Simulate, Refine and Integrate: Strategy Synthesis for Efficient SMT Solving

Bingzhe Zhou[1], **Hannan Wang**[1], **Yuan Yao**[1],
Taolue Chen[2], **Feng Xu**[1] and **Xiaoxing Ma**[1]

[1]State Key Laboratory of Novel Software Technology, Nanjing University, China
[2]School of Computing and Mathematical Sciences, Birkbeck, University of London, UK
{bingzhezhou, hannanwang}@smail.nju.edu.cn, {y.yao,xf,xxm}@nju.edu.cn, t.chen@bbk.ac.uk

Abstract

Satisfiability Modulo Theories (SMT) solvers are crucial in many applications, yet their performance is often a bottleneck. This paper introduces SIRISMT, a novel framework that employs machine learning techniques for the automatic synthesis of efficient SMT-solving strategies. Specifically, SIRISMT targets at Z3 and consists of three key stages. First, given a set of training SMT formulas, SIRISMT simulates the solving process by leveraging reinforcement learning to guide its exploration within the strategy space. Next, SIRISMT refines the collected strategies by pruning redundant tactics and generating augmented strategies based on the subsequence structure of the learned strategies. These refined strategies are then fed back into the reinforcement learning model. Finally, the refined and optimized strategies are integrated into one strategy, which can be directly plugged into modern SMT solvers. Extensive evaluations show the superior performance of SIRISMT over the baseline methods. For example, compared to the default Z3, it solves 26.8% more formulas and achieves up to an 86.3% improvement in the Par-2 score on benchmark datasets. Additionally, we show that the synthesized strategy can improve the code coverage by up to 11.8% in a downstream symbolic execution benchmark.

1 Introduction

Satisfiability Modulo Theory (SMT) solvers are automated tools designed to determine the satisfiability of first-order logic formulas with respect to certain background theories (e.g., bit vectors, etc.), playing a foundational role in various fields such as software verification [Kroening *et al.*, 2023; Song *et al.*, 2023], program synthesis [Jha *et al.*, 2010] and symbolic execution [Cadar *et al.*, 2008].

The efficiency of SMT solvers, which is heavily influenced by the heuristics they employ, often turns out to be a performance bottleneck within these applications [Palikareva and Cadar, 2013; De Moura and Bjørner, 2011]. To improve efficiency, solvers like Z3 [De Moura and Bjørner, 2008], CVC5 [Barbosa *et al.*, 2022], and Yices2 [Dutertre, 2014] use various solving heuristics in conjunction with proof methods and search techniques. To specify the heuristic to solve a complex formula, typically one can use a *tactic language* provided by modern SMT solvers. A program in such a language is often referred to as a *solving strategy*, which gives a high-level plan outlining a combination of tactics.

However, in general, determining a suitable strategy for a given formula is challenging. For instance, Z3 has over a hundred tactics, some with complex parameters, creating a large strategy space, especially when sequences are lengthy. Additionally, strategies often include heuristics with control statements, such as conditionals or loops, making hand-crafting these strategies laborious.

Modern SMT solvers typically provide default strategies designed for general use by domain experts. However, these strategies do not fully consider the specific characteristics of each formula, leading to suboptimal performance in some cases. For instance, adjusting default strategies may significantly improve solving efficiency for certain formulas, as demonstrated in the example below.

Example 1. The following constraint is from the theorem prover Isabelle [Nipkow *et al.*, 2002].

$$\forall g_1.\ g_1^4 + 12g_1^2 + 9 = c_1(g_1^2 + p_1)^2 + y_1$$
$$\forall g_2.\ g_2^4 + 12g_2^2 + 9 = c_2(g_2^2 + p_2)^2 + y_2$$
$$\text{where} \quad answer_1 = y_1 \quad \text{and} \quad answer_2 = y_2$$
$$\text{and} \quad answer_1 = answer_2$$

This example involves nonlinear constraints with (universal) quantifiers. Z3 fails to provide an answer in 200 seconds with its default strategies. However, by adjusting the default solving strategy to the one learned by our approach, CHECK-SAT-USING(`then(simplify; qe; smt)`) (which first simplifies the input constraints, then performs quantifier elimination, and finally tries to solve the formula using tactic `smt`), Z3 can successfully solve this constraint in one second. □

Existing Work. Existing work has attempted the automatic synthesis of SMT strategies using techniques like evolutionary algorithms [Ramírez *et al.*, 2016], machine learning [Balunovic *et al.*, 2018], and Monte Carlo Tree Search (MCTS) [Lu *et al.*, 2024]. However, these methods still have several limitations. (**C1**) The variable interdependencies in an SMT formula is not well encoded, as formulas

were simply treated as natural language tokens; (**C2**) the latest state of the formula[1] is not well captured, as existing work tends to directly map the initial formula to a solving strategy; (**C3**) The generated tactic sequences often suffer from redundancy and bias, reducing their utility.

This Work. We propose SIRISMT[2], a novel framework for the automatic strategy synthesis in SMT solvers.[3] SIRISMT consists of three key stages. The *simulation* stage combines graph neural networks and reinforcement learning to obtain a set of tactic sequences that are efficient in solving the training SMT formulas. The *refinement* stage further enhances the tactic sequences by pruning redundant tactics and generating augmented ones. This stage also interacts with the simulation stage by feeding the refined tactic sequences back into the machine learning model. The *integration* stage integrates the refined tactic sequences into the final solving strategy by employing a specialized decision tree algorithm [Quinlan, 1993]. Specifically, we leverage various techniques to address the limitations of the existing work.

