach generates conceptual hardware (*miter*) by XORing the fault-free and faulty circuit. This miter is then converted to a CNF by the Tseitin transformation [9] (see Subsection 1.1.1), and a test vector is generated as a satisfiability proof.

## 1.7 Machine learning models and related topics used in evaluating practical use of synthesis

In this section, machine learning (ML) models and related topics used in Chapter 3 are described. All described models are from the part of ML called supervised learning. This means that the desired outcomes are known. The opposite of supervised learning is unsupervised, where we do not know the outcomes, and the goal is to understand the structure of the data or some aspects of it [27].

In supervised learning, models are trained on many pairs of data–annotation to predict some target value (e.g., predict a digit from an image consisting of pixel data). The ability of the model is evaluated using some metric (e.g., ratio of correctly predicted digits). The goal is generalization – to create a model performing as well as possible on previously unseen data [27].

The two most common ML tasks are classification and regression. In classification, the model predicts one of multiple classes, while in regression, a real (continuous) value is predicted. Some models can be used for both tasks only with small modifications, while others can only be used for one of them.

### 1.7.1 Model training and evaluation

There are two main methods of using data for training and evaluating models. The simpler one divides the dataset into three parts: training, development, and test sets. Models are trained on the training set. The development set is used to compare different models and hyperparameters and to select the best one, and the test set is used to evaluate the best model’s generalization ability (i.e., the ability to predict on previously unseen data). This approach is used in most cases; however, if the dataset is small, it might be better to use cross-validation [28].

The basic method of cross-validation is  $k$ -fold cross-validation, which divides the data into  $k$  similarly sized parts. Then  $k$  models are trained and evaluated. Each one is trained on  $k - 1$  parts of the data and evaluated on the remaining one (each part is used for evaluation only once). The results of the model on the evaluation parts are then averaged together [28].

There are two approaches to selecting the best model using cross-validation. Firstly, the data can be divided into two sets: a training set and a test set. The best model is selected using cross-validation on the training set, and its generalization ability is evaluated on the test set. Secondly, if we do not want to divide the test set, we can use nested cross-validation to select the best model and evaluate it. An inner cross-validation is used to select the best model, and the outer is used to evaluate it. The average of evaluation results on outer cross-validation is then returned [29].

### 1.7.2 Linear regression

Linear regression is one of the simplest models in machine learning. It is used for regression. It models linear relation between the data and targets. Interestingly, an exact formula for calculating weights of the model exists: for data  $\mathbf{X} \in \mathbb{R}^{m,n}$  ( $m$  samples of  $n$  features) and targets  $\mathbf{Y} \in \mathbb{R}^{m,1}$ , weights of the model are  $\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$ . Prediction for a point  $\mathbf{x}$  is then just  $\hat{Y} = \mathbf{x}^T \mathbf{w}$  [30].

The power of this simple model can be extended by using some non-linear transformations (power, log, sin) of the features. It can also be limited by penalizing the size of model weights – preferring smaller and simpler models [30].

### 1.7.3 Logistic regression

Logistic regression is a modification of linear regression used for binary classification. This model predicts a real number, which is then mapped into  $[0, 1]$  interval using the sigmoid function,  $\sigma(x) = \frac{e^x}{1+e^x}$ . This number is then interpreted as the probability of one of the classes, the probability of the other class is complementary [31].

Contrary to linear regression, there is no exact formula for calculating the model's weights, so they are found using numerical methods (e.g., using gradient descent) [31].

### 1.7.4 Naive Bayes

*Naive Bayes* is a classification model that chooses a different approach than other models described in this section. Instead of predicting the probability of a class given some data (i.e.,  $p(C_k|\mathbf{x})$ ), using Bayes' theorem, it models data distribution for each class, and in combination with the probability of each class, it predicts the most likely one (i.e.,  $\arg \max_k p(\mathbf{x}|C_k)p(C_k)$ ). Since modeling  $p(\mathbf{x}|C_k)$  is difficult, *naive Bayes* assumes all features to be independent so that  $p(\mathbf{x}|C_k) = \prod_{d=1}^n p(x_d|C_k)$  which can be modeled easier [32].

The distribution of each feature is modeled for each class independently. Normal, Bernoulli or multinomial distributions can be used for this purpose; the parameters of each distribution are calculated from data with the corresponding class. By choosing one of the distributions, we assume that features are from that distribution [32].

### 1.7.5 K-nearest neighbours

*K-nearest neighbors* is a simple model that can be used for both classification and regression. As is written in the name, it uses the nearest neighbors of a point to predict its value [32].

Its training is trivial – the training data are the model, and the training targets are the values of each training sample. Nevertheless, it has multiple hyperparameters that influence the model's predictions. The main ones are the number of neighbors used for predicting, metrics used for measuring distance, and weights of the neighbors (e.g., uniform or inverse to the distance) [32].

During prediction,  $k$  nearest neighbors of the given point are found. In the case of classification, the weights of neighbors are summed for each class separately, and the



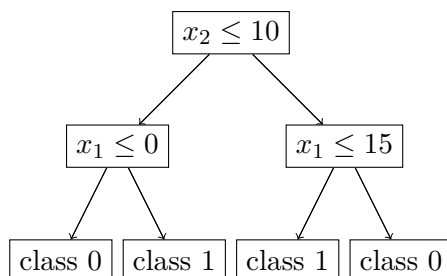
class with the highest sum is chosen. In regression, the prediction is the weighted average of the neighbors [32].

### 1.7.6 Decision tree

The *decision tree* divides the feature space into smaller parts, and each part is then handled separately. The model is a binary tree with internal nodes splitting data into two parts based on a simple condition on one of the features, e.g.,  $x_3 \leq 10$ . The leaves are used for predicting: in classification, the majority class of leaf samples is predicted; in regression, prediction is the average of the samples [33].

Prediction of a *decision tree* is just following the path from the root to a leaf; if the condition in the node is met, go to the left node; otherwise, to the right one. An example of *decision tree* is in Figure 1.1; for  $\mathbf{x} = (1, 20)^T$  model would predict class 1 [33].

If the tree had unlimited depth, it would be an index of the data. Since this is usually not wanted, hyperparameters such as maximum depth, minimum samples to split, or maximum number of leaf nodes are used [33].



■ **Figure 1.1** Example of decision tree for binary classification

### 1.7.7 Random forest

*Random forest* consists of a collection of *decision trees* and combines their predictions – majority voting for classification, and an average for regression [33]. The idea is that by aggregating the results of many (smaller) models, the error will be smaller than of each individual model [34]. It can be used for both classification and regression.

During training, each tree is usually trained on a bootstrapped dataset in order to create diverse trees. The main hyperparameters are the number of trees in the model and the parameters of the trees [33].

### 1.7.8 Gradient boosting decision trees

The *gradient boosting decision trees* uses a collection of *decision trees* similarly to *random forest*. However, in this case, models are trained sequentially and new trees attempt to correct errors of the previous ones. This model can be used for both classification and regression [35].

As with the *random forest*, the main hyperparameters are the number of trees in the model and the parameters of the trees.



### 1.7.9 Data preprocessing (feature normalization)

In some cases, it is beneficial for features to have similar sizes. E.g., when using *k-nearest neighbors*, if one feature is in thousands while the other is between zero and one, the smaller one will have almost no impact on which points are the nearest neighbors.

In order to train and use the model on values of features that have similar sizes, data can be preprocessed. Two commonly used methods are:

**Normalization** – scaling all features to  $[0, 1]$  interval using:  $x_{i,j}^{norm} = \frac{x_{i,j} - \min_k x_{k,j}}{\max_k x_{k,j} - \min_k x_{k,j}}$

**Standardization** – scaling all features to have zero mean and standard deviation of one using:  $x_{i,j}^{standard} = \frac{x_{i,j} - \mu_j}{\sigma_j}$

## 1.8 Usage of logic synthesis in SAT-solving

Using synthesis to speed up SAT-solving is not a novel idea. It has already been explored by [1] and [2]. However, only a limited number of instances were tested in both studies.

The first study presents results of using synthesis for preprocessing integer factorization problem<sup>3</sup>, which is hard for a SAT solver. For synthesis, SIS tool [36] (predecessor of ABC [23]) is used and zCHAFF solved the instances. The paper presents the results of ten instances solved without preprocessing and with preprocessed using the *simplify* synthesis algorithm. On unsatisfiable instances (factorization of a prime is attempted), significant speedup was measured, while on the satisfiable ones (factorization exists), solving with synthesis led to worse results [1].

The second paper is from the authors of MINISAT solver and ABC synthesis tool. Two approaches *DAG-aware circuit compression* and technology mapping for lookup-table (LUT) based FPGAs are explored with the use of MINISAT solver. In total, 30 hard SAT instances consisting of hard industrial instances and those from *SAT-Race 2006* were tested. Applying both techniques led to a 5x speedup (does not include synthesis time) when solving hard industrial instances. In the case of SAT Race instances, synthesis sped up some of them while slowing down others [2].

This thesis intends to explore this topic further by analyzing the influence of logic synthesis on the solving time of a larger number of instances (benchmark as well as practical ones) using multiple currently available solvers (MINISAT, zCHAFF, GLUCOSE) and ABC logic synthesis tool. In cases where synthesis led to speedup, the possibility of effective use in practice is also analyzed.

---

<sup>3</sup>Given an integer  $x$  find its factors  $p$  and  $q$  greater than 1, such that  $x = p \cdot q$ .

# Experiments

In this chapter, three case studies are evaluated: applying logic optimization to SAT instances obtained from (1) standard satisfiability benchmarks (SATLIB) [37], (2) instances obtained from SAT-based optimum circuits generator [25], and (3) SAT-ATPG instances. The influence of different logic optimization processes on the overall run-time is studied.

## 2.1 Experimental Setup

Experiments were run on an Intel Xeon Gold 5218 (2.30GHz) processor with 4 GB of memory running Debian 11 or Debian 12. All solvers were run using one CPU core, and GLUCOSE was run on four CPU cores as well. For practical reasons, the solvers had a time limit of 2000 seconds (real-time); the solving was terminated if an instance was not solved within the limit. The solvers are in the text referred to using their names; in the case of GLUCOSE, the results using one CPU core are referred to as GLUCOSE or, in some cases, GLUCOSE (one CPU), and four CPU cores results are referred to as “GLUCOSE (4CPU)” or “Gl. 4cpu” in tables. *All time values are CPU time unless specified otherwise.*

Logic synthesis was done using ABC 1.01 [22]. Glucose-syrup 4.2.1 [6], zChaff 2007.3.12 [4], and MiniSat 2.2 [5] solvers solved the SAT instances.

As for the logic synthesis and optimization, the following ABC scripts were used, and they will be referred to in the subsequent text as follows:

- Just a simple conversion to an AND-inverter-graph (AIG) [22], without any optimization: *st*:  
`strash;`
- A simple technology-independent optimization, with different efforts: *st-re2-[1, 2, 3]/x*; the whole script repeated *x* times:  
`strash; resyn2;`
- A powerful optimization and mapping into 2-input gates, with different efforts: *2-gate-[1, 5, 10, 15]/x*; the part between `get` and `put` (first and last command) repeated *x* times:

```
&get -n; &st; &synch2; &if -m -a -K 2; &mfs -W 10; &st; &dch;
&if -m -a -K 2; &mfs -W 10; &put;
```

- A powerful optimization and mapping into 6-input LUTs, with different efforts:  $6-LUT-[1, 5, 10, 15]x$ ; the part between `get` and `put` (first and last command) repeated  $x$  times:

```
&get -n; &st; &synch2; &if -m -a -K 6; &mfs -W 10; &st; &dch;
&if -m -a -K 6; &mfs -W 10; &put;
```

In experiments where a SAT instance in CNF was the source (i.e., Section 2.2 and 2.3), the CNF had to be transformed to a format accepted by ABC – the BLIF [38] and then, after the optimization, converted back to the CNF. The first conversion involved rewriting the CNF to a two-level network of a product of sums. The Tseitin transformation did the latter conversion [9] (see Subsection 1.1.1). An in-house tool was used for this purpose.

In the case of SAT solving with synthesis, only the *synthesis time and subsequent SAT solving time are considered* and compared to solving without synthesis. The time of file conversion from CNF to BLIF and back is not taken into account, as the tool used them is not optimized for speed.

## 2.2 SATLIB benchmark instances

The SATLIB benchmark [37] offers a number of randomly generated hard 3-SAT instances, i.e., instances with the ratio of clauses to variables near 4.3 [39] and many other instances from different problem domains.

From available categories, only those with instances that took longer time to solve were selected. Those are uniform Random-3-SAT instances with the numbers of variables and clauses equal to 200-860, 225-960, and 250-1065, as well as SAT-encoded Quasigroup (Latin square) instances and Pigeon-hole problem instances.

### 2.2.1 Synthesis influence on CNF properties

Tables 2.9, 2.10, and 2.11 show how the synthesis and Tseitin transformation influenced the number of variables, number of clauses, and their ratio, respectively. Interestingly, many unsatisfiable instances from the first group of instances were transformed into trivial ones (i.e., for variable  $a$ , the CNF would be  $a \wedge \neg a$ ). The numbers of such instances are in Table 2.12. It happened mostly with  $6-LUT-[1, 5, 10, 15]x$  syntheses and in the case of Quasigroup and Pigeon-hole instances with  $2-gate-[1, 5, 10, 15]x$  synthesis as well. Repeating the synthesis script led to higher numbers of trivial instances. With the increasing complexity of uniform Random-3-SAT instances, the number of trivial instances decreases as the instances were probably too complex for synthesis to “solve” them.

If the trivial instances are left out, the average number of clauses increased significantly for almost all syntheses, with the exception of  $6-LUT-[5, 10, 15]x$  syntheses on the smallest uniform Random-3-SAT instances, and Pigeon-hole instances with  $2-gate-[5, 10, 15]x$  and  $6-LUT-[1, 5, 10, 15]x$  syntheses, which were all transformed into trivial ones.

Similarly, if the trivial instances are left out, the use of all syntheses led to a significant increase in the average number of variables, of course, with the exception of cases where all instances are transformed into trivial ones.

The more complex syntheses or their repetitions (*2-gate-[1, 5, 10, 15]x* and *6-LUT-[1, 5, 10, 15]x*) led to a lower number of variables and clauses compared to other synthesis or less repetitions of the same script.

Although the number of clauses and variables mostly increased, their ratios usually decreased. In the case of uniform Random-3-SAT, with all syntheses except *6-LUT-[1, 5, 10, 15]x*, the ratio changed from 4.3 to 2.7–2.8 with negligible standard deviations. With *6-LUT-[1, 5, 10, 15]x* syntheses, the ratio gradually increases as the 3-SAT instances get larger; this happens even if trivial instances are left out.

