Lightweight Online Learning for Sets of Related Problems in Automated Reasoning

Haoze Wu
Stanford University
Stanford, CA, USA
haozewu@stanford.edu

Christopher Hahn
Stanford University
Stanford, CA, USA
hahn@cs.stanford.edu

Florian Lonsing
Unaffiliated
Linz, Austria
fml@florianlonsing.com

Makai Mann
MIT Lincoln Laboratory
Lexington, MA, USA
makai.mann@ll.mit.edu

Raghuram Ramanujan
Davidson College
Davidson, NC, USA
raramanujan@davidson.edu

Clark Barrett
Stanford University
Stanford, CA, USA
barrett@cs.stanford.edu

***Abstract*—We present Self-Driven Strategy Learning (SDSL), a lightweight online learning methodology for automated reasoning tasks that involve solving a set of related problems. SDSL does not require offline training, but instead automatically constructs a dataset while solving earlier problems. It fits a machine learning model to this data which is then used to adjust the solving strategy for later problems. We formally define the approach as a set of abstract transition rules. We describe a concrete instance of the SDSL calculus which uses conditional sampling for generating data and random forests as the underlying machine learning model. We implement the approach on top of the KISSAT solver and show that the combination of KISSAT+SDSL certifies larger bounds and finds more counter-examples than other state-of-the-art bounded model checking approaches on benchmarks obtained from the latest Hardware Model Checking Competition.**

I. Introduction

Many automated reasoning tasks involve solving a set of related problems that share common structure. For example, in Bounded Model Checking [1], [2], one repeatedly checks deeper and deeper unrolls of a transition system for a property violation. In iterative (e.g., counter-example-guided) abstraction refinement [3], one verifies a condition on an increasingly precise model of a system. And in symbolic execution [4], one analyzes the possible outcomes of a program on symbolic inputs by incrementally adding path conditions. Often, a fixed, predetermined high-level solving strategy (e.g., the choice of a solver and its parameter settings) is used in this iterative solving process. However, given the structural similarity within the set of problems, a natural question is: *can we leverage information gathered while solving earlier problems to adjust the solving strategy for later problems on the fly?*

Adapting high-level solving strategies for particular problem distributions, a practice often termed *meta-algorithmic design* [5], is already a well-established technique. Automated configuration techniques [6], which optimize an algorithm's performance on a given set of problems, are widely used among practitioners. Per-instance algorithm selection techniques (e.g., SATzilla [7]) train machine learning models to predict a suitable strategy for a given problem based on its structural characteristics. More recently, attempts to improve constraint solving using deep learning also generally follow the paradigm of choosing a particular problem distribution (which can be either broad, such as main-track benchmarks from SAT competitions [8], or narrow, such as graph coloring problems [9]), gathering training data using instances in that distribution, and learning a strategy over the data.

A shared, and arguably undesirable,trait of the aforementioned approaches is that they all involve an *offline phase*, in which significant time and (often manual) effort are required to obtain an optimized solving strategy that can be used on new unseen problems. While the cost of the offline phase might be justified by the potential performance gain in the long run, the very distinction between an offline phase and an online phase already makes the reasoning less *automated*.

Our first observation is that for an automated reasoning procedure whose execution involves solving a set S of related problems, it is possible to move the meta-algorithmic design online, as part of the solving, by narrowing the scope of *problem distribution* all the way down to S itself. More concretely, we propose to solve some of the problems in S not just once, but multiple times, each time using a different solving strategy from a space of candidates. The strategies used for solving later problems are selected based on information recorded during the multiple runs (e.g., using lightweight machine learning techniques). We present this general method, which we term **S**elf-**D**riven **S**trategy **L**earning (SDSL), as a set of transition rules, which can be used to model different ways of carrying out on-the-fly meta-algorithmic design.

Though there are many possible ways to instantiate SDSL, we focus on a strategy space consisting of one fixed solver whose parameters are allowed to vary. One obvious method for exploring this strategy space involves choosing the first few problems, optimizing the parameter settings for them with a standard tuning procedure, and then using the optimized strategy for future problems. However, a drawback of this approach is that it only operates on a fixed set of problems and cannot explicitly take into account possible relationships

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_10

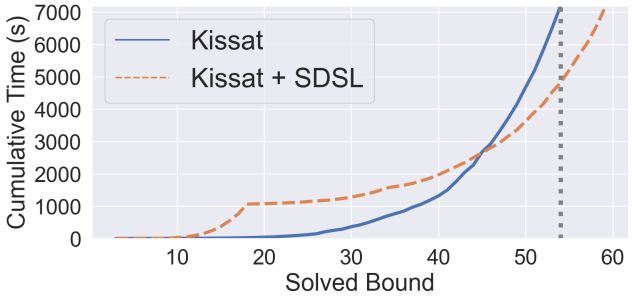

Fig. 1. Executions of Bounded Model Checking with and without SDSL on a hardware model checking benchmark (`arb.n2.w128.d64`). The trajectories show the cumulative wall-clock time required to certify a bound.

between this set of problems and later problem instances.

To allow for such flexibility, our second observation is that a tuning procedure can be viewed, not as a way to select a specific solving strategy, but instead as a means of creating a dataset, where each data point is a pair consisting of a particular solving strategy and a particular problem in the problem set. A machine learning model is trained on this dataset to predict the effect of a given solving strategy on a given problem. The model can then be used as an oracle to select the solving strategy for future problems.

We apply our methodology to a case study of Bounded Model Checking (BMC) problems. We study different SDSL instantiations and compare against existing model checkers. On satisfiable and unsolved bitvector benchmarks from the latest Hardware Model Checking Competition [10], our approach consistently boosts the performance of a BMC-procedure built on top of the KISSAT SAT solver [11]. Additionally, it compares favorably against state-of-the-art open-source model checkers AVR [12] and PONO [13], contributing several unique solutions and speeding up many more. A preview on a single benchmark is shown in Fig. 1. We see that SDSL invests time learning a good solving strategy in the beginning, which results in better performance when solving later problems.

We summarize our main contributions as follows:

1) we propose to move meta-algorithmic design online as part of solving a set of related problems;
2) we propose a general methodology called Self-Driven Strategy Learning and present it formally as a set of transition rules; and
3) we implement our approach and apply it to Bounded Model Checking problems, where it shows significant improvement over other state-of-the-art approaches.

The rest of the paper is organized as follows. After a discussion of related work in Sec. II, we first define a basic calculus for iteratively solving a set of related problems in Sec. III. Next, SDSL is presented as an extension of this calculus with additional rules for data collection, learning, and strategy updates in Sec. IV. We explore the design space of SDSL in Sec. V, discussing how to sample training data and which machine learning models to use. In Sec. VI, we describe in detail the instantiation of SDSL for Bounded Model Checking. In Sec. VII we present experimental results on Bounded Model Checking problems, and finally, we conclude with an account of current limitations and future directions in Sec. VIII.

II. Related Work

Our approach is inspired and informed by several existing lines of work.

Incremental solving: A well-established paradigm for exploiting structural similarity is *incremental solving* [14] in which each new query to a solver can be made by modifying the most recent formulas asserted in the previous query without resetting the solver. SDSL is an orthogonal approach for leveraging structural similarity and may be preferable in cases where incremental solving is not beneficial or not supported.

