Satune: Synthesizing Efficient SAT Encoders

HAMED GORJIARA, University of California, Irvine, USA
GUOQING HARRY XU, University of California, Los Angeles, USA
BRIAN DEMSKY, University of California, Irvine, USA

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 encoders generate constraint problems that are often several orders of magnitude faster to solve than the original encodings used by the tools.

CCS Concepts: • **Software and its engineering** → *Formal software verification*; *Constraint and logic languages*.

Additional Key Words and Phrases: Auto-tuning, SAT encoding, Constraint Solvers

ACM Reference Format:
Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2020. Satune: Synthesizing Efficient SAT Encoders. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 146 (November 2020), 32 pages. https://doi.org/10.1145/3428214

1 Introduction

Modern software analysis — from path-sensitive analysis [Dillig et al. 2008; Shi et al. 2018; Xie and Aiken 2005; Zuo et al. 2019], through symbolic execution [Cadar et al. 2008; Godefroid et al. 2005; Sen et al. 2005], to verification and model checking [Burckhardt et al. 2007; Demsky and Lam 2015; Desai et al. 2013; Flanagan and Qadeer 2002; Huang 2015a; Sigurbjarnarson et al. 2016] — 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

Authors' addresses: Hamed Gorjiara, Electrical Engineering and Computer Science, University of California, Irvine, USA, hgorjiar@uci.edu; Guoqing Harry Xu, Computer Science, University of California, Los Angeles, USA, harryxu@cs.ucla.edu; Brian Demsky, Computer Science, University of California, Irvine, USA, bdemsky@uci.edu.

2475-1421/2020/11-ART146
https://doi.org/10.1145/3428214

Table 1. Performance of different encodings of a randomly generated total order constraint. The total time includes both solving time and encoding time.

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

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 [Audemard and Simon 2014, 2015; Davis et al. 1962; Davis and Putnam 1960; Kautz and Selman 2006, 2003; Xu et al. 2008] 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 [Brain et al. 2016; Inala et al. 2016; Manthey et al. 2012; Martins et al. 2011]. Often times choices that initially appear to be 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 selected encodings interact with the constraints' characteristics, etc.

The difficulty of finding good encodings is well-known. An article [Björk 2009] that interviews several SAT experts states *"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."*

To illustrate 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 model checking of multi-threaded programs—CheckFence [Burckhardt et al. 2007], SATCheck [Demsky and Lam 2015], and MCR [Huang 2015a] 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 2. Table 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 orders-of-magnitude apart.

Challenges. Determining the right encoding for a problem domain is challenging in 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 [Burckhardt et al. 2007] and SATCheck [Demsky and Lam 2015], 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 is 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 [Bordeaux and Marques-Silva 2012], but constraints are not always amenable to unit propagation. For example, unit propagation is less 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.

State of the art. There exists a large body of work [O'Mahony et al. 2008; Singh et al. 2009; Singh and Solar-Lezama 2016; Xu et al. 2008] on optimizing performance for SAT solvers. Most of these optimizations focus on low-level formula rewrites [Inala et al. 2016] or autotuning of a set of candidate rewrites [O'Mahony et al. 2008; Xu et al. 2008], 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!*

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.

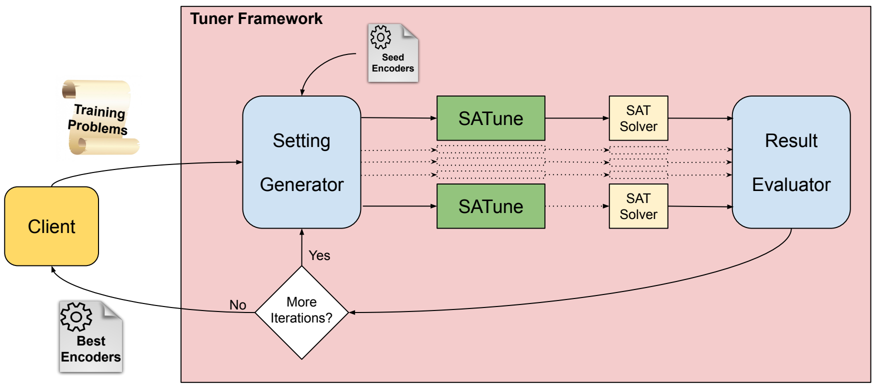


Fig. 1. Satune Synthesis Architecture

Satune allows the developer to express a domain problem with a novel domain-specific language (§4), which provides a means for the developer to specify domain-related constraints while abstracting away low-level SAT-related details. Given a constraint problem 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. Figure 1 presents an overview of the Satune synthesis framework. Given a set of training problems from the client expressed in Satune's DSL, Satune's tuning framework searches the space of encoding strategies on these example problems. It then measures the time taken by the SAT solver for each encoding strategy to evaluate that strategy. This process repeats until the search budget is exhausted. Once the process finishes, Satune has synthesized encoders that are tuned for the given client.

The information expressed in the DSL allows Satune to differentiate between different use cases of the same types of constraints and is critical for Satune to effectively generalize encoders to different problem instances. To further improve efficiency, Satune implements a range of sophisticated analyses (§6) for globally optimizing encoding strategies. Table 2 shows the details of these analyses, including their names and whether they are new techniques proposed by us or used before in the literature. On top of these analyses, we propose novel order graph analyses (§6.2.1) as well as graph-based encoding as an optimization to encode integer variables drawn from a discrete set (§7). For certain 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 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 overheads, 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). 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.

Table 2. A set of analyses and optimizations used by Satune. As marked clearly in the table, some of the techniques are proposed by us in this work and others are used in previous work.

Analysis	Reference
Integer variable domain reduction for discrete sets	Proposed in this work (§6.2.2)
A graph-based encoding for integer variables	Proposed in this work (§7)
Pure literal elimination	(§6.1.1) [Davis et al. 1962]
Representing order as inequalities	(§6.1.2) [Kalhauge and Palsberg 2018]
Exactly-one constraints	(§6.1.3) [Frisch and Giannaros 2010]
Techniques for encoding variable ordering	(§6.1.4) [Iser et al. 2012]

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 10× for our benchmarks.

2 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.

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. 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 constraints to ensure the variable can only have one value. Exactly-one constraints can be formalized as: $AtLeastOne(\{b_0, b_1, ..., b_n\}) \wedge AtMostOne(\{b_0, b_1, ..., b_n\})$. At-Least-One constraints can be simply encoded as $b_0 \vee b_1 \vee .. \vee b_n$. For our example, following constraint is generated to ensure that x has at least one value: $b_0 \vee b_1 \vee b_3$. For encoding At-Most-One constraints, previous literature proposed different encodings [Hölldobler and Nguyen 2013] including, but not limited to *binomial*, *commander variable*, and *sequential counter* encodings.

The binomial encoding [Frisch and Giannaros 2010] generates constraints to ensure the variable can have at most one value ($\forall i, j \neq i, b_i \Rightarrow \neg b_j$). For our example, it enforces the following constraints: $b_0 \Rightarrow \neg b_1$, $b_0 \Rightarrow \neg b_3$, and $b_1 \Rightarrow \neg b_3$. This encoding does not require extra boolean variables. It requires $O(n^2)$ clauses to ensure that the integer variable has exactly one value.

In *the commander variable* encoding [Klieber and Kwon 2007a], variables are partitioned into groups of size m and "a commander variable" c_i is assigned to each group. If we assume groups of size $m = 2$ in the previous example, b_0 and b_1 are grouped together with a commander variable c_0 and b_2 is grouped together with a command variable c_1. This encoding generates the following four types of constraints. (1) In each group, at most one variable can be true. The binomial encoding is used to encode this constraint. (2) If the commander variable is true, at least one of the variables in the group must be true. In our example, $\neg c_0 \vee b_0 \vee b_1$ enforces this constraint for the first group. (3) No variables in the group can be true if the corresponding commander variable is false. In the example, the encoding generates $c_0 \vee \neg b_0$ and $c_0 \vee \neg b_1$. (4) At most one of the commander variables can be true. If there are more than m commander variables, then the encoding recursively applies the same strategy to the commander variables. This encoding employs $O(n)$ auxiliary variables and requires $O(n)$ clauses to encode the at-most-one constraint.