The original clause-to-variable ratio of Quasigroup instances of  $74.1 \pm 85.3$  decreased with all syntheses significantly to 2.8–6.7 and a much lower standard deviation as well (at most 2.2). Lastly, Pigeon-hole instances clause-to-variable was originally 4.1, and in cases where resulting instances were not trivial ones, the ratio was, on average, around 2.1–2.7.

■ **Table 2.1** SATLIB benchmark: influence of logic synthesis on the number of variables

Problem type	3-SAT 200-860	3-SAT 225-960	3-SAT 250-1065	Quasigroup	Pigeon-hole
# instances	199 <sup>1</sup>	200	200	22	5
Original	200.0	225.0	250.0	$1043.2 \pm 574.3$	$74.0 \pm 26.9$
st	$2772.3 \pm 2.6$	$3097.1 \pm 2.4$	$3437.0 \pm 2.8$	$161177.4 \pm 123534.2$	$773.0 \pm 391.3$
st-re2-1x	$2716.1 \pm 7.7$	$3039.4 \pm 8.2$	$3373.0 \pm 8.3$	$101578.3 \pm 64170.8$	$467.0 \pm 197.8$
st-re2-2x	$2699.3 \pm 11.7$	$3021.1 \pm 11.0$	$3344.3 \pm 11.1$	$100150.1 \pm 63138.6$	$459.0 \pm 195.2$
st-re2-3x	$2687.0 \pm 16.6$	$3005.2 \pm 14.8$	$3319.3 \pm 13.9$	$99692.7 \pm 62791.3$	$456.0 \pm 197.6$
2-gate-1x	$2678.9 \pm 8.3$	$3004.4 \pm 8.8$	$3339.6 \pm 8.8$	$93769.2 \pm 71918.8$	$148.2 \pm 329.1$
2-gate-5x	$2529.7 \pm 12.0$	$2846.6 \pm 11.9$	$3163.0 \pm 11.8$	$91215.3 \pm 70761.6$	1.0
2-gate-10x	$2423.3 \pm 11.5$	$2730.0 \pm 11.6$	$3030.7 \pm 12.0$	$90210.6 \pm 70434.0$	1.0
2-gate-15x	$2361.7 \pm 9.9$	$2661.1 \pm 10.6$	$2950.2 \pm 11.0$	$82938.0 \pm 74727.9$	1.0
6-LUT-1x	$180.2 \pm 194.5$	$601.4 \pm 315.2$	$901.2 \pm 90.8$	$25146.9 \pm 20706.8$	1.0
6-LUT-5x	$159.5 \pm 168.6$	$471.6 \pm 314.2$	$842.1 \pm 176.9$	$23854.3 \pm 21167.3$	1.0
6-LUT-10x	$157.1 \pm 165.2$	$379.2 \pm 302.4$	$791.6 \pm 222.1$	$23646.9 \pm 21018.6$	1.0
6-LUT-15x	$156.1 \pm 163.7$	$354.4 \pm 292.4$	$754.2 \pm 252.3$	$22748.1 \pm 21415.0$	1.0

## 2.2.2 Synthesis influence on solving time

As tested instances had the upper time limit of 2000 seconds for the SAT-solver (see Section 2.1), some of them were not solved within the limit; if so, the process was terminated. Table 2.5 shows the numbers of instances of each combination of finished/terminated and with synthesis / without synthesis.

Table 2.6 shows ratios of instances whose time of solving improved after using logic synthesis. If an instance was solved both with and without synthesis, improvement was achieved only if the time of solving with synthesis (synthesis + SAT-solving) was smaller than solving without synthesis (only SAT-solving). If some instance was solved

<sup>1</sup>This is not a mistake, in the benchmark instances archive are only 99 unsatisfiable instances in this category

■ **Table 2.2** SATLIB benchmark: influence of logic synthesis on the number of clauses

Problem type	3-SAT 200-860	3-SAT 225-960	3-SAT 250-1065	Quasigroup	Pigeon-hole
# instances	199	200	200	22	5
Original	860.0	960.0	1065.0	58469.7 ± 42171.1	322.0 ± 170.3
st	7717.8 ± 7.7	8617.3 ± 7.3	9562.0 ± 8.3	480403.7 ± 370199.3	2098.0 ± 1093.2
st-re2-1x	7549.4 ± 23.0	8444.3 ± 24.7	9369.9 ± 24.8	301606.4 ± 191564.8	1180.0 ± 512.6
st-re2-2x	7498.9 ± 35.2	8389.4 ± 33.0	9283.8 ± 33.3	297321.9 ± 188461.8	1156.0 ± 505.0
st-re2-3x	7462.0 ± 49.7	8341.7 ± 44.4	9208.9 ± 41.8	295949.6 ± 187414.9	1147.0 ± 512.0
2-gate-1x	7437.7 ± 25.0	8339.1 ± 26.3	9269.7 ± 26.5	281540.3 ± 216567.1	378.0 ± 840.8
2-gate-5x	6990.0 ± 36.1	7865.8 ± 35.8	8740.0 ± 35.4	274069.4 ± 213224.6	2.0
2-gate-10x	6670.9 ± 34.6	7516.1 ± 34.7	8343.2 ± 35.9	271034.2 ± 212256.6	2.0
2-gate-15x	6486.2 ± 29.8	7309.4 ± 31.8	8101.7 ± 32.9	248910.7 ± 225150.1	2.0
6-LUT-1x	539.6 ± 891.5	3750.8 ± 2330.6	6205.7 ± 638.1	180036.6 ± 141421.9	2.0
6-LUT-5x	411.9 ± 615.6	2549.4 ± 2078.0	5257.1 ± 1143.7	170474.2 ± 145506.0	2.0
6-LUT-10x	392.0 ± 575.8	1779.9 ± 1798.7	4569.4 ± 1377.3	168269.8 ± 144417.7	2.0
6-LUT-15x	384.5 ± 560.4	1569.9 ± 1656.3	4183.8 ± 1497.8	160829.6 ± 147639.6	2.0

■ **Table 2.3** SATLIB benchmark: influence of logic synthesis on the clause-variable ratio

Problem type	3-SAT 200-860	3-SAT 225-960	3-SAT 250-1065	Quasigroup	Pigeon-hole
# instances	199	200	200	22	5
Original	4.3	4.3	4.3	74.1 ± 85.3	4.1 ± 0.8
st	2.8	2.8	2.8	3.0	2.7
st-re2-1x	2.8	2.8	2.8	3.0	2.5
st-re2-2x	2.8	2.8	2.8	3.0	2.5
st-re2-3x	2.8	2.8	2.8	3.0	2.5
2-gate-1x	2.8	2.8	2.8	2.9 ± 0.3	2.1 ± 0.2
2-gate-5x	2.8	2.8	2.8	2.9 ± 0.3	2.0
2-gate-10x	2.8	2.8	2.8	2.9 ± 0.3	2.0
2-gate-15x	2.7	2.7	2.7	2.8 ± 0.4	2.0
6-LUT-1x	2.3 ± 1.0	5.2 ± 2.1	6.8 ± 0.5	6.7 ± 1.7	2.0
6-LUT-5x	2.2 ± 0.8	4.2 ± 1.9	6.1 ± 0.9	6.3 ± 2.1	2.0
6-LUT-10x	2.1 ± 0.7	3.5 ± 1.7	5.5 ± 1.1	6.2 ± 2.1	2.0
6-LUT-15x	2.1 ± 0.7	3.3 ± 1.6	5.2 ± 1.2	5.9 ± 2.2	2.0

with synthesis but not without it, it is considered to be sped up as well. However, if the time of solving with synthesis (synthesis + SAT-solving) is above the time limit for the SAT solver (2000 real-time seconds (corresponds to 8000 CPU seconds for GLUCOSE (4CPU))). It cannot be decided whether the solving was sped up; in such case, the ratio of improved instances is presented as a range. Instances that were solved neither with nor without synthesis are left out.

The influence of synthesis on the solving time is shown in Table 2.7 and 2.8. The percentage changes are calculated based on instances that *were finished both with and without synthesis within the time limit*. The number of such instances is the first value of the corresponding cell in Table 2.5. This approach results in possibly different instances being used for calculating the percentage changes for different combinations of solver and synthesis, which, in some cases, causes worse comparability if many instances are left out. We chose this approach as the number of instances that were solved by all solvers

■ **Table 2.4** SATLIB benchmark: numbers of unsatisfiable instances which were transformed by synthesis into trivial ones (i.e., for variable  $a$ , the CNF would be  $a \wedge \neg a$ )

Problem type	3-SAT 200-860	3-SAT 225-960	3-SAT 250-1065	Quasigroup	Pigeon-hole
# UNSAT instances	99	100	100	12	5
st	0	0	0	0	0
st-re2-1x	0	0	0	0	0
st-re2-2x	0	0	0	0	0
st-re2-3x	0	0	0	0	0
2-gate-1x	0	0	0	2	4
2-gate-5x	0	0	0	2	5
2-gate-10x	0	0	0	2	5
2-gate-15x	0	0	0	4	5
6-LUT-1x	97	37	2	2	5
6-LUT-5x	99	50	8	4	5
6-LUT-10x	99	64	13	4	5
6-LUT-15x	99	67	18	5	5

without synthesis and with all syntheses would sometimes be rather limited. Moreover, the main goal of this thesis is to evaluate the improvements achievable by synthesis, and this approach allows for individual results to be more precise as they are based on more instances.

Likewise, the average time shown in each column can be compared between solvers only if all instances (or most of them) were solved without synthesis. If some instances are left out (due to not being solved within the time limit), the real average time would be much higher than the presented one. Therefore, it usually offers only a rough idea about the solving time to which the percentage changes can be related. In case all instances were solved, which is for benchmark instances mostly true, all values are exact and can be compared easily.

In many cases, the average time of solving for different solvers in one category varies greatly. This, even with approximate results, means that a large potential improvement of using some synthesis would not usually be useful in practice as using the fastest solver (even without any synthesis) would result in a faster time.

The values in Table 2.7 are calculated using Equation 2.1 where  $(nosyn, syn)$  are pairs of solving time without synthesis (only SAT-solving of the original instance) and with synthesis (synthesis + SAT-solving). The values show how the original total time would change if synthesis were used only on instances when it decreased the overall time of solving that instance and others were solved without it. These values show how large improvement *can be potentially achieved* with each synthesis script on tested instances. On its own, it does not reflect any practical application. However, for synthesis to have practical use, a (high) potential decrease in solving time is a necessary condition. In the following paragraphs, mainly, this potential improvement is discussed.

$$\frac{\sum_{(time, syn)} \min\{time, syn\}}{\sum_{(time, syn)} time} - 1 \quad (2.1)$$

Table 2.8 presents how the solving time would be influenced by always using synthesis. In the category of smallest uniform Random-3-SAT instances (3-SAT 200-860), all

instances were solved within the time limit. Most or all unsatisfiable instances were transformed into trivial ones by  $6-LUT-[1, 5, 10, 15]x$  syntheses. From all solvers, MINISAT had the lowest average time of 0.1 seconds with no noteworthy improvement achieved with synthesis. zCHAFF had a slightly higher average time of 0.7 seconds, with no meaningful results of solving with synthesis as well. GLUCOSE had an even higher average time of 4.4 seconds. On the other hand,  $6-LUT-[1, 5, 10, 15]x$  syntheses all sped up 36.7% of instances with potential time decrease around 64%. Lastly, GLUCOSE (4CPU) had the highest average time of 5.9 seconds. The highest ratio of instances (51.3%) was improved by  $st$  synthesis. Nevertheless, the biggest potential improvement slightly above 50% was achieved by  $6-LUT-[1, 5, 10, 15]x$  syntheses, which sped up 47.2% of instances.

All of the middle-sized uniform Random-3-SAT instances (3-SAT 225-960) were solved within the time limit as well. Between third and two-thirds of the unsatisfiable instances were transformed into trivial ones by  $6-LUT-[1, 5, 10, 15]x$  syntheses, with more repetitions resulting in more trivial instances. MINISAT and zCHAFF results are similar to the previous category. They had the lowest average times of 0.6 and 2.9, respectively, while achieving none or insignificant improvements using synthesis. GLUCOSE and GLUCOSE (4CPU) had average times of 19.2 and 17.5 seconds. Potential improvements were much higher than with other solvers, around or slightly above 20% with  $st$  and  $st-re2-[1, 2, 3]x$  syntheses. GLUCOSE sped up 39.5%–45.5 of instances with these syntheses and GLUCOSE (4CPU) 52.0%–53.5% of them.

From the largest uniform Random-3-SAT instances (3-SAT 250-1065), with zCHAFF solver, some instances were solved within the time limit only without the synthesis (ranging from low units to just under a quarter of instances). Other solvers solved all instances within the limit. Only a small number of instances were transformed into trivial ones, most being 13 and 18 (from 100 unsatisfiable instances) with  $6-LUT-[10, 15]x$  syntheses. As in previous cases, both MINISAT and zCHAFF had the lowest average times of 2.7 and 16.6 seconds, respectively, and they did not achieve any significant improvement in combination with any synthesis. On the contrary, some instances transformed by synthesis were not solved by zCHAFF within the time limit. GLUCOSE and GLUCOSE (4CPU) had average times of 61.5 and 53.5 seconds. Potential improvements were around 20% with  $st$  and  $st-re2-[1, 2, 3]x$  syntheses (almost the same as in the middle-sized 3-SAT instances category). The ratio of improved instances was a bit higher, 45.5%–55.0 with GLUCOSE and 54.0%–58.0% with GLUCOSE (4CPU).

When solving Quasigroup instances, all instances were solved within the time limit, and some of them were transformed into trivial ones. The MINISAT solver was the fastest, with an average of 0.3 seconds, followed by zCHAFF with 2.6 seconds. Both solvers achieved no potential improvement at all. Slightly slower than zCHAFF was GLUCOSE (4CPU), with an average time of 2.9 seconds. It achieved small potential improvements of 0.9% and 4.8% with  $st$  and  $st-re2-1x$  syntheses, respectively, in both cases improving 4.5% of instances. These were the only improvements achieved by synthesis on this category of instances. The slowest of all solvers was GLUCOSE, with an average time of 3.5 seconds.

Pigeon-hole problem instances were all solved within the time limit with the exception of one instance with GLUCOSE solver, which was solved only with synthesis. Interestingly,  $2-gate-[5, 10, 15]x$  as well as  $6-LUT-[1, 5, 10, 15]x$  syntheses transformed all 5 instances



into trivial ones. Without synthesis, the fastest solver was zCHAFF, which was not sped up by synthesis at all. The second one was MINISAT. In this case, synthesis helped significantly: *st* decreased the time of solving by 99.1% and sped up four out of five instances, achieving time slightly above one-third of zCHAFF time; *st-re2-[1, 2, 3]x* decreased the time by 98.1%–98.3%, still achieving times below the zCHAFF time and speeding up three instances. The use of these syntheses led only to easier-to-solve instances but not to the trivial ones.

