UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Satune: Synthesizing Efficient SAT Encoders

Permalink
https://escholarship.org/uc/item/5131m4f5

Author
Gorjiara, Hamed

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Satune: Synthesizing Efficient SAT Encoders

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Hamed Gorjiara

Thesis Committee:
Professor Brian Demsky, Chair
Professor Harry Xu
Professor Nader Bagherzadeh

2019

© 2019 Hamed Gorjiara

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

ABSTRACT OF THE THESIS **viii**

1 Introduction **1**
1.1 Introduction to SATUNE 1
1.2 Challenges 3
1.3 State of The Art 4
1.4 SATUNE 4
1.5 Summary of Results 6

2 Background in Boolean Satisfiability Problem **7**
2.1 Constraint Satisfaction Problem (CSP) 7
2.2 Boolean Satisfiability Problem 8
2.3 Propositional Logic and CNF 8
2.4 CNF Example 9

3 Background in SAT Solver **11**
3.1 Preliminary 11
3.2 The DPLL Algorithm 12
3.3 The CDCL Algorithm 13
3.4 Heuristics 13
3.5 Example: Solving a Sudoku problem 14

4 Background in SAT Encoding **18**
4.1 Integer Variable Encodings 18
4.2 Order Encodings 20

5 Motivation and Overview **22**
5.1 Motivation 22
5.2 Overview 24

6 Satune Domain Specific Language (DSL) **27**
6.1 Example 29

7 Satune's Candidate Optimizations **30**
7.1 Elimination of Single Polarity Boolean Variables 31
7.2 Optimization of Orders 32
7.3 Order Conversion 35
7.4 Integer Variable Domain Reduction 35
7.5 Encoding 35
7.6 Constraint Subgraph 37
7.7 Encoding Graph 37
7.8 Constructing Constraint Subgraphs 39
7.9 Has Value Constraints 39
7.10 Variable Ordering 40
7.11 CNF Generation 40
7.12 Incremental Solving 41
7.13 Tuner Framework 41

8 Evaluation **44**
8.1 JMCR 45
8.2 SyPet 47
8.3 Dirk 48
8.4 Hexiom 50
8.5 Sudoku 51
8.6 Killer Sudoku 52
8.7 Parallel Satune 53
8.8 Using Maple SAT Solver 55

9 Related Work **58**

10 Conclusion **60**
10.1 Future Work 60

Bibliography **62**

LIST OF FIGURES

Page

3.1 Process of solving a problem by using a SAT Solver. 14
3.2 4×4 Sudoku problem. Constraints for empty cells are encoded into CNF. . . 16

5.1 SATUNE Overview 25

6.1 SATUNE Constraint Language Grammar 28
6.2 Example Constraints in SATUNE DSL. 29

7.1 Order constraint decomposition example with ellipsis (...) indicating omitted non-order constraints. 32
8.1 SATUNE's test set results of four-fold cross validation are compared with the execution time of Z3(the baseline), SMTRat, and MathSAT for each JMCR problem 46
8.2 SATUNE's test set results of four-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each SyPet problem 47
8.3 SATUNE's test set results of two-fold cross-validation are compared with the execution time of Z3(the baseline), SMTRat, MathSAT for each Dirk problem 49
8.4 SATUNE's test set results of three-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Hexiom problem 50
8.5 SATUNE's test set results of three-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Sudoku problem 51
8.6 SATUNE's test set results of three-fold cross-validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Killer Sudoku problem 52
8.7 Comparing the baseline encoding with SATUNE and Parallel SATUNE for Sudoku 54
8.8 Comparing the baseline encoding with SATUNE and Parallel SATUNE for Killer Sudoku 54
8.9 Comparing the baseline encoding with SATUNE and Parallel SATUNE for Hexiom 55
8.10 Comparing the baseline encoding with SATUNE under two different solvers, Maple and Glucose, for Hexiom 56

8.11 Comparing the baseline encoding with SATUNE under two different solvers, Maple and Glucose, for Sudoku 56

LIST OF TABLES

Page

1.1 Performance of different encodings of a randomly generated total order constraint. . . . 2

8.1 Arithmetic means of SAT solving time for different partitions of the four-fold cross-validation on JMCR . . . 46
8.2 Arithmetic means of SAT solving time for different partitions of the four-fold cross-validation on SyPet . . . 48
8.3 Arithmetic means of SAT solving time for different partitions of the four-fold cross-validation on Dirk . . . 49
8.4 Arithmetic means of SAT solving time for different partitions of the three-fold cross-validation on Hexiom . . . 50
8.5 Arithmetic means of SAT solving time for different partitions of the three-fold cross-validation on Sudoku . . . 52
8.6 Arithmetic means of SAT solving time for different partitions of the three-fold cross-validation on Killer Sudoku . . . 53

ACKNOWLEDGMENTS

I want to express my appreciation to my advisor, Brian Demsky, for his guidance, support, and encouragement throughout my graduate journey. The completion of this dissertation was not practical without his advice.

Also, I wanted to thank other members of my thesis committee, Harry Xu and Nader Bagherzadeh for reading the draft of this dissertation and providing valuable feedback for improving it.

Last, but not least, I would like to express my deepest gratitude and love to my family who have encouraged me in my entire life to follow my dreams and supported me to reach my professional goals.

This work is supported by NSF under the grants OAC-1740210, CNS-1703598, SHF-1217854, and CCF-1319786.

ABSTRACT OF THE THESIS

SATUNE: Synthesizing Efficient SAT Encoders

By

Hamed Gorjiara

Master of Science in Computer Engineering

University of California, Irvine, 2019

Professor Brian Demsky, Chair

Modern SAT solvers are extremely efficient at solving boolean satisfiability problems, enabling a wide spectrum of techniques for checking, verifying, and validating real-world programs. What remains challenging, though, is how to encode a domain problem (*e.g.*, model checking) into a SAT formula because the same problem can have multiple distinct encodings, which can yield performance results that are orders-of-magnitude apart, regardless of the underlying solvers used. We develop SATUNE, a tool that can automatically synthesize SAT encoders for different problem domains. SATUNE employs a DSL that allows developers to express domain problems at a high level and a search algorithm that can effectively find efficient solutions. The search process is guided by observations made over example encodings and their performance for the domain and hence SATUNE can quickly synthesize a high-performance encoder by incorporating patterns from examples that yield good performance. A thorough evaluation with JMCR, SyPet, Dirk, Hexiom, Sudoku, and KillerSudoku demonstrates that SATUNE can easily synthesize high-performance encoders for different domains including model checking, synthesis, and games. These encodings generate constraint problems that are often several orders of magnitude faster to solve than the original encodings used by the tools.

Chapter 1

Introduction

1.1 Introduction to Satune

Modern software analysis — from path-sensitive analysis [77, 29, 63, 83], through symbolic execution [39, 62, 17], to verification and model checking [28, 33, 42, 16, 27, 64] — relies heavily on constraint solving. Analyses are formulated into constraint problems that are subsequently fed to a constraint solver; a significant portion of the computation is done by the solver that uses a search-based algorithm to determine the satisfiability of the input constraints. The past decade has seen a variety of constraint solvers used in software analysis techniques, including SAT, SMT, MaxSAT, or model counting, but under the hood of all of the advanced solvers is the boolean satisfiability problem (SAT), which has been extensively studied for about five decades.

A SAT constraint is often encoded into a *conjunctive normal form (CNF)*, which is a conjunction (and) of clauses and each clause is a disjunction (or) of literals. Each literal is either a propositional variable (a) or the negation of a variable ($!b$). A SAT solver attempts to assign true/false values to boolean variables in the constraint in a way so that the entire

formula can evaluate to true. Solving a constraint requires exploring a huge search space. To improve efficiency, a great number of optimizations [51, 25, 24, 78, 50, 7, 8] have been proposed and implemented in the past to effectively prune the search space.

While modern SAT solvers are often efficient, their performance is highly dependent on the encoding of a constraint. There are often many different ways to encode a problem domain into SAT, and not all of them yield good results. Often times choices that initially appear to be a good turn out not to be the best choices. It is typically labor-intensive to explore all of the different options for encoding. The best encoding choice can even be hard to predict for people who are intimately familiar with the algorithms behind the SAT solvers. For example, the best choice often depends on low-level details of how SAT solvers operate and how these low-level details interact with the structure of the given constraint problem. In many cases, the best encoding also depends on what parts of the constraint problem turn out to be difficult, how the encodings interact with the constraints, etc.

The difficulty of finding good encodings is well-known. An article [12] that interviews several SAT experts state *"the common points picked up during the different interviews is that the encoding does have a big impact on the efficiency of the SAT solver, that finding a good encoding takes much effort, and that encoding quality does not depend much on easily measured properties like size or number of variables. The interviewees usually suggest starting with a simple encoding which is iteratively improved."*

Total Order Encoding	Solving Time(s)	Total Time (s)
Pairwise Encoding	1.17	1.46
Inequality Encoding using Binary	0.01	0.11
Inequality Encoding using One Hot	942.26	1038.75
Inequality Encoding using Unary	0.44	0.59

Table 1.1: Performance of different encodings of a randomly generated total order constraint.

To understand the potential impact of encoding choice, we evaluated the performance of several different encodings for total orders on randomly generated order constraint problems. Total orders are commonly used in constraint-based models for checking multi-threaded

programs—CheckFence [16], SATCheck [27], and MCR [42] which all encode total orders into constraint problems. Two common strategies have been used for encoding order constraints: a pairwise encoding that allocates a variable for each pair of items in the total order that encodes their relative order, and a translation into inequality constraints over variables. The latter requires SAT encoding of the values of these variables, for which three approaches have been proposed: Binary, One Hot, and Unary. Details of these approaches are discussed in Section 4. Table 1.1 summarizes the performance results of these encodings. As shown, the choice of encoding is clearly important—the times between the best and worst choices are four-order-of-magnitude apart.

1.2 Challenges

Determining the right encoding for a domain problem is challenging for the following two major aspects. First, *the relative efficiency of different encoding strategies changes when the SAT formula grows.* For example, although the pairwise encoding strategy, which is used in CheckFence [16] and SATCheck [27], is thought to be more performant than inequality-based encoding in general cases, our experiments show that inequality-based encoding of total orders (generated by model checkers) outperforms their pairwise encoding *by an order of magnitude.* Developers understanding of these different strategies are often based on microbenchmarks. However, when small constraints are integrated into a large satisfiability formula, the relative efficiency of these encodings can change dramatically.

Second, *the relative efficiency of different encoding strategies changes across domains.* There are many factors that go into the efficiency of an encoding. For example, unit propagation is known to be important to optimize for, but constraints are not always amenable to unit propagation. For example, unit propagation is not very useful in the case of a *not-equals* constraint on integer variables. For *equals* constraints on integer variables, there is a trade

off between optimizing for encoding size and for propagation. Understanding these encoding trade offs is typically a tedious and labor-intensive process, requiring extensive experiments with different encoding techniques and domain problems.

1.3 State of The Art

There exists a large body of work [67, 78, 59, 66] on optimizing performance for SAT solvers. Most of these optimizations focus on low-level formula rewrites [45] or autotuning of a set of candidate rewrites [78, 59], assuming that encoding of a domain problem into a formula is done. However, as shown above, encoding can have a huge impact on performance and, hence, opportunities are rather limited if a tool takes an encoded formula as a starting point for optimization. Our major insight in this paper is that *if we shift our focus from tuning the process of solving an encoded formula to tuning the encoding process itself, massive opportunities exist and large gains are possible!*

1.4 Satune

Based on this insight, we developed SATUNE, a novel approach that can synthesize *high-performance, domain-specific SAT encoders.* SATUNE focuses on encoding optimization, which is independent of constraint solving — after a constraint is encoded, the developer can use any backend solver to solve the constraint. Traditionally, developers of analysis tools that incorporate SAT solvers would have to manually decide which encoding strategies (based on prior knowledge and/or their experience) to use to translate their problem into SAT and then manually write code to implement this encoding. This is a daunting task, which is tedious, time-consuming, and labor-intensive, and often ends up with suboptimal encodings and unsatisfactory performance.

Satune allows the developer to express a domain problem with a novel domain-specific language (§6), which provides a means for the developer to specify domain-related constraints while abstracting away low-level SAT-related details. Given a problem description provided by the developer using the DSL, Satune runs a simulated annealing-based optimization process to find the best encoding that respects the constraints specified in the DSL description.

