Learning Oracle Representations to Speed Up Satisfiability Modulo Theories and Oracles

Anish Doshi, Pei-Wei Chen, Junqing Zhang
{apdoshi,pwchen,zhangjunqing}@berkeley.edu

Abstract. The Satisfiability Modulo Theories and Oracles (SMTO) problem is an SMT problem that involves oracles which solvers can only interact with though querying and responses. Previous SMTO algorithms blindly navigate the search space without any clue. In this work, we explored the idea of using synthesized/learned oracle representations to guide SMTO solvers to find solutions more efficiently. Experiments show that the proposed approach significantly improved performance in workforce scheduling benchmarks, and also solved the UUV design problem which involves reasoning about fluid dynamic simulators.

1 Introduction

Recently, the satisfiability modulo theories and oracles (SMTO) problem [1] was proposed to formalize the notion of oracles as black-box solvers, which can only be interacted with through querying and responses. The SMTO problem defines checking the satisfiability of an SMT formula that may involve calls to these oracles under their defined interfaces.

The original SMTO algorithm treats the oracle calls as uninterpreted functions, and iteratively adds oracle assumptions to the formula – restrictions on the input output behavior of the oracle based on actual calls to oracles. However, in this approach, the SMT solver does not use information that should be gleaned from sequences of calls to the oracle, about what the oracle might actually be doing.

As a simple (toy) motivating example, suppose we'd like to check the satisfiability of the formula $(x > 0 \wedge \mathbf{add10}(x) \geq 40)$ in the theory of LIA, that uses a library function $\mathbf{add10} : \mathit{Int} \rightarrow \mathit{Int}$. As its name suggests, suppose **add10** simply adds 10 to its input. The solving algorithm will iteratively find a model for the formula with **add10** treated as an uninterpreted function, call the oracle to check the consistency of the oracle's behavior, and adds oracle

assumptions if inconsistent. In this case, the solver might solve the following sequence of formulae:

$$x = 1 \vdash (x > 0 \wedge \textbf{add10}(x) \geq 40), \text{ but } \textbf{add10}(1) < 40, \text{ so conjunct } \textbf{add10}(1) = 11$$
$$x = 2 \vdash (x > 0 \wedge \textbf{add10}(x) \geq 40 \wedge \textbf{add10}(1) = 11), \text{ but } \textbf{add10}(2) < 40, \text{ so conjunct } \textbf{add10}(2) = 12$$
$$x = 3 \vdash \ldots$$

and so forth till it finally reaches 40. Notice that even after collecting concrete examples (1, 11), (2, 12), (3, 13), the structure of the formula with all the oracle assumptions isn't enough to nudge the SMT solver to sample larger values. If we could learn a *rule* from the examples we've collected so far, that $\textbf{add10}(x) = x + 10$, we could replace the oracle call with our rule $x + 10$ to the formula:

$$\text{CHECK\_SAT}(x > 0 \wedge x + 10 \geq 40)$$

then the SMT solver would know to use sampling/symbolic optimizations to find a satisfying x much more quickly.

In this report, we investigate an approach to solving the SMTO problem using representations of the oracles – explicit functions in the SMT theory that can be substituted for the oracle function symbols. We investigate synthesizing such representations based on input/output examples the oracle provides us, using *SyGuS* in CVC4. We also investigate learning such representations based on three techniques from machine learning: *symbolic regression*, *decision tree learning*, and *neural networks*. While learning or synthesizing a perfect representation for an arbitrary oracle is impossible, if our representations can pick up simple patterns or bounds about the oracle's behavior, and if these representations can be added to the formula as theory specific SMT, then we hope to find satisfying instances much more quickly. Experiments show that the proposed approach significantly improved performance in workforce scheduling benchmarks, and also solved the UUV design problem which involves reasoning about fluid dynamic simulators.