When solving Pigeon-hole instances, GLUCOSE and GLUCOSE (4CPU) were the slowest, with averages of 44.6 and 359.7 seconds, respectively. Even though it seems that the 4 CPU variant was much slower, GLUCOSE (one CPU) did not finish in time one of the instances, which, therefore, is neither part of the average time nor potential improvement percentages. GLUCOSE (4CPU) did solve that instance within 1658.7 seconds, which caused the results to look worse at first glance. Without it, the average time would be 35.0 seconds, and potential improvements achieved by syntheses would be similar to GLUCOSE (one CPU). Nevertheless, on current results, the largest potential improvements of 98.9% were achieved by GLUCOSE solver with *2-gate-[1, 5, 10, 15]x* syntheses and by GLUCOSE (4CPU) 96.1%–96.2% with *6-LUT-[1, 5, 10, 15]x* syntheses. With these syntheses, the solving time of four or all five instances was decreased.

Always using synthesis would, in most cases, significantly increase the solving time of uniform Random-3-SAT instances. The time increase for MINISAT was always above +100%, but in many cases, in thousands of percent. For zCHAFF, increases were always above +1000% percent. Nevertheless, GLUCOSE and GLUCOSE (4CPU) with *st* and *st-re2-[1, 2, 3]x* syntheses achieved in some cases small decreases, up to 7.1% for GLUCOSE and 9.4% for GLUCOSE (4CPU). On the medium and large instances, GLUCOSE (4 CPU) with *st-re2-1x* synthesis would achieve small improvements of 6% and 9.4%, respectively.

The time of solving Quasigroup instances was never improved by synthesis if used always. This is not surprising as the only cases with any potential improvements were GLUCOSE (4CPU) with *st* and *st-re2-1x* syntheses. The increase, resulting from using synthesis always, was always above +100%, but in most cases, in thousands of percent.

Contrary to other categories, results on Pigeon-hole instances were mostly the same as when calculating potential improvements. This is because the solving time with synthesis was in four or all five cases smaller than the solving time without synthesis, so there is a slight or no difference in the calculation. Overall, the best results in this category were still achieved by MINISAT with *st* syntheses.

To sum up, MINISAT, without any synthesis, achieved the best results when solving uniform Random-3-SAT and Quasigroup instances. No combination of any other solver and synthesis had the potential to achieve better results. Using MINISAT with any synthesis would be most likely decremental as the potential improvements are very low or none, and using synthesis for all instances significantly increases the solving time (i.e., selecting instances to run with synthesis would have to be very precise). Pigeon-hole instances solving time was improved by many syntheses, but the fastest combination was MINISAT with *st* synthesis.



■ **Table 2.5** SATLIB benchmarks: separated by slashes are the numbers of instances (1) which were solved in 2000 seconds (real-time), both with and without synthesis, (2) which were solved in 2000 seconds (real-time) with synthesis but not without it, (3) which were solved in 2000 seconds (real-time) without synthesis but not with it, (4) which were solved in 2000 seconds (real-time) neither with nor without synthesis. In case values (2), (3), and (4) are all zeroes, they are omitted.

Problem type Solver	3-SAT 200-860				3-SAT 225-960				3-SAT 250-1065			Quasigroup				Pigeon-hole				
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	
st	199	199	199	199	200	200	200	200	200	174/0/26/0	200	200	22	22	22	22	5	5	4/1/0/0	5
st-re2-1x	199	199	199	199	200	200	200	200	200	174/0/26/0	200	200	22	22	22	22	5	5	4/1/0/0	5
st-re2-2x	199	199	199	199	200	200	200	200	200	174/0/26/0	200	200	22	22	22	22	5	5	4/1/0/0	5
st-re2-3x	199	199	199	199	200	200	200	200	200	171/0/29/0	200	200	22	22	22	22	5	5	4/1/0/0	5
2-gate-1x	199	199	199	199	200	200	200	200	200	167/0/33/0	200	200	22	22	22	22	5	5	4/1/0/0	5
2-gate-5x	199	199	199	199	200	200	200	200	200	160/0/40/0	200	200	22	22	22	22	5	5	4/1/0/0	5
2-gate-10x	199	199	199	199	200	200	200	200	200	152/0/48/0	200	200	22	22	22	22	5	5	4/1/0/0	5
2-gate-15x	199	199	199	199	200	200	200	200	200	154/0/46/0	200	200	22	22	22	22	5	5	4/1/0/0	5
6-LUT-1x	199	199	199	199	200	200	200	200	200	198/0/2/0	200	200	22	22	22	22	5	5	4/1/0/0	5
6-LUT-5x	199	199	199	199	200	200	200	200	200	195/0/5/0	200	200	22	22	22	22	5	5	4/1/0/0	5
6-LUT-10x	199	199	199	199	200	200	200	200	200	197/0/3/0	200	200	22	22	22	22	5	5	4/1/0/0	5
6-LUT-15x	199	199	199	199	200	200	200	200	200	196/0/4/0	200	200	22	22	22	22	5	5	4/1/0/0	5

■ **Table 2.6** SATLIB benchmarks: the ratio of instances whose time of solving decreased after using the synthesis preprocessing. Instances that were solved neither with nor without synthesis are left out.

Problem type Solver	3-SAT 200-860				3-SAT 225-960				3-SAT 250-1065				Quasigroup				Pigeon-hole			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
st	0.5%	6.0%	26.6%	51.3%	6.0%	3.0%	39.5%	53.5%	7.5%	4.0%	55.0%	54.0%	0.0%	0.0%	0.0%	4.5%	80.0%	0.0%	60.0%	80.0%
st-re2-1x	0.0%	3.0%	26.6%	50.8%	3.0%	3.0%	45.0%	53.0%	10.5%	3.0%	49.0%	58.0%	0.0%	0.0%	0.0%	4.5%	60.0%	0.0%	80.0%	60.0%
st-re2-2x	0.0%	2.5%	24.1%	47.7%	2.0%	2.0%	42.0%	52.5%	6.5%	1.0%	45.5%	58.0%	0.0%	0.0%	0.0%	0.0%	60.0%	0.0%	60.0%	60.0%
st-re2-3x	0.0%	2.5%	27.1%	47.2%	2.5%	2.0%	40.5%	52.0%	4.0%	2.0%	50.5%	55.5%	0.0%	0.0%	0.0%	0.0%	60.0%	0.0%	40.0%	60.0%
2-gate-1x	0.0%	0.0%	10.1%	17.1%	0.0%	0.0%	28.5%	28.5%	0.0%	1.0%	40.5%	45.5%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	100.0%
2-gate-5x	0.0%	0.0%	0.0%	1.5%	0.0%	0.0%	13.0%	12.0%	0.0%	0.0%	20.5%	26.5%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	80.0%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.0%	2.0%	0.0%	0.0%	11.0%	11.5%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	80.0%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.5%	0.0%	0.0%	0.0%	7.5%	11.5%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	80.0%
6-LUT-1x	0.0%	3.5%	36.7%	47.2%	0.0%	2.0%	9.0%	9.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	100.0%
6-LUT-5x	0.0%	3.5%	36.7%	47.2%	0.0%	2.0%	9.0%	9.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	100.0%
6-LUT-10x	0.0%	3.5%	36.7%	47.2%	0.0%	2.0%	9.0%	9.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	80.0%
6-LUT-15x	0.0%	3.0%	36.7%	47.2%	0.0%	2.0%	9.0%	9.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	20.0%	0.0%	80.0%	80.0%

■ **Table 2.7** SATLIB benchmarks: potential influence of synthesis on the solving time (i.e., using synthesis only when it decreases the solving time of an instance). Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit.

Problem type	3-SAT 200-860				3-SAT 225-960				3-SAT 250-1065				Quasigroup				Pigeon-hole			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
Average time	0.1	0.7	4.4	5.9	0.6	2.9	19.2	17.5	2.7	16.6	61.5	53.5	0.3	2.6	3.5	2.9	25.4	0.6	44.6	359.7
st	-0.1%	-2.0%	-18.4%	-24.1%	-3.2%	-0.6%	-20.9%	-19.8%	-1.7%	-0.8%	-20.3%	-19.6%	0.0%	0.0%	0.0%	-0.9%	-99.1%	0.0%	-6.6%	-52.8%
st-re2-1x	0.0%	-0.7%	-18.8%	-23.4%	-2.5%	-1.4%	-21.9%	-21.2%	-3.4%	-2.8%	-19.3%	-20.6%	0.0%	0.0%	0.0%	-4.8%	-98.1%	0.0%	-32.9%	-76.9%
st-re2-2x	0.0%	-1.0%	-15.5%	-21.4%	-1.3%	-1.1%	-23.9%	-19.6%	-1.0%	-0.0%	-15.7%	-18.8%	0.0%	0.0%	0.0%	0.0%	-98.2%	0.0%	-39.3%	-72.7%
st-re2-3x	0.0%	-0.8%	-16.1%	-20.5%	-0.7%	-1.0%	-19.1%	-23.2%	-1.8%	-0.7%	-17.6%	-18.3%	0.0%	0.0%	0.0%	0.0%	-98.2%	0.0%	-12.7%	-69.2%
2-gate-1x	0.0%	0.0%	-5.0%	-9.6%	0.0%	0.0%	-13.8%	-9.6%	0.0%	-0.4%	-14.3%	-12.8%	0.0%	0.0%	0.0%	0.0%	-0.7%	0.0%	-98.9%	-65.5%
2-gate-5x	0.0%	0.0%	0.0%	-1.0%	0.0%	0.0%	-7.1%	-2.6%	0.0%	0.0%	-7.6%	-9.0%	0.0%	0.0%	0.0%	0.0%	-0.7%	0.0%	-98.9%	-62.3%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	-1.0%	-0.6%	0.0%	0.0%	-3.9%	-3.7%	0.0%	0.0%	0.0%	0.0%	-0.7%	0.0%	-98.9%	-63.2%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	-0.7%	0.0%	0.0%	0.0%	-3.0%	-4.2%	0.0%	0.0%	0.0%	0.0%	-0.7%	0.0%	-98.9%	-62.4%
6-LUT-1x	0.0%	-1.2%	-64.3%	-50.4%	0.0%	-0.2%	-5.8%	-4.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	-60.8%	0.0%	-88.0%	-96.2%
6-LUT-5x	0.0%	-1.1%	-64.2%	-50.3%	0.0%	-0.2%	-5.8%	-4.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	-60.0%	0.0%	-88.0%	-96.1%
6-LUT-10x	0.0%	-1.1%	-64.1%	-50.2%	0.0%	-0.2%	-5.8%	-4.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	-60.8%	0.0%	-88.0%	-96.2%
6-LUT-15x	0.0%	-1.0%	-64.0%	-50.1%	0.0%	-0.2%	-5.8%	-4.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	-60.8%	0.0%	-88.0%	-96.2%

■ **Table 2.8** SATLIB benchmarks: the influence of synthesis on the solving time if used always; values larger than or equal to +1000% are replaced by “-”. Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit.

Problem type	3-SAT 200-860				3-SAT 225-960				3-SAT 250-1065				Quasigroup				Pigeon-hole			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
Average time	0.1	0.7	4.4	5.9	0.6	2.9	19.2	17.5	2.7	16.6	61.5	53.5	0.3	2.6	3.5	2.9	25.4	0.6	44.6	359.7
st	+218.7%	-	+27.0%	-1.8%	+102.3%	-	+6.5%	-1.0%	+115.2%	-	-7.1%	-8.0%	+933.6%	-	+136.8%	+145.1%	-99.1%	+577.0%	-0.2%	-52.8%
st-re2-1x	+295.6%	-	+33.1%	-2.6%	+126.6%	-	-4.7%	-6.0%	+118.5%	-	-4.5%	-9.4%	-	-	+271.8%	+320.4%	-98.1%	+111.7%	-32.9%	-76.8%
st-re2-2x	+333.1%	-	+39.8%	+2.4%	+136.4%	-	-3.8%	-3.8%	+126.6%	-	+0.3%	-9.0%	-	-	+418.3%	+492.2%	-98.1%	+126.0%	-39.0%	-72.7%
st-re2-3x	+389.8%	-	+35.4%	+2.4%	+147.6%	-	+1.8%	-5.9%	+141.6%	-	-4.9%	-5.3%	-	-	+548.5%	+643.4%	-98.1%	+215.1%	+20.5%	-69.1%
2-gate-1x	-	-	+99.0%	+48.6%	+645.9%	-	+21.5%	+21.4%	+290.3%	-	+4.4%	+0.2%	-	-	-	-	+209.7%	-	-98.9%	-65.5%
2-gate-5x	-	-	+261.1%	+176.8%	-	-	+59.6%	+64.5%	+646.9%	-	+25.6%	+21.1%	-	-	-	-	+434.5%	-	-98.9%	-62.3%
2-gate-10x	-	-	+459.5%	+318.6%	-	-	+118.6%	+124.5%	-	-	+43.7%	+45.1%	-	-	-	-	+421.8%	-	-98.8%	-63.2%
2-gate-15x	-	-	+652.0%	+469.1%	-	-	+169.0%	+186.4%	-	-	+60.9%	+62.4%	-	-	-	-	+434.0%	-	-98.8%	-62.4%
6-LUT-1x	-	-	+754.9%	+549.6%	-	-	-	-	-	-	-	-	-	-	-	-	-45.5%	-	-88.0%	-96.2%
6-LUT-5x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-44.7%	-	-88.0%	-96.1%
6-LUT-10x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-45.5%	-	-88.0%	-96.2%
6-LUT-15x	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-45.4%	-	-88.0%	-96.2%

## 2.3 MinCirc instances

The tested SAT instances have been generated using a SAT-based optimum circuit generator MinCirc [25, 40]. These instances were generated when creating optimal four- or five-variable function implementations composed of AND and XOR gates with a preference for XOR gates. There are 2188 SAT instances created when generating 220 four-variable input functions and 3074 instances created when generating 249 optimum five-variable functions. In total, 5262 SAT instances of 469 different functions.

There was a time-limit of one day for optimum circuit generation. Most of the optimum circuits were generated in that time. Processes that did not finish were terminated and only already generated SAT instances are present in the results. Instances are split into three similarly sized groups based on clause count.