In principle, the two can be combined. A straightforward way would be to switch to incremental solving mode after fixing a solving strategy. A tighter combination would require updating strategies *between* incremental invocations, something that current solvers typically do not allow. In case one wants to both switch solvers on the fly and leverage incrementality, proof-transfer techniques such as solver state migration [15] are likely needed. For our particular BMC case study, we found that the direct use of an incremental SAT/SMT solver has mixed effects on performance (see the extended version [16] of the paper). This suggests that it might be worth revisiting BMC-specific incremental solving techniques such as conflict clause shifting [17] before investigating the interplay between SDSL and incremental solving in BMC, which we leave for future work.

Automated Configuration: Our work is motivated by the success of offline meta-algorithmic design approaches such as automated configuration [6], [18], [19] and per-instance algorithm selection [7], [20], [21]. Automated configuration focuses on finding (near) optimal parameter settings of an algorithm for a fixed set of problems, using either local search or performance prediction techniques [22]–[24]. Per-instance algorithm selection techniques were among the first to utilize machine learning to improve constraint solving. The idea is to train an oracle to predict the performance (e.g., runtime) of a set of candidate algorithms on a formula based on its structural characteristics. SDSL differs from both approaches primarily in that it moves meta-algorithmic design *online*.

ML for AR: Machine Learning has been applied in multiple ways to expedite a variety of automated reasoning tasks, including satisfiability checking [8], [9], [25]–[31], Mixed-Integer Convex Programming [32], [33], program/function synthesis [34]–[36], and symbolic execution [37], [38].

While most existing techniques require an offline training phase, the general idea of using online learning also appears in previous work. The Conflict-Driven Clause Learning paradigm itself can be viewed as online learning. In the MapleSAT solver [27], [28], branching is formulated as a multi-armed bandit problem where the estimated reward of each arm (i.e., variable) is maintained and updated throughout the solving. This reinforcement learning interpretation of a dynamic

$$\frac{i < \mathbf{K} \quad \mathbf{check}(f_i, v) = \text{UNSAT}}{i, v \Longrightarrow i+1, v} \textbf{ (Next)}$$

$$\frac{i = \mathbf{K} \quad \mathbf{check}(f_i, v) = \text{UNSAT}}{i, v \Longrightarrow \text{FAIL}} \textbf{ (Failure)}$$

$$\frac{\mathbf{check}(f_i, v) = \text{SAT}}{i, v \Longrightarrow \text{SUCCESS}} \textbf{ (Success)}$$

Fig. 2. Transition rules for solving a set of related problems. The starting configuration is $\langle 1, v \rangle$.

$$\frac{v_\text{s} \in \mathcal{V} \quad j \leq i \quad c = \mathbf{cost}(f_j, v_\text{s})}{i, v, D, T \Longrightarrow i, v, D \cup \{\langle v_\text{s}, j, c \rangle\}, T} \textbf{ (Collect)}$$

$$\frac{T' = \mathbf{fit}(D)}{i, v, D, T \Longrightarrow i, v, D, T'} \textbf{ (Train)}$$

$$\frac{\mathcal{V}_\text{s} \subseteq \mathcal{V} \quad v' = \arg\min_{v_\text{s} \in \mathcal{V}_\text{s}} T(v_\text{s}, i)}{i, v, D, T \Longrightarrow i, v', D, T} \textbf{ (Strategize)}$$

Fig. 3. Additional transition rules for Self-Driven Strategy Learning.

branching heuristic is perhaps inspired by the study [39] of the popular VSIDS [40] branching heuristic and its later variants [41], which also track a score for each variable during the solving. In contrast to this direction of online learning, SDSL operates on a set of related problems rather than on a single instance. Moreover, SDSL focuses on selecting from a set of existing strategies rather than inventing new ones.

To conclude this section, we remark that in practice, offline learning, "in-solver" online learning, and SDSL could be combined to solve a set of related problems. For example, one could choose to use SAT/SMT solvers with built-in learning components, set the initial solving strategy using offline learning, and then use SDSL to further customize the strategy online. The exploration of such combinations is beyond the scope of this paper, but is a promising future direction.

III. Solving Sets of Related Problems

In this section, we present a simple calculus, *SR*, for iteratively solving a set of related problems. Let $\mathcal{F} = \{f_1, \ldots, f_\mathbf{K}\}$ be a set of $\mathbf{K}$ related formulas. Assume we have a function $\mathbf{check} : \mathcal{F} \times \mathcal{V} \to \{\text{SAT}, \text{UNSAT}\}$, which takes as input a formula $f \in \mathcal{F}$ and a solving strategy v (from a set $\mathcal{V}$ called the *strategy space*) and returns either SAT (satisfiable) or UNSAT (unsatisfiable). Additionally, assume that we can stop once any formula is SAT.

The rules of the basic *SR* calculus are shown in Fig. 2. The rules operate over a *configuration*, which is either one of the distinguished symbols $\{\text{SUCCESS}, \text{FAIL}\}$ or a tuple $\langle i, v \rangle$, where $i \in [1, \mathbf{K}]$ is the current formula index and $v \in \mathcal{V}$ is the current solving strategy. The rules describe the conditions under which a certain configuration can transform into another configuration. The **Next** rule says that if the current formula is unsatisfiable and the maximal index $\mathbf{K}$ has not been reached yet, then the current index will be increased. On the other hand, if the current formula is unsatisfiable and it is the last formula, the **Failure** rule transitions the system to the FAIL configuration. The **Success** rule states that SUCCESS can be reached when the current formula is satisfiable.[1]

An *SR-execution* is a sequence of configurations that respect the rules in *SR*. Note that no rule updates the solving strategy v. We augment the calculus with strategy updates next.

[1]The conditions for progress and termination are inspired by BMC but are applicable in other settings when solving related problems.

Given two configurations C, C', we use $C \vdash C'$ to denote C can transition (in one or more steps) to C'. We state the following two propositions which are straightforward to verify.

Proposition 1 (Soundness and Completeness): $\mathcal{F}$ contains a satisfiable formula if and only if $\langle 1, v \rangle \vdash \text{SUCCESS}$.

Proposition 2 (Termination): There exist no infinite *SR*-executions.

IV. Self-Driven Strategy Learning

A. Informal Presentation

Self-Driven Strategy Learning (SDSL) attempts to learn, on the fly, which solving strategy among a set of candidates $\mathcal{V}$ to use for each formula in $\mathcal{F}$. Learning is based on data gathered during solving. To obtain the data, we occasionally solve a formula multiple times, each time with a different strategy in $\mathcal{V}$. For a strategy v_s, we record its effect, $c \in \mathbb{R}$, when solving f_i by creating a data point $\langle v_\text{s}, i, c \rangle$, where c is a measure of the cost of the strategy. For example, c could be the total run time required to solve f_i.

Given such a dataset, an oracle $T : \mathcal{V} \times [1, \mathbf{K}] \mapsto \mathbb{R}$ is trained to predict the cost of a given strategy when run on a given formula. When solving a new formula, we select the one that T predicts will be most effective, and as more data is collected with each call to $\mathbf{check}$, T is updated.