The sequential counter [Sinz 2005] is another technique to encode at-most-one constraints. It encodes the partial sum $s_i = \sum_{j=1}^{i} b_i$ for increasing values of i up to the final $i = n$. After simplification, this encoding generates the following constraint:

$$(\neg b_1 \vee s_1) \wedge (\neg b_n \vee \neg s_{n-1}) \wedge \bigwedge_{1<i<n}((\neg b_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg b_i \vee \neg s_{i-1}))$$

For our example, this encoding requires two auxiliary variables s_1 and s_2 and it generates the constraint $(\neg b_0 \vee s_1) \wedge (\neg b_3 \vee \neg s_2) \wedge (\neg b_1 \vee s_2) \wedge (\neg s_1 \vee s_2) \wedge (\neg b_1 \vee \neg s_1)$. In general, this encoding utilizes $n - 1$ auxiliary variables and requires $3n - 4$ clauses to encode the at-most-one constraint.

One advantage of the one hot encoding is that it incorporates binary constraints that work well with the propagation behavior of SAT solvers [Matsunaga 2015]. 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 at least $O(n)$ number of boolean variables, (2) depending on the selected encoding, it requires $O(n) - O(n^2)$ clauses to ensure that the integer variable has only one value, and (3) it requires a constraint to ensure that the integer variable at least has a value. Solving these many constraints is time-consuming — equality constraints between variables have $O(n)$ clauses while inequality constraints between integer variables have $O(n^2)$ clauses.

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 $\neg b_0 \land \neg b_1$, $x = 1$ is encoded as $b_0 \land \neg b_1$, and $x = 3$ is encoded as $b_0 \land b_1$. The positives of this encoding are: (1) it requires only $O(n)$ clauses to implement, and (2) inequality constraints between integer variables have $O(n)$ clauses. 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 $O(n)$ clauses.
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 $\neg(b_0 \land b_1)$ to ensure that index is in range. Under this encoding, $x = 0$ (00) would be encoded as $\neg b_0 \land \neg b_1$, $x = 1$ (01) would be encoded as $b_0 \land \neg b_1$, and $x = 2$ (10) would be encoded as $\neg b_0 \land b_1$. 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 $O(log(n))$ clauses. 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. We discussed this in more detail in Section 7.

Order Encodings. Another major category of constraints is 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 $\neg b_{ij} \lor \neg b_{ji}$ to ensure that the encoding cannot order both items first.

Both encodings use the following transitivity constraints: $\forall i, j, k : b_{ij} \land b_{jk} \Rightarrow b_{ik}$. 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) $O(n^3)$ clauses.
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.

3 Motivation

To motivate the discussion, let us 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, the benefit from the SAT solver performing *unit propagation* may be reduced (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* might 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 [Burckhardt et al. 2007; Demsky and Lam 2015], 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.

4 Satune DSL

We begin by presenting the constraint language Satune takes as input. Figure 2 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 accepts constraints via a C, C++, Java, 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 [Demsky and Lam 2015], CheckFence [Burckhardt et al. 2007], MemSAT [Torlak et al. 2010], JMCR [Huang 2015a], RVPredict [Huang et al. 2014], Dirk [Kalhauge and Palsberg 2018], CPPMem [Batty et al. 2011], Nitpick [Blanchette et al. 2011]. Variables drawn from discrete sets are used by Alloy [Jackson 2002], Paradox [Claessen and SÃűrensson 2003], CPPMem [Batty et al. 2011], CheckFence [Burckhardt et al. 2007], Nitpick [Blanchette et al. 2011], Package Managers, etc.

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

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

Fig. 2. SATUNE Constraint Language Grammar

corresponding boolean expression *bexpr* that follows the # to true[1]. Clients can define functions using tables. In table-defined functions, the relation between the function's output and its input is represented as a table. Each row in a table contains a set of integer values for each of the function inputs and the corresponding integer value for the output. 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.

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.

SATUNE does not directly use existing constraint DSLs such as SMTLib [Initiative 2018] because they do not provide a mechanism to label the different uses of variables and sets with types. These labels allow SATUNE to differentiate between variables that serve different roles in the encoding and thus may benefit from using different encoding strategies.

```
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))
```

Fig. 3. Example Constraints in SATUNE DSL.

[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 $\Leftarrow$.

Example. Figure 3 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.

Limitations Satune focuses on autotuning the encoding and targets a commonly used subset of constraints that is more restricted than most existing SMT solvers support. Satune currently does not support set operations (union, intersection, transitive closure, etc.) and universal quantification in propositional logic. Also, it does not support complex arithmetic operations on integer variables such as multiplication or division. The current version of Satune only implements at-least-one and at-most-one constraints for the one hot encoding. These constraints are a restricted form of cardinality constraints. Cardinality constraints encode at-most-k and at-least-k constraints. Cardinality constraints are widely used in various domains [Bailleux and Boufkhad 2003; Cabon et al. 1999; Kuechlin and Sinz 2000], and different encodings exist to encode them into SAT [Frisch and Giannaros 2010; Sinz 2005; Zhou 2020]. There are also different encodings available for a generalization of cardinality constraint, Pseudo-Boolean constraints [Aavani 2011; Bailleux et al. 2009; Warners 1998]. In the future, Satune can be extended to implement these types of constraints in order to be used in a wider range of applications.

5 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, developers first need to modify their application to generate constraints in the Satune DSL. Since Satune's DSL includes abstractions that are commonly already present in such applications along with full support for boolean constraints, 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.

5.1 Synthesizing Encoders

Satune starts the synthesis process with a set of *seed encoders*. The process uses a simulated annealing algorithm 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. While the one-hot encoding works well for Sudoku, Killer Sudoku (a variant of Sudoku) has a wider variety of different types of constraints and despite its similarities surprisingly benefits less from the one-hot encoding. The tuner would likely decide to encode Killer Sudoku 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 share 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.

5.2 Optimization and Encoding Framework

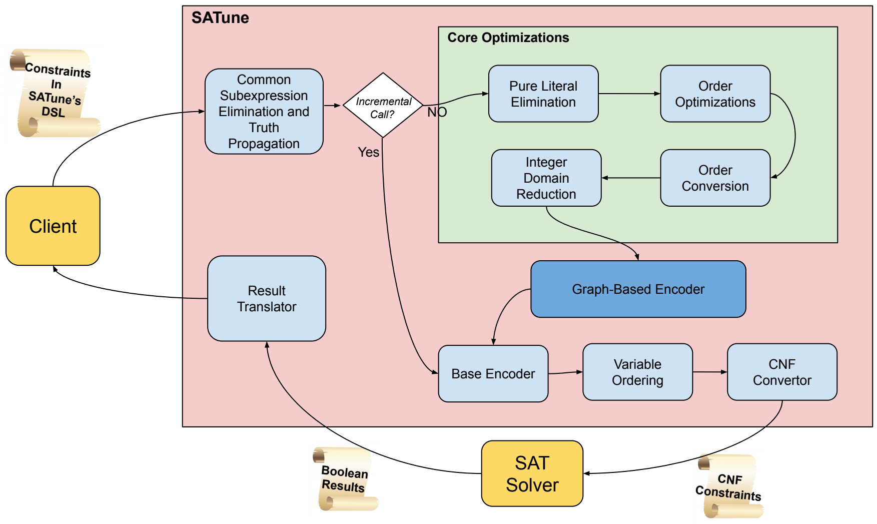


Fig. 4. Overview of SATUNE's optimizations and encoding framework

Figure 4 presents an overview of SATUNE's optimizations and encoding framework. SATUNE is structured as a pipeline of optimizations followed by an encoding framework. SATUNE always performs common sub-expression elimination and truth propagation online as constraints are generated by the client. SATUNE supports incremental solving, but it can only perform its optimizations and encoding selection during the first solve call. SATUNE contains a set of core optimizations shown in the green rectangle. Section 6 presents a set of optimizations SATUNE uses, including those previously known (Section 6.1) as well as new optimizations this work develops (Section 6.2). These optimizations are not guaranteed to benefit all problem domains, and thus these optimizations are under the control of the tuning framework. After the optimizations are performed, SATUNE then encodes the optimized problem domain into CNF SAT. Section 7 describes the encoding process in more detail. The graph-based encoder attempts to globally optimize binary index encodings of integer values. The base encoders and variable ordering then perform the work of encoding the remaining encodings. Finally, the CNF converter implements these encodings in CNF SAT.

6 SATUNE's Candidate Optimizations

Implementing sophisticated optimizations to enable better SAT encodings can sometimes yield significant performance improvements. For some problem domains, 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 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) [Chambers et al. 2009; Manolios and Vroon 2007] 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 analyses 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.