2 Algorithm Overview

A satisfiability modulo oracles (SMTO) formula ρ contains a set of ordinary function symbols $\vec{f}$ and a set of oracle symbols $\vec{\theta}$ with each oracle symbol associated with an oracle interface $\vec{I}$. An SMTO formula ρ is

- *unsatisfiable* if $\exists \vec{f}.\exists \vec{\theta}.A \wedge \rho$ is unsatisfiable
- *satisfiable* if $\exists \vec{f}.\forall \vec{\theta}.A \implies \rho$ is satisfiable

where A is a conjunction of assumptions (implications) generated by $\vec{I}$.

The originally proposed SMTO algorithm treats an oracle function θ as an uninterpreted function and performs a consistency check between the interpretation $\theta(\vec{x})$ and the oracle output o after each solve, where $\vec{x}$ is the oracle input. If the consistency check fails, an assumption $\alpha := (\vec{x} \equiv \vec{c} \implies \theta(\vec{c}) \equiv o)$ is added to the assumption A to constrain the interpretation of θ in the next iteration. The set of assumptions A essentially builds up a lookup table for the oracle.

However, such lookup table does not provide insight into the oracle, as the table only records previously seen examples but does not infer examples that has not been seen before. This results in the solver randomly guessing assignments without knowing which are more likely to satisfy the formula. Such scenario can be inefficient when the oracle is of certain structure which however we are unable to exploit because of not utilizing well the earlier collected examples. We thus propose to *learn or synthesize representations* that mimic oracle behaviors.

Definition 1 (Oracle representation)**.** *Given a functional oracle $O : X \to Y$, under a certain theory T, we define an* oracle representation *as a function $R_O : X \to Y$ with the same domain and codomain as O.*

An oracle representation can be a function of any sort as long as it has the same domain/codomain. Let $\rho\{O \to R_O\}$ be the original SMTO formula ρ with all instances of O replaced with R_O. We informally say that R_O is a *useful representation* if satisfying models for $\rho\{O \to R_O\}$ are likely to be satisfying models for ρ. The task then is to synthesize or learn useful representations based on earlier collected examples to help guide the solver to more promising interpretations.

The new algorithm flow is depicted in Figure 2.1. First, we synthesize representations for oracles based on the oracle call histories (initially empty). The representations are then translated into SMTLIB [2] functions and then substituted into the formula, which now becomes oracle-free. If the formula is satisfiable, a consistency check is performed and if passed the SMTO formula is considered satisfiable; otherwise an assumption is added and the resynthesis procedure continues. If the formula is unsatisfiable, we omit the representation and solve the formula with the oracle being treated as an uninterpreted function. This guarantees the soundness of the algorithm as the representation might be unfaithful. Representation synthesis and consistency check are detailed in Section 3 and 4 respectively.

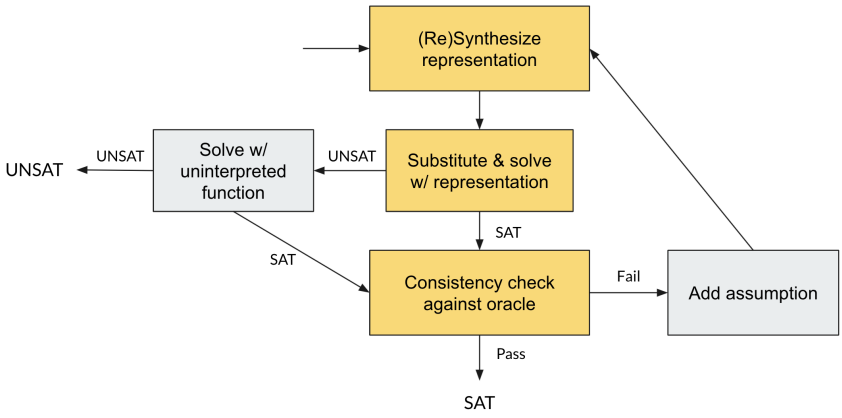


Figure 2.1: New algorithm flow for SMTO.

3 Generating Representations

We investigated two main ways of creating these representations: SyGuS and machine learning. In both cases, the primary set of input/output examples used to generate the representation comes from calls made to the oracle during the solving loop: i.e., the set of assumptions we collect becomes data to generate the representation from. An alternative path is to sample the oracle *beforehand*. We investigate this latter approach for neural networks, which tend to need much more examples to train on than other approaches.