### 2.3.1 Synthesis influence on CNF properties

Tables 2.9, 2.10, and 2.11 show how the synthesis and Tseitin transformation influenced the number of variables, the number of clauses, and their ratio, respectively. Interestingly, most of the unsatisfiable instances from the first group and some from the second were transformed into trivial ones (i.e., for variable  $a$ , the CNF would be  $a \wedge \neg a$ ). The numbers of such instances are in Table 2.12. It happened only with *2-gate-[1, 5, 10, 15]x* and *6-LUT-[1, 5, 10, 15]x* syntheses, and more repetitions of the synthesis script led to higher numbers of trivial instances. Even though many instances in the third group are unsatisfiable as well, none of them were transformed into a trivial one. They were probably too complex for synthesis to “solve” them.

The average number of variables and clauses increased significantly after the use of almost all syntheses. If the trivial instances were excluded, the numbers would, on average, increase after the use of all of them. The more complex syntheses or their repetitions (*2-gate-[1, 5, 10, 15]x* and *6-LUT-[1, 5, 10, 15]x*) led to a lower number of variables and clauses compared to other syntheses or less repetitions of the same script. If the trivial instances are left out, standard deviations usually maintain their relative size compared to the averages.

Although the number of clauses and variables mostly increased, their ratio usually decreased. Even though the original ratio differs for each group (average ratios are 4.5, 6.8, and 8.8), after the application of synthesis, and *if trivial instances are left out*, the average ratio for one synthesis is usually only slowly increasing. For *st*, *st-re2-[1, 2, 3]x* and *2-gate-[1, 5, 10, 15]x* syntheses, the ratio is around 2.6–2.9, and for *6-LUT-[1, 5, 10, 15]x* syntheses around 5.1–6.6 (increases with the size of instances).

### 2.3.2 Synthesis influence on solving time

As the tested instances had the upper time limit of 2000 seconds for the SAT-solver (see Section 2.1), some of them were not solved within the limit; if so, the process was terminated. Table 2.13 shows the numbers of instances of each combination of finished/terminated and with synthesis / without synthesis.

Table 2.14 shows the ratios of instances whose time of solving improved after using logic synthesis. If an instance was solved both with and without synthesis, improvement

■ **Table 2.9** MinCirc: influence of logic synthesis on the number of variables

# clauses	[428, 4608)	[4608, 12688)	[12688, 62392)	all instances
# instances	1588	1885	1798	5271
Original	426.3 ± 172.4	1146.8 ± 253.0	2735.8 ± 877.3	1471.8 ± 1098.4
st	6582.4 ± 3936.5	24616.2 ± 6975.4	76881.6 ± 32884.5	37011.5 ± 35596.5
st-re2-1x	4903.3 ± 2950.2	18153.2 ± 5215.4	55430.2 ± 23223.4	26877.0 ± 25438.5
st-re2-2x	4873.0 ± 2945.1	18047.5 ± 5186.1	55246.5 ± 23161.7	26767.4 ± 25367.6
st-re2-3x	4857.8 ± 2943.0	18012.8 ± 5178.3	55190.8 ± 23141.4	26731.4 ± 25347.5
2-gate-1x	1155.7 ± 2266.1	11787.0 ± 6077.9	45689.5 ± 20828.1	20148.6 ± 22780.3
2-gate-5x	756.6 ± 1769.9	10465.0 ± 6279.7	44552.0 ± 20719.0	19167.6 ± 22594.1
2-gate-10x	665.1 ± 1593.6	9716.7 ± 6188.7	43751.1 ± 20698.7	18599.2 ± 22394.6
2-gate-15x	621.2 ± 1504.8	9259.1 ± 6047.2	43060.2 ± 20636.7	18186.7 ± 22170.6
6-LUT-1x	356.9 ± 882.0	4071.9 ± 1937.2	13399.6 ± 5639.7	6134.5 ± 6481.5
6-LUT-5x	340.4 ± 849.7	3876.2 ± 2089.2	13495.2 ± 5689.6	6092.1 ± 6575.7
6-LUT-10x	316.6 ± 796.6	3717.6 ± 2114.6	13431.7 ± 5709.7	6006.6 ± 6583.7
6-LUT-15x	301.5 ± 767.3	3569.4 ± 2126.2	13346.9 ± 5716.8	5920.1 ± 6576.1

■ **Table 2.10** MinCirc: influence of logic synthesis on the number of clauses

# clauses	[428, 4608)	[4608, 12688)	[12688, 62392)	all instances
# instances	1588	1885	1798	5271
Original	2107.9 ± 1234.5	7853.7 ± 2240.6	24834.9 ± 10811.4	11915.1 ± 11573.3
st	18474.3 ± 11317.2	70433.6 ± 20190.4	222492.7 ± 96049.2	106648.9 ± 103539.6
st-re2-1x	13436.9 ± 8357.2	51044.6 ± 14903.9	158138.6 ± 67063.6	76245.5 ± 73061.7
st-re2-2x	13345.8 ± 8342.4	50727.5 ± 14816.4	157587.6 ± 66878.7	75916.7 ± 72848.9
st-re2-3x	13300.4 ± 8336.2	50623.4 ± 14793.1	157420.3 ± 66817.7	75808.7 ± 72788.6
2-gate-1x	3187.3 ± 6257.9	33537.8 ± 17445.9	133276.8 ± 61685.5	58416.2 ± 66862.4
2-gate-5x	2037.1 ± 4782.4	29672.5 ± 17948.2	129950.9 ± 61430.9	55552.9 ± 66331.4
2-gate-10x	1760.8 ± 4234.8	27452.3 ± 17633.4	127536.7 ± 61369.8	53852.2 ± 65727.7
2-gate-15x	1627.4 ± 3957.2	26078.8 ± 17192.2	125462.1 ± 61189.8	52613.1 ± 65060.5
6-LUT-1x	2102.7 ± 5329.2	25184.8 ± 12137.3	88142.8 ± 39197.3	39706.5 ± 43440.3
6-LUT-5x	1976.9 ± 5078.0	23921.1 ± 13082.7	89961.7 ± 40617.7	39837.2 ± 44846.9
6-LUT-10x	1778.6 ± 4620.1	22605.9 ± 13054.7	88990.5 ± 40747.3	38975.8 ± 44718.3
6-LUT-15x	1665.9 ± 4392.2	21510.6 ± 12995.4	87952.1 ± 40719.9	38195.9 ± 44473.6

was achieved only if the time of solving with synthesis (synthesis + SAT-solving) was smaller than solving without synthesis (only SAT-solving). If some instance was solved with synthesis but not without, it is considered to be sped up as well. However, if the time of solving with synthesis (synthesis + SAT-solving) is above the time limit for the SAT solver (2000 real-time seconds (corresponds to 8000 CPU seconds for GLUCOSE (4CPU))). It cannot be decided whether the solving was sped up; in such case, the ratio of improved instances is presented as a range. Instances that were solved neither with nor without synthesis are left out.

The influence of synthesis on the solving time is shown in Table 2.15 and 2.16. The percentage changes are calculated based on instances that *were finished both with and without synthesis within the time limit*. The number of such instances is the first value of the corresponding cell in Table 2.13. This approach results in possibly different instances being used for calculating the percentage changes for different combinations of solver and

■ **Table 2.11** MinCirc: influence of logic synthesis on the clause-variable ratio

# clauses	[428, 4608)	[4608, 12688)	[12688, 62392)	all instances
# instances	1588	1885	1798	5271
Original	$4.5 \pm 1.2$	$6.8 \pm 0.6$	$8.8 \pm 1.0$	$6.8 \pm 1.9$
st	2.8	2.9	2.9	2.8
st-re2-1x	$2.7 \pm 0.1$	2.8	2.8	2.8
st-re2-2x	$2.7 \pm 0.1$	2.8	2.8	2.8
st-re2-3x	$2.7 \pm 0.1$	2.8	2.8	2.8
2-gate-1x	$2.2 \pm 0.3$	$2.7 \pm 0.3$	2.9	$2.6 \pm 0.4$
2-gate-5x	$2.1 \pm 0.3$	$2.7 \pm 0.3$	2.9	$2.6 \pm 0.4$
2-gate-10x	$2.1 \pm 0.2$	$2.6 \pm 0.3$	2.9	$2.6 \pm 0.4$
2-gate-15x	$2.1 \pm 0.2$	$2.6 \pm 0.3$	2.9	$2.6 \pm 0.4$
6-LUT-1x	$2.5 \pm 1.4$	$5.6 \pm 1.5$	$6.5 \pm 0.2$	$5.0 \pm 2.0$
6-LUT-5x	$2.5 \pm 1.3$	$5.4 \pm 1.6$	$6.6 \pm 0.2$	$4.9 \pm 2.1$
6-LUT-10x	$2.5 \pm 1.3$	$5.2 \pm 1.6$	$6.5 \pm 0.2$	$4.8 \pm 2.0$
6-LUT-15x	$2.5 \pm 1.2$	$5.1 \pm 1.7$	$6.5 \pm 0.2$	$4.8 \pm 2.0$

■ **Table 2.12** MinCirc: numbers of unsatisfiable instances which were transformed by synthesis into trivial ones (i.e., for variable  $a$ , the CNF would be  $a \wedge \neg a$ )

# clauses	[428, 4608)	[4608, 12688)	[12688, 62392)	all instances
# UNSAT instances	1511	1197	584	3292
st	0	0	0	0
st-re2-1x	0	0	0	0
st-re2-2x	0	0	0	0
st-re2-3x	0	0	0	0
2-gate-1x	1251	295	0	1546
2-gate-5x	1334	397	0	1731
2-gate-10x	1345	441	0	1786
2-gate-15x	1350	459	0	1809
6-LUT-1x	1350	278	0	1628
6-LUT-5x	1353	353	0	1706
6-LUT-10x	1356	388	0	1744
6-LUT-15x	1360	421	0	1781

synthesis, which, in some cases, causes worse comparability if many instances are left out. We chose this approach as the number of instances that were solved by all solvers without synthesis and with all syntheses would sometimes be rather limited. Moreover, the main goal of this thesis is to evaluate the improvements achievable by synthesis, and this approach allows for individual results to be more precise as they are based on more instances.

Likewise, the average time shown in each column can be compared between solvers only if all instances (or most of them) were solved without synthesis. If some instances are left out (due to not being solved within the time limit), the real average time would be much higher than the presented one. Therefore, it usually offers only a rough idea about the solving time to which the percentage changes can be related.

In many cases, the average time of solving for different solvers in one category varies greatly. This, even with approximate results, means that a large potential improvement

of using some synthesis would not usually be useful in practice as using the fastest solver (even without any synthesis) would result in a faster time.

The values in Table 2.15 are calculated using Equation 2.1 where  $(nosyn, syn)$  are pairs of solving time without synthesis (only SAT-solving of the original instance) and with synthesis (synthesis + SAT-solving). The values show how the original total time would change if synthesis were used only on instances when it decreased the overall time of solving that instance and others were solved without it. These values show how large improvement *can be potentially achieved* with each synthesis script on tested instances. On its own, it does not reflect any practical application. However, for synthesis to have practical use, a (high) potential decrease in solving time is a necessary condition. In the following paragraphs, mainly, this potential improvement is discussed.

Table 2.16 presents how the solving time would be influenced by always using synthesis.

In the group of smallest-sized instances, the solvers usually solved all or most of the instances in the limit, both with and without synthesis. Almost all instances in this category are unsatisfiable, and most of them were transformed into trivial instances by *2-gate-[1, 5, 10, 15]x* and *6-LUT-[1, 5, 10, 15]x* syntheses scripts; more repetitions of the script resulted in slightly more trivial instances.

All solvers solved the original instances really fast (in less than 0.2 seconds), with MINISAT being the fastest. Synthesis led to a speedup very rarely and potential improvements are very low or none as well. The highest potential improvement was 3.2% using GLUCOSE with *st* synthesis.

Almost all of the middle-sized instances were still solved both with and without the synthesis within the time limit. However, MINISAT did not solve some of the instances after using the synthesis (only units of instances), and zCHAF had similar results with the number of unsolved instances after using synthesis slightly higher (at most around 3%).

MINISAT had the lowest average time of 5.0 seconds. Interestingly, it achieved the highest potential improvement (from all combinations of solvers and syntheses in this category of instances) of 24.0% with *6-LUT-1x* synthesis (only 1.4% of instances were sped up) and of 23.7% with *st* synthesis (9.9% of instances were sped up). The zCHAF solver had the highest average time of 28.0 seconds. Its highest potential improvement of 22.2% was with *st* as well (8.4% of instances were sped up). GLUCOSE and GLUCOSE (4CPU) had almost the same average time of 15.1 and 15.0 seconds, respectively. GLUCOSE achieved the largest potential speedup of 7.6% with *st* synthesis. This led to solving faster 35.6% of instances. GLUCOSE (4CPU) achieved slightly higher potential improvement of 10.6% with *st-re2-1x* synthesis, which sped up solving of 32.9% of instances.

In the last group, MINISAT, GLUCOSE, and GLUCOSE (4CPU) solvers have similarly distributed solved/unsolved instances. Most of the instances were solved both with and without synthesis, while 11%–15% of them were solved in the limit neither with nor without synthesis. The number of instances solved only with synthesis or only without it is similarly low, with the latter being higher in some cases. On the contrary, the zCHAF solver solved within the limit only a small number of instances, most of them being solved only without synthesis. Only about 15% of instances was solved within the time-limit both with and without synthesis. In this category, no instances were



transformed into trivial ones.

In this last category of instances, MINISAT had the lowest average time of 141.4 seconds. It achieved lower potential improvements and a higher ratio of improved instances compared to the middle-sized ones. The biggest potential improvement was 20.7% with *6-LUT-1x* synthesis, which improved solving time of 14.4% of instances. zCHAF had the highest average time of 478.8 seconds. With all syntheses, it achieved high potential improvements (26.8%–60.7), the highest being with *st-re2-2x* synthesis, which sped up 27.6% of instances. These results are, however, based only on a small number of finished instances, as most of them were not solved within the time limit. GLUCOSE had an average time of 340.5 seconds. The highest potential improvement of 19.7% was achieved with *st* synthesis and sped up 35.6% of instances. Lastly, GLUCOSE (4CPU) had a slightly higher average time than GLUCOSE at 397.1 seconds; however, finishing slightly more instances within the time limit. The highest potential decrease of solving time of 28.9% was achieved with *st* synthesis as well, speeding up solving of 37.9% instances.