In the remainder of this section, we discuss a set of optimizations Satune chooses from to optimize for a particular problem domain, including ones proposed before and adapted to our setting (Section 6.1) as well as several new ones developed in this work (Section 6.2).

6.1 Existing Optimizations Used by Satune

Satune implements several optimizations and encoding techniques from the literature that have been demonstrated to be useful.

6.1.1 Pure Literal Elimination Satune includes an optimization pass that simplifies constraints by eliminating boolean variables that appear in a single polarity, *i.e.*, pure literal. 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 [Davis et al. 1962; Davis and Putnam 1960]. While SAT solvers commonly implement this optimization, performing it before encoding can potentially allow Satune to simplify constraints enabling further optimizations and simplifications to the encodings.

6.1.2 Order Conversion Recall from Section 2 that Satune supports 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 [Huang 2015b; Kalhauge and Palsberg 2018]. For example, the order constraint `a- > b` becomes the inequality $x_a < x_b$. Both x_a and x_b are integer variables drawn from a discrete set.

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.

6.1.3 Exactly-one Constraints Both the one hot and the binary index encodings require constraints to ensure that a variable has exactly one value. Recall from Section 2 that exactly-one constraints for one hot encoding can be formalized as: $AtLeastOne(\{b_0, b_1, ..., b_n\}) \wedge AtMostOne(\{b_0, b_1, ..., b_n\})$. Satune implements three encodings for the at least one constraint: the Binomial [Frisch and Giannaros 2010] encoding, the Commander Variable [Klieber and Kwon 2007a] encoding, and the Sequential Counter [Sinz 2005] encoding. As discussed earlier, these encodings make different tradeoffs in number of clauses and/or extra boolean variables to enforce the constraint. The binary index encoding also requires a 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 the 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 if the ratio is smaller than the threshold and the second approach if the ratio is larger than the threshold.

6.1.4 Variable Ordering The order of variables can surprisingly influence SAT solving time [Iser et al. 2012]. 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 to use.

6.2 New Optimizations

In addition to the existing techniques, we develop a set of new optimizations that specifically target SAT problems generated by a software analysis and include them in Satune's toolbox as candidate optimizations.

6.2.1 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 use 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 5 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

```
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 | ...)
```

Fig. 5. Order constraint decomposition example with ellipsis (...) indicating omitted non-order constraints.

a vertex v_a to a second vertex v_b if an order predicate $\mathtt{a}- > \mathtt{b}$ appears with positive polarity or an order predicate $\mathtt{b}- > \mathtt{a}$ 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 (Figure 6). 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.

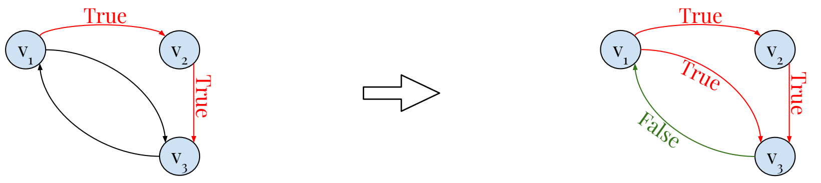


Fig. 6. v_1 is ordered before v_2. v_2 is ordered before v_3. So, it can be inferred v_1 is ordered before v_3.

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 $\mathtt{b}- > \mathtt{a}$ 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 (Figure 7).

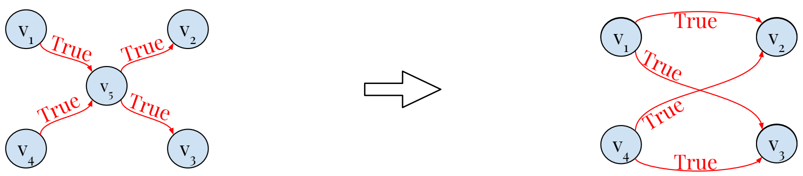


Fig. 7. Example of Vertex Elimination analysis. All the incoming and outgoing edges from/to v_5 are true. v_5 is removed and the corresponding edges are added.

Must Edge Pruning: Consider an edge $\langle v_a, v_b \rangle$ for which (1) the edge satisfies the *mustbetrue* 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 (Figure 8).

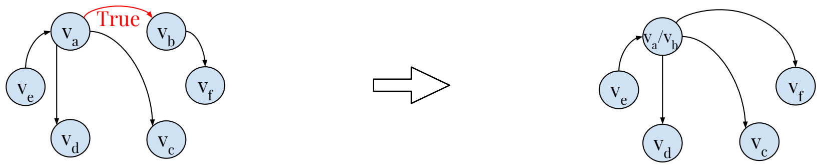


Fig. 8. Edge $\langle v_a, v_b \rangle$ is true and either v_a has no other outgoing edges or v_b has no other incoming edges. So, v_a and v_b can be merged.

Order Decomposition: This optimization decomposes an order into two or more smaller orders to optimize for the fact that the encoding cost for 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 (Figure 9).

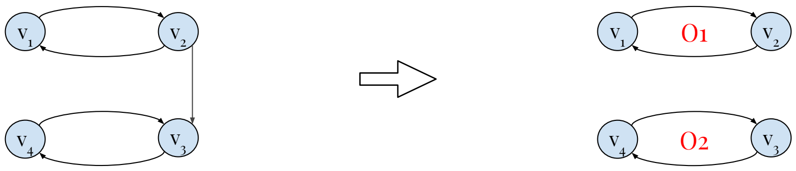


Fig. 9. The order graph corresponds to Figure 5. Edge $\langle v_2, v_3 \rangle$ can be made true and the original order can be decomposed into the orders O1 and O2.

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.

6.2.2 Integer Variable Domain Reduction Clients can assert equalities or inequalities between integer variables and constants. Satune reasons about potential values for integer variables to reduce the number of possible values that it must encode. This optional analysis examines all equalities and inequalities that are asserted between integer variables and constants and then updates the domain of the corresponding integer variable. SMT solvers [de Moura and Bjørner 2008] use similar techniques to propagate values from equality constraints and linear inequalities, but to our knowledge, they do not include optimizations that reduce the range for variables taken over discrete sets.

7 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 of 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 $((x == 0) \land (y == 0)) \lor ((x ==$

$1) \wedge (y == 1))$. 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 the *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 the 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. $\forall i, j.1 \le i \le n_x, 1 \le j \le n_y, x_i = y_j \Rightarrow e_x(x_i) = e_y(y_j)$.
(2) The encodings for x and y must not use the same encoding for different values. $\forall i, j.1 \le i \le n_x, 1 \le j \le n_y, x_i \ne y_j \Rightarrow e_x(x_i) \ne e_y(y_j)$.

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

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

7.1 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 $<, \le, >$, or $\ge$.

7.2 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: $E^{eg}_{\text{inequality}} = \{\langle \langle v_X, x \rangle, \langle v_Y, y \rangle \rangle \mid \langle v_X, v_Y \rangle \in E^{cg}_{\text{inequality}} \vee \langle v_Y, v_X \rangle \in E^{cg}_{\text{inequality}}, x \in v_X, y \in v_Y, x \le y\}$. 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 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-hard as graph coloring can be reduced to an encoding that uses only equality constraints. We next describe our heuristic for solving this problem. Algorithm 1 presents pseudocode for this algorithm. As a working example, consider four integer variables w, x, y, z that are drawn from sets $s_w = \{0, 5\}$, $s_x = \{0, 2, 7\}$, $s_y = \{0, 2, 8\}$, and $s_z = \{0, 2, 5, 8\}$, respectively.

These variables appear in the following constraint predicates: $w < x$, $x = y$, and $y = z$. Figures 10a and 10b present the constraint subgraph and encoding graph for the example, respectively.

Algorithm 1: Encoding Algorithm

```
Result: Circuit-based Encoding
for e in $E^{eg}_{equality} \cup E^{eg}_{inequality}$ do
    if val(src(e)) == val(dst(e)) then
        mergeNodes(src(e), dst(e));
    end
end
strengthenEqualities($E^{eg}_{\text{equality}}$, $E^{eg}_{\text{inequality}}$);
$E^{eg}_{sorted}$ =topologicalSort($E^{eg}_{\text{inequality}}$);
for e in $E^{eg}_{sorted}$ do
    if | inEdges(src(e)) |== 0 then
        src(e).encoding = 0;
    else
        src(e).encoding=MaxEncoding(inEdges(src(e)))+1;
    end
end
for v in $V^{eg}$ (in decreasing order of degree) do
    if v.encoding == UNASSIGNED then
        v.encoding = minAvailable(equalityNeighbors(v));
    end
end
```

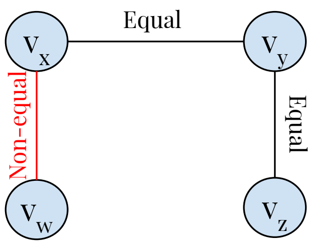
The first pass identifies vertices that must have the same encoding and merges them. This pass finds edges between two vertices that both have the same integer value. Such vertices must share the same encoding to ensure that comparisons function correctly. SATUNE merges these vertices together and the newly 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. Figure 10c presents the example encoding graph after solving for inequalities.