3.1 Synthesizing Representations with SyGuS

For each oracle, given a set of input calls to the oracle and associated outputs, our SyGuS task is to find a function over some grammar that maps all inputs to outputs. In our implementation, we re-trigger synthesis with cvc4 every time a new example is collected.

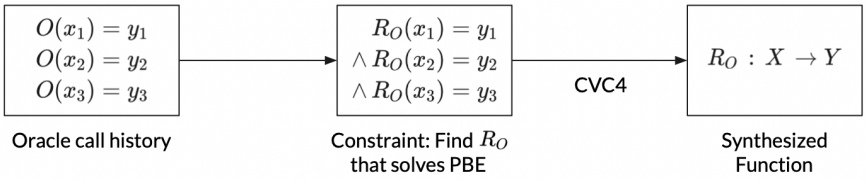


Figure 3.1: Synthesizing representations as a SyGuS problem

We use simple, preset grammars for different logics in the SyGuS task. Of course, the oracle itself might not adhere to our grammar (or even be expressible as a function in a formal grammar), but for the purposes of representations it suffices to use generic grammars that can generalize well to different tasks. Our grammar for fixed width bitvector logics, for example, uses addition, subtraction, comparison, left/right shifts, logical bitwise operations, and an ITE construct.

For many tasks (e.g. representing library functions), SyGuS may be sufficient to learn meaningful representations. In our experiments on processor pipeline verification and workforce scheduling (both in bitvector logics), SyGuS often proved successful at learning partial representations for oracles that were simple C functions.

3.2 Learning Representations with ML

However, synthesizing representations has several limitations. First, we found that cvc4/cvc5 did not yet scale well to tasks involving more than 5-10 examples. The other problem is that the there is no guarantee that the synthesized program will *generalize* to unseen behavior from the oracle. Note that this generalization property is very important to our algorithm, because if our representation has been generated from the oracle call history at a given iteration, the next inputs passed into that function will be inputs it hasn't seen before (since the SMT solver will incorporate the previously added oracle assumptions in its model).

We argue that these two desired properties of a representation: *scalability* and *generalizability*, motivate using *machine learning* based representations. The ML representation problem is a supervised learning task of mapping all arguments of a function call to the output. We restrict to oracles with outputs that can be approximated (integers, real numbers, and bitvectors which we treat as ints). The supervised learning problem is to find a function which minimizes some loss term between the functions outputs and the true values:

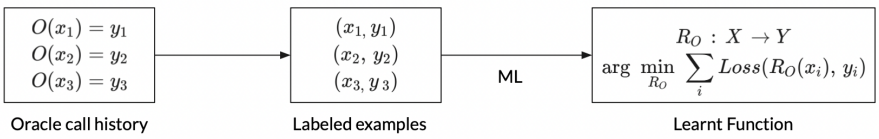


Figure 3.2: Learning representations as an ML problem

We now go over the three types of machine learning approaches we cover: genetic programming, neural networks, and decision trees.

3.2.1 Symbolic Regression (Genetic Programming)

Genetic programming based symbolic regression is a class of methods which use evolutionary algorithms to directly grow parse trees that best fit the input/output examples. From a large population of initial trees, successive generations are created based on which tree minimizes the mean squared error between predictions from the inputs and true output values. The advantage of this approach is that we can use a larger set of operators than most other ML representations, and the program tends to generate shallower function parse trees than other types.

We implemented this method with python's `gplearn`. Since this package assumes all variables are real numbers, for Int and BV logics all operators implicitly typecast their arguments to ints. We choose specific sets of operators and *constants* per logic:

- BV → bvshr, bvshl, bvadd, bvsub, bvmod, =, >, <, ITE, and integer constants between -5 and 5
- LIA → add, sub, mod, =, >, <, ITE, and integer constants between -5 and 5
- LRA → add, sub, mod, =, >, <, ITE, and real constants between -2 and 2

We empirically chose this set based on what performed well on our benchmarks. The genetic algorithm creates a population of 1000 parse trees f_i, evaluates fitness on them based on the mean squared error between f_i(inputs) and outputs, and selects the best 100 to seed the next generation. Finally, we convert the best performing program into valid SMTLIB, and return it to the SMTO solver for substitution.