Based on the overall results, MINISAT, GLUCOSE and GLUCOSE (4CPU) finished around 90% of all instances both with and without synthesis within the time-limit. zCHAF, on the other hand, finished only around 70% of them. The MINISAT solver had the lowest average time of solving the original instances of 45.1 seconds. The highest potential improvement of 20.8% was achieved with *6-LUT-1x* synthesis, speeding up only 4.9% of instances; the second was *st* synthesis with a slightly lower potential improvement of 17.3%, which, however, sped up 10.4% of instances. MINISAT seems to be, the fastest solver for this problem. zCHAF had an average time of 60.3 seconds. With *st* synthesis it achieved the highest potential speedup of 43.5%, speeding up 8.5% of instances. This is, however, overshadowed by the large number of instances unfinished within the time limit. GLUCOSE and GLUCOSE (4CPU) achieved similar results, with the latter having a higher average time as well as potential improvement and ratio of sped-up instances. GLUCOSE had an average time of 109.9 seconds and the biggest potential improvement of 19.0% with *st* synthesis, which sped up 19.7% of instances. GLUCOSE (4CPU) had an average time of 129.6 seconds and the biggest potential speedup of 28.0% with *st* synthesis as well. It sped up 21.8% of instances, a bit more than with GLUCOSE.

Interestingly, the overall results show that repeating *2-gate* or *6-LUT* synthesis script more than once (i.e. *2-gate-[5, 10, 15]x* or *6-LUT-[5, 10, 15]x* syntheses) did not increase the potential improvement nor the ratio of sped up instances; with *st-re2* script (i.e. *st-re2-[1, 2, 3]x*), it is not clear how many repetitions leads to best results.

Always using synthesis would increase the solving time in all cases. However, compared to benchmark instances (Subsection 2.2.2), the increases are not as high. If the average time of solving original instances was higher compared to other solvers, the increase was, in most cases, below +100% (i.e., below twice the original time). With synthesis that achieved the highest potential speedups (*st* and *6-LUT-1x*), the slowdowns of using theses always were not as high compared to other syntheses.

In Chapter 3, methods of incorporating synthesis into SAT-solving are explored. Based on these results, only the combination of MINISAT, which had the lowest average time on solved instances, while solving almost all of them, with *st* and *6-LUT-1x* syntheses will be used. With the MINISAT solver, *st* synthesis had the highest improvement rate of 10.4% and the second highest potential improvement of 17.3%; *6-LUT-1x* had the highest potential speedup of 20.8% but significantly lower (although second highest

for this solver) improvement rate of 4.9%.

■ **Table 2.13** MinCirc: separated by slashes are the numbers of instances (1) which were solved in 2000 seconds (real-time) both with and without synthesis, (2) which were solved in 2000 seconds (real-time) with synthesis but not without it, (3) which were solved in 2000 seconds (real-time) without synthesis but not with it, (4) which were solved in 2000 seconds (real-time) neither with nor without synthesis. In case values (2), (3), and (4) are all zeroes, they are omitted.

# clauses Solver	[428, 4608)				[4608, 12688)				[12688, 62392)			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
st	1588	1588	1588	1588	1884/0/1/0	1860/2/12/11	1885	1885	1447/13/84/254	275/53/98/1372	1475/27/55/241	1565/15/11/207
st-re2-1x	1588	1588	1588	1588	1876/0/9/0	1812/1/60/12	1885	1885	1320/9/211/258	214/28/159/1397	1431/23/99/245	1558/10/18/212
st-re2-2x	1588	1588	1588	1588	1881/0/4/0	1812/2/60/11	1884/0/1/0	1885	1363/8/168/259	219/33/154/1392	1447/28/83/240	1566/13/10/209
st-re2-3x	1588	1588	1588	1588	1878/0/7/0	1810/2/62/11	1884/0/1/0	1885	1318/10/213/257	223/32/150/1393	1450/24/80/244	1561/10/15/212
2-gate-1x	1588	1588	1588	1588	1884/0/1/0	1840/2/32/11	1885	1885	1465/14/66/253	249/38/124/1387	1455/25/75/243	1563/17/13/205
2-gate-5x	1588	1588	1588	1588	1885	1839/2/33/11	1885	1885	1471/16/60/251	246/44/127/1381	1450/31/80/237	1564/13/12/209
2-gate-10x	1588	1588	1588	1588	1885	1843/2/29/11	1884/0/1/0	1885	1466/18/65/249	246/39/127/1386	1435/26/95/242	1562/18/14/204
2-gate-15x	1588	1588	1588	1588	1884/0/1/0	1838/1/34/12	1885	1885	1464/14/67/253	238/34/135/1391	1440/25/90/243	1554/17/22/205
6-LUT-1x	1588	1588	1588	1588	1885	1833/2/39/11	1885	1885	1502/14/29/253	254/44/119/1381	1450/25/80/243	1565/14/11/208
6-LUT-5x	1588	1588	1588	1588	1885	1846/2/26/11	1885	1885	1505/21/26/246	257/47/116/1378	1450/30/80/238	1562/18/14/204
6-LUT-10x	1588	1588	1588	1588	1885	1850/2/22/11	1885	1885	1508/24/23/243	268/44/105/1381	1448/29/82/239	1560/13/16/209
6-LUT-15x	1588	1588	1588	1588	1885	1844/2/28/11	1885	1885	1513/23/18/244	261/44/112/1381	1460/29/70/239	1561/14/15/208

# clauses Solver	all instances			
	MiniSat	zChaff	Glucose	Gl. 4cpu
st	4919/13/85/254	3723/55/110/1383	4948/27/55/241	5038/15/11/207
st-re2-1x	4784/9/220/258	3614/29/219/1409	4904/23/99/245	5031/10/18/212
st-re2-2x	4832/8/172/259	3619/35/214/1403	4919/28/84/240	5039/13/10/209
st-re2-3x	4784/10/220/257	3621/34/212/1404	4922/24/81/244	5034/10/15/212
2-gate-1x	4937/14/67/253	3677/40/156/1398	4928/25/75/243	5036/17/13/205
2-gate-5x	4944/16/60/251	3673/46/160/1392	4923/31/80/237	5037/13/12/209
2-gate-10x	4939/18/65/249	3677/41/156/1397	4907/26/96/242	5035/18/14/204
2-gate-15x	4936/14/68/253	3664/35/169/1403	4913/25/90/243	5027/17/22/205
6-LUT-1x	4975/14/29/253	3675/46/158/1392	4923/25/80/243	5038/14/11/208
6-LUT-5x	4978/21/26/246	3691/49/142/1389	4923/30/80/238	5035/18/14/204
6-LUT-10x	4981/24/23/243	3706/46/127/1392	4921/29/82/239	5033/13/16/209
6-LUT-15x	4986/23/18/244	3693/46/140/1392	4933/29/70/239	5034/14/15/208

■ **Table 2.14** MinCirc: ratio of instances whose time of solving decreased after using the synthesis preprocessing. Instances that were solved neither with nor without synthesis are left out.

# clauses Solver	[428, 4608)				[4608, 12688)				[12688, 62392)				all instances			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
st	0.0%	0.6%	2.4%	2.6%	9.9%	8.4%	21.3%	24.2%	21.8%	39.0%	35.6%	37.9%	10.4%	8.5%	19.7%	21.8%
st-re2-1x	0.0%	0.0%	0.7%	2.1%	3.3%	2.3%	15.4%	18.8%	7.5%	24.4%	30.3%	32.9%	3.6%	3.7%	15.3%	18.0%
st-re2-2x	0.0%	0.0%	0.3%	0.6%	3.0%	3.1%	12.2%	16.3%	7.6%	27.6%	31.7%	33.5%	3.5%	4.4%	14.5%	16.8%
st-re2-3x	0.0%	0.0%	0.1%	0.4%	1.9%	2.3%	12.2%	14.5%	7.5%	27.7%	29.0%	32.2%	3.0%	4.0%	13.6%	15.6%
2-gate-1x	0.0%	0.0%	0.0%	0.0%	0.4%	2.0%	0.9%	1.3%	7.1%	29.0%–29.4%	14.9%–15.0%	18.1%	2.3%	4.1%–4.1%	5.0%–5.0%	6.2%
2-gate-5x	0.0%	0.0%	0.0%	0.0%	0.4%	0.8%	0.4%	0.3%	3.9%–4.0%	23.7%–24.5%	8.2%	9.3%	1.4%–1.4%	2.9%–3.0%	2.7%	3.0%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	0.3%	0.5%	0.2%	0.2%	2.6%–2.7%	20.1%–20.9%	4.8%–5.1%	6.5%	0.9%–0.9%	2.4%–2.5%	1.6%–1.6%	2.1%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	0.1%	0.6%	0.0%	0.0%	1.9%–2.0%	16.7%–17.2%	3.7%–4.2%	4.5%	0.6%–0.7%	2.1%–2.1%	1.1%–1.3%	1.4%
6-LUT-1x	0.0%	0.0%	0.0%	0.0%	1.4%	2.2%	2.5%	4.6%	14.4%	32.6%	17.9%	20.5%	4.9%	4.6%	6.5%	8.1%
6-LUT-5x	0.0%	0.0%	0.0%	0.0%	0.6%	1.4%	0.6%	0.6%	10.2%	29.5%–30.2%	10.9%–11.0%	12.0%	3.4%	3.9%–3.9%	3.6%–3.6%	4.0%
6-LUT-10x	0.0%	0.0%	0.0%	0.0%	0.4%	0.6%	0.4%	0.4%	7.3%	27.3%–27.6%	8.4%–8.6%	9.2%	2.4%	3.2%–3.3%	2.7%–2.8%	3.0%
6-LUT-15x	0.0%	0.0%	0.0%	0.0%	0.3%	0.6%	0.2%	0.3%	5.4%–5.5%	24.7%–25.2%	6.0%–6.3%	6.5%	1.8%–1.8%	2.9%–3.0%	1.9%–2.0%	2.1%

■ **Table 2.15** MinCirc: potential influence of synthesis on the solving time (i.e., using synthesis only when decreases the solving time of an instance). Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit.

# clauses	[428, 4608)				[4608, 12688)				[12688, 62392)				all instances			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
Average time	0.0	0.1	0.2	0.2	5.0	28.0	15.1	15.0	141.4	478.8	340.5	397.1	45.1	60.3	109.9	129.6
st	0.0%	-0.8%	-3.2%	-2.9%	-23.7%	-22.2%	-7.6%	-10.1%	-16.9%	-54.5%	-19.7%	-28.9%	-17.3%	-43.5%	-19.0%	-28.0%
st-re2-1x	0.0%	0.0%	-1.2%	-2.4%	-14.2%	-12.7%	-7.3%	-10.6%	-13.3%	-52.1%	-17.2%	-26.7%	-13.4%	-35.6%	-16.6%	-26.0%
st-re2-2x	0.0%	0.0%	-0.3%	-0.6%	-10.4%	-12.9%	-5.8%	-7.6%	-13.1%	-60.7%	-17.6%	-25.1%	-12.9%	-42.6%	-16.9%	-24.3%
st-re2-3x	0.0%	0.0%	-0.1%	-0.7%	-11.6%	-11.1%	-7.0%	-9.2%	-15.4%	-57.0%	-17.3%	-25.3%	-15.2%	-40.9%	-16.7%	-24.5%
2-gate-1x	0.0%	0.0%	0.0%	0.0%	-15.7%	-13.5%	-1.9%	-2.8%	-14.1%	-50.5%	-11.6%	-22.0%	-14.2%	-35.7%	-11.0%	-21.1%
2-gate-5x	0.0%	0.0%	0.0%	0.0%	-11.4%	-8.4%	-2.0%	-1.1%	-7.9%	-42.6%	-7.1%	-18.5%	-8.1%	-30.5%	-6.8%	-17.7%
2-gate-10x	0.0%	0.0%	0.0%	0.0%	-4.7%	-8.8%	-0.9%	-1.1%	-4.5%	-29.5%	-4.3%	-12.9%	-4.5%	-21.9%	-4.1%	-12.3%
2-gate-15x	0.0%	0.0%	0.0%	0.0%	-0.9%	-9.6%	0.0%	0.0%	-4.2%	-26.8%	-2.7%	-10.8%	-4.0%	-20.2%	-2.5%	-10.3%
6-LUT-1x	0.0%	0.0%	0.0%	0.0%	-24.0%	-14.5%	-3.4%	-4.7%	-20.7%	-53.2%	-14.0%	-25.8%	-20.8%	-39.2%	-13.4%	-24.8%
6-LUT-5x	0.0%	0.0%	0.0%	0.0%	-15.3%	-14.2%	-1.6%	-2.0%	-16.9%	-49.2%	-9.7%	-17.9%	-16.8%	-36.9%	-9.2%	-17.2%
6-LUT-10x	0.0%	0.0%	0.0%	0.0%	-9.8%	-7.5%	-1.2%	-1.8%	-13.5%	-43.8%	-7.7%	-15.5%	-13.4%	-30.9%	-7.3%	-14.9%
6-LUT-15x	0.0%	0.0%	0.0%	0.0%	-10.1%	-4.9%	-0.5%	-1.9%	-9.3%	-40.9%	-5.5%	-13.6%	-9.3%	-29.4%	-5.3%	-13.1%

■ **Table 2.16** MinCirc: the influence of synthesis on the solving time if used always; values larger than or equal to +1000% are replaced by “-”. Only instances solved both with and without synthesis are considered. The average time is calculated from instances solved without synthesis within the time limit.