Note that the problem information expressed in the DSL is critical for Satune to effectively prune the search space. To further improve efficiency, Satune implements a range of sophisticated analyses (§7) for globally optimizing encoding strategies. For some problems, these analyses are worthwhile as they expose unseen opportunities; in other cases, they do not lead to additional efficiency. To determine when these analyses are beneficial, Satune also employs a tuning framework to learn which analyses are worth performing for a given problem domain.

While much effort has been made to improve the efficiency of the optimization process, applying it on every single problem instance will still incur large overhead, defeating the purpose of optimization. The good news is that we found *the best encoding strategy often holds across problem instances in a given domain* (§5.2). As such, Satune applies this optimization process on a small number of problems to learn a high-performance encoding strategy that can be subsequently used, without incurring any overhead, to encode other problems in the same domain. The domain constraints expressed in the DSL enable Satune to generalize the encoding it learns from a set of examples to new problems. As such, the optimization effort only has a one-time cost that can be effectively amortized across future solving of similar problems in the same domain.

1.5 Summary of Results

We have evaluated SATUNE on a set of real-world software analysis and game applications. Our results show that the SATUNE-synthesized encodings outperform those hand developed by an overall geometric mean of 24× for our benchmarks.

Chapter 2

Background in Boolean Satisfiability Problem

This chapter introduces the SAT problem which is known to decide the values for boolean variables in ways that satisfy the propositional formula.

2.1 Constraint Satisfaction Problem (CSP)

A constraint satisfaction problem (CSP) P is characterized by a set of variables V where each variable is defined on a d_i. Each domain variable d_i lists the possible distinct values for each corresponding variable v_i. A constraint C is defined on a subset of variables S where S={v_i, v_{i+1}, ..., v_j} shows the actual assignments to each variable. Each constraint CT defines a subset of Cartesian product $d_i \times d_{i+1} \times ... \times d_j$. P is satisfiable if and only if there is an assignment to the variables that can satisfy all the constraints. [74]

$V = \{v_1, v_2, ..., v_n\}$

$D = \{d_1, d_2, ..., d_n\}$

2.2 Boolean Satisfiability Problem

Boolean Satisfaiability problem (SAT) is a special case of CSP where all the variables have the same domain D={True, False}. A problem is SAT if there is at least one assignment to the boolean variables V_i where all the constraints are satisfied and it is UNSAT if even one solution cannot be found.

2.3 Propositional Logic and CNF

Propositional Logic is characterized as defining constraints on Boolean Variables V also known as propositional **atoms**. Constraints are represented as standard logical operations $\vee$ (i.e. disjunction), $\wedge$ (i.e. conjuction) and $\neg$ (i.e. negation). A **literal** l refers to an atom or its negation (in other words, a boolean variable v_i or $\neg v_i$). A **Clause** C is defined as a set of literals with disjunction operation between them $\vee_{l \subseteq V} l$. A propositional formula ϕ is characterized as a set of clauses representing boolean operations between them. The formula ϕ in Conjunctive Normal Form (CNF) is identified as conjunctions of a set of clauses (i.e. distjunction of a set of literals). A clause C is satisfiable if and only if it incorporates at least one literal evaluated to *True*. The formula ϕ has a solution if and only if all of its clauses are satisfiable [72, 41, 11].

In general, clauses can have different structures. 1-SAT structure is characterized as a

clause with only one literal. 2-SAT structure is identified when a clause only incorporates two literals. The structure of clauses is being considered as one of the primary metrics for measuring the complexity of a boolean formula. In the following chapter, we elaborate on how these structures can impact on the formula's solving time.

2.4 CNF Example

Consider the following example to clarify a CNF structure. There are three binary clauses C_1, C_2, and C_3 in the formula ϕ as the following:

$$C_1 = x_1 \lor \neg x_2$$

$$C_2 = x_2 \lor \neg x_3$$

$$C_3 = x_3 \lor \neg x_1$$

$$\phi = C_1 \land C_2 \land C_3$$

The formula ϕ is solvable under the following assignment the the variables:

$$x_1 = \mathit{True}$$

$$x_2 = \mathit{True}$$

$$x_3 = \mathit{True}$$

However, adding the clause C_4 to the original formula yields the problem to be UNSAT (i.e. no assignment can be found for x_1, x_2, and x_3 that can satisfy all the clauses.)

$$C_4 = \neg x_1 \vee \neg x_2 \vee \neg x_3$$

Adding clauses to the original problem and rechecking its satisfiablity under newly added clauses is called *Incremental Solving*. This technique is very common in SAT-based problems in both academia and industry and many real-world problems in various domains are solved by employing this technique [49, 32, 43].

Chapter 3

Background in SAT Solver

This chapter introduces the SAT solvers and the primary search algorithms that they employ to solve a SAT problem. The rest of the chapter focuses on some of the primary heuristics used by modern solvers. Lastly, this chapter presents the process of encoding for a puzzle into a CNF formula.

3.1 Preliminary

A boolean satisfiability (SAT) solver is a decision procedure which determines the solution for a given boolean formula in CNF. During the past decades, SAT solvers remarkably evolved and currently, they are capable of solving complex problems with hundreds of thousands of literals and clauses. SAT is NP-complete and based on the definition, no polynomial solution has been found yet to solve the SAT problem. Although SAT problems are solved in exponential time in the worse case, however, in practice, the real-world problems from various domains such as scheduling, model checking, software verification, dependency managing, AI planning, theorem provers and etc. are solvable in polynomial time by cutting-edge solvers.

These solvers usually have a search algorithm as their backbone and employ various distinct heuristics and machine learning techniques to systematically search the boolean space of the given formula [82]

3.2 The DPLL Algorithm

DPLL is a recursive search algorithm that is used to find a solution for a boolean formula. It starts the backtracking search by picking a literal (called decision variable) and assuming a value. Then, it simplifies the formula and recursively repeats the algorithm until it either discovers a solution or reaches a conflict. In the case of confronting a conflict, DPLL proposes a simple conflict analysis method. The solver maintains a flag for each decision variable showing that both polarities have been discovered. After reaching a conflict, it finds the most recent decision variable where both polarities have not been explored and flips its value and then recursively repeats the algorithm. The backtracking algorithm employs two important characteristics to simplify the formula:

Unit Propagation: If a clause has a 1-SAT structure (only has one literal), that literal is evaluated to have a truth value to satisfy the clause and the value of the literal is propagated to the rest of the formula.

Pure Literal Elimination: If a literal appears with only one polarity in the formula is called it is pure. Pure literals can be replaced with the truth value which satisfies all the clauses where it appeared.

3.3 The CDCL Algorithm

The conflict-driven clause learning (CDCL) is one of the primary search algorithms that are being used in many solvers [5, 31, 54]. In this technique, which is similar to DPLL, SAT solver makes an assumption about a literal and it simplifies the formula until it reaches a conflict. In contrary to DPLL, CDCL performs a sophisticated conflict analysis to fix the wrong branches [65, 81]. The CDCL conflict analysis provides: 1) A learned clause that is added to the formula to avoid searching the same boolean space later 2) The correct decision level that Solver restarts searching from there.

The main difference between DPLL solvers and CDCL solvers is that the solver in the DPLL algorithm exhaustively searches the entire search space. However, in the CDCL technique, by learning a clause and enriching the original formula, it avoids searching the boolean search space that doesn't incorporate the solution. Consequently, CDCL is capable of doing non-chronological jump in the backtracking search.

3.4 Heuristics

Each year, a lot of SAT solvers emerge in the annual SAT competition. The majority of them employ the same backbone search algorithm and they differentiate based on the heuristics that they use for cleaning up the learned clauses, restarting the search, analyzing the conflict, etc.

Restart: Sometimes SAT solver may end up deeply exploring part of the search space that does not contain the solution. By restarting, the solver keeps all the learned information and starts solving the formula from the beginning. Keeping the learned clauses potentially yields the solver to explore different parts of the search space.

Learned Clause Removal: Learned clauses help solvers to prune the boolean search space. However, when SAT solvers cope with an enormous formula, a lot of clauses are learned that cannot fit in the memory. Different solvers leverage different heuristics to identify and keep useful clauses and remove redundant clauses from the formula [4, 6].

3.5 Example: Solving a Sudoku problem

As Figure 3.1 depicts, for solving a problem using a SAT solver, the constraints must be encoded into a CNF formula. The solver takes the formula and either provides the variable assignments to satisfy the formula or confirms no solution exists for the problem. In the case of finding a solution, the assignments to the boolean variables need to be translated back to the original domain to obtain the solution for the problem.

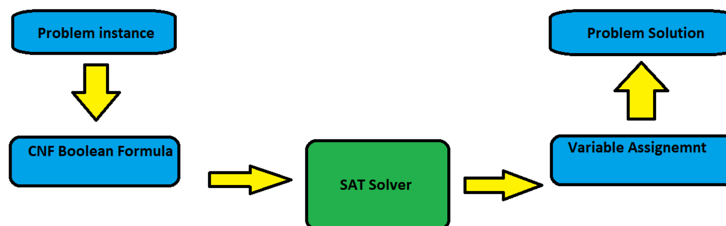

Figure 3.1: Process of solving a problem by using a SAT Solver.

This section elaborates on the process of encoding a popular puzzle, Sudoku, to a CNF boolean formula. Sudoku is a well-known SAT problem and there are many encodings available to convert Sudoku into a CNF. Originally in Sudoku, the user has to assign numbers from 1 to 9 to each cell in the 9×9 table in order to fill out the table thoroughly with respect to the following rules:

1. Only we can use numbers 1 to 9 for each cell.

2. Each number only appears once in each row.

3. Each number only appears once in each column.

4. Numbers should be identical in each block of 3×3.

Figure 3.2 depicts a 4×4 Sudoku problem. As an example, We encode the mentioned rules for the first empty cell, cell **A**, and the same constraints for the rest of empty cells must be added to the formula.

1. We assign four variables for each cell A, B, and C and each variable respectively represents the value of the cell from 1 to 4.

   $$A = \{a_1, a_2, a_3, a_4\}$$

   $$B = \{b_1, b_2, b_3, b_4\}$$

   $$C = \{c_1, c_2, c_3, c_4\}$$

   $$P_1 = a_1 \vee a_2 \vee a_3 \vee a_4$$

   $$P_2 = b_1 \vee b_2 \vee b_3 \vee b_4$$

   $$P_3 = c_1 \vee c_2 \vee c_3 \vee c_4$$

   for example if boolean variable b_2 evaluates to True, it means cell B has a value 2.

2. Constraints for row one:

   $$P_4 = a_2 == False, P_5 = a_3 == False, P_6 = a_4 == False$$

3. Constraints for column one

$$P_7 = a_2 == False$$

$$P_8 = a_1 \Rightarrow \neg b_1, P_9 = a_2 \Rightarrow \neg b_2, P_{10} = a_3 \Rightarrow \neg b_3, P_{11} = a_4 \Rightarrow \neg b_4$$

$$P_{12} = a_1 \Rightarrow \neg c_1, P_{13} = a_2 \Rightarrow \neg c_2, P_{14} = a_3 \Rightarrow \neg c_3, P_{15} = a_4 \Rightarrow \neg c_4$$

4. Constraints for the 2×2 block that contains cell A:

$$P_{16} = a_2 == False, P_{17} = a_3 == False, P_{18} = a_4 == False$$

A	3	2	4
2	4		3
B	1		
C		3	

Figure 3.2: 4×4 Sudoku problem. Constraints for empty cells are encoded into CNF.

After adding all the constraints for each empty cell to the above constraints and provide the CNF formula for the SAT solver, the following solution is found for boolean variables of cell A, B, and C:

$$a_1 = True, a_2 = False, a_3 = False, a_4 = False$$

$$b_1 = False, b_2 = False, b_3 = True, b_4 = False$$

$$c_1 = False, c_2 = False, c_3 = False, c_4 = True$$

which means cell A, B, and C are respectively assigned to the value 1, 3, and 4.

Chapter 4

Background in SAT Encoding

This section provides a gentle introduction to commonly used encoding strategies for two major categories of constraints: (1) constraints on integers drawn from discrete sets and (2) orders on discrete sets. They are both used widely in software analysis tools.

4.1 Integer Variable Encodings

We first discuss several issues that arise in how these encodings interact. To illustrate how integer variables are encoded, consider variable x taken from the set of values $\{0, 1, 3\}$. Such encoding is used widely in SAT-based model checkers. Next, we discuss a few commonly used encoding strategies for integer variables drawn from discrete sets. n denotes the size of a set:

One Hot: The one-hot encoding uses a boolean variable b_i to represent each value v_i that the integer variable may have. It then generates constraints to ensure the variable can only have one value ($\forall i, j \neq i, b_i \Rightarrow \neg b_j$) and that the variable must have some value ($b_0 \vee b_1 \vee .. \vee b_n$).