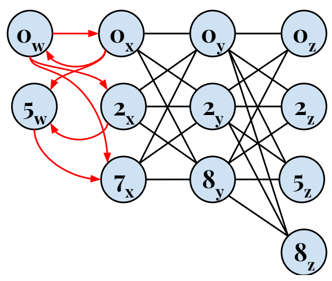
Finally, SATUNE assigns encodings for vertices that have no inequality edges. SATUNE's algorithm uses a standard greedy graph coloring algorithm to assign colors at this point—it processes these vertices one by one in decreasing order of degree. For each vertex, it assigns as an encoding the smallest non-negative integer that is not already in use by a vertex that shares an edge with the current vertex. Figure 10d presents the final encoding results for the running example. Note that the generated encoding can encode the same number in different ways for different sets, *e.g.*, 5_w

and 5_z, and that the generated encodings are not necessarily monotonic, *e.g.*, $8_{y,z}$ has a smaller encoding value than 5_z.

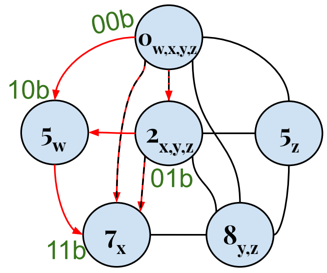
In some cases, SATUNE is able to use the same encoding for multiple different values from the same set if those values cannot be distinguished using the comparison operations in the problem. If a variable from such a set is used in a function or arithmetic operation, this outcome is not sound. In that case, SATUNE adds edges between all encoding graph nodes for the same set to force them to have different encodings.



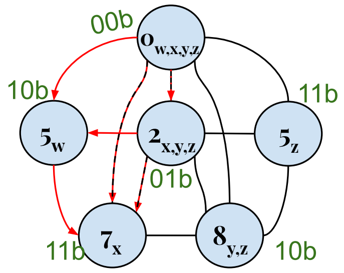
(a) Constraint Subgraph



(b) Encoding Graph: Splitting each node in constraint subgraph into its integer values.



(c) Merging vertexes that have the same value and are connected and then assigning encoding for vertexes with inequalities. The green numbers indicate the encoding for the corresponding integer values.



(d) Assign encoding to vertexes that only have equalities.

Fig. 10. The process of building constraint subgraph and encoding graph to encode the example as a circuit.

7.3 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.

7.4 CNF Generation

Satune implements a variation of the NICE [Chambers et al. 2009; Manolios and Vroon 2007] algorithm to generate CNF constraints. The original implementation of NICE uses hashing to eliminate duplicate expressions. Most of the benefits from detecting duplicate expressions are 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, it simply uses those precomputed polarities.

These two modifications together allow Satune 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.

7.5 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.

8 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 presented in Sections 6 and 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 domain. In its learning mode, Satune explores different configurations of optimizations and encodings to find the 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 (due to space limitations, not all are mentioned in the paper) that can be fully tuned for each user-specified type. Due to the large set of tunable settings and Satune's support for user-specified types, the space of potential settings is much too large to exhaustively enumerate.

8.1 Tuner Architecture

The tuner framework consists of three primary components: language support, integration with analysis and optimizations, and tuner algorithm. Making a tuner work requires tight integration between these three components. Language support allows clients to tag variables with labels that identify different use cases. The idea is that these variables can potentially benefit from different encodings based on their different use cases.

The next component is the integration with the analyses and optimizations. The design of the analyses and optimizations is important to provide the tuner with (1) sufficient freedom to optimize the constraint and (2) reasonable settings such that the tuner results generalize across problems and can easily be learned.

Finally, the tuner algorithm explores the search space efficiently to find a good set of optimizations and encodings for all variables. One component of our tuning algorithm is that for some problem domains, different problem instances can benefit from different encodings. Thus for such problem domains, it can be useful to have more than one encoding strategy that are then solved in parallel.

8.2 Tuner Algorithm

SATUNE implements a variation of *Simulated Annealing (SA)* [Van Laarhoven and Aarts 1987] 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.

The standard Simulated Annealing (SA) algorithm searches for a single encoding/optimization that works best for all the problems in the training set. Using the standard algorithm to produce multiple different encodings would just generate a set of (potentially identical) encodings that work well for the same set of problems. However, the goal of the tuner is to find multiple different encodings that are complementary — for each problem instance there should be at least on encoding that performs well. The tuner needs to find a set of complementary encodings such that the combination of the encodings yields the best performance for all the problems in the training test set.

Thus, SATUNE implements its own variation of Simulated Annealing (SA) to find sets of encodings. The execution time is used to evaluate each generated encoding strategy (*i.e.*, the less execution time the better encoding). The tuning algorithm uses a weighted scoring algorithm that gives points based on how good an encoding is relative to other encodings (*e.g.*,, 1st place gets 3^{n-1} points, 2nd place gets 3^{n-2} points, 3rd place gets 3^{n-3} points, and so on). If there is an encoding that does very well on a subset of problems on which other encodings do not perform well, it can receive many points for those problems. At each iteration, the tuner picks the best n encodings for the next iteration of the SA algorithm. At each round, a new random encoding strategy is generated based on the best n encoding strategies the previous round found. Once the SA budget is exhausted, SATUNE computes the set of n encodings out of all generated encoding that minimize the expected total time for running in parallel on the training set.

The degree of parallelism is a decision that needs to be made based on the problem domain. If there is an encoding that performs relatively well on all the test cases, using a single encoding suffices. When SATUNE is used on clients that benefit from different encodings, multiple encodings can run in parallel if multiple encodings provide sufficient benefit. In SATUNE âĂŹs benchmarks, having one encoding for most of the cases was sufficient. For many benchmarks using 2 encodings in parallel does yield better performance. However, the performance gain is typically less than 2x compared to running a single encoding and thus not sufficient to cover the cost of running 2 solvers. As such, to be consistent in the evaluation section, we report numbers for $n = 1$ for all the benchmarks.

The choice of problems in the training set is important. As is the case in other ML and AI learning algorithms, if the training set is small or it contains non-representative problems, the learned encoding would be biased toward the training set.

Example. Consider the following scenario: A client wants to learn 2 best encodings to minimize the total time if we run them in parallel. The learning process is conducted on four problems: Problem A, Problem B, Problem C, and Problem D. Let Strategy 1 and Strategy 2 be the best encoding strategies from the previous iteration of the tuning algorithm. These strategies are utilized to randomly generate Strategy 3 and 4. Table 3 shows example execution time of each encoding for each problem.

Table 3. Execution time of each encoding for each problem. Numbers are in seconds.

	Strategy 1	Strategy 2	Strategy 3	Strategy 4
Problem A	2.0	112.0	753.0	519.0
Problem B	800.0	681.0	119.0	260.0
Problem C	14.0	69.0	454.0	29.0
Problem D	892.0	329.0	243.0	1100.0

The weighted scoring algorithm computes the score of each encoding strategy for each problem (*i.e.*, Table 4). Once the score is assigned, Satune picks the set of two encodings with the highest score. For our example, Strategy 1 and 3 will be selected for the next iteration with a total score of 36 for all problems.

Table 4. Satune's weighted scoring algorithm assign points to strategies for each problem.

	Strategy 1	Strategy 2	Strategy 3	Strategy 4
Problem A	9	3	0	1
Problem B	0	1	9	3
Problem C	9	1	0	3
Problem D	1	3	9	0

9 Evaluation

We evaluate Satune on constraint problems generated by three real-world tools: JMCR [Huang 2015b], a Java-based model checker; SyPet [Feng et al. 2017a], a component-based synthesis tool for APIs; and Dirk [Kalhauge and Palsberg 2018], 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. We have made Satune available at http://plrg.ics.uci.edu/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 [Audemard and Simon 2009], with the exception of the benchmarks that use SAT4J, *i.e.*, SyPet. For SyPet, we used the Glucose strategies from the SAT4J Solver [Le Berre and Parrain 2010]. 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 [de Moura and Bjørner 2008] to solve them. Thus, we could not replace the underlying solver with Glucose for these benchmarks.