- We build a graph for each SMT formula and use graph neural networks (GNNs, [Kipf and Welling, 2017; Wu *et al.*, 2020]) to represent SMT formulas beyond simple natural language tokens. GNNs can better capture formulas' structural information, providing more meaningful representations for the later strategy synthesis.
- We leverage reinforcement learning (RL, [Wang *et al.*, 2022; van Hasselt *et al.*, 2015]) to capture the information of the transformed formulas. Under RL, the decision on the next tactic is made based on the current form of the input formula. RL and GNNs together allow SIRISMT to better understand and respond to the current formula adaptively, leading to more effective subsequent decisions.
- We refine the generated tactic sequences to improve their effectiveness and diversity. Inspired by probabilistic delta debugging [Wang *et al.*, 2021], we systematically eliminate redundant tactics in each tactic sequence. With the observation that some common tactic combinations are often effective across different formulas, we identify these combinations via frequent subsequence detection [Liu *et al.*, 2021], and then use the identified tactic combinations to generate new tactic sequences.

In this work, we evaluate Z3 plugged with the strategy outputted by SIRISMT. Experiments on five benchmark SMT datasets demonstrate that the solving strategy achieves a remarkable performance. For instance, the synthesized strategies of SIRISMT can solve 26.8% more formulas compared to the default solving strategy in Z3, and achieve 17.4%–86.3% (relative) improvements in the PAR-2 score [Amadini *et al.*, 2023]. Compared to the state-of-the-art competitors fastSMT [Balunovic *et al.*, 2018] and Z3Alpha [Lu *et al.*, 2024], SIRISMT also solves 15.6% and 16.6% more formulas, respectively. Moreover, we evaluate SIRISMT in the downstream symbolic execution task [Cadar *et al.*, 2008]. The results show that SIRISMT achieves up to 11.8% code coverage improvement on *coreutils* programs compared to KLEE's default solving strategy.

```
1  then(
2      using-params simplify mul2concat true;
3      using-params propagate-values flat-and-or
       false;
4      using-params solve-eqs solve-eqs-max-occs 2;
5      elim-uncnstr;
6      ...
7  )
```

Figure 1: Example (partial) strategy for QFBV theory in Z3.

We summarize the main contributions as follows.

- We put forward a three-stage framework for automatic strategy synthesis in SMT solvers, with an additional refinement stage to guide the learning process.
- We substantiate the framework by GNNs that capture the complex structures of SMT formulas as well as RL which effectively guides the exploration of the strategy space.
- We implement the above techniques in SIRISMT and demonstrate its superior performance in solving SMT formulas through extensive evaluations.

Roadmap. Section 2 introduces the background knowledge and problem statement. Section 3 details the proposed approach. Section 4 reports the experimental results. Section 5 covers related work, and Section 6 concludes the paper.

2 Preliminary and Problem Statement

2.1 SMT Solving Strategy

In addition to the default solving strategies, modern SMT solvers such as Bitwuzla [Niemetz and Preiner, 2023], CVC5 [Barbosa *et al.*, 2022], and Z3 [De Moura and Bjørner, 2008] allow users to customize solver configurations or design their own solving strategies using a tactic language. For example, a solving strategy in Z3 is a program in its tactic language, typically encompassing multiple tactics in a sequence and possibly further incorporating control statements to improve its expressiveness. Each tactic can adeptly modify formulas to either simplify their evaluation or directly solve them. For instance, the tactic `bit-blast` is often used to reduce bit-vector expressions in the SMT formula to equivalent propositional logic. This translation is essential as it allows the formula, originally expressed using bit-vectors, to be processed using an SAT solver. Within each tactic, parameters are commonly needed.

An example (partial) strategy for quantifier-free bit-vector (QFBV) problems in Z3 is shown in Fig. 1.[4] This strategy

[1]Formulas are transformed into equivalent forms as tactics are applied.

[2]The source code, dataset and the extended version are available at: https://github.com/SoftWiser-group/SiriSMT

[3]We focus on Z3 in this paper, but the underlying idea is applicable to other solvers.

[4]The sequence is part of the default strategy, which comprises multiple branches and is over 20 steps.

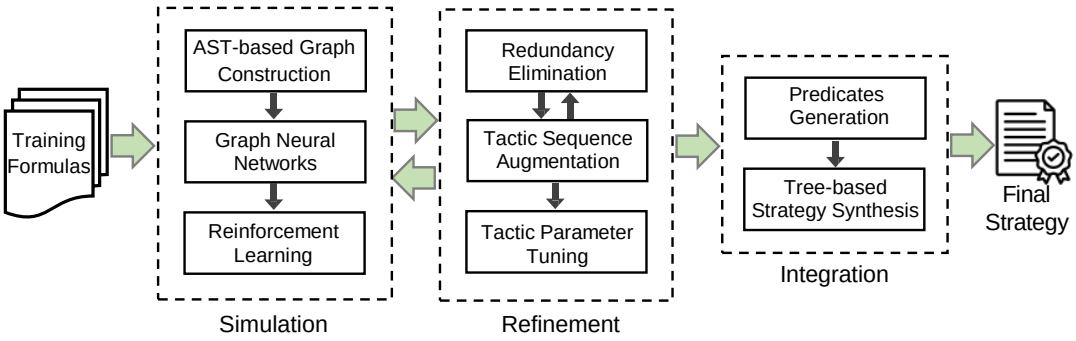


Figure 2: Overview of SIRISMT.

begins with applying the `simplify` tactic using specific parameters to transform multiplication into concatenation operations. Subsequently, the `propagate-values` tactic is used, avoiding the flattening of `and`/`or` operators. The `solve-eqs` tactic is then employed, restricting its application to variables with at most two occurrences. Finally, `elim-uncnstr` is employed to eliminate uninterpreted constants that occur only once in a goal, where their immediate context can be substituted with a new constant.