3.2.2 Neural Networks

For many oracles in continuous domains, e.g. the LOGiCs vehicle design problem, and oracles that we do not know the logic structure of, e.g. image classifiers, a neural network can provide highly accurate approximations of the oracle's behavior, while also converging to more complex representations much faster than symbolic regression. We consider a simple feedforward networks with ReLU activations to allow simple SMTLIB translation.

To avoid collecting data and training the network while solving the SMTO problem, we can give the search process a "warm start" by pretraining the neural network with precollected data. We also observed that smaller neural networks are more desirable not only because they result in smaller SMTLIB functions, but also they tend to not overfit data and thus generalize better. A simple SMTLIB translation is provided below, where x_i are the inputs, w_i the weights, b the bias, k the logit, and z the output .

```
(assert (= k (+ (* w_i x_i)···)))
(assert (= z (ite (> k 0) k 0))))
```

3.2.3 Decision Trees and True/False Representations

Directly modeling the behavior of a complex oracle, like a physics simulator with continuous outputs, can be extremely challenging even with machine learning. However, note since we are only interested in finding satisfying instances to the SMT formula, it may suffice to model not the direct *output* of the oracle, but instead the boolean question of *whether or not the oracle's output satisfies its enclosing constraints*.

Let ϕ be our overall formula, and consider the case when ϕ can be decomposed into a set of constraints dependent on the oracle, and a set of constraints independent of it:

$$\phi = \phi_I \wedge \phi_O(O(x), z)$$

where O is the oracle, x are its inputs, and z are other free variables in the constraint. A *true/false representation* $R_{TF} : X, Z- > Bool$ is a function that returns *True* if the assignments to the oracle inputs x and free variables z satisfy its enclosing constraints, and *False*. Learning such boolean representations is complicated in the presence of recursive calls to the oracle, if multiple oracle calls are entangled, or if ϕ_O is large. However, in the LOGiCS problem described below, we consider a case where the only constraint dependent on an oracle is a simple comparison of the form $O(x) <= k$, which is a straightforward task for a machine learner. To generate examples at a given point, we collect z from the full SMT model as well as the oracle call history $x, O(x)$, evaluate $\phi_O(O(x), z)$ with these inputs, and label x, z with this boolean result.

We chose to use *decision trees* to learn boolean representations. Intuitively, decision trees are extremely effective at learning bounds in large sample spaces, because they involve simple comparison predicates on their input features. We chose a particular tree training algorithm suited to handling *data drift* [3], so that our can be effectively trained on new examples as they come in.

3.2.4 Ensuring consistency between representations and collected assumptions

One of the risks of ML representations is that they are usually not trained to perfect accuracy - the representation might return incorrect outputs even on inputs it's seen. This is dangerous, because it might be contradicting the *correct* input/output oracle assumptions that get conjoined to the formula. Omitting these assumptions from the formula isn't a good solution to this, because for a given input, if the incorrect representation satisfies its constraints but the corresponding oracle doesn't, the SMT solver might simply keep trying the same model as long as an incorrect representation is used.

To solve this, we ensure that the learnt ML representation is *always consistent* with the examples it was trained on. Before substituting a given learnt representation R_{learnt}, we check on which examples $\{(x_i, y_i)\}$ the model returned the incorrect values. We then *wrap* the representation with ITE constructs that correctly handle all the k incorrect examples:

$$R_{consistent} = ITE(x = x_1, y_1, ITE(x = x_2, y_2, ...ITE(x = x_k, y_k, R_O(x))))$$

This approach guarantees that at any given state, the substituted representation is perfectly consistent with the oracle assumptions.