# clauses	[945, 8023)				[8023, 16443)				[16443, 62392)				all instances			
	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu	MiniSat	zChaff	Glucose	Gl. 4cpu
Average time	0.0	0.1	0.2	0.2	5.0	28.0	15.1	15.0	141.4	478.8	340.5	397.1	45.1	60.3	109.9	129.6
st	+601.6%	+368.3%	+88.4%	+72.7%	+142.6%	+320.3%	+18.2%	+7.6%	+161.8%	+5.3%	+30.1%	+23.6%	+160.9%	+112.3%	+29.5%	+22.9%
st-re2-1x	-	+973.1%	+147.1%	+121.1%	+490.2%	+843.4%	+23.4%	+8.1%	+278.9%	+58.2%	+44.6%	+43.2%	+289.4%	+387.5%	+43.4%	+41.7%
st-re2-2x	-	-	+208.9%	+176.5%	+424.2%	+905.9%	+30.9%	+16.2%	+277.1%	+35.1%	+43.3%	+40.5%	+285.1%	+365.5%	+42.7%	+39.4%
st-re2-3x	-	-	+254.2%	+216.5%	+661.9%	+955.8%	+22.7%	+16.4%	+312.8%	+38.5%	+44.3%	+40.7%	+332.1%	+360.2%	+43.2%	+39.7%
2-gate-1x	-	-	-	-	+462.9%	+556.5%	+137.0%	+128.2%	+198.1%	+36.6%	+69.5%	+68.2%	+213.3%	+248.1%	+74.2%	+71.7%
2-gate-5x	-	-	-	-	-	+872.6%	+391.1%	+389.8%	+331.2%	+87.6%	+127.2%	+108.7%	+382.2%	+374.6%	+144.3%	+123.0%
2-gate-10x	-	-	-	-	-	-	+712.4%	+680.4%	+520.1%	+151.4%	+200.8%	+176.1%	+604.0%	+509.6%	+233.3%	+201.5%
2-gate-15x	-	-	-	-	-	-	+959.9%	+960.2%	+730.7%	+194.5%	+262.6%	+245.6%	+855.6%	+589.5%	+306.8%	+282.2%
6-LUT-1x	-	-	+751.0%	+682.6%	+251.4%	+438.5%	+83.5%	+67.8%	+74.8%	+41.9%	+57.5%	+52.1%	+84.0%	+187.1%	+59.5%	+53.2%
6-LUT-5x	-	-	-	-	+668.2%	+577.6%	+226.9%	+214.7%	+145.9%	+70.0%	+96.9%	+98.6%	+172.1%	+251.3%	+105.3%	+104.6%
6-LUT-10x	-	-	-	-	-	+621.5%	+384.5%	+371.1%	+250.5%	+97.5%	+145.4%	+131.6%	+293.3%	+289.8%	+160.7%	+143.8%
6-LUT-15x	-	-	-	-	-	+878.2%	+536.7%	+527.4%	+347.5%	+109.1%	+190.2%	+172.3%	+407.5%	+363.0%	+212.1%	+189.9%

## 2.4 ATPG instances

For this set of experiments, instances from a simple in-house SAT-based ATPG based on the original idea of Larrabee [26] were used. This approach generates conceptual hardware (*miter*) by XORing the fault-free and faulty circuit. This miter is then converted to a CNF by the Tseitin transformation [9], and a test vector is generated as a satisfiability proof. It is possible to use synthesis to optimize the miter before doing the Tseitin transformation, to make the CNF smaller. No CNF to BLIF conversion in this case is needed, since the miter can be dumped to BLIF and optimized by logic synthesis. Thus, the synthesis can be directly incorporated into the ATPG process, without a need for any conversions.

The ATPG process was run on circuits from a mixture of logic synthesis and testing benchmarks [41]. Generated SAT instances were solved by MINISAT solver.

### 2.4.1 Synthesis influences on solving time

Table 2.17 shows the ratios of circuits that were solved faster if synthesis was used. Circuits are divided into groups based on the solving without synthesis in seconds. The results are reported for a circuit as a whole, i.e., the ATPG process, including solving all SAT instances together, i.e., for all tested faults. In the first group containing circuits solved in up to 5 seconds, almost none were sped up by synthesis preprocessing. With increasing the original time of solving, the circuits were usually more likely to be sped up by synthesis. The more complicated syntheses (*2-gate-[1, 5, 10, 15]x* and *6-LUT-[1, 5, 10, 15]x*) did not achieve as good results as other syntheses, speeding up at most 35.0% that originally took 1000 or more seconds. The simpler syntheses (*st* and *st-re2-[1, 2, 3]x*) were more successful, as they all sped up more than 50% of the circuits that originally took 1000 or more seconds. Nevertheless, most circuits were sped up by *st* synthesis script, which sped up the highest number of them in each group of circuits, as well as more than half of the circuits that originally took 100 seconds or more; in total speeding up 10.5% of all circuits.

Table 2.18 shows potential improvements that could be achieved if the synthesis was used only if it decreased the solving time. Values in the table are calculated using Equation 2.1 where (*nosyn, syn*) are pairs of solving time without synthesis and with synthesis. And Table 2.19 shows how the original time of solving would change if synthesis was used on all instances.

Potential improvements achievable with syntheses are similar to ratios of sped-up circuits. The fastest-solved circuits could usually not be sped up by synthesis, while those that took longer to solve could. *st* and *st-re2-[1, 2, 3]x* syntheses achieved high potential decrease in solving time, with *st* having the highest, in total 32.0%.

Most of the potential improvement achieved by synthesis disappears if the synthesis is used on all circuits. In many cases, mainly for faster-solved circuits, using synthesis always would lead to a large increase in the solving time. Nonetheless, *st* synthesis sped up solving of circuits from 100 seconds up, with the exception of 2.3% increase for circuits which originally took [200, 500) seconds. It was the only synthesis that decreased the total time of solving while using synthesis on all instances – it achieved a significant time decrease of 19.7%.

■ **Table 2.17** ATPG circuits: ratio of circuits whose time of solving decreased after using logic synthesis

Original time	[0, 5)	[5, 50)	[50, 100)	[100, 200)	[200, 500)	[500, 1000)	[1000, inf)	all circuits
# circuits	788	274	60	47	27	32	20	1248
st	0.8%	10.2%	23.3%	61.7%	51.9%	78.1%	75.0%	10.5%
st-re2-1x	0.3%	4.4%	15.0%	51.1%	48.1%	43.8%	70.0%	7.1%
st-re2-2x	0.3%	3.6%	10.0%	48.9%	40.7%	40.6%	55.0%	6.1%
st-re2-3x	0.3%	2.2%	8.3%	38.3%	29.6%	37.5%	65.0%	5.1%
2-gate-1x	0.0%	0.7%	3.3%	25.5%	7.4%	6.2%	35.0%	2.2%
2-gate-5x	0.0%	1.1%	1.7%	21.3%	11.1%	3.1%	35.0%	2.0%
2-gate-10x	0.0%	0.4%	1.7%	2.1%	0.0%	3.1%	30.0%	0.8%
2-gate-15x	0.0%	0.4%	1.7%	0.0%	0.0%	3.1%	35.0%	0.8%
6-LUT-1x	0.0%	1.1%	1.7%	21.3%	3.7%	6.2%	35.0%	1.9%
6-LUT-5x	0.0%	1.1%	1.7%	19.1%	3.7%	3.1%	35.0%	1.8%
6-LUT-10x	0.0%	1.1%	1.7%	0.0%	3.7%	3.1%	35.0%	1.0%
6-LUT-15x	0.0%	0.7%	1.7%	0.0%	3.7%	3.1%	35.0%	1.0%

■ **Table 2.18** ATPG circuits: potential influence of synthesis on solving time (i.e., using synthesis only when decreases solving time of an instance)

Original time	[0, 5)	[5, 50)	[50, 100)	[100, 200)	[200, 500)	[500, 1000)	[1000, inf)	all circuits
# circuits	788	274	60	47	27	32	20	1248
st	-0.5%	-3.3%	-7.5%	-12.2%	-9.3%	-20.1%	-51.2%	-32.0%
st-re2-1x	-0.2%	-2.3%	-3.5%	-14.7%	-8.7%	-7.6%	-50.0%	-28.4%
st-re2-2x	-0.2%	-1.9%	-3.1%	-13.0%	-7.0%	-6.5%	-48.8%	-27.2%
st-re2-3x	-0.2%	-1.5%	-2.4%	-11.8%	-5.9%	-5.5%	-48.6%	-26.6%
2-gate-1x	0.0%	-0.8%	-1.5%	-7.3%	-1.2%	-1.5%	-41.8%	-21.5%
2-gate-5x	0.0%	-0.8%	-1.2%	-3.1%	-5.7%	-0.8%	-40.7%	-20.9%
2-gate-10x	0.0%	-0.5%	-0.9%	-0.1%	0.0%	-0.3%	-39.6%	-19.5%
2-gate-15x	0.0%	-0.5%	-0.6%	0.0%	0.0%	-0.2%	-38.4%	-18.9%
6-LUT-1x	0.0%	-1.0%	-1.4%	-5.4%	-3.0%	-1.0%	-38.7%	-19.9%
6-LUT-5x	0.0%	-0.8%	-1.3%	-1.0%	-2.8%	-0.8%	-39.0%	-19.7%
6-LUT-10x	0.0%	-0.6%	-1.1%	0.0%	-2.8%	-0.6%	-38.5%	-19.3%
6-LUT-15x	0.0%	-0.5%	-0.9%	0.0%	-2.8%	-0.7%	-38.1%	-19.1%

As circuits that took longer to solve are more likely to be solved faster with synthesis, and using synthesis on all circuits that took longer to solve, on average, reduces this improvement only slightly, it offers an opportunity to explore – one can attempt to solve the instance without using synthesis for some time, and if after some time  $t$  instance is not solved, end the process and rerun it with synthesis. This approach is explored in Chapter 3.



■ **Table 2.19** ATPG circuits: influence of synthesis on solving time if used always

Original time # circuits	[0, 5) 788	[5, 50) 274	[50, 100) 60	[100, 200) 47	[200, 500) 27	[500, 1000) 32	[1000, inf) 20	all circuits 1248
st	+357.8%	+65.7%	+20.0%	-3.4%	+2.3%	-19.0%	-46.8%	-19.7%
st-re2-1x	+955.4%	+130.4%	+41.5%	+2.8%	+4.6%	+31.8%	-46.1%	+2.7%
st-re2-2x	+1367.2%	+163.1%	+47.1%	+11.4%	+11.9%	+39.6%	-44.5%	+11.9%
st-re2-3x	+1670.1%	+182.9%	+66.5%	+14.6%	+17.0%	+45.6%	-44.2%	+18.6%
2-gate-1x	+2510.3%	+883.8%	+192.6%	+117.2%	+89.3%	+113.1%	-17.9%	+115.1%
2-gate-5x	+3285.9%	+1175.8%	+314.1%	+175.4%	+103.8%	+137.8%	-8.6%	+160.1%
2-gate-10x	+4245.7%	+1441.0%	+416.8%	+225.2%	+141.4%	+156.0%	-0.5%	+203.3%
2-gate-15x	+4917.9%	+1668.1%	+473.0%	+279.3%	+159.2%	+177.4%	+6.7%	+238.6%
6-LUT-1x	+2541.9%	+927.2%	+308.4%	+137.7%	+96.6%	+125.8%	-10.4%	+132.4%
6-LUT-5x	+3263.8%	+1280.1%	+521.8%	+183.1%	+135.2%	+152.3%	-3.2%	+185.8%
6-LUT-10x	+3778.4%	+1550.0%	+600.7%	+226.6%	+151.1%	+178.4%	+0.8%	+222.2%
6-LUT-15x	+4515.9%	+1760.0%	+628.0%	+260.2%	+179.8%	+190.4%	+3.7%	+250.8%

# Using synthesis in SAT-solving

## 3.1 Explored approaches

### 3.1.1 Restarting

One possibility of incorporating synthesis into SAT or ATPG solving is to try to solve an instance without synthesis for some time  $t$ , and if it does not finish in that time, end the process and run it once again but with the synthesis. This approach is based on the fact that synthesis is unnecessary and usually decremental for simple instances.

Finding optimal  $t$  based on the measured instance solving times can be formalized by the following expression, where  $(nosyn, syn)$  are pairs of solving times without and with synthesis.

$$\arg \min_t \sum_{(nosyn, syn)} (\min\{nosyn, t\} + [nosyn > t] \cdot syn) \quad (3.1)$$

From experimenting on data from ATPG with  $st$  synthesis, this approach can achieve reasonable results; however, the solving time of many instances was increased. This can be addressed by adding a constraint (Equation 3.2) specifying the minimum ratio of instances  $x \in [0, 1]$  that should be solved in the same or faster time<sup>1</sup>.

$$\frac{\sum_{(nosyn, syn)} [(\min\{nosyn, t\} + [nosyn > t] \cdot syn) < nosyn]}{\sum_{(nosyn, syn)} 1} \geq x \quad (3.2)$$

A downside of this approach is that it increases the time of solving the instances that were restarted and solved with synthesis by the time  $t$ .

### 3.1.2 Selecting instances to run with synthesis

A different approach is to try to predict whether to use synthesis based on some properties of an instance. There are two possibilities that we can predict: (1) we can predict one of two classes (binary classification), whether it is beneficial to use synthesis or not,

<sup>1</sup>Since this model did not achieve the best results when compared to others. This constraint was not tried as it limits the maximal improvement the model can achieve.

or (2) real value (regression) corresponding to how sure it is that synthesis will improve the solving time. In the first case, synthesis is beneficial if  $syn < nosyn$ . In the second case, we take  $\log(\frac{nosyn}{syn})$  (both the divisor and whole fraction needs to be increased by a small constant, e.g.,  $10^{-9}$ , in order to avoid undefined operations). This value is zero if  $syn = nosyn$ , less than zero if  $syn > nosyn$ , and greater than zero  $syn < nosyn$ . The larger the absolute value of the logarithm is, the more sure the use or not use of synthesis is. It can also be mapped to  $[0, 1]$  interval using sigmoid function ( $\sigma(x) = \frac{1}{1+e^{-x}}$ ). In which case, the value can be interpreted as “*how high is the probability of synthesis speeding up the SAT-solving*”.

In either case, the main problem is the imbalance in data – at best only around 10.5% of instances were sped up after using logic synthesis. This can be problematic as models can end up predicting the more frequent class in the case of classification or only values less than zero in the case of regression.

Multiple machine learning (ML) models were tried for selecting instances: linear regression, logistic regression, naive Bayes, k-nearest neighbors, random forest, and gradient boosted decision trees. If models have hyperparameters that can be tuned, multiple different values were tried.

### 3.1.3 Selecting instances for restarting

This approach combines both previously described ones. Firstly, an ML model is trained to predict whether to use synthesis preprocessing. Then, the restarting model’s  $t$  is found *only* on instances selected to be run with synthesis.

The final model predicts whether to restart (use synthesis) after  $t$  seconds of solving without synthesis. Instances that should not be restarted are solved without synthesis. Those that should be restarted are run without synthesis for time  $t$  and then restarted and run with synthesis, in case they have not already been solved.

The idea is that if the ML model falsely predicts to use the synthesis preprocessing for an instance that would be solved quickly without it, the restarting part fixes it in some cases. If the ML model predicts the use of synthesis always, this model transforms into the original restarting model; similarly, if the model’s  $t$  is around 0, it transforms into the selecting model.

## 3.2 Model training and evaluation

Since the dataset is relatively small, instances sped up by synthesis are scarce, and improvements achieved using synthesis differ significantly; using train, dev, and test sets for evaluating models is not ideal as the results would strongly depend on how instances were divided. For this reason, a cross-validation (CV) will be used. This should help with this problem as the models are trained and evaluated multiple times on different parts of data and results are then averaged together.

In the case of the restarting model or if no hyperparameters of the model are tuned, a simple 5-fold CV is used. In other cases, nested CV is used. The outer CV is 5-fold and splits the test part of the data from the train and dev parts, and the inner CV is 4-fold and splits the train and dev parts. The inner CV selects the best parameters and the outer is used to measure the quality of the best parameters. The reported results

are averages from test parts, while the models with the best parameters are trained on the rest of the data.