For our example, the one-hot encoding allocates 3 variables: b_0, b_1, and b_3, representing the fact $x = 0$, $x = 1$, and $x = 3$, respectively. For example, if x turns out to have the value 3, the corresponding variable b_3 would be true. It then generates at-most-one-value constraints: $b_0 \Rightarrow \neg b_1$, $b_0 \Rightarrow \neg b_3$, and $b_1 \Rightarrow \neg b_3$. Next, it generates the following constraint to ensure that x has at least one value: $b_0 \vee b_1 \vee b_3$. One advantage of this encoding is that it works well with the propagation behavior of SAT solvers. For example, when a SAT solver branches on a boolean variable used for one-hot encoding, if the solver has decided the value for the integer variable, it may be able to propagate this decision to other variables. However, the negatives of this encoding are: (1) as shown in the example, it requires a large number of boolean variables, (2) it requires $O(n^2)$ constraints to ensure that the integer variable has only one value, and (3) it requires a constraint to ensure that the integer variable has a value. Solving these many constraints is time-consuming — equality constraints between variables have a $O(n)$ complexity while inequality constraints between integer variables have a $O(n^2)$ complexity.

Unary: The unary encoding uses $n - 1$ boolean variables to encode the value of the integer variable. The idea is that a boolean variable b_i is true if the encoded value is larger than the value v_i from the discrete set. the transition from 1 to 0 encodes the value. For our example, this encoding would generate the variables b_0 and b_1, as well as the clause $b_1 \Rightarrow b_0$. The constraint $x = 0$ is encoded as $\neg b_0 \wedge \neg b_1$, $x = 1$ is encoded as $b_0 \wedge \neg b_1$, and $x = 2$ is encoded as $b_0 \wedge b_1$. The positives of this encoding are: (1) it requires only $O(n)$ constraints to implement, and (2) inequality constraints between integer variables have a $O(n)$ complexity. Its negatives are: (1) it may not work as well as one-hot with the propagation behavior of SAT solvers and (2) it also requires a large number of boolean variables. Equality constraints between integer variables have a $O(n)$ complexity.

Binary Index: The binary index approach encodes, in the *binary format*, an index into the set of discrete values. For our example, this encoding would generate the variables b_0 and b_1,

and the clause $\neg(b_0 \wedge b_1)$ to ensure that index is in range. Under this encoding, $x = 0$ (00) would be encoded as $\neg b_0 \wedge \neg b_1$, $x = 1$ (01) would be encoded as $b_0 \wedge \neg b_1$, and $x = 2$ (10) would be encoded as $\neg b_0 \wedge b_1$. The positives here are: (1) it requires only $O(log(n))$ variables, (2) it does not require any clauses to ensure that the integer variable has only one value, and (3) equality and inequality constraints can be encoded efficiently with $O(log(n))$ complexity. The negative of this encoding is: (1) it may not work well with the propagation of SAT solvers. It sometimes requires inequality constraints to ensure that the integer variable has a value if the size of the discrete set is not a power of two. If the values are not *dense*, (*i.e.*,, there are holes between), the binary index encoding can require additional constraints.

4.2 Order Encodings

Another major category of constraints is on partial and total orders over discrete sets. Similarly, we use n to denote the size of the discrete set.

Pairwise Encoding: Both total and partial orders over sets can be encoded by using boolean variables to represent the order of each pair of elements in the set. In the case of a total order, a single boolean variable is used for each pair (v_i, v_j) — b_{ij} being true indicates that v_i is ordered first and b_{ij} being false indicates that v_j is ordered first. To be consistent with the notations for partial orders (discussed shortly), we use the shorthand b_{ji} to denote a total order where $j > i$; it is simply the negation of variable b_{ij}. In the case of a partial order, a pair of boolean variables is used. For each pair (v_i, v_j), b_{ij} being true indicates that v_i is ordered first and b_{ji} being true indicates that v_j is ordered first. Partial orders must then add the clause $\neg b_{ij} \vee \neg b_{ji}$ to ensure that the encoding cannot order both items first.

Both encodings use the following transitivity constraints: $\forall i, j, k : b_{ij} \wedge b_{jk} \Rightarrow b_{ik}$. The positive for the pairwise encoding is that it works well with the propagation behaviors of

SAT solvers because a single variable corresponds to a client-level predicate. The negatives for this encoding are: (1) it requires $O(n^2)$ boolean variables and (2) $O(n^3)$ constraints.

Inequality-Based Encodings: Total orders can also be encoded as a system of inequalities. Each item v_i in the order is encoded as an integer variable x_i in the range of $[0, n-1]$. We then encode the constraint $v_i - > v_j$ (*i.e.*,, `->` denote ordered-before) with the inequality $x_i < x_j$. The positives of this approach are (1) it requires only $O(nlog(n))$ boolean variables and (2) transitivity constraints come for free. The negatives here are: (1) order constraints become more complicated and (2) it may not work well with SAT solver propagation behaviors.

Chapter 5

Motivation and Overview

5.1 Motivation

To motivate the discussion, let's consider sample constraints from two different domains: Sudoku and a SAT-based model checker. In Sudoku, boxes are filled in with numbers from one to nine and two boxes in the same row, column, or block cannot be assigned the same number. As such, a common constraint in Sudoku is that variable x drawn from $[1, 9]$ and variable y drawn from the set $[1, 9]$ are not equal, *i.e.*,, $\neg(x == y)$. Some of these boxes have been pre-filled, so certain numbers are not possible and additional constraints are needed to rule out such possibilities. For example, if the number 2 is prefilled in a box, then we need $\neg x = 2$ for the row containing the box. Since these are all *not-equal* constraints, it is hard for the SAT solver to perform *unit propagation* (which is a simple technique that can simplify a clause if the clause contains a single literal) if it guesses a SAT variable. As a result, encoding x and y as *binary indices* can be a reasonable choice.

In the domain of concurrency model checking, for instance, model checkers may need a constraint to represent the following assertion: if a load L reads from a store S, then the value

read by L should be the same as the value written by S, that is, $S \xrightarrow{rf} L \Rightarrow L_{\text{value}} = S_{\text{value}}$. In this case, if the SAT solver decides a value for the boolean variable indicates that L reads from the store S, and knows the values of the boolean variables that encode the value of either L or S, it can propagate the value to the other operation (and potentially many more through other constraints). Thus, optimizing for unit propagation is potentially beneficial.

We make two observations on the above examples. First, *constraint characteristics differ across domains*. It is clear to see that while both domains generate equality constraints over integers, the properties of constraints differ significantly. For example, variables in Sudoku often have similar sets of possible values and thus encoding each variable as a binary index into the same set is a reasonable choice. On the contrary, for model checkers [27, 16], program variables may have very different sets of possible values. Encoding every program variable using binary index can generate an excessive number of variables as well as constraints that enforce each variable has a valid value. Clearly, there are many factors in choosing an encoding and different factors may point to different encoding choices. Knowing the relative performance impact of these factors is difficult and can typically only be achieved by labor-intensive experimentation.

The second observation is that *different problems in the same domain often require constraints of similar natures*. For example, the *not-equal* property of Sudoku constraints holds not only for Sudoku, but also for other board games. For program-analysis-related applications, they need constraints to model variable relationships and hence their constraints all share similar properties to the model checking constraints stated above. This observation indicates that the best encoding learned from small examples in a domain can often hold universally in the domain.

5.2 Overview

SATUNE has two phases: an *example-driven learning phase* in which it synthesizes an encoder, as well as a *deployment phase* where it uses the synthesized encoder to encode new problems. To use SATUNE, the developer first needs to modify their application to generate constraints in the SATUNE DSL. Since the SATUNE language is a superset of the languages accepted by typical SAT solvers, this step is straightforward, requiring only minimum user effort. SATUNE requires a set of examples to use to synthesize an encoder — in the case of the Sudoku example, this would be a set of Sudoku puzzles.

SATUNE starts the synthesis process with a set of *seed encoders*. The process uses simulated annealing to explore a space of possible encoders. In each round of simulated annealing, it evaluates the fitness of the current encoders by measuring the time the solver takes to solve the example with the given encoding. SATUNE then mutates these encoders and repeats. For Sudoku, for instance, this process would naturally learn that the one-hot encoding does not result in large performance benefits from unit propagation. This would bias the search process away from the one-hot encoding. It would likely decide to encode constraints using the binary index encoding to minimize both the size of the constraints and the number of variables.

Note that in the model checking example, integer variables are used for two distinct purposes — encoding the read-from relation and encoding variable values. These two different purposes may not have shared the same optimal encoding. SATUNE supports *labeling different use cases* and synthesizes encoders that individually optimize the encodings for the distinct use cases. For the example, the developer may label, with one type, the integer variable that tracks which store the load L reads from, and, with another type, the integer variables that contain values. As such, SATUNE can encode the read-from relation using the one-hot encoding so that when a SAT solver guesses a value used by the one-hot relation,

unit propagation allows it to propagate values accessed by the load and store. However, if the set of possible values is large, SATUNE may use the binary index encoding to minimize the number of variables.

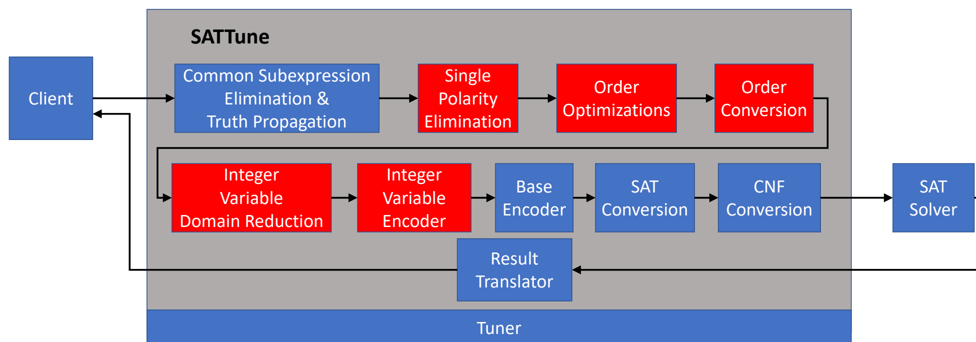

Figure 5.1: SATUNE Overview

Figure 5.1 presents a detailed overview of SATUNE's deployment architecture. The pipeline begins with a set of constraints generated by a client which are specified using the SATUNE constraint language. The SATUNE constraint language is designed around a set of commonly used constructs for which several different encodings are known. The SATUNE language raises the abstraction level the the client uses to specify constraint problems and thus allows SATUNE to tune the generation of low-level SAT encoding of the problem. The interface with SATUNE's clients is via a library-based API for which we have developed bindings in C, C++, and Python. The end of the pipeline encodes these constraints into CNF SAT and then uses an off-the-shelf SAT solver to solve the constraints. SATUNE then translates the SAT solution into a solution to the original client's constraints.

We next discuss the passes in SATUNE's pipeline. The first pass, common subexpression elimination and truth propagation eliminates duplicate expressions and propagates truth values. This pass is always used and is performed on the fly as the client generates constraints.

The passes shown in red target specific optimization and encoding opportunities. These passes perform simplifications to the input that may allow better encodings, optimize order

constraints, convert order constraints into inequalities, and optimize the encoding of integer variables globally. The time taken to run these passes is not justified for all clients and for many clients it would be better to spend the time in the SAT solver. Thus, these passes are under the control of SATUNE sythesizer.

The base encoder provides default encodings for variables that were not encoded by previous passes. Finally, the abstractions are converted into boolean constraints in CNF form. The optimization pass pipeline builds result translators for each variable in the original set of constraints. The result translators translate the value assignment for the boolean variables back to the original client constraints. In some cases, the optimization passes may partially (or even completely) solve the problem. The result translators in those cases simply return the solution that the optimization passes discovered.

Chapter 6

Satune Domain Specific Language (DSL)

We begin by presenting the constraint language Satune takes as input. Figure 6.1 presents the grammar for the language. A constraint problem is given by a *prog* term in the grammar. While we present a textual grammar for purposes of exposition, Satune's implementation actually accepts constraints via a C, C++, or Python native interface.