In the tactic language of Z3, there are more than a hundred tactics, and each tactic may include a substantial number of parameters. For instance, the `simplify` tactic comprises over 50 parameters. Given that solving strategies often encompass several tens of tactics, identifying the appropriate solving strategy for a formula poses a challenging task.

2.2 Problem Statement

Given a set of SMT formulas $\mathcal{F} = \{f_1, f_2, \ldots, f_N\}$, our goal is to automatically synthesize an effective solving strategy $q_{\text{best}} \in \mathcal{Q}$ that not only maximizes the number of formulas solved but also minimizes the total solving time. Specifically, we aim to optimize the following objective:

$$q_{\text{best}} = \arg\min_{q \in \mathcal{Q}} \sum_{i=1}^{N} \text{cost}(f_i, q),$$

where the cost function $\text{cost}(f_i, q)$ defines the cost of solving formula f_i with strategy q. For example, one can define it as:

$$\text{cost}(f_i, q) = \begin{cases} \text{runtime}(q, f_i) & \text{if strategy } q \text{ solves } f_i, \\ \text{penalty} & \text{if strategy } q \text{ does not solve } f_i. \end{cases}$$

Here, $\text{runtime}(q, f_i)$ represents the time required for strategy q to solve formula f_i, and penalty is a fixed cost applied when strategy q fails to solve formula f_i.

Although minimizing the runtime is a natural objective, directly optimizing it is challenging due to the inherent noise and variability caused by factors such as hardware performance and solver configurations. To address this, we propose to optimize a cost function that considers not only the runtime but also the computational operations (e.g., length of the tactic sequence) required to solve the formulas. This allows for a more robust and stable strategy synthesis process, overcoming the pitfalls of relying solely on runtime.

3 Approach

An overview of SIRISMT is given in Fig. 2. It takes a set of training formulas as inputs, and outputs a solving strategy in a tree form employing the control structures of the tactic language. Specifically, SIRISMT consists of the following three stages.

- **Simulation stage.** This stage generates initial tactic sequences for solving the training SMT formulas. SIRISMT parses the input formulas into graphs and uses GNNs to encode them. The encoding combines graph representations, probe embeddings, and applied tactics. Reinforcement learning (RL) is then used to explore the strategy space, adapting to the latest state of the input formula as tactics are applied.
- **Refinement stage.** To address noise and instability from the simulation stage, SIRISMT refines the tactic sequences by reducing redundancies and augmenting the strategy space. The refined sequences are fed back into the simulation stage, allowing RL models to retrain with improved sequences. This iterative process continues until a resource budget is exhausted or no further improvement is observed. SIRISMT also tunes tactic parameters during this stage.
- **Integration stage.** In this stage, SIRISMT synthesizes the solving strategy by generating predicates for partitioning tactic sequences and integrating them into a tree-form strategy using control structures. The resulting strategy is applicable to SMT formulas with distributions similar to the training set and can be directly used with solvers (e.g., Z3).

3.1 The Simulation Stage

In this stage, we aim to identify a set of tactic sequences that are efficient in terms of solving the training formulas. Specifically, we try to generate a tactic sequence for each formula in the training set, which involves three integral components: AST-based graph construction, SMT formula representation learning using GNNs, and tactic sequence generation via RL. In the sequel, we present the details of these three components.

AST-based Graph Construction Traditional approaches that treat SMT formulas as natural language tokens often fail to capture the structural information and semantic relationships crucial for optimal strategy synthesis. To address this,

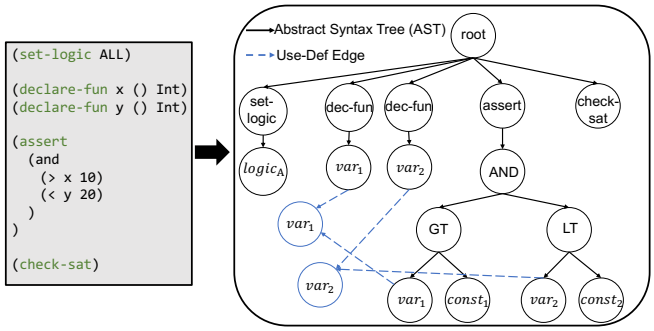


Figure 3: Graph representation of an example SMT formula.

we propose transforming SMT formulas into graph representations, which are then encoded using GNNs. This method allows us to capture both structural and semantic details of the formulas, providing a solid foundation for the subsequent stages.