Using simple metrics such as accuracy or F1-score to evaluate the correctness of predictions is not enough since the speedups or slowdowns of using synthesis differ significantly between instances – the goal is to correctly predict instances where using synthesis would lead to significant speedup or slowdown. One metric that takes this into consideration is the average change in solving time (Equation 3.3; the model is represented as a function predicting the solving time of each instance). Results below zero mean a speedup, while those above a slowdown. This metric is used for selecting the best model. Test parts used for model evaluation have different total times of solving instances without synthesis. Using a simple average to aggregate the results of this metric together would not take this into account, thus the weighted average and standard deviation are used with the total solving times without synthesis as the weights.

$$\frac{\sum_{(nosyn, syn, data)} model(nosyn, syn, data)}{\sum_{(nosyn, syn, data)} nosyn} - 1 \quad (3.3)$$

Another presented metric is the ratio of instances whose solving time was the same or faster. This is complementary to the ratio of instances that were solved in a longer time due to the use of the model. In case solving of no instance was slowed down, and there is a speedup in solving time as well, using this model has no downside.

To measure how often synthesis is used, the ratio of instances that were solved with it is also reported.

### 3.3 MinCirc instances

In this section, data and results regarding the use of synthesis while solving SAT instances from the MinCirc [25] optimum circuit generator are described. Results of solving these instances are in detail described in Section 2.3.

#### 3.3.1 Data

Some instances were solved only with or without synthesis, in order to be able to include them in data used for training and evaluating the models. The unknown time value was set to 2500 seconds (10000 seconds for GLUCOSE (4CPU)). This is an arbitrary value, which might, on average, approximate actual times of solving. An approach using linear regression for predicting the missing value from the other one was tried. However, only a handful of outliers were solved only with synthesis, thus choosing an arbitrary value does not make much difference, and instances that were solved only without synthesis are mostly also outliers and using linear regression would lead to predicting values too large, which would be problematic with the chosen metric.

Since the number of instances solved only without synthesis is, in most cases, higher or the same as those solved only with synthesis, this approach should not lead to favoring the use of synthesis.

Instances that were solved neither with nor without synthesis are left out.

Same as in Chapter 2, the time of solving with synthesis includes only the time of synthesis and subsequent SAT solving. The time of file conversion from CNF to BLIF and back is not taken into account, as the tool used them is not optimized for speed.

For predicting whether to use synthesis, 43 features extracted from each SAT instance are used. Most of them (38) are extracted using the SATfeatPy library<sup>2</sup>, and they correspond to features 1–33 used in SATzilla solver [42] extended by a few similar ones.

Since MinCirc sequentially generates instances with increasing size, five additional domain-specific features are included: (1) the number of the MinCirc instance, (2–3) one-hot encoded information, whether the previous MinCirc instance was satisfiable or not, and (4–5) one-hot encoded information, whether it is an instance from four or five-variable function generation.

Each model was trained on unpreprocessed data and data preprocessed using min-max normalization and standardization.

Only results of *st* and *6-LUT-1x* syntheses with the MINISAT solver are used since MINISAT was the fastest solver and with both *st* and *6-LUT-1x* syntheses had the best results. *st* sped up solving of more instances, while *6-LUT-1x* achieved higher potential improvement.

### 3.3.2 Results

The results of the ten best ML models ordered by the total time change with *st* synthesis are in Table 3.1 and with *6-LUT-1x* are in Table 3.2.

*Model change* is the change in solving time resulting from the use of the model compared to the total original time. *Same or better* is the ratio of instances whose solving time was the same or better after using the model. *Synthesis use* is the ratio of instances that were solved with synthesis. *t* is the time after which solving of an instance would be restarted and rerun with synthesis. All values are averaged over all test parts during cross-validation; for calculating *Model change*, weighted average and standard deviation are used, with the weights being total solving time without synthesis of the corresponding test part.

Potential improvement averaged across cross-validation splits is  $-20.3 \pm 4.2\%$  for *st* synthesis and  $-26.9 \pm 2.8\%$  for *6-LUT-1x* synthesis. These values are weighted using the same method as *Model change* (see previous paragraph).

With both syntheses, all tested approaches achieved negligible or no improvements with high standard deviations. At best  $-1.6 \pm 5.0\%$  change in total solving time was achieved using MINISAT with *6-LUT-1x* synthesis.

Based on the tested models, it seems that using synthesis while solving this type of SAT instances is not beneficial.

## 3.4 ATPG instances

This section describes data and results regarding the use of synthesis while solving SAT instances generated when solving ATPG. Results of these instances are in detail described in Section 2.4.

<sup>2</sup><https://github.com/bprovanbessell/SATfeatPy>

■ **Table 3.1** Practical use of synthesis: results of ten best models by a decrease in solving time on MinCirc instances using the MINISAT solver with *st* synthesis

Name	Restarting	Targets type	Preprocessing	Model change	Same or better	Synthesis use	t
Random Forest	yes	binary	min-max	$-0.1 \pm 0.3\%$	$99.9 \pm 0.1\%$	$0.1 \pm 0.2\%$	$33.4 \pm 74.6$
Random Forest	yes	binary	none	$-0.1 \pm 0.3\%$	$99.9 \pm 0.1\%$	$0.1 \pm 0.2\%$	$33.4 \pm 74.6$
Random Forest	yes	binary	standard	$-0.1 \pm 0.3\%$	$99.9 \pm 0.1\%$	$0.1 \pm 0.2\%$	$33.4 \pm 74.6$
Gradient Boosting	no	continuous	min-max	$-0.03\%$	$99.94\%$	$0.2 \pm 0.2\%$	-
Gradient Boosting	no	continuous	standard	$-0.03\%$	$99.94\%$	$0.2 \pm 0.2\%$	-
Gradient Boosting	no	continuous	none	$-0.03\%$	$99.94\%$	$0.2 \pm 0.2\%$	-
Gradient Boosting	yes	continuous	standard	$-0.02\%$	$99.90\%$	$0.2 \pm 0.1\%$	$1.0 \pm 2.2$
Gradient Boosting	yes	continuous	min-max	$-0.02\%$	$99.90\%$	$0.2 \pm 0.1\%$	$1.0 \pm 2.2$
Gradient Boosting	yes	continuous	none	$-0.02\%$	$99.90\%$	$0.2 \pm 0.1\%$	$1.0 \pm 2.2$
Gradient Boosting	yes	binary	none	$-0.01\%$	$99.94\%$	$0.1 \pm 0.1\%$	0.00

■ **Table 3.2** Practical use of synthesis: results of ten best models by a decrease in solving time on MinCirc instances using the MINISAT solver with *6-LUT-1x* synthesis

Name	Restarting	Targets type	Preprocessing	Model change	Same or better	Synthesis use	t
K-Nearest Neighbours	no	continuous	min-max	$-1.6 \pm 5.0\%$	$99.8 \pm 0.3\%$	$0.4 \pm 0.6\%$	-
K-Nearest Neighbours	no	continuous	standard	$-1.6 \pm 5.0\%$	$99.8 \pm 0.3\%$	$0.4 \pm 0.6\%$	-
K-Nearest Neighbours	no	binary	min-max	$-1.6 \pm 5.0\%$	$99.8 \pm 0.3\%$	$0.4 \pm 0.6\%$	-
K-Nearest Neighbours	yes	binary	min-max	$-0.3 \pm 0.7\%$	$99.98\%$	$0.1 \pm 0.1\%$	$496.4 \pm 801.2$
Random Forest	no	binary	none	0.00%	100.00%	0.00%	-
Random Forest	yes	binary	none	0.00%	100.00%	0.00%	0.00
Random Forest	no	binary	min-max	0.00%	100.00%	0.00%	-
Random Forest	yes	binary	min-max	0.00%	100.00%	0.00%	0.00
Random Forest	no	binary	standard	0.00%	100.00%	0.00%	-
Random Forest	yes	binary	standard	0.00%	100.00%	0.00%	0.00

### 3.4.1 Data

Fourteen features of each circuit are used to predict whether to use synthesis, these are (1) # of primary inputs, (2) # of primary outputs, (3) # of gates, (4) # of edges, (5) the total number of terms, (6) the total number of literals, (7) # of combinational levels, (8) # of connected components, (9) # of xors, (10) # of gate equivalents \* 2, (11) the maximum number of fanins, (12) the average number of fanins, (13) the maximum number of fanouts, (14) the average number of fanouts.

Each model was trained on unpreprocessed data and data preprocessed using min-max normalization and standardization.

Only the results of *st* and *st-re2-1x* syntheses are used since they have better results than other syntheses. The most promising is *st* synthesis, which achieved 19.7% speedup if used always. This sets a baseline improvement that we can attempt to surpass.

### 3.4.2 Results

The results of the ten best ML models ordered by the total time change with *st* synthesis are in Table 3.3 and with *st-re2-1x* are in Table 3.4.

*Model change* is the change in solving time resulting from the use of the model

compared to the total original time. *Same or better* is the ratio of instances whose solving time was the same or better after using the model. *Synthesis use* is the ratio of instances that were solved with synthesis. *t* is the time after which solving of an instance would be restarted and rerun with synthesis. All values are averaged over all test parts during cross-validation; for calculating *Model change*, weighted average and standard deviation are used, with the weights being total solving time without synthesis of the corresponding test part.

Potential improvement of *st* synthesis is  $-32.0 \pm 27.0\%$  and of *st-re2-1x* synthesis  $-28.4 \pm 29.7\%$ . Change in solving time if synthesis was used always is  $-19.7 \pm 31.2\%$  and  $2.7 \pm 39.3\%$  for *st* and *st-re2-1x* syntheses, respectively. These values are weighted using the same method as *Model change* (see previous paragraph).

Contrary to MinCirc instances, achieved improvements are with both syntheses much larger. With *st-re2-1x* synthesis, the best model (k-nearest neighbors) achieved  $-14.8 \pm 18.7\%$  change, more than half of the potential improvement. In this case, the synthesis was used in around 4.6% of cases. Almost all instances ( $98.6 \pm 0.9\%$ ) were solved in the same or faster time.

Results with *st* synthesis are even better – the best model (Gaussian naive Bayes with standardization / min-max normalization) achieved  $-26.3 \pm 29.2\%$  change, more than three-quarters of potential improvement. The use of synthesis is also higher at around 12.7%. However, solving of more instances was slowed down compared to *st-re2-1x* synthesis, as in the same or faster time were solved “only”  $93.9 \pm 2.4\%$  of instances.

In both cases, the best models share their place with other models that differ in the use of restarting and/or preprocessing. Interestingly, the best models are all classifiers, so the use of real value to represent how sure is the use or not use of synthesis, does not seem to help.

Slightly confusing is the 6% difference in results of the Gaussian naive Bayes classifier with different forms of preprocessing. Mathematically, neither standardization nor min-max normalization should influence this model. Multiplying each value of a feature by a constant<sup>3</sup> or adding a constant changes the calculation of all model coefficients in the same way, and thus, the calculated probabilities of different classes should maintain the same order. Different results are most likely caused by the imprecision of floating point numbers and/or variance smoothing. For better numerical stability, the calculated variance of each feature and class usually is increased by a small constant (e.g.,  $10^{-9}$ ) multiplied by the maximum of calculated variances. If the variances have significantly different sizes (which is true for unprocessed data), using the smoothing can change some of the calculated variances more than is desirable.

---

<sup>3</sup>min, max, mean, and var of each feature can be viewed as a constant in this context

■ **Table 3.3** Practical use of synthesis: results of ten best models by a decrease in solving time on ATPG instances with *st* synthesis

Name	Restarting	Targets type	Preprocessing	Model improvement	Same or better	Synthesis use	t
Gaussian NB	no	binary	standard	$-26.3 \pm 29.2\%$	$93.9 \pm 2.4\%$	$12.7 \pm 3.2\%$	-
Gaussian NB	no	binary	min-max	$-26.3 \pm 29.2\%$	$93.9 \pm 2.4\%$	$12.7 \pm 3.2\%$	-
Gaussian NB	yes	binary	standard	$-26.3 \pm 29.2\%$	$93.9 \pm 2.4\%$	$12.7 \pm 3.2\%$	0.03
Gaussian NB	yes	binary	min-max	$-26.3 \pm 29.2\%$	$93.9 \pm 2.4\%$	$12.7 \pm 3.2\%$	0.03
Restarting	yes	-	none	$-20.2 \pm 31.0\%$	$62.4 \pm 2.5\%$	$46.5 \pm 5.4\%$	$2.1 \pm 0.5$
Gaussian NB	no	binary	none	$-20.1 \pm 19.6\%$	$94.1 \pm 2.4\%$	$12.1 \pm 3.1\%$	-
Gaussian NB	yes	binary	none	$-20.1 \pm 19.6\%$	$94.1 \pm 2.4\%$	$12.1 \pm 3.1\%$	0.03
K-Nearest Neighbours	no	continuous	standard	$-19.0 \pm 11.2\%$	$97.5 \pm 1.8\%$	$9.1 \pm 2.0\%$	-
K-Nearest Neighbours	yes	continuous	standard	$-19.0 \pm 11.2\%$	$97.5 \pm 1.8\%$	$9.1 \pm 2.0\%$	0.03
Gradient Boosting	no	continuous	min-max	$-18.0 \pm 12.5\%$	$98.1 \pm 0.8\%$	$7.1 \pm 1.8\%$	-

■ **Table 3.4** Practical use of synthesis: results of ten best models by a decrease in solving time on ATPG instances with *st-re2-1x* synthesis

Name	Restarting	Targets type	Preprocessing	Model improvement	Same or better	Synthesis use	t
K-Nearest Neighbours	no	binary	min-max	$-14.8 \pm 18.7\%$	$98.6 \pm 0.9\%$	$4.6 \pm 1.3\%$	-
K-Nearest Neighbours	yes	binary	min-max	$-14.8 \pm 18.7\%$	$98.6 \pm 0.9\%$	$4.6 \pm 1.3\%$	0.03
K-Nearest Neighbours	no	continuous	min-max	$-9.2 \pm 9.4\%$	$98.7 \pm 0.8\%$	$4.5 \pm 1.3\%$	-
K-Nearest Neighbours	yes	continuous	min-max	$-9.2 \pm 9.4\%$	$98.7 \pm 0.8\%$	$4.5 \pm 1.3\%$	0.03
Logistic Regression	no	binary	standard	$-9.1 \pm 14.1\%$	$98.6 \pm 1.1\%$	$3.8 \pm 1.1\%$	-
Logistic Regression	yes	binary	standard	$-9.1 \pm 14.1\%$	$98.6 \pm 1.1\%$	$3.8 \pm 1.1\%$	0.03
Logistic Regression	no	binary	min-max	$-8.9 \pm 14.1\%$	$98.4 \pm 0.9\%$	$3.9 \pm 0.9\%$	-
Logistic Regression	yes	binary	min-max	$-8.9 \pm 14.1\%$	$98.4 \pm 0.9\%$	$3.9 \pm 0.9\%$	0.03
Gradient Boosting	no	binary	min-max	$-8.9 \pm 14.0\%$	$98.4 \pm 1.0\%$	$4.9 \pm 1.1\%$	-
Gradient Boosting	yes	binary	min-max	$-8.9 \pm 14.0\%$	$98.4 \pm 1.0\%$	$4.9 \pm 1.1\%$	0.03



## Discussion

This thesis focused on answering whether it is beneficial to use logic synthesis when solving benchmark SAT instances as well as those from optimum circuit generator MinCirc [25] and instances created when solving ATPG. Compared to [2], where the authors measured improvements on all of the industrial SAT instances when solved with synthesis, the results presented in this thesis are not that favorable for the synthesis.