4 Two-step Consistency Check

After obtaining a solution from the SMT solver, a consistency check is performed to ensure consistency between the interpretation and the oracle output. In the original SMTO implementation, the check consists of three steps:

1. Solve SMTO function ρ with oracle symbol $\theta(\vec{x})$ treated as uninterpreted function (or substituted by synthesized representation).
2. Call oracle binary O with input $\vec{c}$, the interpretation of oracle input $\vec{x}$.
3. Compare interpretation of $\theta(\vec{x})$ to $O(\vec{c})$.

The problem with this implementation is that the SMT solver can assign any interpretation to the uninterpreted function as long as it satisfies the constraint. To show the concept, consider the **add10** example in Section 1. If the solver always assigns 10000 as the interpretation to the **add10** oracle symbol, the solving process will not end until x reaches 9990. However, if x starts from 0 and is incremented by 1 in each iteration, the process should already

end when x reaches 30. This shows the inefficiency we have to face when the solver is allowed to randomly assign interpretations to oracle symbols.

We proposed a two-step consistency check to eliminate this mismatch problem. The consistency check consists of two SMT calls, where the first call is same as the above and the second call is performed after replacing the oracle symbol with the output of the oracle binary. This can be done by simply adding an assumption to enforce the constraint. We detail it as follows.

1. Solve SMTO function ρ with oracle symbol $\theta(\vec{x})$ treated as uninterpreted function (or substituted by synthesized representation).
2. Call oracle binary O with input $\vec{c}$, the interpretation of oracle input $\vec{x}$.
3. Replace oracle inputs and output by conjuncting assumption $(\vec{x} \equiv \vec{c} \wedge \theta(\vec{x}) \equiv O(\vec{c}))$ to ρ.
4. Solve ρ.

The consistency check passes if the result is SAT. This proposed consistency check is more efficient in that it does not require SMT solver to guess interpretations. Moreover, it is particularly useful in cases where the oracle output is of real values, e.g. neural network representations, since it is hard for SMT solvers to guess the exact same value. This is an important step towards guiding solvers with representations, as solvers are unlikely to correctly guess the oracle output themselves.

5 Experiments

Since SMTO is a very recent idea, there is as of yet a lack of standard benchmarks to evaluate our approach on. Instead, we experiment with SMT problems where certain operations have been *realistically oraclized* into external binaries (workforce scheduling and pipelined processor verification), and a *novel SMTO problem* involving finding constraints in the presence of simulators. All experiments were conducted on Macbook Pros with 32GB memory.

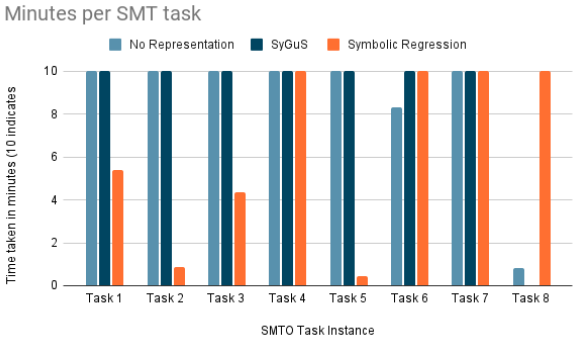
Our implementation [1] is based on the SMTO solver Delphi [1]. The main metrics we compare are the *total runtime*, as well as the *number of oracle calls*. The latter is an important metric, since we aim to reduce the number of calls made at the cost of learning a representation.

5.1 Workforce Scheduling

The *workforce scheduling* task [4] is a set of problems around automatically designing work schedules subject to labor, legal, and preferential constraints, represented as BitVector SMT2 problems. We considered a subset of 8 such SMT2 problems which was also evaluated by the original paper, which have had certain binary operations oraclized *realistically* (the SMTO problems contain many calls to these oracles, which indicate that *repetitive* operations were abstracted). We compared calling `delphi` with no representations, with symbolic regression representation learning, and with SyGuS representation synthesis:

Analysis: For tasks 1, 2, 3 and 5 , our symbolic regression learner was able to learn a representation that guided the solver to a solution significantly faster than having no representation. In task 1, the learnt representation was a perfect recreation of the oracle; however, in tasks 2, 3, 5, it sufficed to learn a partial representation of the oracle, and was wrapped in ITE's to handle the inconsistent cases. The biggest benefit is also seen in the significantly

[1] Our code is available at https://github.com/anishpdoshi/delphi.