Based on our initial experiments, some encodings appear more efficient for simple problem instances. However, these encodings may not be optimal for more complex SAT problems in the same domain. In order to reduce bias in training, the learning set must contain a diverse set of problem instances. Thus, we partitioned test cases into two, three, and four sets based on the availability of problem instances for each benchmark. Each partition includes problem instances with different difficulty levels. For each benchmark, in **n**-fold cross-validation, one partition is left out for testing, and the rest of **n-1** partitions are employed to learn the encoding. This procedure is repeated for each partition. For each test case, only the test results have reported in the figures.

Satune can translate given problems in its DSL to SMT LIB v2.0, the standard input language for SMT solvers. Variables over discrete sets and total orders are translated into the integer theory in SMT LIB. Satune's translator also supports Alloy [Jackson 2002]. We used Satune's translator to compare Satune against: Z3 [de Moura and Bjørner 2008], SMTRat [Corzilius et al. 2015], MathSAT [Bruttomesso et al. 2008], 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 consist of a single constraint problem: we set a time limit for Hexiom of 1,000 seconds, for Sudoku of 2,000 seconds, and for KillerSudoku of 2,000 seconds. The duration for the learning experiment depends on the time budget and it varies from a couple of hours (*e.g.*, SyPet), to two weeks (*e.g.*, Dirk). We report the time it took Satune to synthesize each of our encodings. We did not attempt to minimize the SA budget for synthesizing encodings — it is possible that similar results could be achieved more quickly by providing a smaller SA budget.

All of the 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. For all the benchmarks, we validated Satune's results against the baseline's results.

9.1 JMCR

JMCR [Huang 2015b] 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.

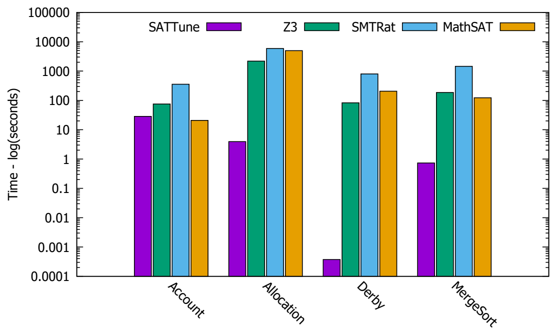


Fig. 11. Satune's test set results of four-fold cross-validation are compared with the execution time of Z3(the baseline), SMTRat, and MathSAT for each JMCR problem

Figure 11 presents the test result for each test case. For example, the Satune bar for the Account test case shows the aggregated SAT solving time of all Account's SAT problems with the encoding that Satune synthesized by learning from the Allocation, Derby, and MergeSort test cases. This process is repeated for three other test cases (*i.e.*, four-fold cross-validation) and we reported the testing result for each of them in Figure 11.

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 test cases in JMCR's original test suite. In total, JMCR test cases contain more than 100 SAT problems.

Figure 11 presents 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 of the cases. SMT solvers are general purpose solvers and unlike SATUNE, they don't optimize the encoding for each client and consequently, they are slower than SATUNE. As an example of a synthesized encoding, SATUNE synthesized an encoding for the first partition that encodes orders pairwise and runs SATUNE's order optimizations. On average, SATUNE synthesized each of these encodings in about five days.

9.2 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 [Feng et al. 2017a]. 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 holes 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. To make the comparison fair, we report non-incremental results for the baseline, for SATUNE, and for other solvers. We used all four available test cases from the original test suite [Feng et al. 2017b]. In total, these test cases contain more than 140 SAT problems. Figure 12 presents the total SAT solving time for each test case.

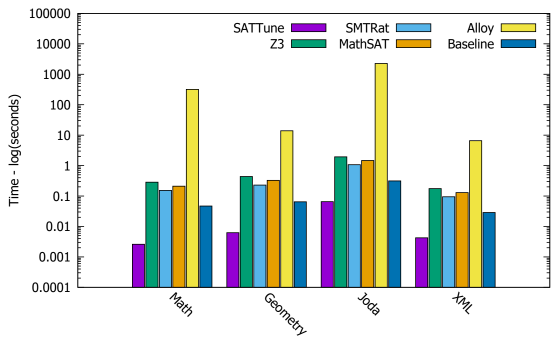


Fig. 12. SATUNE's test set results of four-fold cross-validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline (*i.e.*, SAT4J with glucose strategies) for each SyPet problem

Figure 12 shows the test results for each SyPet test case. For example, the SATUNE bar for the Math test case shows the aggregated SAT solving time for all Math's SAT problems with the encoding that SATUNE synthesized by learning from the Geometry, Joda, and XML test cases. This process is repeated for the three other test cases (*i.e.*, four-fold cross-validation) and we report the testing result for each in Figure 12. 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 of a synthesized encoding, 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. On average, SATUNE synthesized each of these encodings in less than an hour.

9.3 Dirk

Dirk [Kalhauge and Palsberg 2018] 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 [Kalhauge and Palsberg 2018]. 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.

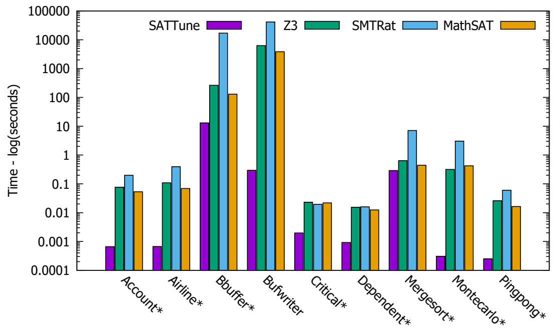


Fig. 13. SATUNE's test set results of two-fold cross-validation are compared with the execution time of Z3(the baseline), SMTRat, MathSAT for each Dirk problem

Figure 13 reports the SAT solving time for each test case. Test cases are separated into two sets and the learned encoding from one set is tested on the other set. This process is repeated for the other set (*i.e.*, two-fold cross-validation) and we reported the testing result for each of them in Figure 13. 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 test cases. In total, the test cases incorporate more than 200 SMT problems.

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. On average, SATUNE synthesized the encoding for each partition in about two weeks of learning.

Puzzles. We evaluated SATUNE on three puzzles: Hexiom, Sudoku, and Killer Sudoku. All of these puzzles mainly employ integer variables and at-most-one constraints. 'SATUNE-V2' bar represents results for a version of SATUNE that supports *the commander variable* encoding [Klieber and Kwon 2007a] as well as *The sequential counter* [Sinz 2005] for at-most-one constraints. The rest of this section provides more details about each of these puzzles.

9.4 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 [Gualandi 2012]. 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 14 presents the solving time for SATUNE (both version 1 and 2), the baseline solver, the SMT solvers, and Alloy. SATUNE is better than the baseline encoding in most of the test cases. SATUNE synthesized the following encodings for each partition: it encoded integers as unary in all

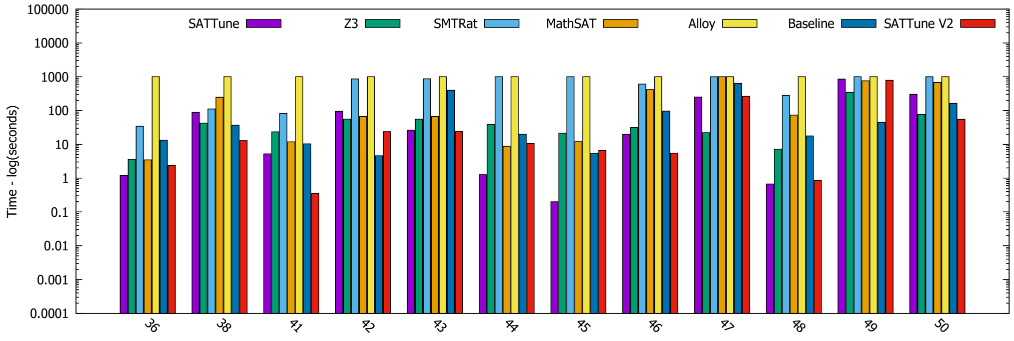


Fig. 14. Satune's test set results of three-fold cross-validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, and the baseline for each Hexiom problem

partitions. Integer domain reduction pass is enabled in two partitions. In the other partition that incorporates relatively easy problems, Satune disabled this pass which caused a minor slowdown for larger size problems. Satune kept graph encoding optimizations deactivated in all partitions. On average, Satune synthesized each of these encodings in about six hours.