The constraint DSL incorporates common abstractions for exploring different SAT encodings. The DSL contains the following three state abstractions: variables drawn from discrete sets of integers, orders (both total and partial) over discrete sets, and boolean variables. The constructs in the language are also motivated by the fact that they are used across many tools. For example, total and partial orders are extensively used by analyses of concurrent executions including SATCheck [27], CheckFence [16], MemSAT [70], JMCR [42], RVPredict [44], Dirk [49], CPPMem [9], Nitpick [13]. Variables drawn from discrete sets are used by Alloy [47], Paradox [20], SATCheck [27], CPPMem [9], CheckFence [16], Nitpick [13], Package Managers, etc.

$$
\begin{array}{rcl}
\textit{intlist} & := & \textit{int} \mid \textit{intlist},\ \textit{int} \\
\textit{setdecl} & := & \texttt{set}\ \textit{sname}\ \textit{type}\ \{\textit{intlist}\} \\
\textit{booldecl} & := & \texttt{boolean}\ \textit{bname} \mid \texttt{boolean}\ \textit{bname}\ =\ \textit{bexpr} \\
\textit{orderdecl} & := & \texttt{total}\ \textit{oname}\ \textit{setdecl} \mid \texttt{partial}\ \textit{oname}\ \textit{setdecl} \\
\textit{vdecl} & := & \texttt{var}\ \textit{vname}\ \texttt{in}\ \textit{sname} \mid \texttt{var}\ \textit{vname} = \textit{vexpr} \\
\textit{vexpr} & := & \textit{vname} \mid \textit{int} \mid \textit{sname} : \textit{vexpr} + \textit{vexpr}\ \#\ \textit{bexpr} \mid \\
 & & \textit{sname} : \textit{vexpr} - \textit{vexpr}\ \#\ \textit{bexpr} \mid \\
 & & f(\textit{vexpr},^{*}\textit{vexpr})\ \#\ \textit{bexpr} \\
\textit{bexpr} & := & \textit{vexpr}\ \textit{comp}\ \textit{vexpr} \mid p(\textit{vexpr},^{*}\textit{vexpr})\ \#\ \textit{bexpr} \mid \\
 & & \textit{oname} : \textit{int}{-}{>}\textit{int} \mid \textit{bname} \mid !\textit{bexpr} \mid \\
 & & \textit{bexpr}\ \textit{boolop}\ \textit{bexpr} \\
\textit{boolop} & := & \mid\ \mid \& \mid \Rightarrow \mid \oplus \mid = \\
\textit{comp} & := & = \mid < \mid \leq \mid > \mid \geq \\
\textit{assert} & := & \texttt{assert}(\textit{bexpr}) \\
\textit{prog} & := & \textit{setdecl}^{*}\ \textit{booldecl}^{*}\ \textit{orderdecl}^{*}\ \textit{vdecl}^{*}\ \textit{assert}^{*}
\end{array}
$$

Figure 6.1: SATUNE Constraint Language Grammar

The constraint language supports basic operations on integer expressions: addition, subtraction, and the application of table-defined functions. Both addition and subtraction define a range set of the valid results. It is possible for addition or subtraction to overflow, and this will set the corresponding boolean expression *bexpr* that follows the # to true[1]. Clients can define functions using tables. Each such function declares its range set. While table-driven functions do not overflow, they may accept inputs that do not match any entry in the table. In this case, the corresponding boolean expression is set to true. The constraint language supports two classes of predicates on integer expressions: standard comparison operators as well as table-defined predicates.

The DSL also allows clients to write constraints on both total and partial orders over discrete sets. SATUNE requires the elements of these sets to be integers for convenience of representation, but assigns no meaning to the integer elements. Clients can then use predicates on the order of these elements in boolean expressions.

[1]The SATUNE implementation allows the client to select the directionality of the implication between overflow occurring and the truth value of the overflow boolean expression as iff, $\Rightarrow$, or $\Leftarrow$.

The constraint language also provides clients with boolean variables and standard boolean connectives (not, and, or, xor, iff, and implication) that can be used with any predicate.

```
set RF rfset {1, 2}
set VS1 valueset {100, 101, 102}
set VS2 valueset {100, 101, 102}
set VL valueset {101, 102, 103}
var loadrf in RF
var storeval1 in VS1
var storeval2 in VS2
var loadval in VL
assert((loadrf = 1) => (storeval1 = loadval))
assert((loadrf = 2) => (storeval2 = loadval))
```

Figure 6.2: Example Constraints in SATUNE DSL.

6.1 Example

Figure 6.2 presents an example reads-from constraint with two stores and a load that might be used by a model checker. The keyword `set` declares a set. We declare the set `RF` to contain the set of stores. We also declare sets `VS1`, `VS2`, and `VL` to contain the set of values for two stores and a load. Each set has a type label that SATUNE uses to synthesize encoders that generalize across different problem instances. For example, here we the label `rfset` for the reads-from set and `valueset` for sets of load and store values, because these two different types of sets are different use cases and thus may benefit from different encodings. SATUNE can synthesize different encodings and optimizations for sets with different labels. The keyword `var` declares variables. We declare the variable `loadrf` to be taken from the set of values `RF`. This variable will be used to encode which store a load reads from. After the variable declarations, we declare the constraints to be satisfied. The first assertion declares the constraint that if the load reads from the first store, then the load must have the same value as the first store.

Chapter 7

Satune's Candidate Optimizations

Implementing sophisticated optimizations to enable better SAT encodings can sometimes yield significant performance improvements. For some problem domains, specialized optimizations in SATUNE may enable it to simplify the problem before encoding and thus generate simpler encodings. However, for other domains, the propagation built into the SAT solver will outperform these specialized optimizations. SATUNE implements a wide range of optimizations, many of which may not improve performance for a specific domain. SATUNE uses a tuning framework that learns which set of optimizations to use for a specific domain. For problem domains that do not benefit from the optimization, the tuner will simply disable the optimization and avoid the associated overhead. This section discusses the candidate optimizations available in SATUNE.

We begin by discussing SATUNE's internal representation of the constraints. SATUNE represents a constraint as an *And-Inverter-Graph* (AIG) where nodes are represented as objects and can either be a predicate, a boolean variable, or an AND boolean operation. Edges are encoded as pointers and we steal the lowest bit to record whether the edge is a negation.

As clients use SATUNE's API to specify constraints, SATUNE translates these constraints into

AIGs on Satune's predicates and boolean variables and Satune uses hashing to detect and eliminate redundant expressions. When an expression is asserted as a constraint, Satune propagates its truth value to any expression that it appears in.

Satune does not currently attempt to encode the constraints on the fly as the client generates them. Several of Satune's analysis require complete knowledge of the constraints and thus cannot safely encode the constraints until the client has finished generating them and called Satune's solve procedure.

The polarity of a boolean expression is positive if the expression appears with an even number of negations and negative if the expression appears with an odd number of negations. Polarity is important because an expression e that appears in a given context with a positive polarity can only contribute in that context to satisfying the overall set of constraints by being true. Knowing the polarity of expressions is thus important for several of Satune's analysis as well as Satune's SAT encoding procedures. Thus, Satune computes the polarity of all nodes in its AIG as the first step in its solve procedure. All other transformations in its pipeline maintain this polarity information in the AIG.

7.1 Elimination of Single Polarity Boolean Variables

Satune includes an optimization pass that simplifies constraints by eliminating boolean variables that appear in a single polarity. If a boolean variable only appears in the positive polarity it can be assigned the boolean value true, and if it only appears in the negative polarity it can be assigned to false. This can potentially allow Satune to simplify constraints enabling further optimizations and simplifications.

```
set baseset orderlabel {1, 2, 3, 4}
total exorder baseset
assert(exorder: 1->2 | ...)
assert(exorder: 2->1 | ...)
assert(exorder: 2->3 | ...)
assert(exorder: 3->4 | ...)
assert(exorder: 4->3 | ...)
```

Figure 7.1: Order constraint decomposition example with ellipsis (...) indicating omitted non-order constraints.

7.2 Optimization of Orders

Consider for example, the potential use by a model checker of total orders to model the execution of concurrent code. Such a client might create a set with an item for each step in the execution. It would then enforce intra-thread order (program order) and thread creation and joining by asserting the appropriate order constraints. This straightforward usage case would result in the order being constructed on a larger set than is strictly necessary.

The complexity of encoding orders grows super-linearly with the size of an order. Thus, it can be potentially useful to decompose and simplify constraints on an order into constraints on one or more smaller orders. To illustrate the idea, consider the example shown in Figure 7.1 in which non-order constraints are omitted. Without reasoning about the full set of constraints, we cannot determine the relative ordering of items 1 and 2 or items 3 and 4. But we can determine that item 2 can be safely ordered before item 3 because (1) a constraint on the order of item 2 and 3 appears only in the positive polarity and (2) this selection does not contradict any other order constraints.

We discuss the order optimizations in more detail for total orders. SATUNE also implements a variation of these optimizations for partial orders; we omit details due to space constraints, but they generally involve minor adaptations to the optimizations for total orders. SATUNE constructs an order graph to reason about order constraints. An order graph corresponds to a specific declared order. An order graph contains a vertex v_a for each item a in the order's

set. There is an edge from a vertex v_a to a second vertex v_b if an order predicate `a`$->$`b` appears with positive polarity or an order predicate `b`$->$`a` appears with negative polarity. If SATUNE determines that a specific order predicate must be true, the *mustbetrue* predicate is true for the corresponding edge.

Transitive Must Be True Analysis: SATUNE includes an optional analysis that performs a depth first traversal over the edges in the order graph that satisfy the *mustbetrue* predicate. This traversal allows SATUNE to compute more precise information associated with edges. For order constraints that are implied by transitivity, this analysis marks those edges with the *mustbetrue* predicate. This analysis marks order constraints that would contradict order constraints that must be true (*i.e.*, the source of the edge is reachable from the destination by following only edges with the *mustbetrue* predicate) with the *mustbefalse* predicate.

Local Must Analysis: SATUNE includes an optional analysis that propagates the information it learns about order predicates that must be true to the opposite order predicate. For example, if SATUNE determines that a must be ordered before b, then the predicate `b`$->$`a` must be false.

Vertex Elimination: Consider a vertex in the order graph for which all incoming edges must be true and all outgoing edges must be true. The corresponding item can be eliminated and the constraints replaced with constraints on the order of the sources of the incoming edges relative to the sources of the outgoing edges. SATUNE includes an optional optimization that eliminates such vertices.

Must Edge Pruning: Consider an edge $\langle v_a, v_b \rangle$ for which (1) the edge satisfies the *must-betrue* predicate and (2) either v_a has no other outgoing edges or v_b has no other incoming edges. We can then safely merge the items v_a and v_b without affecting order constraints on other items. SATUNE includes an optional optimization that prunes such edges.

Order Decomposition: This optimization decomposes an order into two or more smaller orders to optimize for the fact that the encoding cost of orders is superlinear. SATUNE runs a strongly connected component analysis on the edges that may (or must) be true. The result of this analysis is a DAG of strongly connected components. Edges between strongly connected components can simply be made true and the corresponding constraints replaced with the appropriate truth value. If a strongly connected component contains more than one vertex, SATUNE generates a new order for the nodes in the strongly connected component.

Partial Orders: SATUNE implements similar optimizations for partial orders. One key difference is that SATUNE can perform strength reduction on partial orders to replace them with total orders. In general, encoding a partial order is more costly than encoding a total order. There are fewer encoding choices and the choices require more boolean variables and constraints. The only difference between a partial order and a total order is that a partial order allows for both `a->b` and `b->a` to be false, while a total order requires one of the two to be true. If a partial order only contains order constraints with positive polarity, it can be converted into a total order. SATUNE implements this optimization in the order decomposition stage of the partial order analysis.

7.3 Order Conversion

Recall from Section 4 that SATUNE support two different encodings for total orders. One approach to encoding total orders is to create an integer variable for every item in the underlying set. An order constraint then becomes an inequality constraint. For example, the order constraint `a−` $>$ `b` becomes the inequality $x_a < x_b$.

SATUNE optionally applies this conversion. The conversion is applied before the integer-specific optimizations and encoding passes such that the converted order can leverage these optimizations.

7.4 Integer Variable Domain Reduction

Clients can assert equalities or inequalities between integer variables and constants. These assertions can be used to reason about potential values for integer variables and to reduce the number of possible values that we must encode. SATUNE includes an optional analysis that examines all equalities and inequalities that are asserted between integer variables and constants and then updates the domain of the corresponding integer variable. This optimization can reduce the number of values that must be encoded, resulting in simpler encodings.

7.5 Encoding

This section discusses how SATUNE optimizes the implementation of specific encodings. Optimizing encoding is not only a matter of selecting which encodings to use for individual integer variables. It is also a matter optimizing how the encodings for different variables

affect the encoding of constraints between these variables.