Using synthesis was not beneficial when solving uniform Random-3-SAT and Quasi-group benchmark instances – potential improvements of the fastest solver, MINISAT, were negligible, and other solvers could not achieve faster solving time even with slightly better results of using synthesis. In the case of the Pigeon-hole problem, the combination of *st* synthesis with MINISAT solver led to about a two-thirds decrease in solving time compared to zCHAFF, which without synthesis solved the instances fastest.

When solving SAT instances from MinCirc, the ratio of instances sped up when preprocessed by synthesis was usually fairly low, with the potential improvements that could be achieved only slightly higher. For MINISAT, which was the fastest solver, the highest ratio of improved instances was 10.4% (*st* synthesis) and the largest potential improvement 20.8% (*6-LUT-1x* synthesis). Using synthesis to preprocess all instances would cause a significant slowdown.

It also does not seem that the synthesis is able to make the originally hard-to-solve instances easier, as usually more instances were solved within the time-limit *only without* the synthesis preprocessing than *only with* it.

An interesting result of using synthesis on smaller unsatisfiable instances was that some syntheses transformed those instances into trivial ones (i.e., for variable  $a$ , the CNF would be  $a \wedge \neg a$ ). This was more frequent with more repetitions of the synthesis script.

Results from ATPG instances are more in favor of synthesis. Using *st* synthesis to preprocess all instances would lead to a 19.7% decrease in total solving time. The possible speedup was even higher – at most, the total solving time could be decreased by 32.0%.

Using synthesis to preprocess all instances would lead to a significant slowdown in the case of MinCirc instances, and using synthesis only in some cases would be more beneficial on ATPG instances as well. Multiple approaches to incorporating synthesis into SAT-solving were explored. In the case of MinCirc instances, no method of achieving usable improvement was found. On the other hand, using a Gaussian naive Bayes

classifier to predict whether to use synthesis on ATPG instances was more successful, decreasing the solving time on average by 26.3%.

## Conclusion

This thesis analyzes the possible use of logic synthesis in SAT-solving. Different instance types were processed with multiple ABC synthesis scripts and then solved by three SAT-solvers – MINISAT, zCHAFF, and GLUCOSE.

Maximum potential improvement (i.e., using synthesis only when it improves the overall solving time of an instance) and results of using synthesis on all instances are presented. Simpler syntheses usually achieved higher potential improvement, while the more complicated ones could “solve” more unsatisfiable instances (transform them into trivial ones) on their own. The improvement rate was usually low, as syntheses sped up usually only around 10%–30% of instances.

From the benchmark instances, only the Pigeon-hole problem instances were solved faster by a combination of the solver and synthesis than by the solver alone. This problem is, however, only a benchmark and has no practical use. When solving practical instances created during optimum circuit generation, potential improvements achieved with synthesis were around 20% with the fastest solver (MINISAT); however, no reliable method to take advantage of them was found. Lastly, in the case of ATPG instances, *st* synthesis script could achieve a speed-up slightly below 20% if used on all instances. Using a Gaussian naive Bayes classifier to predict whether to use the synthesis led to an average decrease in solving time by 26.3%.

To conclude, using synthesis to preprocess MinCirc instances does not seem to be beneficial, while using *st* synthesis for preprocessing of SAT instances generated during ATPG solving can lead to about a quarter shorter solving time if used only on instances selected using Gaussian naive Bayes classifier.

## Acknowledgment

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

# Bibliography

1. DRECHSLER, Rolf. Using Synthesis Techniques in SAT Solvers. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. 2004.
2. EEN, Niklas; MISHCHENKO, Alan; SÖRENSSON, Niklas. Applying Logic Synthesis for Speeding Up SAT. In: MARQUES-SILVA, João; SAKALLAH, Karem A. (eds.). *Theory and Applications of Satisfiability Testing – SAT 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 272–286.
3. COOK, Stephen A. The complexity of theorem-proving procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. STOC '71. ISBN 9781450374644. Available from DOI: 10.1145/800157.805047.
4. MAHAJAN, Yogesh S.; FU, Zhaohui; MALIK, Sharad. Zchaff2004: An Efficient SAT Solver. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2004.
5. EÉN, Niklas; SÖRENSSON, Niklas. An Extensible SAT-solver. In: GIUNCHIGLIA, Enrico; TACCHELLA, Armando (eds.). *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, 2004, vol. 2919, pp. 502–518. Lecture Notes in Computer Science. ISBN 978-3-540-20851-8. Available from DOI: 10.1007/978-3-540-24605-3\_37.
6. AUDEMARD, Gilles; SIMON, Laurent. On the Glucose SAT Solver. *International Journal on Artificial Intelligence Tools*. 2018, vol. 27, no. 01. Available from DOI: 10.1142/S0218213018400018.
7. KAUTZ, Henry; SABHARWAL, Ashish; SELMAN, Bart. Incomplete Algorithms. In: BIERE, Armin; HEULE, Marijn; MAAREN, Hans van; WALSCH, Toby (eds.). *Handbook of Satisfiability*. IOS Press, 2008. ISBN 978-1-60750-376-7.
8. HOOS, Holger. *SATLIB - Benchmark Problems* [online]. 2000. [visited on 2024-04-18]. Available from: <https://www.cs.ubc.ca/~hoos/SATLIB/benchn.html>.
9. TSEITIN, G.S. On the Complexity of Derivation in Propositional Calculus. In: SIEKMANN, JorgH.; WRIGHTSON, Graham (eds.). *Automation of Reasoning*. Springer Berlin Heidelberg, 1983, pp. 466–483. Symbolic Computation. ISBN 978-3-642-81957-5.

10. OLIVERAS, Albert; RODRÍGUEZ-CARBONELL, Enric. *Propositional Logic, Combinatorial Problem Solving (CPS)* [online]. 2023. [visited on 2024-04-18]. Available from: <https://www.cs.upc.edu/~erodri/webpage/cps/theory/sat/prop-logic/slides.pdf>.
11. BJÖRK, Magnus. Successful SAT Encoding Techniques. *Journal on Satisfiability, Boolean Modeling and Computation*. 2009, vol. 7, pp. 189–201. Available from DOI: 10.3233/SAT190085.
12. HOOS, Holger. *SAT instances of the Pigeon Hole Problem* [online]. [visited on 2024-04-18]. Available from: <https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/DIMACS/PHOLE/descr.html>.
13. MARQUES-SILVA, Joao. Practical applications of Boolean Satisfiability. In: *2008 9th International Workshop on Discrete Event Systems*. 2008, pp. 74–80. Available from DOI: 10.1109/WODES.2008.4605925.
14. DAVIS, Martin; PUTNAM, Hilary. A Computing Procedure for Quantification Theory. *J. ACM*. 1960, vol. 7, no. 3, pp. 201–215. ISSN 0004-5411. Available from DOI: 10.1145/321033.321034.
15. DAVIS, Martin; LOGEMANN, George; LOVELAND, Donald. A machine program for theorem-proving. *Commun. ACM*. 1962, vol. 5, no. 7, pp. 394–397. ISSN 0001-0782. Available from DOI: 10.1145/368273.368557.
16. ZHANG, Lintao; MALIK, Sharad. The Quest for Efficient Boolean Satisfiability Solvers. In: *Proceedings of the 14th International Conference on Computer Aided Verification*. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 17–36. CAV '02. ISBN 3540439978.
17. SCHIMPF, Albert. *Modern SAT Solvers*. 2014. Tech. rep. Technical Report. TU Kaiserslautern.
18. MARQUES SILVA, J.P.; SAKALLAH, K.A. GRASP-A new search algorithm for satisfiability. In: *Proceedings of International Conference on Computer Aided Design*. 1996, pp. 220–227. Available from DOI: 10.1109/ICCAD.1996.569607.
19. *zChaff* [online]. Princeton University [visited on 2024-04-17]. Available from: <https://www.princeton.edu/~chaff/zchaff.html>.
20. EÉN, Niklas; SÖRENSON, Niklas. *MiniSat* [online]. [visited on 2024-04-17]. Available from: <http://minisat.se/MiniSat.html>.
21. HARRIS Sarah, L.; HARRIS, David. 4 - Hardware Description Languages. In: HARRIS Sarah, L.; HARRIS, David (eds.). *Digital Design and Computer Architecture*. Kaufmann, Morgan, 2022, pp. 170–235. ISBN 978-0-12-820064-3. Available from DOI: <https://doi.org/10.1016/B978-0-12-820064-3.00004-0>.
22. MISHCHENKO, A et al. *ABC: A system for sequential synthesis and verification* [online]. 2012. [visited on 2024-04-18]. Available from: <http://www.eecs.berkeley.edu/%5C~%7B%7Dalanmi/abc>.
23. BRAYTON, Robert; MISHCHENKO, Alan. ABC: An Academic Industrial-strength Verification Tool. In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*. Edinburgh, UK: Springer-Verlag, 2010, pp. 24–40. ISBN 978-3-642-14295-6. Available from DOI: 10.1007/978-3-642-14295-6\_5.

24. BRAYTON Robert, K.; MISHCHENKO, Alan; CHATTERJEE, Satrajit. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: *43rd ACM/IEEE Design Automation Conference*. San Francisco, CA, USA: ACM, 2006, pp. 532–535.
25. FIŠER, Petr; HÁLEČEK, Ivo; SCHMIDT, Jan. SAT-Based Generation of Optimum Function Implementations with XOR Gates. In: *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, pp. 163–170. Available from DOI: 10.1109/DSD.2017.74.
26. LARRABEE, T. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1992, vol. 11, no. 1, pp. 4–15.
27. LIBOVICKÝ, Jindřich; STRAKA, Milan. *Introduction to Machine Learning* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~courses/npfl129/2324/slides.pdf/npfl129-2324-01.pdf>.
28. SCIKIT-LEARN DEVELOPERS. *Cross-validation: evaluating estimator performance* [online]. 2024. [visited on 2024-05-08]. Available from: [https://scikit-learn.org/stable/modules/cross\\_validation.html](https://scikit-learn.org/stable/modules/cross_validation.html).
29. SCIKIT-LEARN DEVELOPERS. *Nested versus non-nested cross-validation* [online]. 2024. [visited on 2024-05-08]. Available from: [https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_nested\\_cross\\_validation\\_iris.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html).
30. LIBOVICKÝ, Jindřich; STRAKA, Milan. *Linear Regression II, SGD* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~straka/courses/npfl129/2324/slides.pdf/npfl129-2324-02.pdf>.
31. LIBOVICKÝ, Jindřich; STRAKA, Milan. *Perceptron and Logistic Regression* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~courses/npfl129/2324/slides.pdf/npfl129-2324-03.pdf>.
32. LIBOVICKÝ, Jindřich; STRAKA, Milan. *k-NN, Naive Bayes* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~courses/npfl129/2324/slides.pdf/npfl129-2324-08.pdf>.
33. LIBOVICKÝ, Jindřich; STRAKA, Milan. *Decision Trees, Random Forests* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~courses/npfl129/2324/slides.pdf/npfl129-2324-09.pdf>.
34. LIBOVICKÝ, Jindřich; STRAKA, Milan. *Correlation, Model Combination* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~courses/npfl129/2324/slides.pdf/npfl129-2324-08.pdf>.
35. LIBOVICKÝ, Jindřich; STRAKA, Milan. *Gradient Boosted Decision Trees* [online]. 2023. [visited on 2024-05-08]. Available from: <https://ufal.mff.cuni.cz/~courses/npfl129/2324/slides.pdf/npfl129-2324-10.pdf>.
36. SENTOVICH, E.M.; SINGH, K.J.; LAVAGNO, L.; MOON, C.; MURGAI, R.; SALDANHA, A.; SAVOJ, H.; STEPHAN, P.R.; BRAYTON, Robert K.; SANGIOVANNI-VINCENTELLI, Alberto L. *SIS: A System for Sequential Circuit Synthesis*. 1992. Tech. rep., UCB/ERL M92/41. EECS Department, University of California, Berkeley.

37. HOOS, H.; STUTZLE, T. SATLIB: An Online Resource for Research on SAT. In: *SAT2000*. IOS Press, 2000, pp. 283–292.
38. Berkeley Logic Interchange Format (BLIF). In: University of California, Berkeley, 1992.
39. SELMAN, Bart. Stochastic Search and Phase Transitions: AI Meets Physics. In: *International Joint Conference on Artificial Intelligence*. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, vol. 1, pp. 998–1002.
40. FIŠER, Petr. *MinCirc: Optimum circuits generator* [online]. [visited on 2024-04-29]. Available from: <https://ddd.fit.cvut.cz/www/prj/MinCirc/index.php>.
41. FIŠER, Petr; SCHMIDT, Jan. A Comprehensive Set of Logic Synthesis and Optimization Examples. In: *12th. Int. Workshop on Boolean Problems (IWSBP)*. Freiberg, Germany, 2016, pp. 151–158. Available also from: <https://ddd.fit.cvut.cz/www/prj/Benchmarks/>.
42. XU, L.; HUTTER, F.; HOOS, H. H.; LEYTON-BROWN, K. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*. 2008, vol. 32, pp. 565–606. ISSN 1076-9757. Available from DOI: 10.1613/jair.2490.



# Concents of the attachment

readme.md.....	description of the attachment contents
experiment-results .....	results of the experiments
├─ readme.md.....	description of the result files
src	
├─ metacentrum-scripts .....	scripts used for running experiments
├─ environment.yml .....	virtual environment specifications
├─ practical-use-of-synthesis.ipynb .....	Jupyter notebook for evaluating possibilities of using synthesis
├─ table-gen.ipynb .....	Jupyter notebook for generating tables used in this thesis
thesis	
├─ thesis.pdf.....	thesis in PDF format
├─ thesis.zip.....	source files of the thesis text in $\text{\LaTeX}$