An essential characteristic of SDSL is that the training data is gathered for a specific, and *a priori* unknown, set of formulas in an *online* and *automatic* manner, as part of the solving process. This approach has two challenging implications. The learning process must not incur a large overhead; otherwise, insufficient time is left for actual solving. Additionally, the choice of $\mathcal{V}$ is crucial as it must be large enough to contain good candidate strategies but also not too large to explore. We address these challenges in Sections V and VI, respectively.

B. Formal presentation

Formally, we present SDSL as an extension of *SR*. Configurations are as in *SR* except that tuple configurations $\langle i, v, D, T \rangle$ have two additional components (assumed to be left unchanged by rules in *SR*): $D \in \mathcal{P}(\mathcal{V} \times [1, \mathbf{K}] \times \mathbb{R})$ is a dataset, each of whose members records the result of running a single strategy on a single formula; and $T : \mathcal{V} \times [1, \mathbf{K}] \mapsto \mathbb{R}$ is an oracle (e.g., a machine learning model) that predicts the cost of a strategy on a formula. Initially, D is empty, and T is arbitrary (e.g., always return 0). The additional transition rules of SDSL are described in Fig. 3.

The **Collect** rule samples a strategy $v_s \in \mathcal{V}$, evaluates its cost when solving f_j, and augments D with this new data. The rule is parameterized by a function $\mathbf{cost} : \mathcal{F} \times \mathcal{V} \mapsto \mathbb{R}$. The **Train** rule updates the oracle T with a new one trained on the current dataset D. It is parameterized by a machine learning algorithm $\mathbf{fit}$ (e.g., k-NN, tree ensemble, deep learning, etc.). Finally, the **Strategize** rule updates the current strategy by sampling a set of strategies $\mathcal{V}_s$ from $\mathcal{V}$ and choosing the one with the best predicted cost for the current index i. The extended calculus is still sound and complete (i.e., Proposition 1 still holds). Since the added rules can effectively be applied at any time, Proposition 2 only holds if we allow only a finite number of applications of the new rules.

Note that in the **Collect** rule, the results of solving the formula f_j are discarded, as f_j must have been solved already in some previous application of the **Next** rule. It is possible to extend the SR calculus to allow UNKNOWN results from $\mathbf{check}$, but the completeness property would be lost.

A reasonable strategy for applying SDSL rules is as follows:

1) After every application of **Next**, issue one *learning epoch*; that is, apply **Collect** m times on the current problem, then apply **Train**;
2) If **Train** has been applied at least once, apply **Strategize** whenever i is updated;
3) If the *estimated learning time* exceeds some threshold n, override the first policy and do not issue any more learning epochs;
4) Terminate whenever **Success** or **Failure** applies.

The estimated learning time is calculated as the time spent on learning so far plus $m \cdot t$, where t is the runtime of solving the current problem using the current strategy. If $|\mathcal{V}|$ is small, it may be reasonable to use $m = |\mathcal{V}|$ and try each strategy from $\mathcal{V}$. In the more general scenario where $m \ll |\mathcal{V}|$, the choice of which samples to use impacts the quality of the dataset. We discuss this choice and present a conditional sampling procedure in Sec. V-A. The purpose of restricting the training time in step three is to ensure that training does not dominate the total time taken. This simple criterion for when to stop learning already works reasonably well in practice. We leave the exploration of more sophisticated heuristics to future work.

V. Design Space in SDSL

This section discusses the design space in the implementation of SDSL and proposes solutions to the following questions: 1) How should training data be sampled? 2) Which machine learning model and training algorithm should be used? The solutions we propose focus on the case where a strategy is simply a set of values for a specific set of solver parameters. In this case, the strategy space is the cartesian product of p sub-strategy spaces, each representing a single parameter: $\mathcal{V} = \mathcal{V}_1 \times \ldots \times \mathcal{V}_p$. The set of possible values for each parameter can vary (e.g., parameter values could be Booleans, strings, or numbers), but for now we assume each $\mathcal{V}_i$ is finite.

A. Gathering Informative Training Data

To make the most informed decision, we could try all candidate strategies on all previously considered problems, but this is infeasible when $|\mathcal{V}|$ is large. In the following, we consider the scenario where m samples are drawn from a strategy space $\mathcal{V}$, where $m \ll |\mathcal{V}|$.

In this restrictive setting, we need to ensure our dataset contains a sufficient number of low-cost strategies (if there are any). Sampling uniformly is unlikely to achieve this goal because in practice, many or most candidate strategies could have high cost. For this reason, we propose to *explicitly* favor low-cost strategies in the sampling process. One way to do this is by using Markov-Chain Monte-Carlo (MCMC) sampling, which in our setting can be used to generate a sequence of solving strategies with the desirable property that in the limit, strategies with the lowest cost are most frequently drawn. A popular MCMC method is the Metropolis-Hastings (M-H) Algorithm [42], instantiated in the context of SDSL as follows:

1) Choose a current strategy v;
2) Propose to replace the current strategy with a new one v', which comes from a *proposal distribution* $q(v'|v)$;
3) If $\mathbf{cost}(f, v') \leq \mathbf{cost}(f, v)$, accept v' as the current strategy;
4) Otherwise, accept v' as the current strategy with some probability $a(v \to v')$ (e.g., a probability inversely proportional to the increase in cost);
5) Go to step 2.

This process is repeated until m samples are drawn. Importantly, under this scheme, a proposal that results in lower cost is always accepted, while a proposal that does not may still be accepted. This means that the algorithm greedily moves to a better strategy whenever possible, but also has a means for escaping local minima. In our implementation, the acceptance probability is computed using a common method [43] described as follows. We first transform $\mathbf{cost}(f, v)$ into a probability distribution $p(v)$:

$$p(v) \propto \exp(-\beta \cdot \mathbf{cost}(f, v)) \ ,$$

where $\beta > 0$ is a configurable parameter. The acceptance probability is then computed as:

$$\begin{aligned} a(v \to v') &= \min\left(1, \frac{p\,(v')}{p\,(v)}\right) \\ &= \min\left(1, \exp\left(\beta \cdot (c - c')\right)\right) \ , \end{aligned}$$

where $c = \mathbf{cost}(f, v)$ and $c' = \mathbf{cost}(f, v')$. Under this acceptance probability, the larger that c' is compared to c, the lower the probability to accept. On the other hand, the larger β is, the more reluctant we are to move to a worse proposal.

To ensure the aforementioned convergence property of MCMC in the limit, the proposal distribution must be both *symmetric* and *ergodic*.[2] For discrete search spaces, a common proposal distribution is the symmetric random walk, which

[2] A proposal distribution q is symmetric if $q(v'|v) = q(v|v')$ for any $v, v' \in \mathcal{V}$ and is ergodic if there is a non-zero probability of reaching a strategy $v \in \mathcal{V}$ from any other strategy $v' \in \mathcal{V}$ in a finite number of steps.

moves to one of the *neighbors* of the current sample with equal probability. For our strategy space, we define the neighbors of a strategy as all strategies for which exactly k parameter values are different. We use $k = 1$ in our implementation.

Note that MCMC sampling can be used not only in the data collection process, but also in the **Strategize** rule (i.e., to choose $\mathcal{V}_s \subseteq \mathcal{V}$). Since in this case we use the machine learning model as an oracle of $\mathbf{cost}$ (which is much cheaper than calling a solver), a larger sample size is affordable.