Number of Oracle Calls	No Representation	SyGuS	Symbolic Regression
Task 1	512	15	78
Task 2	512	6	38
Task 3	342	6	73
Task 4	864	3	44
Task 5	1146	6	23
Task 6	128	7	128
Task 7	3	3	3
Task 8	490	1	343

Figure 5.1: Time taken and number of oracle calls on the 8 workforce scheduling tasks. On the left, the max bar height indicates that the solver was taking >= 10 minutes, at which point we timed it out. On the right, cells are highlighted in green if `delphi` actually completed for that task with that representation type within 10 minutes.

lower number of oracle calls required for all problems that completed.
SyGuS representation synthesis struggled whenever more than 3 examples were needed; otherwise, it completed extremely fast with a perfect representation of the oracle. In the future, more tuning of cvc4/cvc5's options, combined with potentially relaxing the PBE problem, might improve its scalability.
Interestingly, on task 6 our method actually *worsened* performance on task 6. The actual oracle was simply performing $x\&(x-1)$, which is a fairly shallow program and should theoretically be easily solvable with symbolic regression. We note, however, that this task used the less common bitvector width 9, which might have made it impossible for our configuration to find a solution; better BV typed genetic programming might help with this problem.

5.2 LOGICS Problem: Underwater Unmanned Vehicle (UUV) Design

In this problem, we aim to find an assignment to a set of seven parameters to generate a design that satisfies two constraints: 1) the design has to contain a fixed-sized payload, and 2) the design has to have drag force less than a certain threshold. The first constraint can easily be formulated as a logical formula, while the second one involves calling a computational fluid dynamics (CFD) simulator to obtain drag force. As it is extremely expensive to perform CFD simulations (5 to 10 minutes each), we built a surrogate model as a proxy for the simulator.

We collected 100 examples and trained two neural networks with a layer of 50 and 10 neurons respectively. Standard scaling was also applied to normalize features to better train the networks. For the 50-neuron network, the translated SMTLIB function was too large to solve. The 10-neuron network scales much better and also has comparable validation loss to the 50-neuron one. With the neural network, the SMTO solver was able to find assignments that the neural network deemed promising; on the other hand, without the network, the SMTO solver started with a solution and incremented one of the parameter by one in each iteration. We noticed that the search was more efficient when using the neural network representation. We also applied the decision tree true/false prediction - the decision tree was pretrained on the 100 samples, and achieved 86% validation accuracy on the binary prediction task: checking whether or not given input parameters would result in a drag force of under 45 N.

Both representations were able to solve the SMTO problem in an hour: it took only **one iteration** to find a satisfying solution that is consistent with the CFD simulator. The original algorithm, without representations, was unable to terminate.

6 Further Technical Challenges and Future Work

6.1 Dealing with UNSAT

One of the main limitations of our current algorithm is when the problem is fundamentally UNSAT, we are simply adding overhead - no matter what representation we try, we always have to do an oracle consistency check, which will always return UNSAT. To avoid this in our current implementation, if the SMT solver returns UNSAT on a formula with a substituted representation, then we *never* try substituting a representation again.

However, this approach may be overaggressive, because it may simply be the case that our representation may simply just be overfitted - it might have overconstrained the search space. Instead, we should try to learn/synthesize a representation that is less *complex*, in the hopes that it will not overconstrain the problem. For example, each time we hit UNSAT, we should try *pruning* the representation. In the context of neural networks, this might involve network pruning or retraining with less hidden units; in the context of symbolic regression/decision trees, we might restrict the max depth of the tree; in the context of SyGuS, we might relax the PBE equality constraint to be a set of *approximate* constraints of the form $y - k < R_O(x) < y + k$ (or, try a MAX-SMT approach).