9.5 Sudoku

Sudoku is a popular puzzle which has both backtracking and constraint-based solvers, the latter is faster and more scalable [Lynce and Ouaknine 2006; Pfeiffer et al. 2013; Weber 2005]. 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 [Ardi 2015] to generate test cases with large sizes, *e.g.*, 36×36, which are more time-consuming for the SAT Solver to solve.

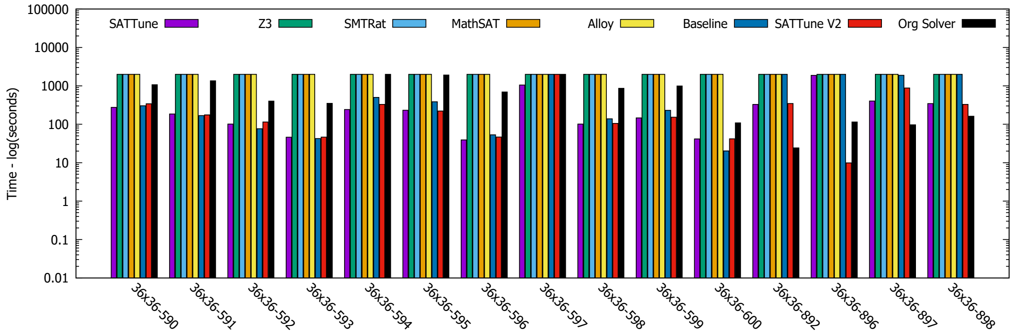


Fig. 15. Satune's test set results of three-fold cross validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, original solver (*i.e.*, PicoSAT [Biere 2008]), and our baseline (*i.e.*, glucose) for each Sudoku problem

Figure 15 presents the solving time of each test case in the three-fold cross-validation compared 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 the following encodings for the three partitions: All three partitions encoded integer variables using One hot encoding. Two partitions used the sequential counter technique to solve at-most-one constraints and reversed the order of boolean variables. However, the other partition that contains relatively easy problems used binomial (naive) encoding and kept the original variable ordering. In all partitions, the integer optimization pass is enabled and the graph encoding optimization is deactivated. On average, Satune synthesized the encoding for each partition in about five days.

9.6 Killer Sudoku

Killer Sudoku extends Sudoku with cages. The Killer Sudoku encoding extends the Sudoku encoding 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 [Airobert 2016].

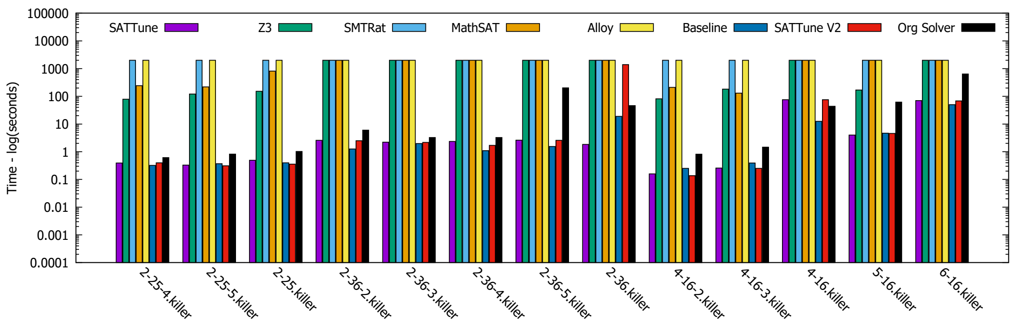


Fig. 16. SATUNE's test set results of three-fold cross-validation are compared with the execution time of Z3, SMTRat, MathSAT, Alloy, original solver (*i.e.*, PicoSAT [Biere 2008]), and the baseline (*i.e.*, glucose) for each Killer Sudoku problem

Figure 16 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 the following encoding strategies for three partitions. Each encoded integer variables using binary index. Two partitions kept the original variable ordering and the other one changed it to the order of variable creation. For two partitions, SATUNE encoded integer variables using the integer domain reduction pass and SATUNE could outperform the baseline encoding. However, this pass causes a minor slowdown for relatively easy problems. SATUNE kept graph encoding optimizations disabled for all partitions. On average, SATUNE synthesized each of these encodings in about two hours.

9.7 Deployment Phase

Recall from Section 5 that SATUNE has two phases: *learning* phase and deployment phase. Once the encoders are synthesized in the learning phase, the client can switch into SATUNE's *deployment* mode, and use the encoders to encode new problems. This section evaluates the performance of SATUNE in the **deployment** phase for JMCR, Dirk, and SyPet benchmarks. The ideal methodology is to, first, run the benchmarks with the baseline solver and measure the total end-to-end time, serialization time, and SAT solving time. Then, swap in SATUNE with the synthesized encoders and carrying out the same measurements and compare the numbers. This methodology works well for *deterministic* benchmarks. Many benchmarks including JMCR, Dirk, and SyPet are *non-deterministic* as: (1) they generate a series of SAT problems, (2) these problems often have many different solutions, and (3) the solution that is found for a SAT problem will can affect which problems are generated later. If one uses two different constraint solving frameworks, the client executions can quickly diverge and one constraint solving framework can randomly receive a much easier set of problems. To make results comparable across frameworks, we modified the clients to serialize out the generated constraint problems so that all constraint frameworks solve exactly the same set of problems. Figure 17 presents the break-down of the execution time that includes the application time, the baseline solving time, and SATUNE's solving time for three benchmarks of JMCR, Dirk, and SyPet. On average, SATUNE reduced the solving time by 94% in these three benchmarks.

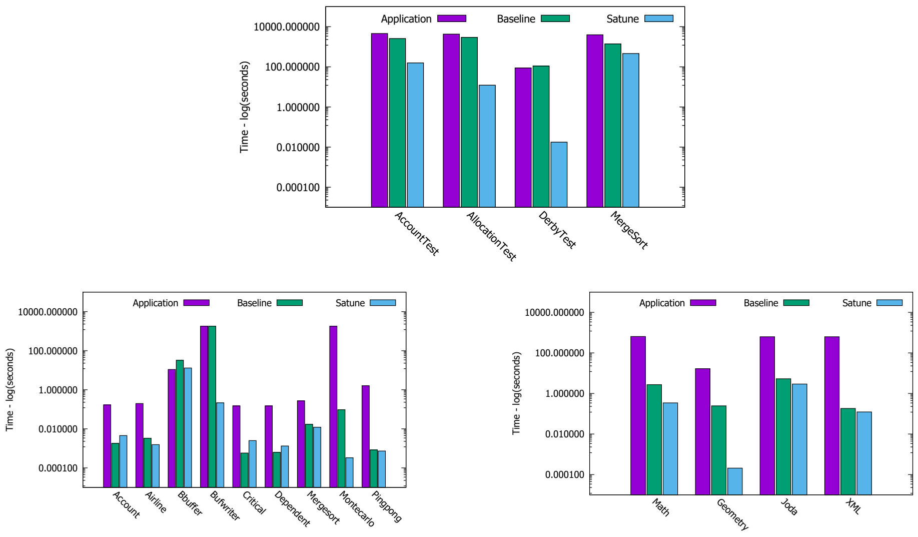


Fig. 17. The break-down of the end-to-end time for Dirk, JMCR, and SyPet. The application time is the time that is spent purely in the application code, *i.e.*, the total end-to-end time minus the serialization time and solving time.

10 Related Work

Alloy is a relational logic intended for describing structural properties [Jackson 2002]. It leverages Kodkod [Torlak and Jackson 2007], a SAT-based relational model finder, to translate its constraints into SAT. SATUNE's language differs from Alloy since it is intended to have a support commonly-used abstractions for which multiple good SAT encodings exist and the SATUNE language is in general a lower-level language that is primarily targeted towards supporting client applications. Kodkod also uses sophisticated optimizations including auto-compacting circuits and symmetry breaking to simplify the given constraints. However, contrary to SATUNE, it does not attempt to tune its encoding strategy for specific domains. Enfragmo [Aavani et al. 2012] has a high-level language extending first-order logic for solving combinatorial search problems. It has no support for order constraints and synthesizing domain-specific encodings. Some frameworks have high-level languages to hide the complexity of SAT solvers from the user without incorporating any domain-specific encodings and optimizations [Metodi and Codish 2012]. Other frameworks implement a collection of encodings and techniques without any tuning [Gecode 2016].