With a naive encoding strategy for variables, comparisons between two different variables over a set must be encoded as an enumeration of all cases. For example, if x is taken from the set $\{0, 1\}$ and y is taken from the set $\{0, 1, 2\}$, then $x = y$ is typically encoded as $((x == 0) \wedge (y == 0)) \vee ((x == 1) \wedge (y == 1))$. If x and y were taken from the same set and encoded using the binary index encoding in the same way, the constraint could be encoded by a "bitwise" comparison of the boolean variables that comprise the binary index. This alternative circuit-based encoding grows as *log* of the size of the set.

This brings up the question of what are the necessary conditions for comparing two variable encodings using a circuit-based encoding instead of an enumeration-based encoding. To make this discussion precise, we introduce the following notation. For an integer variable x drawn from the set $\{x_1, x_2, ..., x_{n_x}\}$, we define $e_x(x_i)$ to be binary value that encodes the integer x_i.

It is safe to use circuit-based encodings of $x = y$, if the following conditions are satisfied:

1. The encodings for x and y must encode all shared values in the same way. $\forall i, j. 1 \leq i \leq n_x, 1 \leq j \leq n_y, x_i = y_j \Rightarrow e_x(x_i) = e_y(y_j)$.
2. The encodings for x and y must not use the same encoding for different values. $\forall i, j. 1 \leq i \leq n_x, 1 \leq j \leq n_y, x_i \neq y_j \Rightarrow e_x(x_i) \neq e_y(y_j)$.

It is also possible to use circuit-based encodings for comparisons such as $x < y$ or $x \leq y$. The corresponding condition for $x < y$ is:

1. $\forall i, j. 1 \leq i \leq n_x, 1 \leq j \leq n_y, x_i < y_j \Leftrightarrow e_x(x_i) < e_y(y_j)$.

7.6 Constraint Subgraph

We next define the *constraint subgraph* that SATUNE uses to track which comparison predicates to encode as circuits. We represent the constraint subgraph as a set of vertices V^{cg}, a set of equality edges E^{cg}_{equality}, and a set of inequality edges $E^{cg}_{\text{inequality}}$. There is a vertex $v_S \in V_{cg}$ in the graph that SATUNE uses to represent all integer variables drawn from the same declared set S. If there is an equality predicate between a variable x represented by v_X and a variable y represented by v_Y that is to be encoded as a circuit, then there is an edge $\langle v_X, v_Y \rangle \in E^{cg}_{\text{equality}}$. If there is an inequality predicate between a variable x represented by v_X and a variable y represented by v_Y that is to be encoded as a circuit, then there is an edge $\langle v_X, v_Y \rangle \in E^{cg}_{\text{inequality}}$. Note that the constraint subgraph loses information—it does not distinguish whether an inequality predicate is $<, \leq, >$, or $\geq$.

7.7 Encoding Graph

SATUNE next converts the constraint subgraph into an *encoding graph*. There is a vertex in the encoding graph for each integer value in each vertex of the constraint graph—the set of vertices in the encoding graph is: $V^{eg} = \{\langle v_X, x \rangle \mid v_X \in V^{cg}, x \in X\}$. Equality constraints on the encoding are modeled as equality edges; the equality edges are defined as follows: $E^{eg}_{\text{equality}} = \{\langle \langle v_X, x \rangle, \langle v_Y, y \rangle \rangle \mid \langle v_X, v_Y \rangle \in E^{cg}_{\text{equality}}, x \in v_X, y \in v_Y\}$. Inequality constraints on the encoding are modeled as inequality edges; the inequality edges are defined as follows: $E^{eg}_{\text{inequality}} = \{\langle \langle v_X, x \rangle, \langle v_Y, y \rangle \rangle \mid \langle v_X, v_Y \rangle \in E^{cg}_{\text{inequality}} \vee \langle v_Y, v_X \rangle \in E^{cg}_{\text{inequality}}, x \in v_X, y \in v_Y, x < y\}$. Note that the inequality edges in the encoding graph are directed towards the larger value.

The encoding graph defines the constraints on valid encodings. Solving the constraints from Section 7.5 on the encoding graph yields a valid encoding. In general, we suspect that

encodings that use a minimal number of variables are likely to be better. Thus, we wish to find the solution to these constraints that requires the minimal number of boolean variables. Note that the optimal encoding problem is NP-complete as solving problems that contain just equality constraints is identical to graph coloring. We thus describe our approximate solution algorithm.

The first pass identifies vertices that must have the same encoding and merges them. This pass finds edges between two vertices that both have the same integer value. Such vertices must share the same encoding to ensure that comparisons function correctly. SATUNE merges these vertices together and the new merged vertex has all of the edges that the previous two vertices contained (minus the self edge).

The remainder of the encoding process will be structured as two passes: a first pass assigns encodings for vertices with inequalities and the second pass assigns encodings for vertices that only have equalities. Since the first pass will assign encodings for all vertices that have inequalities, we need to make sure that this initial assignment correctly accounts for equality constraints. Thus, we strengthen an equality edge $\langle\langle v_X, x\rangle, \langle v_Y, y\rangle\rangle$ to an inequality edge if both of the vertices $\langle v_X, x\rangle$ and $\langle v_Y, y\rangle$ at the endpoints of an equality edge also have inequality edges. SATUNE then topologically sorts the encoding graph considering only the inequality edges. In topological order, it assigns encodings to vertices that have at least one incoming or outgoing inequality edge. If a vertex has no incoming inequality edges but it does have outgoing equality edges, it is assigned the encoding 0. Vertices with at least one incoming inequality edge are assigned an encoding that is one larger than the largest encoding value of the sources of the incoming inequality edges.

Finally, SATUNE assigns encodings for vertices that have no inequality edges. SATUNE's algorithm processes these vertices one by one. For each vertex, it first iterates over all of the equality edges for the vertex and constructs the set of encodings that are used by the vertices on the other side of the edge. It then selects the smallest non-negative integer that

is not already in use to encode the value corresponding to the vertex.

7.8 Constructing Constraint Subgraphs

We next discuss the heuristics we use for constructing constraint subgraphs. Comparisons between variables in the same constraint subgraph use circuit-based encodings. We begin by considering some factors in this decision. Our first consideration is the number of clauses that are generated by the encoding. The size of an enumerative encoding for equality becomes large if the intersection of the two sets is large. A second consideration is the size of the encodings. If we place variables over sets with little overlap into the same constraint subgraph, we can potentially increase the size of the encoding. This incurs two costs: (1) it increases the number of boolean variables used by the encoding and (2) it increases the number of clauses that must be generated to ensure that variables have valid values.

Our constraint subgraph construction uses a greedy merging algorithm. It considers two factors when merging nodes: (1) could this merge require allocating new boolean variables in the encoding and (2) do the nodes have substantial overlap in their values.

7.9 Has Value Constraints

Both the one hot and the binary index encodings require constraints to ensure that a variable has a value. The binary index encoding requires the constraint because there are often unused encoding values and Satune must ensure that the variable has one of the used encoding values.

For binary index encodings, a constraint is only needed if there are unused encoding values. There are two ways to generate a constraint that ensures that the encoding has a value. The

first approach is to generate a constraint that is a disjunction (or) of all the valid values for the encoding. The second approach is to generate a constraint that ensures that the encoding does not have one of the unused values. This approach starts with a less than constraint to ensure that encoding does not have a value larger than the largest used encoding. The approach then generates a constraint for each unused encoding below this maximum value that ensures that the encoding is not assigned the given unused value.

For each encoding instance, Satune computes an approximate ratio of the total clause size generated by the first approach to the total clause size generated by the second approach. The tuner selects a threshold and Satune uses the first approach is the ratio is smaller than the threshold and the second approach if the ratio is larger than the threshold.

7.10 Variable Ordering

The order of variables can surprisingly strongly influence SAT solving time [46]. Satune uses three strategies to order variables: (1) order variables in the order that they are used by the client, (2) order variables in the order that the client creates them, or (3) order variables in the reverse order that the client creates them. The tuner selects which strategy Satune uses to order variables.

7.11 CNF Generation

Satune implements a variation of the NICE [56, 18] algorithm to generate CNF constraints. The original implementation of NICE uses hashing to eliminate duplicate expressions. Most of the potential benefit from detecting duplicate expressions is already obtained by Satune's detection of common subexpressions when constructing the intermediate representation. The

second modification is that in certain cases, the NICE CNF generation algorithm needs to know the polarity of expressions. Since SATUNE already has computed the polarity of expressions in its intermediate representation, we can simply use those precomputed polarities.

These two modifications together allow us to implement a variation of the NICE algorithm that does not require keeping the complete set of boolean constraints in memory. SATUNE can thus immediately translate constraints into CNF and output them to the solver as it encodes them. This significantly reduces the memory consumption and the time taken by the CNF generation phase.

7.12 Incremental Solving

SATUNE supports incremental solving. It supports the addition of new constraints on integer variables. While it is conceptually straightforward to support incremental solving on order constraints, it would require disabling optimizations as not all of our order optimizations are safe in the presence of new order constraints.

7.13 Tuner Framework

Recall from Section 1 that SATUNE can operate in two modes: a learning mode in which it learns an encoding specifically for the given client and a deployment mode in which it uses the learned encoding strategy. As it was presented in Section 7, SATUNE incorporates a wide range of specialized optimizations and encoding strategies but not all of them are expected to be beneficial for a given problem type. In the learning mode, SATUNE explores different configurations of optimizations and encodings to find the best set of settings that provide the best performance for a specific problem type. In addition to various optimizations, SATUNE

incorporates a wide range of general and optimization-specific heuristics that they can be fully tuned for each problem type.

SATUNE implements a variation of *Simulated Annealing (SA)* algorithm for exploring the search space and finding the best settings. In general, an SA algorithm is a probabilistic technique for finding the approximate maximum and it works as follows. At each step, a solution close to the current one is selected and evaluated. Based on the performance of the solution, the SA algorithm decides whether to keep the setting. To avoid getting stuck in a local maxima, it is necessary to sometime accept worse settings in order to find better settings. A *temperature* is used to select how willing the SA algorithm is to accept a non-optimal solution. In the beginning, the temperature is high and there is a higher chance of accepting worse solutions. As the algorithm progresses, the temperature decreases and the algorithm is less likely to accept bad solutions and explore new solutions spaces. Because of this property, the algorithm can avoid getting caught at local maxima which are better than any nearby solutions but are not globally optimal.

For some problem types, a single encoding strategy does not always yield the fast solution. Given the availability of compute nodes, it can be reasonable to used different encoding strategies in parallel to solve a given problem. Thus, SATUNE implements a variation of the Simulated Annealing algorithm which finds a set of n complementary tuning strategies for a specific client instead of only one found by the original algorithm. SATUNE uses a score-based ranking system for evaluating the performance of each encoding strategy for each client. The score-based system works as follows. At each round, a new random tuner will be generated based on the best n encoding strategies from the previous round. Then, the tuner scores the n best encoding strategies for each problem with higher weights given for the best strategy. The weighted scoring mechanism causes the tuner to avoid finding similar strategies that perform well on the same problems while not optimizing any encoding strategy for other problems. At the end of the round, it only keeps n encoding strategies

with the highest scores to be explored in the next round. After synthesizing the n best encoding strategies for a given client, SATUNE utilizes them for solving new problems from that client in the deployment mode. The idea is to encode the problem using the n different strategies and solve them in parallel, finishing when the fastest encoding returns a result. For our evaluation, we used $n = 1$.

Chapter 8

Evaluation

We evaluate Satune on constraint problems generated by three real-world tools: JMCR [43], a Java-based model checker; SyPet [32], a component-based synthesis tool for APIs; and Dirk [49], a deadlock predictor. We also evaluate Satune on three puzzle games: Sudoku, Hexiom, and Killer Sudoku. For each benchmark, we used the original implementation of the benchmark and swapped the SMT/SAT Solver with Satune making minimal modifications to add support for Satune.

The SAT solver used plays a key role in the performance of constraint solving. In order to make the comparison fair, we modified all the SAT-based benchmark implementations to use the same solver as Satune, Glucose [5]. Thus, the Satune implementation and the baseline implementations only differ in the encodings they use. However, Dirk and JMCR encode constraints in SMT and use Z3 to solve them. Thus, we could not replace the underlying solver with Glucose for these benchmarks.

For each benchmark, we use a subset of the problems to learn the best encoding configuration. We used approximately 70% of our data to train on and 30% as our test set. We report the test results only.