The sampling scheme presented above largely coincides with many local search approaches used in the automated configuration literature [5]. Borrowing more insights from that literature and devising more sophisticated sampling schemes are interesting directions for future work.

B. Lightweight Online Learning

In the online setting, the machine learning model must generalize from sparse data in limited time. This means the model needs to be both robust against outliers and efficient to train. Training a neural network from scratch, for example, is likely unsuitable, because it requires large amounts of data and, depending on the architecture, could be costly to train. On the other hand, lightweight ensemble models, which consist of a set of sub-models with different strengths and weaknesses, are well-suited for SDSL.

Our data is what is often called *tabular data*, that is, it can be represented as a table with rows and columns, where each row corresponds to a sample, and each column corresponds to a feature. When the strategy space consists of parameter settings, each sample has $p+2$ features: the p parameters, the problem index, and the cost. Tree-based ensemble methods such as *random forests* are generally considered to be a good match for such data [44].

A random forest consists of a set of B *regression trees* $\{f_1, \ldots, f_B\}$. Each tree is trained independently by sampling data from the data set D. A regression tree makes a prediction by following a path from the root node to a leaf node, based on the values of the input features, and returning the cost associated with the leaf node (which generally is the average of the costs of the training points that map to that leaf node). The predictions of a random forest f are computed by averaging the predictions of the individual trees in f.

A random forest is both efficient to train and efficient for prediction [45]. The time complexity of training a random forest with B trees is $O(B \cdot m \cdot n \cdot \log m)$, where m is the number of data points and n is the number of input features. The inference time complexity for a random forest is $O(B \cdot n)$.

Many machine learning algorithms are themselves parameterized and the performance of the model depends on a good choice of the hyperparameters. For a tree-based algorithm like random forest, an important hyperparameter is the maximal depth allowed for each individual regression tree: too shallow, and the model's prediction will be inaccurate; too deep, and the model might overfit to outliers.[3] The standard way to find suitable values of the hyperparameters is via (cross-)validation: split the data into a training set and a validation set, train models with different hyperparameters on the training set, evaluate them on the validation set, and pick the best one. However, in the SDSL setting where data is already sparse, validation is less feasible because it is hard to make sure that both the training set and the validation set are representative of the input space. Instead, we propose the following pragmatic heuristic: start by training a random forest with shallow trees and then retrain with incrementally deeper trees as needed until the training score is high enough.

VI. Case Study: Bounded Model Checking

Bounded Model Checking (BMC) [1], [2], [17] is a well-known technique for checking whether a property P holds along bounded executions of a given system M. The algorithm starts by checking all executions of length k; if no counter-example is found, k is increased and the system checked until either a counter-example is found, the problem becomes intractable, or some upper bound on k is exceeded.

BMC is useful in practice for at least two reasons. First, it is often the most efficient way to find counter-examples (if they exist) when trying to prove that a system has a particular property. Second, when techniques capable of providing a full (i.e. unbounded) proof fail (which is often the case in practice), BMC still establishes a certain confidence in the system by providing formal guarantees for bounded executions. The larger the *certified bound*, the stronger the guarantee.

A basic BMC formula for checking whether a property P holds for a system M along executions of length k is:

$$I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge \left(\bigvee_{i=0}^{k} \neg P_i\right) \ ,$$

where I_0 represents the initial state of M, $\rho(i, i+1)$ represents how the system evolves in a single step, and P_i represents the property at step i in the execution. This formula is satisfiable iff there is an execution of length less than or equal to k such that the property P does not hold at the end of the execution.

In practice, when the bound is increased, additional constraints are added stating that previously checked states are safe (in order to prune the search space). For example, suppose the check for bound k' is unsatisfiable. To check bound $k > k'$, we use the following formula:

$$bmc(k', k) := I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge \bigwedge_{i=0}^{k'} P_i \wedge \left(\bigvee_{i=k'+1}^{k} \neg P_i\right) \ .$$

We use BMC as a case study for our approach. For a given system and property, we solve the following set of problems:

$$\mathcal{F} = \{bmc(k-s, k) \mid k = i \cdot s, 1 \leq i \leq \mathbf{K}\} \ ,$$

[3]Instead of tuning the tree depth while fixing the number of trees, one could alternatively grow deep trees and tune the number of trees (to be large enough). However, this makes training and prediction much more costly.

where s is the *step size*. We focus on hardware model checking problems where the set of formulas to solve is in the theory of bitvectors [46]. We use standard techniques to encode the bitvector problems as Boolean satisfiability (SAT) problems [47]. Thus, $\mathcal{F}$ is a set of Boolean formulas, and we can implement $\mathbf{check}$ using an off-the-shelf SAT solver. We use the state-of-the-art KISSAT SAT solver [11].

In the following, we discuss the choice of the cost function $\mathbf{cost}$ and the strategy space $\mathcal{V}$ for this case study.

A. Choosing the Cost Function

One plausible cost function for a strategy v and a formula f_i is the ratio of the runtime to that of some default strategy v_0, i.e., if the runtime is t with strategy v and t_0 with v_0, then $\mathbf{cost}(f_i, v) = \frac{t}{t_0}$. While this definition works in practice, the use of runtime makes SDSL's behavior non-deterministic across different runs. This is undesirable for many reasons, including experimental reproducibility. Therefore, we instead use the *number of conflicts* generated by the SAT solver, which is accepted as a good proxy for runtime [48]. Given the same parameter settings, the number of conflicts generated by KISSAT on the same problem is deterministic.

B. Choosing the strategy space

As discussed in Sec. IV-B, the choice of the strategy space is crucial to the effectiveness of SDSL. KISSAT has over 90 configurable parameters, so considering all of them is impractical. One plausible approach is to rely on expert/domain knowledge and empirical studies to identify a reasonable set of parameters to consider. We follow this approach to define two strategy spaces for KISSAT.

The first one, $\mathcal{V}_{exp}$ (Table I), is based on a study by Dutertre [49] on the effect of SAT solver parameters on bitvector problems.[4] We allow two possible values for each parameter, the default one and an alternative one. For options that were found to be beneficial in [49], we include the corresponding parameter in KISSAT and for non-Boolean parameters, we set the alternative value to be more aggressive;[5] for Boolean parameters, we simply set the alternative value to be the opposite of the default. In total, $\mathcal{V}_{exp}$ contains 8192 (2^{13}) possible parameter settings.

The second strategy space, $\mathcal{V}_{dev}$ (Tab. II), is based on suggestions made by the developer of KISSAT.[6] It contains significantly fewer possible parameter settings (216).

[4]We consider all options considered in Table 2 of [49], except four: "lucky" and "walk" control procedures that find satisfying assignments independent of the main search; "scan-index" is not available in KISSAT; and "compacting" is a data-structure optimization that we do not believe has strong correlations with the number of conflicts. Noting that [49] does not consider any options related to branching, we additionally consider the *bumpreasonsrate* parameter, which controls the eagerness of reason-side literal bumping [27] and reportedly [11], [50] has significant impact on SAT Competition benchmarks.

[5]The alternative values are selected as follows: **int* parameters are divided by 10; **lim* parameters are divided by 100; and **effort* parameters are doubled. This works well in practice, and in further testing, setting the parameters to other reasonable values did not significantly alter the overall results. In the future, it might be advisable to obtain expert knowledge also on the specific values of the parameters.