Specifically, we first convert an SMT formula into a directed graph based on its abstract syntax tree (AST), denoted as $G = (V, E)$. The set of nodes V includes all components of the formula such as logical operators, relational operators, functions, constants, variables, and commands. The set of edges E consists of two types of links: the syntax tree links and the use-definition links [Leeson *et al.*, 2023], which connect variables to their definitions.

Example 2. Fig. 3 shows an example of transforming an SMT formula into its graph representation. An example SMT formula given on the left is parsed into an AST by pySMT [pys, 2014]. Each component of the SMT formula is represented as a node in the graph shown on the right. In this example, the types of nodes include logical operators (e.g., the conjunction operator `and`), relational operators (e.g., the greater-than sign '>'), commands (e.g., `assert`, `declare-fun`, and `check-sat`), constants, and variables. Note that to ensure the generalization across different formulas, variable names and constants are substituted into a unified form (e.g., 'var_1' and 'const_1'). In addition to the AST links, we further connect the AST nodes through use-def links (denoted with blue dashed lines). These links clearly outline the usages of the free variables in a formula, and thus are pivotal as they can capture diverse contexts in which free variables operate, enriching the structure and semantic information of the graph. □

Formula Representation Learning via GNN We apply Graph Attention Networks (GAT) [Veličković *et al.*, 2018] to the constructed graph representation. Each node v_i is assigned a vector representation $h^t_{v_i}$, which is iteratively updated through GAT's message passing:

$$h^{t+1}_{v_i} = g\left(\sum_{v_j \in \mathcal{N}(v_i)} \alpha^t_{ij} W^t_v h^t_{v_j}\right), \quad \alpha^t_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(v_i)} \exp(e_{ik})},$$

where $e_{ij} = \text{LeakyReLU}(b^\top [W_e h_{v_i} \| W_e h_{v_j}])$. Here, $g(\cdot)$ is the ReLU activation introducing non-linearity, $\mathcal{N}(v_i)$ denotes the neighbors of v_i, and W^t_v is the shared weight matrix for feature transformation. The attention coefficient α^t_{ij} quantifies the importance of node v_j's features to v_i. In this framework, $W^t_v h^t_{v_j}$ represents the message passed from v_j to v_i, enabling effective aggregation of structural information. Employing GNNs in our setting is important for accurately discerning the intricate interdependencies within SMT formulas, paving the way for the development of sophisticated tactic sequence generation in the subsequent RL phase.

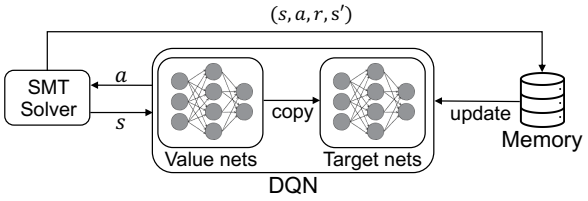


Figure 4: Illustration of the reinforcement learning module.

Tactic Sequence Generation via RL SMT solving is inherently a sequential decision-making task, where the effects of applying a specific tactic depend on the current form of the formula. Unlike traditional methods (e.g., fastSMT [Balunovic *et al.*, 2018]) that mainly encode the input formulas, reinforcement learning (RL) is well-suited to adaptively learn tactics based on the evolving formula states.

We cast tactic sequence generation as an RL problem, defined by the tuple $M = (S, A, T, R)$. Here, states (S) represent the formula's current form (including GNN embeddings, applied tactics, and probes), actions (A) correspond to tactics, and the reward function (R) evaluates solving success, time efficiency, and sequence length. The transition function (T) is deterministic and thus omitted. The policy (π) maps states to actions to maximize cumulative rewards.

As illustrated in Fig. 4, our framework adopts the Double DQN architecture [van Hasselt *et al.*, 2015], where value and target networks reduce overestimation and stabilize learning. The solver applies selected tactics, resulting in new states stored in a memory buffer to periodically update Q-values. The training dataset comprises transitions $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^N$, and refined sequences from the next stage are included for retraining.

Reward Function. The reward function encourages efficient solving while discouraging unnecessary tactic sequences:

$$\text{Reward} = \mathbb{I}_{\text{solved}} - \beta\tau + \gamma\frac{1}{l},$$

where $\mathbb{I}_{\text{solved}}$ is an indicator for solving success, τ is normalized time, and l is the sequence length. We set $\beta = \gamma = 0.1$ to emphasize solving capability over other factors.

3.2 The Refinement Stage

The refinement stage is motivated by the observation that although tactic sequences generated by machine learning may exhibit good performance, they are often noisy, i.e., containing inefficiencies and redundancies that may hinder the overall performance of the solver. In view of this, we design a rigorous refinement process that reevaluates the tactic sequences generated by the model. In practice, we implement refinement through a two-phase iterative approach. First,

tactic sequence reduction is applied to eliminate redundant tactics, streamlining the sequences and improving their efficiency. Second, the reduced sequences are analyzed to detect frequent subsequences, which form the basis for mutation-based augmentation, generating new tactic sequences from the existing ones. These refined sequences are then fed back to the simulation stage for retraining the RL model, enabling it to distinguish effective tactics better and further optimize the strategy space exploration.

Redundancy elimination. The first phase focuses on identifying and eliminating ineffective tactics. We use Probabilistic Delta Debugging (ProbDD [Wang *et al.*, 2021]) for efficient pruning. ProbDD builds a probabilistic model to assess the contribution of each tactic to solving performance, and keeps the reduced sequence with the minimum runtime.