Satune can translate problems to SMT LIB v2.0, the standard input language for SMT solvers. Variables over discrete sets and total orders can be translated into the integer theory in SMT LIB. Satune's translator also supports Alloy [47]. We used Satune's translator to compare Satune against: Z3 [26], SMTRat [23], MathSAT [15], and Alloy.

All of our experiments were run on identical machines, each with a Xeon(R) CPU E3-1246 v3 3.5GHz processor and 32GB memory running Ubuntu Linux 18.04. Each machine ran only one instance of Satune. We set time limits for each problem. Each test case for the tool benchmarks consisted of many constraint problems: we set a per constraint problem time limit for JMCR of 100 seconds, for SyPet of 100 seconds, and for Dirk of 1,000 seconds. The game test cases consists of a single constraint problem: we set a time limit for Hexium of 1,000 seconds, for Sudoku 2,000 seconds, and for KillerSudoku 2,000 seconds.

All of the tool benchmarks generate multiple constraint problems. The later problems depend on the results from the previous problems. This is an issue for comparison because different constraint solvers might find different answers to a problem, and these different answers could lead to easier or harder problems to solve later. To make the problems comparable for our evaluation, we modified these tools to serialize Satune problems to disk and recorded the results from the original baseline solver. We then used the recorded problems for our evaluation.

8.1 JMCR

JMCR [43] is a stateless model checker that implements the Maximal Causality Reduction model checking algorithm. It constructs ordering constraints over executions to generate new possible schedules and enforces that at least one load returns a different value in the new schedule.

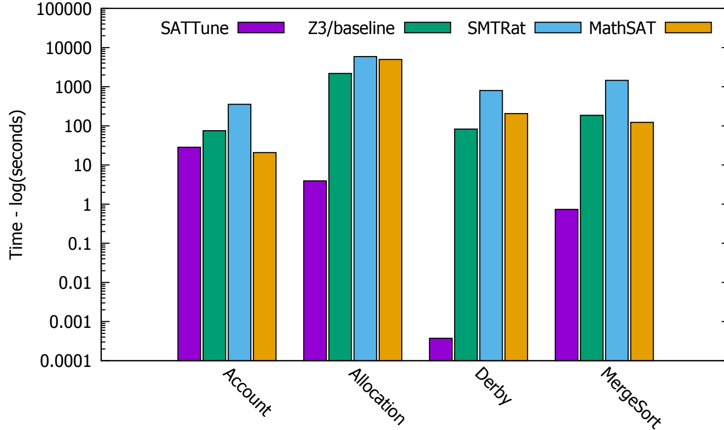

Figure 8.1: SATUNE's test set results of four-fold cross validation are compared with the execution time of Z3(the baseline), SMTRat, and MathSAT for each JMCR problem

The baseline encoding uses the integer theory of SMT to represent ordering constraints. We encoded these constraints as total orders in SATUNE. In order to evaluate the performance of SATUNE, we selected the four most difficult problems in JMCR's original test suite.

Encoding Average	Part. 1 SAT(s)	Part. 2 SAT(s)	Part. 3 SAT(s)	Part. 4 SAT(s)
SATUNE All	0.256	0.250	0.260	0.135
Baseline All	21.998	21.998	21.998	21.998
SATUNE Test	4.071	0.046	0.001	0.049
Baseline Test	10.742	25.714	10.362	12.397

Table 8.1: Arithmetic means of SAT solving time for different partitions of the four-fold cross-validation on JMCR

Table 8.1 shows the arithmetic mean of the SAT solving time for each partition in the four-fold cross-validation learning. As it is depicted, the encoding that SATUNE learned yielded speedups that were, on average, 150× greater for some partitions.

Figure 8.1 shows the SAT solving time for each test case. **Lower is better. We use logarithmic scales throughout this paper, and so Satune is significantly faster than the baseline on most benchmarks.** The encodings that SATUNE synthesized outperform JMCR's original encoding and other SMT solvers by several orders of magnitude in most cases. SMT solvers are general purposed solvers and unlike SATUNE, they don't optimize the encoding for each client and are consequently slower than SATUNE.

As an example, SATUNE synthesized an encoding for the first partition that encodes orders pairwise and runs SATUNE's order optimizations.

8.2 SyPet

SyPet is a type-directed tool for component-based synthesis, which uses a compact Petri-net representation to model relationships between methods in an API [32]. For a given target method signature S, SyPet uses reachability analysis to determine the sequences of method calls that could be used to synthesize an implementation of S. SyPet guarantees that the synthesized components type-check and pass all test cases.

SyPet uses the one-hot encoding for the possible ways of completing holes in a program sketch. There is a finite set of possible ways to fill the hole and we map the holes to variables over discrete sets in SATUNE. SyPet uses incremental solving for the baseline encoding. Although SATUNE supports incremental solving, our synthesis framework and translator does not. However, the synthesized encoder could be used with SyPet in incremental mode. In order for the results to not be skewed, we report (the faster) incremental results for the baseline and non-incremental results for SATUNE and all other solvers.

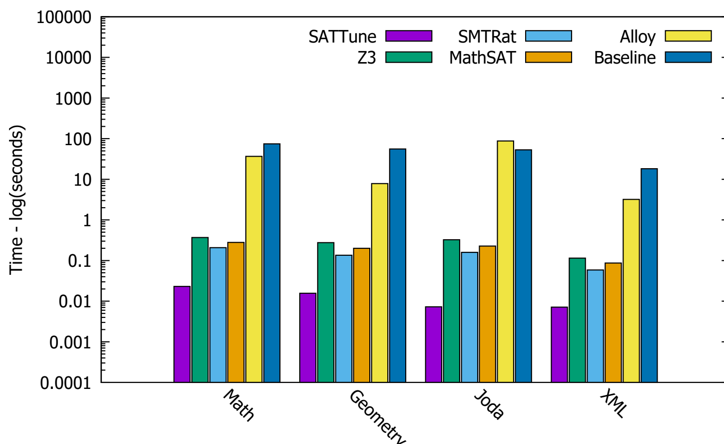

Figure 8.2: SATUNE's test set results of four-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each SyPet problem

Encoding Average	Part. 1 SAT(s)	Part. 2 SAT(s)	Part. 3 SAT(s)	Part. 4 SAT(s)
Satune All	0.001	0.001	0.001	0.001
Baseline All	3.590	3.590	3.590	3.590
Satune Test	0.001	0.001	0.001	0.001
Baseline Test	3.113	3.916	3.969	3.027

Table 8.2: Arithmetic means of SAT solving time for different partitions of the four-fold cross-validation on SyPet

Table 8.2 reports the arithmetic mean of SAT solving time for each partition in the four-fold cross-validation learning. The encoding that Satune learned yields speedups that are, on average, 300× greater for some partitions.

Figure 8.2 compares the solving time of SyPet with Satune. Satune improves the performance of SyPet by several orders of magnitude on all test cases. We also compare Satune to several SMT solvers and Satune is faster than these solvers for all test cases.

As an example, Satune synthesized an encoding for the first partition that uses the binary index encoding. It also uses the integer variable domain reduction optimization together with the encoding graph.

8.3 Dirk

Dirk [49] is a deadlock and data-race predictor for Java. It uses Z3 to model execution constraints. Dirk uses ordering constraints to represent the happens-before relation for lock *release* and *acquire* events. Dirk uses the integer theory of SMT to represent the happens before relation in the program [49]. Dirk adds a non-standard constraint that two events happen at the same time. This constraint cannot be directly represented as a total order. While Satune could be modified to support this constraint, we do not believe that it is commonly used. So instead, we preprocessed the constraints to eliminate this constraint. Due to this translation process, it is not possible to support incremental solving for this

benchmark in SATUNE. We instead disable incremental solving in Dirk for the comparison.

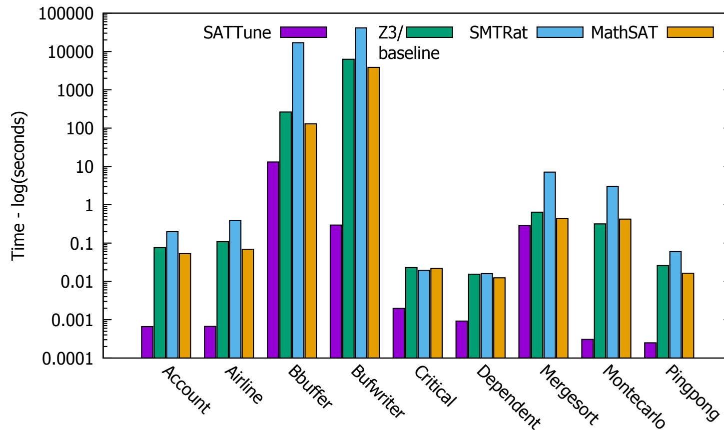

Figure 8.3: SATUNE's test set results of two-fold cross-validation are compared with the execution time of Z3(the baseline), SMTRat, MathSAT for each Dirk problem

Encoding Average	Part. 1 SAT(s)	Part. 2 SAT(s)
SATUNE All	0.064	0.066
Baseline All	19.129	19.129
SATUNE Test	1.111	0.626
Baseline Test	10.682	22.990

Table 8.3: Arithmetic means of SAT solving time for different partitions of the four-fold cross-validation on Dirk

Table 8.3 shows the arithmetic mean of SAT solving time for each partition in the four-fold cross-validation learning. The encoding that SATUNE learned yielded speedups that were, on average, 2000× greater for some partitions.

Figure 8.3 reports the SAT solving time for each test case. The encoding that SATUNE synthesized is faster for every test case than the baseline encoding (Z3) for Dirk and sometimes outperforms the baseline encoding by several orders of magnitude. SATUNE is also faster than the other SMT solvers for all of the benchmarks.

SATUNE synthesized an encoding for the first partition that encodes orders using pairwise encoding and then uses the order optimizations to simplify the order constraints.

8.4 Hexiom

Hexiom is a game in which a player moves numbered tiles on a hexagonal board until the numbers on the tiles match the number of its neighbors. The baseline encoding uses the one hot encoding to represent the tile number of each cell if it is occupied [40]. In the SATUNE version of the puzzle, the tile number for each cell is a variable drawn from a discrete set. SATUNE can generate the same encoding as the baseline, if it uses one hot encoding. The Hexiom test cases were based on ones from the original online puzzle. But since most of them were easy to solve for the SAT solver, we modified the test cases to generate more difficult problems.

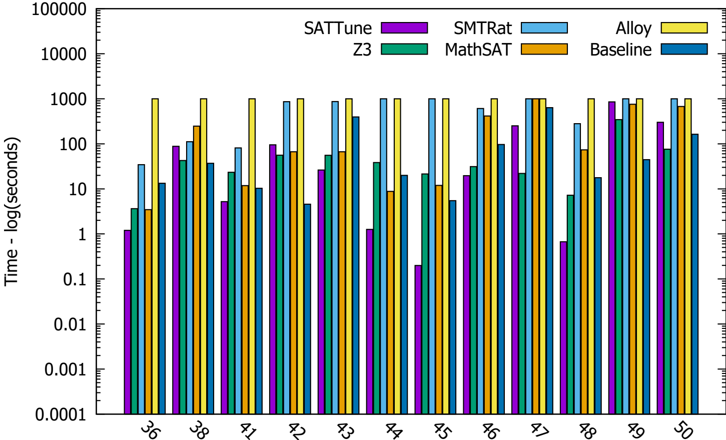

Figure 8.4: SATUNE's test set results of three-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Hexiom problem

Encoding Average	Part. 1 SAT(s)	Part. 2 SAT(s)	Part. 3 SAT(s)
SATUNE All	43.558	96.962	104.504
Baseline All	119.480	119.480	119.480
SATUNE Test	24.225	246.923	139.748
Baseline Test	9.465	142.651	206.323

Table 8.4: Arithmetic means of SAT solving time for different partitions of the three-fold cross-validation on Hexiom

Table 8.4shows the arithmetic mean of SAT solving time for each partition in the three-fold cross-validation learning. The encoding that SATUNE learned yielded speedups that were, on average, 8× greater for some partitions.

Figure 8.4 presents the solving time for Satune, the baseline solver, the SMT solvers, and Alloy. Satune is better than the baseline encoding in most of the test cases.

Satune synthesized encodings for each partition used in cross validation. It encoded integers as binary index (partition 2) or unary (partitions 1 and 3) and reverses the order of variable. It does not use the graph encoding optimizations.

8.5 Sudoku

Sudoku is a popular puzzle which has both backtracking and constraint-based solvers, the latter is faster and more scalable [73, 60]. The baseline solver uses the one hot encoding and allocates a boolean variable for each possible value in each cell. We used a variation of Sudoku generator [3] to generate test cases with large sizes which are more time-consuming for the SAT Solver to solve.