[6]See https://github.com/arminbiere/kissat/issues/25

TABLE I
THE STRATEGY SPACE $\mathcal{V}_{exp}$ BASED ON [49].

KISSAT option	default	alternative
and	1	0
bumpreasonsrate	10	1
chrono	1	0
eliminateint	500	50
eliminateocclim	2000	20
forwardeffort	100	200
ifthenelse	1	0
probeint	100	10
rephaseint	1000	100
stable	1	0
substituteeffort	10	20
subsumeocclim	1000	10
vivifyeffort	100	200

TABLE II
THE STRATEGY SPACE $\mathcal{V}_{dev}$.

KISSAT option	default	alternative(s)
chrono	1	0
phase	1	0
stable	1	0, 2
target	1	0, 2
tier1	2	1
tier2	6	3, 9

Designing principled ways to automatically construct the strategy space (e.g., using techniques for assessing parameter importance [51]) is an important direction for future work.

C. Implementation

We implemented an SDSL-based BMC procedure in PYTHON3.[7] Our prototype takes as input a model checking problem in the BTOR/BTOR2 format [52], [53] and can run BMC on that input with or without SDSL. We implemented SDSL following the strategy described in Sec. IV-B. The BMC step size and the maximal bound are also command-line arguments. Additional input arguments include:

1) $\mathcal{V}$: path to a CSV file representation of the strategy space (e.g., Tabs. I and II);
2) n: the time budget for the learning epochs (see Sec. IV-B), by default 15% of the total time limit;
3) m: the number of samples to draw per learning epoch, by default 100;
4) The number of samples to draw in the **Strategize** rule, by default 500;
5) The number of trees in the random forest, by default 50;
6) The initial tree depth, by default a third of the number of parameters in $\mathcal{V}$;
7) The random seed, by default 0.

The default values are used in all experiments unless otherwise specified.

The formula $bmc(k', k)$ is generated online by first creating a bitvector formula using the PONO Model Checker [13],

[7]Available at https://github.com/anwu1219/sdsl/

then bit-blasting it into a SAT formula using the BOOLECTOR solver [54]. The versions of the solvers are reported in the extended version [16] of the paper. Our prototype does not leverage incrementality for reasons discussed in Sec. II. We use the Scikit-Learn machine learning library [55] for training the Random Forest. Apart from the number of trees and the depth of the trees, we use the default hyperparameters of Scikit-Learn's Random Forest module. The prototype runs on one thread, though the sampling, training, and inference are in principle parallelizable.

VII. Experimental Evaluation

We consider the bitvector track benchmarks from the latest Hardware Model Checking Competition (HWMCC) [10]. We omit all unsatisfiable benchmarks, since these are not solvable using BMC. What remain are 65 benchmarks that were reported to be satisfiable during the competition and 24 benchmarks that were unsolved during the competition. All experiments are performed on a cluster equipped with Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz running Ubuntu 20.04. Each job is given one physical core and 8 GB memory.

A. Unrolling the unsolved benchmarks

In the first experiment, we focus on the 24 unsolved benchmarks. For each benchmark, the goal is to either find a property violation or to prove that the property holds for as large a bound as possible. A CPU time limit of 2 hours is given for each benchmark. We consider two BMC step sizes: 1 and 10.[8] For each step size, we run as baselines our KISSAT-based BMC implementation without SDSL (denoted KISSAT) and the BMC engine of PONO (denoted PONO), which makes incremental calls to BOOLECTOR to solve bitvector queries.

1) Performance of SDSL *using the strategy space in Tab. I:* We first evaluate KISSAT + SDSL$_{exp}$, the SDSL-extended BMC procedure using $\mathcal{V}_{exp}$ (Tab. I) as the strategy space. The results are shown in Tab. III. We report the largest *solved* (i.e., certified or falsified) bound k for each configuration. For KISSAT + SDSL$_{exp}$ and KISSAT, we also show the total time to solve all formulas up until the largest commonly solved bound ($t_{c.s.}$). For KISSAT + SDSL$_{exp}$, this includes the time spent on learning. We further report the number of learning epochs (*ep*) and the time spent on learning (t_{learn}) for KISSAT + SDSL$_{exp}$. Graphic illustrations in the style of Fig. 1 and the duration of each training epoch are presented in the extended version [16] of the paper.

When the BMC step size is 1, KISSAT + SDSL$_{exp}$ is able to certify larger bounds compared with the baseline configurations on 22 out of the 24 benchmarks, with an average bound increase of 3.9 ($52.6 - 48.7$). This improvement is highly non-trivial, considering that to reach a larger bound, KISSAT + SDSL$_{exp}$ needs to 1) certify all the formulas up to the baseline bound; 2) spend time (on average 975 seconds) learning a solving strategy; and 3) solve an additional set of harder formulas with the remaining time, one for each increase in the bound size. Comparing $t_{c.s.}$ sheds further light on the performance gain enabled by SDSL: on average, KISSAT + SDSL$_{exp}$ is $1.3\times$ faster ($\frac{6354}{4811}$) on the set of commonly solved problems. The fraction of $t_{c.s.}$ that KISSAT + SDSL$_{exp}$ spends on actual solving (not including learning) is 3836 seconds ($4811 - 975$). Thus, on average a $1.7\times$ speedup ($\frac{6354}{3836}$) is achieved in the sheer performance of the SAT solver. Upon closer examination, the learning time is dominated by the data collection, with actual training and inference only taking 2.1% of t_{learn} on average.

When using BMC step size 10, both the KISSAT-based baseline and KISSAT + SDSL$_{exp}$ find counter-examples on 8 benchmarks (highlighted in red). In all but one of those benchmarks, KISSAT + SDSL$_{exp}$ finds counter-examples faster. Additionally, KISSAT + SDSL$_{exp}$ certifies a larger bound than KISSAT on 4 benchmarks. For the remaining 12 benchmarks, the two configurations certify the same bounds, but KISSAT + SDSL$_{exp}$ reduces the runtime on only 3 of them. One explanation for this is that the number of affordable learning epochs is significantly smaller when the step size is 10 due to the increased hardness of individual formulas. As a result, fewer strategies are considered. For example, on `arb.n2.w128.d64`, a total of 871 unique solving strategies are evaluated when the step size is 1, whereas only 175 strategies are evaluated when the step size is 10. Nonetheless, overall, KISSAT + SDSL$_{exp}$ is still $1.3\times$ faster ($\frac{3712}{2927}$) at certifying the same bounds.

It is important to note that using step size 10 does not necessarily lead to a larger certified bound. Take `circ.w128.d128` for example: KISSAT + SDSL$_{exp}$ can unroll to an execution length of 46 with step size 1 while only unrolling to 30 with step size 10. This also applies to an incremental solver like PONO, which certifies an execution length of 39 with step size 1 versus 20 with step size 10. This suggests that the optimal step size varies in practice.