Tactic Sequence Augmentation. Following the reduction phase, we perform tactic sequence augmentation, which consists of two steps: subsequence substitution and parameter generation. We first detect frequent subsequences in reduced sequences using a detection algorithm [Liu *et al.*, 2021]. These subsequences are hypothesized to generalize well to other formulas. For each reduced sequence, we replace prefixes with common tactic combinations, generating new sequences. We retain only those that outperform the original sequences. These augmented sequences are sent back to the reduction phase, and the process is iteratively repeated. This ensures continuous refinement of the best-performing sequences. Finally, random parameter configurations are assigned to tactics to explore a broader spectrum of sequences. Each configuration is tested on a subset of training formulas to validate their effectiveness and ensure they do not cause errors during application.[5] Note that we assign default tactic parameters in the simulation stage to avoid enlarging the search space.

3.3 The Integration Stage

In this stage, we integrate the refined tactic sequences into a final strategy. The integration has to navigate the delicate balance between specialized and general strategies. Given the variability in the initial consecutive tactics across different tactic sequences, we synthesize multiple strategies and then select the strategy that solves the most instances from the training set as the final solving strategy. This approach ensures that we accommodate the diversity in the current tactic sequences of training formulas, and that the final strategy could efficiently solve a wide range of formulas without being overly specialized.

In this work, we employ the C4.5 decision tree algorithm [Quinlan, 1993] to build the solving strategy. We choose C4.5 due to its ability to correct biases towards predicates with larger probe values and to mitigate risks of overfitting. Specifically, given a set of candidate tactic sequences $Q = \{q_1, \ldots, q_n\}$ and a dataset of formulas Φ, we combine the tactic sequences using a branch b defined as follows,

$$b := \text{if Pred then } q_{\text{true}} \text{ else } q_{\text{false}}.$$

[5]Some parameter combinations may lead to illegal transformations or errors.

This branch partitions Φ into Φ_{true} and Φ_{false} based on the predicate 'Pred', which refers to a conditional statement that utilizes probes to classify formulas. To obtain 'Pred', we first probe each formula, and generate a predicate set by segmenting the probe values. To enhance decision-making during strategy synthesis, we also support arithmetic operations such as addition, subtraction, multiplication, and division between probes. Based on the predicate set, we then compute the information gain for each possible predicate, and select the one with maximum gain as the current branch. Here, information gain measures how much a specific predicate reduces uncertainty or increases predictiveness about the success of solving a formula.

4 Experiment

4.1 Experimental Setup

Datasets We evaluate the effectiveness of SIRISMT on five SMT benchmark datasets, namely, *core* [Barrett *et al.*, 2016b], *Sage2* [Barrett *et al.*, 2016e], *leipzig* [Barrett *et al.*, 2016d], *hycomp* [Barrett *et al.*, 2016c], and *AProVE* [Barrett *et al.*, 2016a], across three theories of QFBV (Quantifier-Free Bit-Vector), QFNIA (Quantifier-Free Nonlinear Integer Arithmetic), and QFNRA (Quantifier-Free Nonlinear Real Arithmetic). We follow the existing work on dataset selection and partition.To test the downstream impact of our method, we also evaluate the effectiveness of SIRISMT in the symbolic execution task. We use *coreutils* (version 8.31) and two real-world programs (*Make* and *Gawk*) as datasets, which are standard benchmarks for evaluating the performance of symbolic execution techniques [Cadar *et al.*, 2008].

Baselines We compare SIRISMT with four state-of-the-art baseline methods: fastSMT [Balunovic *et al.*, 2018], Z3Alpha [Lu *et al.*, 2024], and the default and theory-specific strategies of Z3 [De Moura and Bjørner, 2008]. Z3Alpha is a strategy synthesis approach that employs a two-stage Monte Carlo Tree Search (MCTS) for both tactic sequence generation and integration. fastSMT also uses machine learning techniques, similar to SIRISMT, by training deep neural networks on a dataset of formulas to predict tactic sequences, which are then synthesized into a strategy. Z3, a widely used SMT solver, integrates a set of hand-crafted strategies for general-purpose use as well as theory-specific strategies tailored to domains like QFBV, QFNIA, and QFNRA. We use Z3 version 4.12.2 for our experiments.

Evaluation Metrics We include the number of solved formulas and the Penalized Average Runtime 2 (PAR-2) score [Amadini *et al.*, 2023] as the evlaution metrics. The number of solved formulas measures the total number of formulas solved, indicating its overall effectiveness. The PAR-2 score emphasizes efficiency under time constraints, penalizing unsolved formulas with a double time limit. Smaller PAR-2 score indicates better performance.