Much work has been done on developing encodings to SAT[Abío and Stuckey 2014; Bailleux and Boufkhad 2003; Chebiryak and Kroening 2008; Chen 2010; Gent and Lynce 2005; Samer and Veith 2009; Tamura et al. 2013, 2009; Tanjo et al. 2011, 2012]. Several different encodings are known for variables drawn from discrete sets [Biere et al. 2014; Björk 2009; Frisch and Giannaros 2010; Klieber and Kwon 2007b] 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 [Barrett et al. 2011; Bouton et al. 2009; Bruttomesso et al. 2008; Corzilius et al. 2015; de Moura and Bjørner 2008; Dutertre and de Moura 2006]. SMT generalizes boolean satisfiability by replacing boolean variables with predicates over a variety of 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 that the translation approach does not 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 solvers but rather to explore the benefits of automatically tuning encodings. Eager SMT solvers take a similar solving strategy to Satune and directly translate constraints from other theories into SAT [Brummayer and Biere 2009; Ganesh and Dill 2007; Jha et al. 2009]. Satune differs from this work in that it supports multiple encodings and is targeted towards automatically tuning encodings rather than supporting other constraint theories.

There is some prior work that attempts to tune encodings automatically [Brain et al. 2016; Inala et al. 2016]. OPTCNF [Inala et al. 2016], which is the closest work to Satune, extracts patterns from a given problem in a specific domain, and synthesizes encodings to convert bit-vector terms to low-level CNF clauses. Other work has developed techniques for automatically generating propagation complete encodings that optimize for unit propagation [Brain et al. 2016]. The key difference between Satune and the prior work is that the prior work focuses on different approaches for generating constraints for the same underlying representation (encoding) of the problem domain state into SAT variables, while Satune can also tune the underlying representations for the problem domain state into SAT variables.

Many constraint problems are solved by translation into SAT, including planning [Kautz and Selman 2006; Rintanen 2014; Robinson et al. 2008], circuit security [WINOGRAD and MAHMOODI 2009; Yu et al. 2017], SAT-based model checking [Burckhardt et al. 2007; Demsky and Lam 2015; Timm et al. 2017; Xie and Aiken 2005], test generation [Nguyen et al. 2015; Sen et al. 2013; Tanno et al. 2015], scheduling [Béjar and Manya 2000; Gent and Lynce 2005; Zhang et al. 2004], and verification [Chung 2017; Turkmen et al. 2015]. Encodings are mostly hard-coded in SAT-based frameworks and to our knowledge, no other framework automatically tunes encodings for clients.

AI techniques and learning have been widely used in the SAT solving domain [Gent et al. 2010; Haim and Walsh 2009; Inala et al. 2016; Liang et al. 2018; Musliu [n.d.]; Singh et al. 2009; Wu 2017]. There are frameworks that can learn the best solver for each problem type using bit-level information [Kurin et al. 2019; Liang et al. 2016; O'Mahony et al. 2008; Xu et al. 2008]. Prior work and Satune are largely complementary and it could be beneficial for Satune to employ them for client-specific SAT solvers or constraint rewrites [Singh and Solar-Lezama 2016] in addition to Satune's encoding optimizations. As future work, Satune can integrate with prior work to provide more information from its higher level abstraction in order to improve the learning process carried out in the bit-level.

11 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 performance 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.

Acknowledgments

We thank the anonymous reviewers for their thorough and insightful comments that helped us substantially improve the paper. This work is supported by the National Science Foundation grants CNS-1703598, OAC-1740210, CNS-1703598, CNS-1763172, CCF-2006948, CNS-2007737, and CNS-2006437, as well as ONR grants N00014-16-1-2913 and N00014-18-1-2037.

References

Amir Aavani. 2011. Translating Pseudo-Boolean Constraints into CNF. In *Theory and Applications of Satisfiability Testing - SAT 2011*, Karem A. Sakallah and Laurent Simon (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 357–359.

Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David Mitchell. 2012. Enfragmo: A System for Modelling and Solving Search Problems with Logic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Nikolaj Bjørner and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 15–22.

Ignasi Abío and Peter J Stuckey. 2014. Encoding linear constraints into SAT. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 75–91.

Airobert. 2016. *SAT Based Killer Sudoku*. https://github.com/UvA-KR16/KilerSudoku

Taufan Ardi. 2015. *SAT Based Sudoku Solver in Python*. https://github.com/taufanardi/sudoku-sat-solver

Gilles Audemard and Laurent Simon. 2009. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence* (Pasadena, California, USA). Morgan Kaufmann Publishers Inc., 399–404.

Gilles Audemard and Laurent Simon. 2014. Lazy Clause Exchange Policy for Parallel SAT Solvers. In *Theory and Applications of Satisfiability Testing (SAT '14)*. 197–205.

Gilles Audemard and Laurent Simon. 2015. Glucose and Syrup in the SAT Race 2015.

Olivier Bailleux and Yacine Boufkhad. 2003. Efficient CNF Encoding of Boolean Cardinality Constraints, Vol. 2833. 108–122. https://doi.org/10.1007/978-3-540-45193-8_8

Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. 2009. New Encodings of Pseudo-Boolean Constraints into CNF, Vol. 5584. 181–194. https://doi.org/10.1007/978-3-642-02777-2_19

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Ramón Béjar and Felip Manya. 2000. Solving the round robin problem using propositional logic. In *AAAI/IAAI*. 262–266.

Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), 75–97.

Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. 2014. Detecting Cardinality Constraints in CNF. In *Theory and Applications of Satisfiability Testing – SAT 2014*, Carsten Sinz and Uwe Egly (Eds.). Springer International Publishing, Cham, 285–301.

Magnus Björk. 2009. Successful SAT Encoding Techniques. *Journal on Satisfiability, Boolean Modeling, and Computation* 7 (July 2009), 189–201.

Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. 2011. Nitpicking C++ Concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*. 113–124. http://doi.acm.org/10.1145/2003476.2003493

Lucas Bordeaux and Joao Marques-Silva. 2012. Knowledge compilation with empowerment. In *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 612–624.

Thomas Bouton, Diego Caminha B de Oliveira, David Déharbe, and Pascal Fontaine. 2009. veriT: an open, trustable and efficient SMT-solver. In *International Conference on Automated Deduction*. Springer, 151–156.

Martin Brain, Liana Hadarean, Daniel Kroening, and Ruben Martins. 2016. Automatic Generation of Propagation Complete SAT Encodings. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 536–556.

Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS (Lecture Notes in Computer Science, Vol. 5505)*. Springer, 174–177.

Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. 2008. The MathSATÃĆÂă4 SMT Solver. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–303.

Sebastian Burckhardt, Rajeev Alur, and Milo MK Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 12–21.

Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost Warners. 1999. Radio Link Frequency Assignment. *Constraints* 4 (02 1999), 79–89. https://doi.org/10.1023/A:1009812409930

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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*. 209–224.

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

Yury Chebiryak and Daniel Kroening. 2008. An efficient SAT encoding of circuit codes. In *2008 International Symposium on Information Theory and Its Applications*. IEEE, 1–4.

Jingchao Chen. 2010. A new SAT encoding of the at-most-one constraint. *Proc. Constraint Modelling and Reformulation* (2010).

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

Koen Claessen and Niklas SÃűrensson. 2003. New Techniques that Improve MACE-style Finite Model Finding. In *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*.

Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. 2015. SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Theory and Applications of Satisfiability Testing – SAT 2015*, Marijn Heule and Sean Weaver (Eds.). Springer International Publishing, Cham, 360–368.

Martin Davis, George Logemann, and Donald W. Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397. https://doi.org/10.1145/368273.368557

Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (1960), 201–215. https://doi.org/10.1145/321033.321034

Leonardo de Moura and Nikolaj Bjørner. 2008. 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 (Lecture Notes in Computer Science, Vol. 4963).* Springer, 337–340.

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 20–36.

Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 321–332.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-sensitive Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 270–280.

Bruno Dutertre and Leonardo de Moura. 2006. *The Yices SMT solver.* Technical Report. SRI International.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017a. Component-based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017).* ACM, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. *SyPet.* https://github.com/utopia-group/sypet

Cormac Flanagan and Shaz Qadeer. 2002. Predicate Abstraction for Software Verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 191–202.