2) Performance of SDSL *using the strategy space in Tab. II:* We repeat the same experiment for the other SDSL configuration KISSAT + SDSL$_{dev}$, which uses the smaller strategy space $\mathcal{V}_{dev}$ (Tab. II). The result is shown in Tab. IV. To summarize, KISSAT + SDSL$_{dev}$ still boosts the performance of KISSAT though the overall gain is less. For step size 1, the average solved bound by KISSAT + SDSL$_{dev}$ is 49.4 compared to 48.7 by KISSAT. The overall reduction in $t_{c.s.}$ is not significant (2.8%) though the reduction in the pure solving time (computed by subtracting t_{learn} from $t_{c.s.}$) is still clear (16.5%). For step size 10, KISSAT + SDSL$_{exp}$ and KISSAT unroll to the same bound on each instance, but it takes KISSAT + SDSL$_{exp}$ 12.7% less time to get there. It is not too surprising that the performance gain resulting from KISSAT + SDSL$_{dev}$ is smaller than from KISSAT + SDSL$_{exp}$. The smaller strategy space has far fewer strategy options and might simply not contain a better strategy than the default one.

In the extended version [16] of the paper, we also consider two additional SDSL configurations. One includes all boolean flags in the strategy space; the other uses local-search-based tuning instead of machine learning to pick the solving strategy. Both configurations perform worse than the KISSAT-based

[8] The value of 10 is chosen based on a study by Lonsing [56].

TABLE III

EVALUATION OF KISSAT + SDSL$_{exp}$ ON BV BENCHMARKS OF HARDWARE MODEL CHECKING COMPETITION 2020 THAT WERE NOT SOLVED DURING THE COMPETITION. *ep* IS THE NUMBER OF TRAINING EPOCHS. t_{learn} IS THE TIME SPENT ON DATA COLLECTION, TRAINING, AND INFERENCE. $t_{c.s.}$ IS THE CUMULATIVE TIME (t_{learn} **INCLUDED**) TO SOLVE ALL THE FORMULAS UP UNTIL THE LARGEST BOUND COMMONLY SOLVED BY KISSAT + SDSL$_{exp}$ AND KISSAT. k IS THE LARGEST SOLVED BOUND WITHIN 2 HOURS AND IS **highlighted** IF A VIOLATION IS FOUND (I.E., THE BENCHMARK IS SOLVED).

	step size = 1							step size = 10						
	KISSAT + SDSL$_{exp}$				**KISSAT**		**PONO**	**KISSAT + SDSL$_{exp}$**				**KISSAT**		**PONO**
Benchmark	*ep*	t_{learn}	$t_{c.s.}$	k	$t_{c.s.}$	k	k	*ep*	t_{learn}	$t_{c.s.}$	k	$t_{c.s.}$	k	k
arb.n2.w128.d64	10	1040	**4816**	**59**	7198	54	45	2	751	**3676**	**70**	4940	**70**	50
arb.n2.w64.d64	9	936	**4431**	**59**	6515	53	48	2	638	**3985**	**70**	4224	**70**	50
arb.n2.w8.d128	10	1031	**5599**	**54**	6795	52	45	2	1127	**3932**	60	5528	60	50
arb.n3.w16.d128	9	937	**4898**	**58**	7118	53	48	2	807	**3653**	**70**	5792	60	60
arb.n3.w64.d128	10	1062	**4856**	**58**	7132	53	44	1	84	**3427**	60	5120	60	50
arb.n3.w64.d64	10	985	**4809**	**58**	7055	53	46	2	919	**3736**	**70**	5018	**70**	50
arb.n3.w8.d128	9	983	**4760**	**58**	6996	53	46	2	906	**3227**	60	6017	60	50
arb.n4.w128.d64	9	963	6143	**54**	**6107**	53	46	2	1086	**3704**	**70**	5627	**70**	50
arb.n4.w16.d64	10	984	**4768**	**57**	6599	53	49	2	580	**3349**	**70**	5231	**70**	**70**
arb.n4.w8.d64	10	1081	**5863**	**54**	6659	52	48	2	617	**3177**	**70**	4563	**70**	50
arb.n5.w128.d64	9	1105	**5095**	**57**	6731	53	47	1	139	**4204**	**70**	5587	**70**	50
circ.w128.d128	7	845	**5746**	**46**	6549	44	39	1	536	2348	30	**1743**	30	20
circ.w128.d64	8	887	**6071**	**47**	6452	46	39	1	747	2688	30	**1962**	30	20
circ.w16.d128	11	967	**4997**	**58**	7179	54	47	2	876	5778	50	**4978**	50	40
circ.w64.d128	9	870	**5409**	**51**	6895	48	44	1	232	4575	40	**4294**	40	30
dspf.p22	4	1078	**3741**	**31**	5654	28	27	1	285	289	20	**90**	20	20
pgm.3.prop5	10	1019	7128	128	**5663**	**133**	131	2	618	6844	190	**6052**	190	170
picor.AX.nom.p2	2	929	**3307**	**16**	3636	15	14	1	117	279	**20**	**151**	**20**	**20**
picor.pcregs-p0	5	992	**4020**	**32**	6402	30	30	0	0	83	20	85	20	20
picor.pcregs-p2	5	840	6149	30	**5003**	**31**	31	0	0	89	20	91	20	20
shift.w128.d64	7	858	**4098**	**27**	5393	25	21	1	582	678	**30**	**353**	20	20
shift.w16.d128	9	1084	**3020**	**45**	5817	39	28	2	864	**1731**	**50**	2606	40	20
shift.w32.d128	8	821	**2778**	**43**	6273	35	27	1	216	2656	30	**2424**	30	20
zipversa.p03	5	1110	**2961**	**82**	6683	59	45	2	1182	**2138**	110	6622	90	**330**
Mean	8.1	975	**4811**	**52.6**	6354	48.7	43.1	1.5	580	**2927**	**57.5**	3712	55.4	55.4

BMC baseline. It is worth noting that local-search-based tuning does also result in speedup in $t_{c.s.}$ and improves upon KISSAT on 16 of the 24 instances. However, the performance gain is less significant compared to KISSAT + SDSL$_{dev}$ and, on certain benchmarks, tuning landed on parameter settings that drastically harm the performance. This suggests that using empirical performance models can be more robust than direct tuning in our setting.

B. Mini Hardware Model Checking Competition

We evaluate KISSAT + SDSL$_{exp}$ with step size 10 on all the satisfiable and unsolved bitvector benchmarks from HWMCC. As in the competition, we use a time limit of 1 hour for this experiment. We consider the basic KISSAT-based BMC procedure (also using step size 10) as a baseline. In addition, we perform an apples-to-oranges comparison to two algorithm portfolios, one of PONO and the other of the AVR model checker [12], which was the winner of the most recent competition.[9] We use the competition portfolio of AVR, which consists of 16 single-threaded solving modes. The PONO portfolio contains 13 single-threaded modes selected by the developers of PONO. Each mode can construct counter-examples. The AVR portfolio contains two BMC modes, both with step size 5. The PONO portfolio contains 1 BMC mode, with step size 11.

The number of solved instances and the total time on solved instances are shown in Tab. V. To study the complementarity of the configurations, we also report the number of unique solutions and the performance of a virtual best configuration. Results on individual benchmarks are reported in the extended version [16] of the paper.

KISSAT + SDSL$_{exp}$ solves all the instances solved by KISSAT plus 7 more, suggesting that while SDSL might create overhead for easy instances, this overhead is overcome by benefits in the long run. Impressively, those 7 problems are also not solved by the AVR and PONO portfolios. This suggests that including an SDSL-driven BMC procedure in a model checking algorithm portfolio can be beneficial.