For symbolic execution, we use three coverage metrics: instruction coverage (ICov), branch coverage (BCov), and line coverage (LCov). ICov measures the percentage of executed LLVM instructions, BCov reflects the percentage of covered branches (indicating control flow coverage), and LCov shows

Datasets	Methods	#Solved	Impr.	PAR-2	Impr.
core	Z3	210		1,501	
	Z3 (QFBV)	212	1.0%	1,476	1.7%
	fastSMT	221	5.2%	1,322	11.9%
	Z3Alpha	269	28.1%	250	83.3%
	SIRISMT	**270**	**28.6%**	**205**	**86.3%**
Sage2	Z3	2,459		86,050	
	Z3 (QFBV)	2,471	0.4%	85,781	0.3%
	fastSMT	2,984	21.4%	74,512	13.4%
	Z3Alpha	2,452	-0.3%	85,942	0.1%
	SIRISMT	**3,807**	**54.8%**	**61,996**	**28.0%**
leipzig	Z3	59		206	
	Z3 (QFNIA)	61	3.4%	172	16.5%
	fastSMT	61	3.4%	178	13.6%
	Z3Alpha	60	1.7%	178	13.6%
	SIRISMT	61	3.4%	**170**	**17.5%**
AProVE	Z3	1,534		5,517	
	Z3 (QFNIA)	1,545	0.7%	5,328	3.4%
	fastSMT	1,533	0	4,018	27.2%
	Z3Alpha	1,593	3.8%	3,076	44.2%
	SIRISMT	**1,603**	**4.5%**	**2,891**	**47.6%**
hycomp	Z3	1,700		6,288	
	Z3 (QFNRA)	1,700	0	6,258	0.5%
	fastSMT	1,742	2.5%	4,994	20.6%
	Z3Alpha	1,794	5.5%	4,116	34.5%
	SIRISMT	**1,823**	**7.2%**	**3,703**	**41.1%**

Table 1: Comparison of different solving strategies. The best results are in boldface. The relative improvements over Z3 are also shown. The proposed SIRISMT significantly outperforms the baselines.

the proportion of lines in the program covered by the explored paths during symbolic execution.

Implementation We implement SIRISMT in PyTorch [Paszke *et al.*, 2017], represent SMT formulas in the SMT-LIB2 format [Barrett *et al.*, 2016f], and use pySMT [pys, 2014] to parse formulas into ASTs. In our RL framework, we run five iterations with different random seeds for each training formula to obtain sufficient candidate tactic sequences. During the refinement stage, tactic sequence reduction and augmentation are repeated twice for efficiency, and the integration stage uses a maximum tree depth of 20 in the decision tree algorithm. For both training and testing phases, we adopt the 10-second timeout setting following fastSMT, with all experiments run using Z3. For symbolic execution, we use KLEE [Cadar *et al.*, 2008] (version 3.1) on LLVM 13. All experiments are conducted on a server equipped with an Intel Core i9-12900KF CPU (12th Generation, 24 cores) and an RTX 3090 GPU.

4.2 Experimental Results

Overall Performance We first present the overall performance of different approaches. Table 1 reports the number of solved formulas and PAR-2 score across five benchmarks. We can observe that SIRISMT consistently outperforms all the baselines in all cases. In terms of the total number of formulas solved, the synthesized strategies of SIRISMT can solve 26.8% and 26.2% more formulas compared to the default strategy in Z3 (7,564 vs. 5,962) and theory-specific strategies in Z3 (7,564 vs. 5,989), respectively. It also solves 15.6% more formulas compared to fastSMT (7,564 vs. 6,541), and 16.6% more formulas against Z3Alpha (7,564 vs. 6,168). Noteworthy, for the *Sage2* dataset which is related to symbolic execution, SIRISMT demonstrates a remarkable advantage, solving 54.8% more instances than the default strategy in Z3. The performance boost can be attributed to the diverse nature of the *Sage2* dataset, which is not derived from a single test program but includes a variety of programs. The diversity of constraints requires a solver that can adapt dynamically to each unique scenario. When considering the PAR-2 score which is a holistic measure of the overall performance of solving strategies, our method also exhibits a notable enhancement, outperforming Z3 with relative improvements ranging from 17.4% to 86.3%.

Variants	#Solved	Impr.	PAR-2	Impr.
w/o GNN	7,191	-4.9%	75,174	-9.0%
w/o Refinement	7,306	-3.4%	74,114	-7.5%
w/o Reduction	6,988	-7.6%	78,621	-14.0%
w/o Augmentation	7,030	-7.1%	77,272	-12.0%
w/o Params Tuning	6,927	-8.4%	79,964	-15.9%
SIRISMT	**7,564**		**68,965**	

Table 2: The ablation results of SIRISMT. Each design choice is helpful in terms of improving SIRISMT's performance.

Ablation Study We next evaluate the effect of different components in our framework. Specifically, since RL is the base building block of SIRISMT, we consider the rest design choices including when the GNN is excluded and when the refinement stage is excluded. We also evaluate the effectiveness of each component (i.e., redundancy elimination, sequence augmentation, and parameter tuning) in the refinement stage. The results are summarized in Table 2.

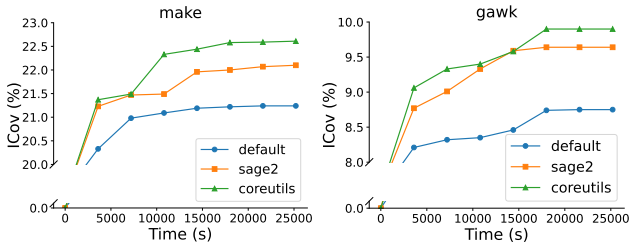
In general, we observe that each component has a positive influence on SIRISMT. For example, excluding parameter tuning has a significant negative impact on performance, with a decrease in the PAR-2 score of 15.9%. Similarly, when we exclude tactic sequence reduction and augmentation, the performance also significantly drops. This result indicates that the incorporation of both components can better guide the exploration of better solving strategies.