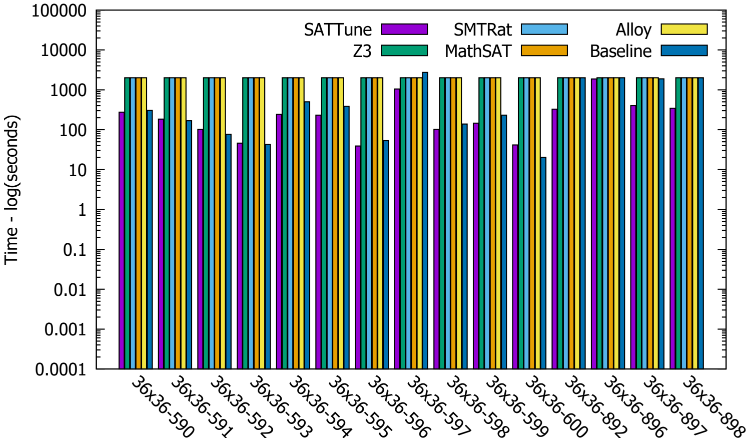

Figure 8.5: Satune's test set results of three-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Sudoku problem

Table 8.5 shows the arithmetic mean of SAT solving time for each partition in the three-fold cross-validation learning. The encoding that Satune learned yielded speedups that were, on average, 2.1× greater for some partitions.

Figure 8.5 presents the solving time of each test case in the three-fold cross-validation com-

Encoding Average	Part. 1 SAT(s)	Part. 2 SAT(s)	Part. 3 SAT(s)
SATUNE All	685.367	585.566	584.229
Baseline All	974.751	974.751	974.751
SATUNE Test	484.918	643.844	314.039
Baseline Test	752.638	1236.115	920.125

Table 8.5: Arithmetic means of SAT solving time for different partitions of the three-fold cross-validation on Sudoku

pared with solving time of the baseline encoding and the other SMT solvers. For most of the test cases, SATUNE's synthesized encodings outperformed the baseline and other solvers.

SATUNE synthesized encodings for three partitions. These encodings use the one hot for integer variables and order variables in the original order. They differ in the threshold they use for creating proxy variables in CNF encoding.

8.6 Killer Sudoku

Killer Sudoku extends Sudoku with ***cages***. The Killer Sudoku encoding extends the Sudoku by enumerating the possible values for cells in each cage to implement the sum constraint. The Killer Sudoku baseline performs some preprocessing to reduce the number of variables by assigning common values to a new boolean variable in each cell [2].

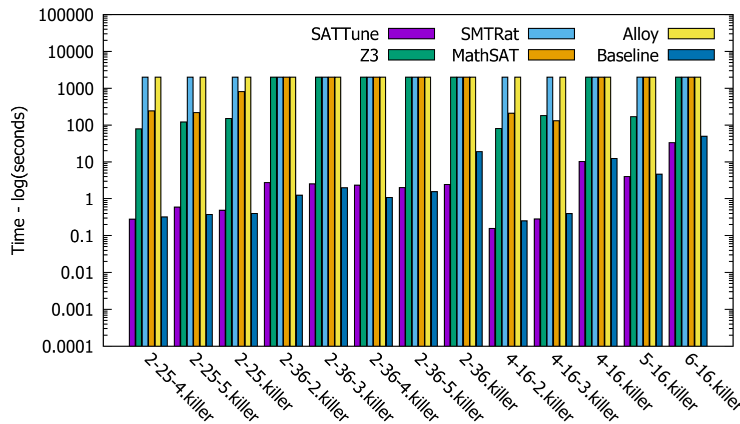

Figure 8.6: SATUNE's test set results of three-fold cross-validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Killer Sudoku problem

Encoding Average	Part. 1 SAT(s)	Part. 2 SAT(s)	Part. 3 SAT(s)
Satune All	7.355	116.736	7.196
Baseline All	7.190	7.190	7.190
Satune Test	15.160	368.056	3.214
Baseline Test	10.683	8.413	1.603

Table 8.6: Arithmetic means of SAT solving time for different partitions of the three-fold cross-validation on Killer Sudoku

Table 8.6 shows the arithmetic mean of SAT solving time for each partition in the three-fold cross-validation learning. The encoding that Satune learned yielded speedups that were, on average, 1.5× greater for some partitions.

Figure 8.6 presents the solving time for each test case in the three-fold cross-validation compared with solving time of the baseline and other solvers. For most of the test cases, Satune's synthesized encodings outperformed the baseline and other solvers.

Satune synthesized encoding strategies for three partitions. Each encoded integer variables using binary index and kept the original variable ordering. For the two partitions with difficult problems, Satune encoded integer variables using the graph optimization and Satune could outperform the baseline encoding. However, this optimization causes a minor slowdown for relatively easy problems. In the other partition, that did not happen to include difficult problems, Satune just enabled the integer domain reduction pass.

8.7 Parallel Satune

Nowadays, a lot of computing parallel resources are available such as servers or systems with multi-core CPUs. Satune is capable of synthesizing more than one encoding for each learning set. Having multiple Satune running with different encoding strategies yields the utilization of parallel resources to get better performance. This technique is very useful when Satune is learning on a partition incorporating both difficult and simple problems that

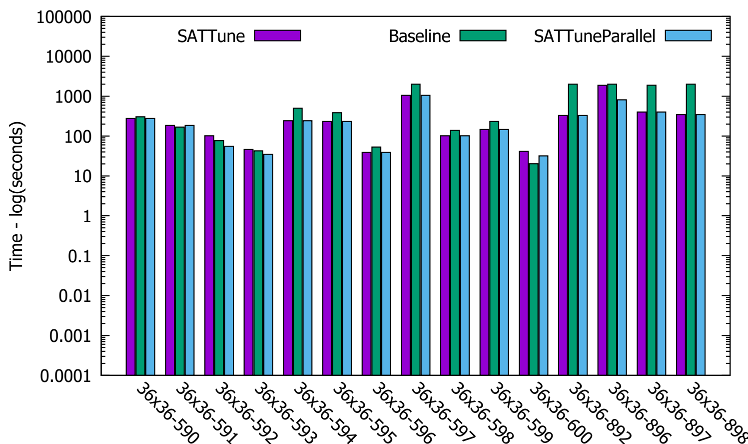

Figure 8.7: Comparing the baseline encoding with Satune and Parallel Satune for Sudoku

demand thoroughly different encoding strategies for each of them. For the Sudoku, Hexiom and Killer Sudoku, instead of one, two of the best encoding strategies are synthesized and run in parallel to measure the performance improvement of Parallel Satune.

Figure 8.7 depicts the Satune's solving time for each test case in the three-fold cross-validation compared with the solving time of the baseline and Parallel Satune. The encoding that Parallel Satune learned yields speedups that are, on average, 1.2× greater than Satune for some partitions. For Sudoku, since the difficulty of problems in the learning set is similar, two synthesized strategies are very similar to each other and using Parallel Satune instead of Satune is not very efficient.

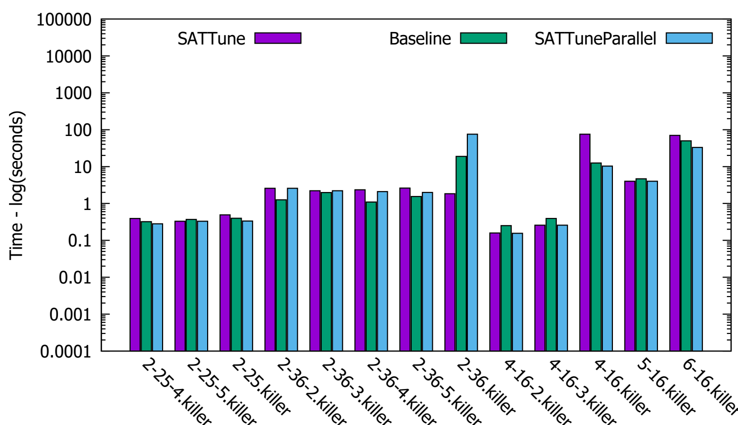

Figure 8.8: Comparing the baseline encoding with Satune and Parallel Satune for Killer Sudoku

Figure 8.8 depicts the Satune's solving time for each test case in the three-fold cross-

validation compared with the solving time of the baseline and Parallel SATUNE. The encoding that Parallel SATUNE learned yields speedups that are, on average, 1.6× greater than SATUNE for some partitions. For Killer sudoku, since the difficulty of problems in the learning set is similar, two synthesized strategies are very similar to each other and using Parallel SATUNE instead of SATUNE is not very efficient.

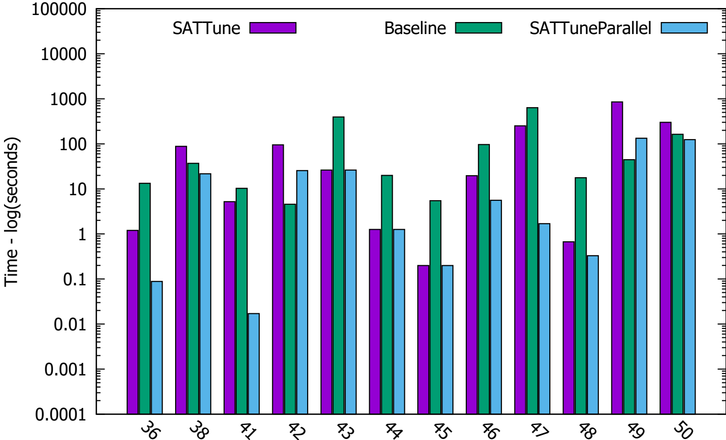

Figure 8.9: Comparing the baseline encoding with SATUNE and Parallel SATUNE for Hexiom

Figure 8.9 depicts the SATUNE's solving time for each test case in the three-fold cross-validation compared with the solving time of the baseline and Parallel SATUNE. The encoding that Parallel SATUNE learned yields speedups that are, on average, 40× greater than SATUNE for some partitions. For Hexiom, since the difficulty of problems in the learning set is considerably different, two synthesized strategies are very different from each other. In this case Parallel SATUNE is reasonable to be employed instead of normal SATUNE.

8.8 Using Maple SAT Solver

In general, the choice of SAT solver plays a pivotal role in the performance of solving a SAT problem. The encoding that SATUNE synthesizes is customized for the SAT solver that SATUNE uses. So, if one encoding strategy works well for SATUNE's SAT solver, it does not imply that encoding suits best for the other SAT solver. In order to assess the previous

statement, we replace Glucose [5] with Maple [54] in Satune and run the best synthesized encoding for Sudoku and Hexiom with Maple solver instead. The Maple is a conflict-driven clause-learning SAT solver which employs machine learning-based heuristics in branching and restart policies [54].

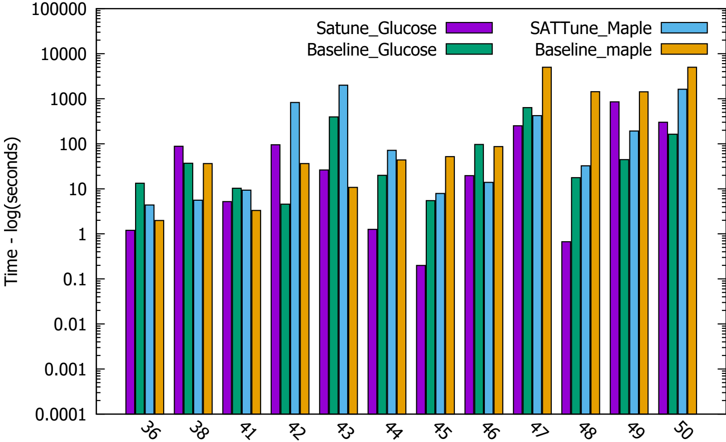

Figure 8.10: Comparing the baseline encoding with Satune under two different solvers, Maple and Glucose, for Hexiom

As Figure 8.10 depicts, the baseline encoding is faster for Glucose instead of Maple. Although Maple uses machine learning heuristics, it does not work well for Hexiom. On average, using Maple instead of Glucose causes more than 4.3× slowdown for Satune and more than 2.5× slowdown for the baseline encoding.