C. Ablation studies of training budget and model architecture

To study the effect of dataset size and model accuracy, we select one benchmark `picor.pcregs-p0`, and vary the learning budget (in seconds) in the set $\{180, 360, 720, 1080, 1440\}$ and the decision tree depth from

[9]We hope to also compare with the bit-level solver ABC [57] but have no information about the commands and version used for the competition. We have contacted the ABC team and will include such results after hearing back.

TABLE IV
EVALUATION OF KISSAT + SDSL$_{dev}$. THE SETUP IS THE SAME AS TAB. III

	step size = 1						step size = 10					
	KISSAT + SDSL$_{dev}$				KISSAT		KISSAT + SDSL$_{dev}$				KISSAT	
Benchmark	*ep*	t_{learn}	$t_{c.s.}$	k	$t_{c.s.}$	k	*ep*	t_{learn}	$t_{c.s.}$	k	$t_{c.s.}$	k
arb.n2.w128.d64	11	861	**6714**	54	7198	54	2	631	**3720**	**70**	4940	**70**
arb.n2.w64.d64	10	896	**6158**	**54**	6515	53	2	575	4981	**70**	4224	**70**
arb.n2.w8.d128	11	1006	**6291**	**53**	6795	52	2	684	**3674**	60	5528	60
arb.n3.w16.d128	9	824	**6061**	**54**	7118	53	2	635	**3905**	60	5792	60
arb.n3.w64.d128	10	966	**6417**	**54**	7132	53	2	735	**3937**	60	5120	60
arb.n3.w64.d64	11	1053	**5452**	**55**	7055	53	2	598	5619	**70**	**5018**	**70**
arb.n3.w8.d128	9	857	**5780**	**55**	6996	53	2	635	**3703**	60	6017	60
arb.n4.w128.d64	10	901	**5859**	**55**	6107	53	2	702	6712	**70**	**5627**	**70**
arb.n4.w16.d64	10	898	**5626**	**55**	6599	53	2	462	**3266**	**70**	5231	**70**
arb.n4.w8.d64	11	973	**4749**	**57**	6659	52	2	532	**3774**	70	4563	70
arb.n5.w128.d64	10	869	**5672**	**55**	6731	53	1	92	**5283**	**70**	5587	**70**
circ.w128.d128	8	694	6948	43	**5879**	**44**	1	400	2158	30	**1743**	30
circ.w128.d64	8	829	6650	45	**5803**	**46**	1	788	2663	30	**1962**	30
circ.w16.d128	12	969	**6961**	54	7179	54	2	433	**4927**	50	4978	50
circ.w64.d128	10	713	**6759**	48	6895	48	1	179	4361	40	**4294**	40
dspf.p22	4	959	**4148**	**31**	5654	28	1	185	177	20	**90**	20
pgm.3.prop5	16	920	**6933**	133	6996	133	3	557	**5664**	190	6052	190
picor.AX.nom.p2	2	590	**3540**	15	3636	15	1	59	191	**20**	**151**	**20**
picor.pcregs-p0	6	873	6573	28	**3337**	**30**	0	0	86	20	85	20
picor.pcregs-p2	5	646	6045	28	**2786**	**31**	0	0	86	20	91	20
shift.w128.d64	8	666	5965	25	**5393**	25	1	1392	601	20	**353**	20
shift.w16.d128	9	940	**5209**	**40**	5817	39	1	139	**2058**	40	2606	40
shift.w32.d128	8	796	**5551**	**36**	6273	35	1	217	2615	30	**2424**	30
zipversa.p03	2	460	7037	59	**6683**	59	1	268	**3650**	90	6622	90
Mean	8.8	840	**5962**	**49.4**	6134	48.7	1.5	454	**3242**	55.4	3712	55.4

TABLE V
COMPARISON WITH TWO ALGORITHM PORTFOLIOS ON SATISFIABLE AND UNSOLVED BV HWMCC BENCHMARKS (89 IN TOTAL).

Config.	**Threads**	**Slv.**	**Time**	**Unique**
KISSAT + SDSL$_{exp}$	1	**68**	27362	**7**
KISSAT	1	61	6358	0
AVR PORTFOLIO	16	48	12113	2
PONO PORTFOLIO	13	63	10723	0
VIRTUAL BEST	31	72	24700	–

1 to 10.[10] We consider all 50 combinations of the two. For each combination, we run KISSAT + SDSL$_{exp}$ (step size 1, time limit 2 hours) 12 times, each time with a different random seed (0 . . . 11); we show the median certified bound in the top half of Fig. 4 and the average training score (R^2 score, the larger the better) in the last learning epoch in the bottom half. The largest bound certified by KISSAT without SDSL is 30.

Noticeably, on this instance, improvements in the certified bound are achieved when the depth of the tree is at least 4 and the learning budget is at least 1080 seconds. This suggests that both a sufficient amount of training data and an accurate model are necessary for SDSL to work in practice. If not enough data

[10]For this experiment we use a fixed tree depth instead of the dynamic one described in Sec. V-B.

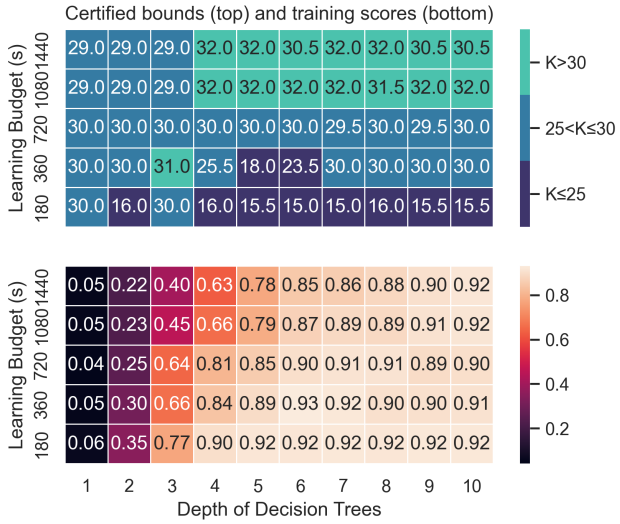

Fig. 4. Varying the learning budget and tree depth on picor.pcregs-p0.

is collected (bottom right), the machine learning model cannot extrapolate well to new problem instances. On the other hand, if the machine learning model is not accurate enough (top left), the strategy it suggests can also be misleading. Determining the optimal learning budget on a per-benchmark basis is a topic worth studying in the future.

VIII. Conclusion, limitations, and future work

We introduced Self-Driven Strategy Learning, a conceptually simple, easy-to-implement online learning approach for solving sets of related problems in automated reasoning. We presented the methodology formally as a set of transition rules and instantiated it in the context of Bounded Model Checking. Our experiments show that equipping a BMC-procedure with SDSL results in a significant performance boost, both in terms of certified bounds and solved instances, when comparing against state-of-the-art open-source model checkers.