Impact on Symbolic Execution Finally, we evaluate the performance of SIRISMT on symbolic execution by comparing its solving strategies with KLEE's default strategies. In particular, two experiments were conducted: 1) 51 randomly selected *coreutils* programs as the training set, with the remaining 52 used for testing; 2) strategies trained on the *Sage2* dataset were tested on all *coreutils* programs.

The results, shown in Table 3, indicate that SIRISMT's strategies outperform KLEE's default in all metrics. For example, trained on *coreutils*, SIRISMT achieves 3.92%, 3.57%, and 11.48% improvements in instruction, branch, and line coverage, respectively. Using the *Sage2* strategy, we observe a similar trend with an 11.79% improvement in line coverage. Additionally, SIRISMT generates more test cases

Setup	Method	Coverage						#Test Cases
		ICov	Impr.	BCov	Impr.	LCov	Impr.	
Sage2 → coreutils	KLEE-Default	42.81		28.81		51.25		2,837
	SIRISMT	**44.56**	**4.1%**	**30.30**	**5.2%**	**57.29**	**11.8%**	**4,482**
coreutils → coreutils	KLEE-Default	44.88		29.97		46.26		1,542
	SIRISMT	**46.64**	**3.9%**	**31.04**	**3.6%**	**51.57**	**11.5%**	**2,225**

Table 3: The performance of SIRISMT's solving strategy on symbolic execution. We evaluate both the strategies trained from the *Sage2* and (partial) *coreutils* datasets on the (rest) *coreutils*. The former is designed with the purpose to show SIRISMT's generalization ability. We observe significant coverage improvements in both cases.



(a) *Make* Program (b) *Gawk* Program

Figure 5: SIRISMT 's generalization ability in real-world scenarios. We use *Sage2* and *coreutils* as training data, and evaluate the synthesized strategies on the two programs. The synthesized strategies significantly outperform the default strategy in KLEE.

than KLEE's default (e.g., 4,482 vs. 2,837 with *Sage2* training) given the same time window. Note that the result when trained on *Sage2* is slightly better than that trained on *coreutils* is due to the fact that the *Sage2* dataset is larger. The results of the programs *Make* and *Gawk* (Fig. 5) further confirm the performance of SIRISMT. These results demonstrate that SIRISMT can significantly improve symbolic execution, a key task in program analysis and software testing.

5 Related Work

This section briefly reviews the existing learning-based techniques for SMT solvers, which roughly include solver/algorithm selection and strategy synthesis. We also review graph-based representation learning for logical formulas.

Algorithm selection. Algorithm selection is a common approach in the quest for solver accelerations, where the principle is to match SMT formulas with the solver performing best for their resolution. The insight behind it is that solvers with different heuristic algorithms have distinct performances when tackling different formulas. A rule-based method [Beyer and Dangl, 2018] is proposed to select different strategies based on different theories to which the formulas belong. Nevertheless, there is a notable disparity in the efficacy of strategies when applied to formulas within the same theories. Later, learning-based methods have been proposed. They either use feature engineering which manually designs a feature vector to allocate distinct formulas to different strategies [Scott *et al.*, 2021; Xu *et al.*, 2011; Richter *et al.*, 2020; Pimpalkhare *et al.*, 2021], or utilize an encoder to transform formulas into embeddings, thereby learning reasonable representation vectors corresponding to distinct formulas [Leeson *et al.*, 2023]Our work is orthogonal to this line of research.

Strategy synthesis. Strategy synthesis methods aim to synthesize more efficient solving strategies for a given SMT solver. StratEVO [Ramírez *et al.*, 2016] uses an evolutionary algorithm to generate strategies, but suffers from the limited capacity of a basic set of mutation rules; fastSMT [Balunovic *et al.*, 2018] adopts deep neural networks to develop strategies for input SMT formulas, which is later adapted to boost the efficiency of symbolic execution tools [Chen *et al.*, 2021]. Nonetheless, as mentioned in the introduction, these methods encode constraints as natural language tokens, thus overlooking the critical structure information. Our method falls into this category. However, we encode SMT formulas along their transformations with GNNs under an RL framework and further add a refinement stage after the learning process.

Graph-based representation of logical formulas. Existing work has attempted to use GNNs to learn representations of logical formulas. For example, [Wang *et al.*, 2017] use GNN for premise selection for higher-order logic formulas. However, predicting tactic sequences based on the representations of formulas alone does not allow for tracking their form transformations after applying different tactics. Therefore, we integrate GNN with RL, enabling the selection of appropriate tactics upon the latest form. Additionally, NeuroSAT [Selsam *et al.*, 2019] leverages GNN to predict satisfiability in propositional logic; Graph-Q-SAT [Kurin *et al.*, 2019] learns to suggest variable ordering for branching heuristics in a SAT solver. However, these approaches are limited to encoding propositional formulas, whereas our framework aims to tackle more challenging SMT formulas.

6 Conclusion

This paper presents a three-stage strategy synthesis framework SIRISMT for solving SMT formulas. Our approach effectively models the structural and semantic intricacies of SMT formulas, along with their transformations during the solving process. Experimental evaluations show that the synthesized strategy of SIRISMT leads to significant improvements in solving performance, compared to both the existing hand-crafted strategies and two state-of-the-art learning-based methods. Future directions include how to combine learning-based heuristics with traditional experience-based rules, and how to employ the recent large language models to enhance SMT solving.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grants #62025202), the Frontier Technologies R&D Program of Jiangsu (BF2024059), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. T. Chen is partially supported by overseas grants from the State Key Laboratory of Novel Software Technology, Nanjing University, under #KFKT2023A04 and #KFKT2025A05. Yuan Yao is the corresponding author.