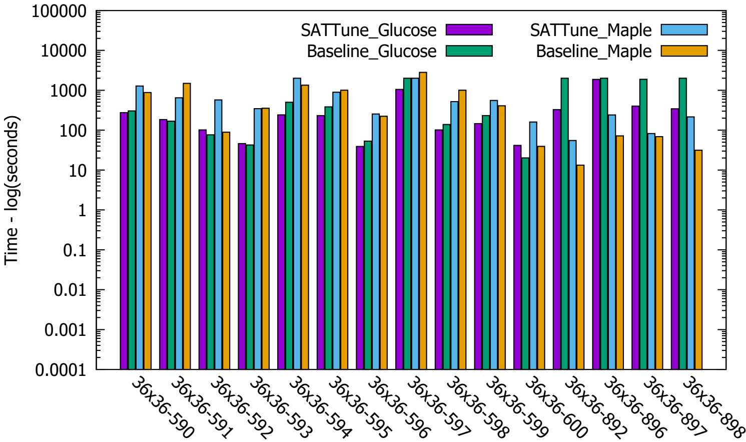

Figure 8.11: Comparing the baseline encoding with Satune under two different solvers, Maple and Glucose, for Sudoku

As Figure 8.11 shows, the baseline encoding is faster for Maple instead of Glucose. On average, Using Maple instead of Glucose causes more than 2× slowdown for Satune and

more than 1.2× speedup for the baseline encoding.

As we could see, using Maple for Sudoku can have performance improvement in comparison with the baseline but it causes slowdowns for Hexiom. The encoding that Glucose-based SATUNE synthesized is best for Glucose SAT Solver and not necessarily the most optimized encoding for Maple solver. SATUNE needs to retrain on the benchmarks in order to learn the best encoding strategy for Maple-based SATUNE. There are many solvers other than MapleSAT and Glucose and choosing the best solver and the suitable encoding for each solver on each benchmark is a difficult problem and can be considered as the future work for SATUNE.

Chapter 9

Related Work

Alloy is a relational logic intended for software modeling [47]. The Alloy analyzer translates Alloy into SAT to find finite models. Kodkod is the current model finding engine for Alloy and translates Alloy constraints into SAT constraints [69]. The SATUNE language differs from Alloy in that the SATUNE language is intended to have a closer mapping to abstractions for which multiple good SAT encodings are known and the SATUNE language is in general a lower-level language that is primarily targeted towards supporting client applications. Kodkod uses sophisticated optimizations including auto-compacting circuits and symmetry breaking to generate constraints that are more efficient to solve. Kodkod does not attempt to tune its encoding strategy for specific problem types. Enfragmo [1] is specified for solving combinatorial search problems contrary to SATUNE which is a general purpose framework. Some frameworks provide high level language and perform some domain-specific optimization [57] and some implement a collection of encodings and techniques without any tuning [36].

Much work has been done on developing encodings to SAT. Several different encodings are known for variables drawn from discrete sets [12, 34, 52] and we have implemented the most common ones. SATUNE could be extended to support more of these encodings.

Satisfiability modulo theories (SMT) solvers support constraints that overlap with Satune's constraint language [30, 26]. SMT generalizes boolean satisfiability by replacing boolean variables with predicates over a variety of other theories. The predicates over these theories are then solved by specialized solvers. Satune does not use specialized solvers, but rather translates its constraint language directly into SAT. This approach has trade offs in the translation approach can support very rich constraints but cannot leverage the performance benefits of domain-specific solvers. Although some constraints in the Satune language overlap with SMT theories, Satune is not intended to replace SMT solver but rather to explore benefits of automatically tuning encodings. Eager SMT solvers take a similar solving strategy to Satune and directly translate constraints from other theories into SAT [48, 14, 35]. Satune differs from this work in that it supports multiple encodings and is targeted towards automatically tuning encodings rather than supporting other constraint theories.

Many constraint problems are solved by translation into SAT, including planning [61, 50], circuit security [75, 79], SAT-based model checking [27, 16, 77, 68], scheduling [80, 38, 10], and verification [71, 19]. Encodings mostly are hard-coded in SAT-based frameworks and to best of our knowledge, no other framework automatically tunes encodings for clients. AI and learning have been widely used in the SAT solving domain [76, 55, 37, 58] and there are frameworks that can learn the best solver for each problem type [78, 53, 59].

Chapter 10

Conclusion

This paper presents Satune, a tool for automatically synthesizing the encodings of constraints into SAT. Traditionally discovering a good SAT encoding for a problem domain required much effort to explore the many different options. Satune supports a range of encoding strategies and optimizations and automatically selects combinations that yield good performances for a given problem domain. Our evaluation shows that Satune is able to synthesize encodings that are significantly faster than the original encodings used by our benchmarks.

10.1 Future Work

Satune can be extended in different ways:

1. Satune can add support for more sophisticated encoding for different types of constraints such as ExactlyOne, etc.

2. Satune synthesizer can leverage other AI search to identify the best encoding

3. Satune can used a portfolio solver and its synthesizer can find the best encoding and best SAT solver for each problem

4. There are many other SAT problems that are potentially capable of using Satune instead of normal SAT Solvers. As an example, Conda [22] is a package manager of Anacando [21] that uses SAT solver to resolve dependency among packages. Packaging constraints can be fully supported by Satune.

5. By adding more frameworks that use Satune and gathering more data, Satune synthesizer can employ machine learning to identify the best encoding as well.

Bibliography

[1] A. Aavani, X. N. Wu, S. Tasharrofi, E. Ternovska, and D. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 15–22, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Airobert. SAT based Killer Sudoku, 2016.

[3] T. Ardi. SAT based sudoku solver in Python, 2015.

[4] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs. On freezing and reactivating learnt clauses. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 188–200, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 399–404. Morgan Kaufmann Publishers Inc., 2009.

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

[7] G. Audemard and L. Simon. Lazy clause exchange policy for parallel sat solvers. In *Theory and Applications of Satisfiability Testing (SAT '14)*, pages 197–205, 2014.

[8] G. Audemard and L. Simon. Glucose and syrup in the sat race 2015, 2015.

[9] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[10] R. Béjar and F. Manya. Solving the round robin problem using propositional logic. In *AAAI/IAAI*, pages 262–266, 2000.

[11] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[12] M. Björk. Successful SAT encoding techniques. *Journal on Satisfiability, Boolean Modeling, and Computation*, 7:189–201, July 2009.

[13] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, pages 113–124, 2011.

[14] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[15] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The mathsatÂ 4 smt solver. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, pages 299–303, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[16] S. Burckhardt, R. Alur, and M. M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, volume 42, pages 12–21. ACM, 2007.

[17] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[18] B. Chambers, P. Manolios, and D. Vroon. Faster SAT solving with better CNF generation. In *2009 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2009.

[19] I. Chung. A SAT-based method for basis path testing using KodKod. *International Journal of Applied Engineering Research*, 12(18):7294–7305, 2017.

[20] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[21] A. Community. Anaconda an opensource framework for data science and machine learning, 2019.

[22] C. Community. Conda a cross-platform, language-agnostic binary package manager, 2019.

[23] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám. Smt-rat: An open source c++ toolbox for strategic and parallel smt solving. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368, Cham, 2015. Springer International Publishing.

[24] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[25] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[26] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[27] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *ACM SIGPLAN Notices*, volume 50, pages 20–36. ACM, 2015.

[28] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–332, 2013.

[29] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, 2008.

[30] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[31] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[32] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 599–612, New York, NY, USA, 2017. ACM.

[33] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 191–202, 2002.

[34] A. M. Frisch and P. A. Giannaros. SAT encodings of the at-most-k constraint. some old, some new, some fast, some slow. In *Proceedings of the Tenth International Workshop of Constraint Modelling and Reformulation*, 2010.

[35] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 519–531, 2007.

[36] Gecode. Generic constraint development environment, 2016.

[37] I. Gent, L. Kotthoff, I. Miguel, and P. Nightingale. Machine learning for constraint solver design–a case study for the alldifferent constraint. *arXiv preprint arXiv:1008.4326*, 2010.

[38] I. P. Gent and I. Lynce. A SAT encoding for the social golfer problem. *Modelling and Solving Problems with Constraints*, 2, 2005.

[39] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[40] H. M. Gualandi. Using an industrial-strength SAT solver to solve the Hexiom puzzle, 2012.

[41] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In K. Eder, J. Lourenço, and O. Shehory, editors, *Hardware and Software: Verification and Testing*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[42] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–174, 2015.

[43] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 165–174, New York, NY, USA, 2015. ACM.

[44] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[45] J. P. Inala, R. Singh, and A. Solar-Lezama. Synthesis of domain specific CNF encoders for bit-vector solvers. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 302–320. Springer, 2016.

[46] M. Iser, M. Taghdiri, and C. Sinz. Optimizing MiniSAT variable orderings for the relational model finder Kodkod. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 483–484, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[47] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[48] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector artithmetic. In *Proc. 21st International Conference on Computer-Aided verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674, June 2009.

[49] C. G. Kalhauge and J. Palsberg. Sound deadlock prediction. *Proc. ACM Program. Lang.*, 2(OOPSLA):146:1–146:29, Oct. 2018.

[50] H. Kautz and B. Selman. SATPLAN04: Planning as satisfiability. *Working Notes on the Fifth International Planning Competition (IPC-2006)*, pages 45–46, 2006.

[51] H. A. Kautz and B. Selman. Ten challenges redux: Recent progress in propositional reasoning and search. In *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, pages 1–18, 2003.

[52] W. Klieber and G. Kwon. Efficient CNF encoding for selecting 1 from N objects. In *Proceedings of the Fourth Workshop on Constraint in Formal Verification*, 2007.

[53] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016.

[54] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 94–110, 2018.

[55] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 94–110. Springer, 2018.

[56] P. Manolios and D. Vroon. Efficient circuit to CNF conversion. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[57] A. Metodi and M. Codish. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming*, 12(4-5):465–483, 2012.

[58] N. Musliu. Applying machine learning for solver selection in scheduling: A case study.

[59] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

[60] U. Pfeiffer, T. Karnagel, and G. Scheffler. A Sudoku-solver for large puzzles using SAT. In A. Voronkov, G. Sutcliffe, M. Baaz, and C. Ferm\"uller, editors, *LPAR-17-short. short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning.*, volume 13 of *EPiC Series in Computing*, pages 52–57. EasyChair, 2013.

[61] J. Rintanen. Madagascar: Scalable planning with SAT. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 21, 2014.

[62] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[63] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–706, 2018.

[64] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2016.

[65] J. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.

[66] R. Singh, J. P. Near, V. Ganesh, and M. Rinard. AvatarSAT: An auto-tuning boolean SAT solver. Technical Report MIT-CSAIL-TR-2009-039, Massachusetts Institute of Technology, August 2009.

[67] R. Singh and A. Solar-Lezama. SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *FMCAD*, pages 185–192. IEEE, 2016.

[68] N. Timm, S. Gruner, and P. Sibanda. Model checking of concurrent software systems via heuristic-guided sat solving. In *International Conference on Fundamentals of Software Engineering*, pages 244–259. Springer, 2017.

[69] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 632–647, 2007.

[70] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 2010 Conference on Programming Language Design and Implementation*, pages 341–350, 2010.

[71] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone. Analysis of XACML policies with SMT. In *International Conference on Principles of Security and Trust*, pages 115–134. Springer, 2015.

[72] J. P. Wallner, G. Weissenbacher, and S. Woltran. Advanced sat techniques for abstract argumentation. In *Proceedings of the 14th International Workshop on Computational Logic in Multi-Agent Systems - Volume 8143*, CLIMA XIV, pages 138–154, Berlin, Heidelberg, 2013. Springer-Verlag.

[73] T. Weber. A SAT-based Sudoku solver. In G. Sutcliffe and A. Voronkov, editors, *LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings*, pages 11–15, Dec. 2005.

[74] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1173–1178, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[75] T. WINOGRAD and H. MAHMOODI. Programmable gates using hybrid CMOS-STT design to prevent ic reverse engineering. 2009.

[76] H. Wu. Improving sat-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 787–788, New York, NY, USA, 2017. ACM.

[77] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *International Conference on Computer Aided Verification*, pages 139–143. Springer, 2005.

[78] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[79] C. Yu, X. Zhang, D. Liu, M. Ciesielski, and D. Holcomb. Incremental SAT-based reverse engineering of camouflaged logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(10):1647–1659, 2017.

[80] H. Zhang, D. Li, and H. Shen. A SAT based scheduler for tournament schedules. In *SAT*, 2004.

[81] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

[82] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 17–36, London, UK, UK, 2002. Springer-Verlag.

[83] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, G. H. Xu, L. Wang, and X. Li. Grapple: A graph system for static finite-state property checking of large-scale system code. In *Proceedings of European Computer System Conference*, 2019.