One thing to consider when applying SDSL is that a good return on investment in learning depends on a sensible *a priori* choice of the strategy space. Another limitation is that when the problem set is small, gathering sufficient training data can be challenging. An intriguing question is whether a problem can be decomposed into sub-problems automatically in order to obtain sufficient data. Other future directions include alternative orders of applying the SDSL rules, applying SDSL to other automated reasoning tasks (e.g., symbolic execution, max-satisfiability, iterative abstraction refinement), and combining SDSL with offline learning and incremental solving.

Acknowledgments: We thank the anonymous reviewers for their careful reviews and constructive feedback. This work was supported in part by the National Science Foundation (grant 1269248) and by the Stanford Center for Automated Reasoning. Additionally, the NASA University Leadership initiative (grant #80NSSC20M0163) provided funds to assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not any NASA entity.

References

[1] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, pp. 7–34, 2001.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking." *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 2000, pp. 154–169.

[4] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[5] H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Automated configuration and selection of sat solvers," in *Handbook of Satisfiability*. IOS Press, 2021, pp. 481–507.

[6] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, "Boosting verification by automatic tuning of decision procedures," in *Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007, pp. 27–34.

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

[8] D. Selsam and N. Bjørner, "Guiding high-performance sat solvers with unsat-core predictions," in *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*. Springer, 2019, pp. 336–353.

[9] E. Yolcu and B. Póczos, "Learning local search heuristics for boolean satisfiability," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] M. Preiner, A. Biere, and N. Froleyks, "Hardware model checking competition 2020," 2020.

[11] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[12] A. Goel and K. Sakallah, "Avr: abstractly verifying reachability," in *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26*. Springer, 2020, pp. 413–422.

[13] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, "Pono: a flexible and extensible smt-based model checker," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 2021, pp. 461–474.

[14] J. N. Hooker, "Solving the incremental satisfiability problem," *The Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.

[15] A. Biere, M. S. Chowdhury, M. J. Heule, B. Kiesl, and M. W. Whalen, "Migrating solver state," in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[16] H. Wu, C. Hahn, F. Lonsing, M. Mann, R. Ramanujan, and C. Barrett, "Lightweight online learning for sets of related problems in automated reasoning [extended version]," *arXiv preprint arXiv:2305.11087*, 2023.

[17] O. Strichman, "Accelerating bounded model checking of safety properties," *Formal Methods in System Design*, vol. 24, pp. 5–24, 2004.

[18] F. Hutter, H. H. Hoos, and T. Stützle, "Automatic algorithm configuration based on local search," in *Aaai*, vol. 7, 2007, pp. 1152–1157.

[19] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Satenstein: Automatically building local search sat solvers from components," *Artificial Intelligence*, vol. 232, pp. 20–42, 2016.

[20] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh, "Algorithm selection for SMT," *Int. J. Softw. Tools Technol. Transf.*, vol. 25, no. 2, pp. 219–239, 2023. [Online]. Available: https://doi.org/10.1007/s10009-023-00696-0

[21] L. Xu, H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, 2010, pp. 210–216.

[22] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[23] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, K. Tierney *et al.*, "Model-based genetic algorithms for algorithm configuration." in *IJCAI*, 2015, pp. 733–739.

[24] K. Leyton-Brown, E. Nudelman, and Y. Shoham, "Empirical hardness models: Methodology and a case study on combinatorial auctions," *Journal of the ACM (JACM)*, vol. 56, no. 4, pp. 1–52, 2009.

[25] M. Balunovic, P. Bielik, and M. Vechev, "Learning to solve smt formulas," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[26] C. Hahn, F. Schmitt, J. U. Kreber, M. N. Rabe, and B. Finkbeiner, "Teaching temporal logics to neural networks," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=dOcQK-f4byz

[27] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning rate based branching heuristic for SAT solvers," in *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, ser. Lecture Notes in Computer Science, N. Creignou and D. L. Berre, Eds., vol. 9710. Springer, 2016, pp. 123–140. [Online]. Available: https://doi.org/10.1007/978-3-319-40970-2_9

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

[29] H. Wu, "Improving sat-solving with machine learning," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017, pp. 787–788.

[30] H. Wu and R. Ramanujan, "Learning to generate industrial sat instances," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 10, no. 1, 2019, pp. 206–207.

[31] J. You, H. Wu, C. Barrett, R. Ramanujan, and J. Leskovec, "G2sat: learning to generate sat formulas," *Advances in neural information processing systems*, vol. 32, 2019.

[32] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals, and Y. Zwols, "Solving mixed integer programs using neural networks," *CoRR*, vol. abs/2012.13349, 2020. [Online]. Available: https://arxiv.org/abs/2012.13349

[33] D. Bertsimas and B. Stellato, "The voice of optimization," *Machine Learning*, vol. 110, no. 2, pp. 249–277, Feb 2021. [Online]. Available: https://doi.org/10.1007/s10994-020-05893-5

[34] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data-driven approach for boolean function synthesis," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*. Springer, 2020, pp. 611–633.

[35] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, "Engineering an efficient boolean functional synthesis engine," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[36] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.

[37] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[38] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2526–2540. [Online]. Available: https://doi.org/10.1145/3460120.3484813

[39] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki, "Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers," in *Hardware and Software: Verification and Testing: 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings 11*. Springer, 2015, pp. 225–241.

[40] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 530–535.

[41] A. Biere and A. Fröhlich, "Evaluating cdcl variable scoring schemes," in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer, 2015, pp. 405–422.

[42] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[43] R. E. Kass, B. P. Carlin, A. Gelman, and R. M. Neal, "Markov chain monte carlo in practice: a roundtable discussion," *The American Statistician*, vol. 52, no. 2, pp. 93–100, 1998.

[44] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[45] G. Louppe, "Understanding random forests: From theory to practice," *arXiv preprint arXiv:1407.7502*, 2014.

[46] C. Barrett, A. Stump, and C. Tinelli, "The satisfiability modulo theories library (smt-lib). www," *SMT-LIB. org*, vol. 15, pp. 18–52, 2010.

[47] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2016.

[48] F. Beskyd and P. Surynek, "Domain dependent parameter setting in sat solver using machine learning techniques," in *Agents and Artificial Intelligence: 14th International Conference, ICAART 2022, Virtual Event, February 3–5, 2022, Revised Selected Papers*. Springer, 2023, pp. 169–200.

[49] B. Dutertre, "An empirical evaluation of sat solvers on bit-vector problems." in *SMT*, 2020, pp. 15–25.

[50] A. Biere, "Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018," *Proceedings of SAT Competition*, vol. 14, pp. 316–336, 2017.

[51] F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *International conference on machine learning*. PMLR, 2014, pp. 754–762.

[52] R. Brummayer, A. Biere, and F. Lonsing, "Btor: bit-precise modelling of word-level problems for model checking," in *Proceedings of the joint workshops of the 6th international workshop on satisfiability modulo theories and 1st international workshop on bit-precise reasoning*, 2008, pp. 33–38.

[53] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2, btormc and boolector 3.0," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Springer, 2018, pp. 587–595.

[54] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. [Online]. Available: https://doi.org/10.3233/sat190101

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[56] F. Lonsing, "Pono: An smt-based model checker," in *Center for Automated Reasoning Workshop*. Stanford, CA, 2022. [Online]. Available: http://www.florianlonsing.com/talks/Lonsing-CentaurRetreat-2022-talk.pdf

[57] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.