References

[Amadini *et al.*, 2023] Roberto Amadini, Maurizio Gabbrielli, Tong Liu, and Jacopo Mauro. On the evaluation of (meta-)solver approaches. *J. Artif. Intell. Res.*, 76:705–719, 2023.

[Balunovic *et al.*, 2018] Mislav Balunovic, Pavol Bielik, and Martin Vechev. Learning to solve smt formulas. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[Barbosa *et al.*, 2022] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[Barrett *et al.*, 2016a] C. Barrett, P. Fontaine, and C. Tinelli. AProVE Benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/tree/master/AProVE, 2016.

[Barrett *et al.*, 2016b] C. Barrett, P. Fontaine, and C. Tinelli. core benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/tree/master/bruttomesso/core, 2016.

[Barrett *et al.*, 2016c] C. Barrett, P. Fontaine, and C. Tinelli. hycomp benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA/tree/master/hycomp, 2016.

[Barrett *et al.*, 2016d] C. Barrett, P. Fontaine, and C. Tinelli. leipzig benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/tree/master/leipzig, 2016.

[Barrett *et al.*, 2016e] C. Barrett, P. Fontaine, and C. Tinelli. Sage2 benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/Sage2, 2016.

[Barrett *et al.*, 2016f] C. Barrett, P. Fontaine, and C. Tinelli. The satisfiability modulo theories library (smt-lib). www.SMT-LIB.org, 2016.

[Beyer and Dangl, 2018] Dirk Beyer and Matthias Dangl. Strategy selection for software verification based on boolean features: A simple but effective approach. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II 8*, pages 144–159. Springer, 2018.

[Cadar *et al.*, 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[Chen *et al.*, 2021] Zhenbang Chen, Zehua Chen, Ziqi Shuai, Guofeng Zhang, Weiyu Pan, Yufeng Zhang, and Ji Wang. Synthesize solving strategy for symbolic execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 348–360, New York, NY, USA, 2021. Association for Computing Machinery.

[De Moura and Bjørner, 2008] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[De Moura and Bjørner, 2011] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, sep 2011.

[Dutertre, 2014] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[Jha *et al.*, 2010] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.

[Kipf and Welling, 2017] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[Kroening *et al.*, 2023] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Cbmc: The c bounded model checker, 2023.

[Kurin *et al.*, 2019] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Improving SAT solver heuristics with graph networks and reinforcement learning. *CoRR*, abs/1909.11830, 2019.

[Leeson *et al.*, 2023] Will Leeson, Matthew B Dwyer, and Antonio Filieri. Sibyl: Improving software engineering tools with smt selection. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2185–2197. IEEE Press, 2023.

[Liu *et al.*, 2021] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. Iterative compilation optimization based on metric learning and collaborative filtering. *ACM Trans. Archit. Code Optim.*, 19(1), dec 2021.

[Lu *et al.*, 2024] Zhengyang Lu, Stefan Siemer, Piyush Jha, Joel Day, Florin Manea, and Vijay Ganesh. Layered and staged monte carlo tree search for smt strategy synthesis, 2024.

[Niemetz and Preiner, 2023] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[Palikareva and Cadar, 2013] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, page 53–68, Berlin, Heidelberg, 2013. Springer-Verlag.

[Paszke *et al.*, 2017] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch, 2017.

[Pimpalkhare *et al.*, 2021] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. Medleysolver: Online SMT algorithm selection. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 453–470. Springer, 2021.

[pys, 2014] Pysmt: A solver-agnostic library for fast prototyping of smt-based algorithms. https://pypi.org/project/PySMT/, 2014.

[Quinlan, 1993] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Ramírez *et al.*, 2016] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. Evolving smt strategies. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 247–254, 2016.

[Richter *et al.*, 2020] Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Algorithm selection for software validation based on graph kernels. *Autom. Softw. Eng.*, 27(1):153–186, 2020.

[Scott *et al.*, 2021] Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. Machsmt: A machine learning-based algorithm selector for smt solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–325. Springer, 2021.

[Selsam *et al.*, 2019] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a sat solver from single-bit supervision, 2019.

[Song *et al.*, 2023] Kunjian Song, Mikhail R. Gadelha, Franz Brauße, Rafael S. Menezes, and Lucas C. Cordeiro. Esbmc v7.3: Model checking c++ programs using clang ast, 2023.

[van Hasselt *et al.*, 2015] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[Veličković *et al.*, 2018] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[Wang *et al.*, 2017] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding, 2017.

[Wang *et al.*, 2021] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 881–892, New York, NY, USA, 2021. Association for Computing Machinery.

[Wang *et al.*, 2022] Xu Wang, Sen Wang, Xingxing Liang, Dawei Zhao, Jincai Huang, Xin Xu, Bin Dai, and Qiguang Miao. Deep reinforcement learning: a survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[Wu *et al.*, 2020] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[Xu *et al.*, 2011] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *CoRR*, abs/1111.2249, 2011.