6.2 Scalability

In our implementation, the C++ solver `delphi` repetitively calls a Python process to learn representations (with the `gplearn`, `pytorch`, and `river`). We can improve this by instead C++ specific libraries, and by using *online* training algorithms - i.e., not retrain the model from scratch every time an example comes in, but instead save its state and retrain only on new examples.

6.3 SyMO

Synthesis modulo oracles (SyMO) is the problem of synthesizing functions to satisfy constraints that include oracle functions. The SyMO algorithm uses SMTO queries in its loop, but the inputs passed into the SMTO solver's oracles are now *functions* - the oracles in SyMO are often *program* verifiers, e.g. in CEGIS. Thus, representations of these oracles would have to accept functions as inputs. One potential approach is to tokenize/parse the function into a string/BV, and try to learn representations that approximate behavior with this; another approach is to sample input/output examples from candidate functions and try to learn representations between inputs to candidate functions and corresponding outputs of the oracle.

6.4 Lack of Benchmarks

We searched for problems that can be suitably framed as an SMTO problem, and the pipelined processor verification problem was a promising candidate. We attempted to oraclize the ALU module, translate the verification problem into SMTO using UCLID5 [5], and solve it using our implementation. Though our implementation did not improve much on the performance, we hope that determining the right extent (module) of abstraction could improve verification of the system. We also searched on SMTLIB, SV-COMP for benchmarks, but they are either too simple or incomprehensible (understanding benchmarks is crucial to crafting oracles). We are still looking for benchmarks that suit our needs.

7 Conclusions

We have detailed a method for using representations to speed up finding satisfying instances to SMTO problems. By substituting a SMT compatible functional representation, learnt or synthesized from oracle input/output examples, we can often dramatically reduce the time taken to find satisfying instances. Our approach is extremely

useful when the oracle is expensive to call, as is the case with the LOGiCS simulator. In the future, we aim to continue finding classes of problems for which certain types of representations are useful, and optimizing our implementation to solve such problems even more quickly.

8 Miscellaneous

8.1 Individual Contributions

Anish built the SyGuS representation learner, the symbolic regression learner, formalized the true/false prediction task and built the decision tree learner, converting all the models into SMT, ensured consistency of representations with assumptions, wired up calling the representation from `delphi` and passing in CLI options, helped train the NN for the LOGiCs problem, called the workforce scheduling benchmarks and wrote sections 1, 3, 5.1, 6, 7, and parts of 8 of this report.

Pei-Wei came up with and formalized the framework of representation substitution, implemented representation substitution, two-step consistency check, and fallback in the case of incorrect representation, designed and worked on the SyGuS representation learner and the LOGiCS problem (data collection, model training, benchmark design, oracle design and implementation), and wrote abstract, sections 2, 3.2.2, 4, 5.2, 6.4, and part of 8.

Junqing discussed with teammates to make the proposal, worked on the benchmarks of Pipelined Processor Verification, built oracle functions to substitute the original functions in the benchmarks, generated the smt2 files for `delphi` to test the satisfiability, added new operation for real type in cbmc and contributed part 6.4 of the report.

8.2 Links to class

Our project is heavily tied to SMT solving, and uses SyGuS as a major component (and potential future application). We are aiming to improve a particular method that was introduced in the last lecture.

8.3 Feedback

We really enjoyed this class! Very much liked the hybrid lecture format, the homeworks that used real world tools (although some more debugging support for them could have helped), and feedback on the projects was very useful to guide us. In the future, it would perhaps be useful to have some discussion sections or handouts with worked through examples of the formal methods we learned in class.

References

[1] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. Satisfiability and synthesis modulo oracles. 2022.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[3] Albert Carles Bifet Figuerol and Ricard Gavaldà Mestre. Adaptive parameter-free learning from evolving data streams. 2009.

[4] Verfassung der Arbeit and Christoph Erkinger. Rotating workforce scheduling as satisfiability modulo theories. 2013.

[5] Sanjit A. Seshia and Pramod Subramanyan. Uclid5: Integrating modeling, verification, synthesis and learning. 2018.