Alan M. Frisch and Paul A. Giannaros. 2010. 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.*

Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany). 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

Gecode. 2016. *Generic Constraint Development Environment.* https://www.gecode.org/

Ian Gent, Lars Kotthoff, Ian Miguel, and Peter Nightingale. 2010. Machine learning for constraint solver design–A case study for the alldifferent constraint. *arXiv preprint arXiv:1008.4326* (2010).

Ian P Gent and Inês Lynce. 2005. A SAT encoding for the social golfer problem. *Modelling and Solving Problems with Constraints* 2 (2005).

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* 213–223.

Hugo Musso Gualandi. 2012. *Using an industrial-strength SAT solver to solve the Hexiom puzzle.* https://github.com/hugomg/hexiom

Shai Haim and Toby Walsh. 2009. Restart strategy selection using machine learning techniques. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 312–325.

Steffen Hölldobler and Van-Hau Nguyen. 2013. An Efficient Encoding of the at-most-one Constraint.

Jeff Huang. 2015a. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA). 165–174. https://doi.org/10.1145/2737924.2737975

Jeff Huang. 2015b. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15).*

ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/2737924.2737975

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom). 337–348. https://doi.org/10.1145/2594291.2594315

Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. 2016. Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers. In *SAT (Lecture Notes in Computer Science, Vol. 9710)*. Springer, 302–320.

SMT LIB Initiative. 2018. *SMT-LIB The Satisfiability Modulo Theories Library*. http://smtlib.cs.uiowa.edu/index.shtml

Markus Iser, Mana Taghdiri, and Carsten Sinz. 2012. Optimizing MiniSAT Variable Orderings for the Relational Model Finder Kodkod. In *Theory and Applications of Satisfiability Testing – SAT 2012*, Alessandro Cimatti and Roberto Sebastiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 483–484.

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

Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. 2009. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Artithmetic. In *Proc. 21st International Conference on Computer-Aided verification (CAV) (Lecture Notes in Computer Science, Vol. 5643)*. 668–674.

Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

Henry Kautz and Bart Selman. 2006. SATPLAN04: Planning as satisfiability. *Working Notes on the Fifth International Planning Competition (IPC-2006)* (2006), 45–46.

Henry A. Kautz and Bart Selman. 2003. 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*. 1–18.

Will Klieber and Gihwon Kwon. 2007a. Efficient CNF Encoding for Selecting 1 from N Objects.

Will Klieber and Gihwon Kwon. 2007b. Efficient CNF Encoding for Selecting 1 from N Objects. In *Proceedings of the Fourth Workshop on Constraint in Formal Verification*.

Wolfgang Kuechlin and Carsten Sinz. 2000. Proving Consistency Assertions for Automotive Product Data Management. *J. Autom. Reasoning* 24 (02 2000), 145–163. https://doi.org/10.1023/A:1006370506164

Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. 2019. Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. arXiv:1909.11830 [cs.LG]

Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *JSAT* 7 (01 2010), 59–6.

Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. 2016. Learning Rate Based Branching Heuristic for SAT Solvers. In *SAT (Lecture Notes in Computer Science, Vol. 9710)*. Springer, 123–140.

Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. 2018. Machine Learning-Based Restart Policy for CDCL SAT Solvers. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 94–110.

InÃłs Lynce and JoÃńl Ouaknine. 2006. Sudoku as a SAT problem.

Panagiotis Manolios and Daron Vroon. 2007. Efficient Circuit to CNF Conversion. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*.

Norbert Manthey, Marijn JH Heule, and Armin Biere. 2012. Automated reencoding of boolean formulas. In *Haifa Verification Conference*. Springer, 102–117.

R. Martins, V. Manquinho, and I. Lynce. 2011. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*. 313–320.

Y. Matsunaga. 2015. Accelerating SAT-based Boolean matching for heterogeneous FPGAs using one-hot encoding and CEGAR technique. In *The 20th Asia and South Pacific Design Automation Conference*. 255–260.

Amit Metodi and Michael Codish. 2012. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming* 12, 4-5 (2012), 465–483.

Nysret Musliu. [n.d.]. Applying Machine Learning for Solver Selection in Scheduling: A Case Study. ([n. d.]).

Cuong Nguyen, Hiroaki Yoshida, Mukul Prasad, Indradeep Ghosh, and Koushik Sen. 2015. Generating Succinct Test Cases Using Don't Care Analysis. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. 2008. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*. 210–216.

Uwe Pfeiffer, Tomas Karnagel, and Guido Scheffler. 2013. A Sudoku-Solver for Large Puzzles using SAT. In *LPAR-17-short. short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning. (EPiC Series in Computing, Vol. 13)*, Andrei Voronkov, Geoff Sutcliffe, Matthias Baaz, and Christian Ferm\"uller (Eds.). EasyChair, 52–57. https://doi.org/10.29007/79mc

Jussi Rintanen. 2014. Madagascar: Scalable planning with SAT. *Proceedings of the 8th International Planning Competition (IPC-2014)* 21 (2014).

Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. 2008. A Compact and Efficient SAT Encoding for Planning.. In *ICAPS*. 296–303.

Marko Samer and Helmut Veith. 2009. Encoding treewidth into SAT. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 45–50.

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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*. 263–272.

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. 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* (Philadelphia, PA, USA). 693–706.

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 1–16.

Rishabh Singh, Joseph P. Near, Vijar Ganesh, and Martin Rinard. 2009. *AvatarSAT: An Auto-Tuning Boolean SAT Solver*. Technical Report MIT-CSAIL-TR-2009-039. Massachusetts Institute of Technology.

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

Carsten Sinz. 2005. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Principles and Practice of Constraint Programming - CP 2005*, Peter van Beek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 827–831.

Naoyuki Tamura, Mutsunori Banbara, and Takehide Soh. 2013. Compiling pseudo-boolean constraints to SAT with order encoding. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, 1020–1027.

Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. 2009. Compiling finite linear CSP into SAT. *Constraints* 14, 2 (2009), 254–272.

Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. 2011. A compact and efficient SAT-encoding of finite domain CSP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 375–376.

Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. 2012. Azucar: A SAT-Based CSP Solver Using Compact Order Encoding. In *Theory and Applications of Satisfiability Testing – SAT 2012*, Alessandro Cimatti and Roberto Sebastiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 456–462.

Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. 2015. TesMa and CATG: automated test generation tools for models of enterprise applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 717–720.

Nils Timm, Stefan Gruner, and Prince Sibanda. 2017. Model Checking of Concurrent Software Systems via Heuristic-Guided SAT Solving. In *International Conference on Fundamentals of Software Engineering*. Springer, 244–259.

Emina Torlak and Daniel Jackson. 2007. 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*. 632–647.

Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking Axiomatic Specifications of Memory Models. In *Proceedings of the 2010 Conference on Programming Language Design and Implementation*. 341–350. http://doi.acm.org/10.1145/1809028.1806635

Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. 2015. Analysis of XACML policies with SMT. In *International Conference on Principles of Security and Trust*. Springer, 115–134.

Peter JM Van Laarhoven and Emile HL Aarts. 1987. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 7–15.

Joost P. Warners. 1998. A linear-time transformation of linear inequalities into conjunctive normal form. *Inform. Process. Lett.* 68, 2 (1998), 63 – 69. https://doi.org/10.1016/S0020-0190(98)00144-6

Tjark Weber. 2005. A SAT-based Sudoku Solver. In *LPAR-12, The 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Short Paper Proceedings*, Geoff Sutcliffe and Andrei Voronkov (Eds.). 11–15.

TED WINOGRAD and HAMID MAHMOODI. 2009. Programmable Gates Using Hybrid CMOS-STT Design to Prevent IC Reverse Engineering. (2009).

Haoze Wu. 2017. Improving SAT-solving with Machine Learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 787–788. http://doi.acm.org/10.1145/3017680.3022464

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

Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research* 32 (2008), 565–606.

Cunxi Yu, Xiangyu Zhang, Duo Liu, Maciej Ciesielski, and Daniel Holcomb. 2017. Incremental SAT-based reverse engineering of camouflaged logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 10 (2017), 1647–1659.

Hantao Zhang, Dapeng Li, and Haiou Shen. 2004. A SAT Based Scheduler for Tournament Schedules.. In *SAT*.

Neng-Fa Zhou. 2020. Yet Another Comparison of SAT Encodings for the At-Most-K Constraint. arXiv:2005.06274 [cs.LO]

Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale System Code. In *Proceedings of European Computer System Conference*.