Satisfiability and Synthesis Modulo Oracles

Citation for published version:
Polgreen, E, Reynolds, A & Seshia, SA 2022, Satisfiability and Synthesis Modulo Oracles. in B Finkbeiner & T Wies (eds), *Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings.* Lecture Notes in Computer Science, vol. 13182, Springer International Publishing, Cham, Switzerland, pp. 263-284, 23rd International Conference on Verification, Model Checking, and Abstract Interpretation, Philadelphia, Pennsylvania, United States, 16/01/22. https://doi.org/10.1007/978-3-030-94583-1_13

Digital Object Identifier (DOI):
10.1007/978-3-030-94583-1_13

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Verification, Model Checking, and Abstract Interpretation

Satisfiability and Synthesis Modulo Oracles

Elizabeth Polgreen
University of Edinburgh and UC Berkeley

Andrew Reynolds
University of Iowa

Sanjit A. Seshia
UC Berkeley

***Abstract*—In classic program synthesis algorithms, such as counterexample-guided inductive synthesis (CEGIS), the algorithms alternate between a synthesis phase and an oracle (verification) phase. Many synthesis algorithms use a white-box oracle based on satisfiability modulo theory (SMT) solvers to provide counterexamples. But what if a white-box oracle is either not available or not easy to work with? We present a framework for solving a general class of oracle-guided synthesis problems which we term *synthesis modulo oracles*. In this setting, oracles may be black boxes with a query-response interface defined by the synthesis problem. As a necessary component of this framework, we also formalize the problem of *satisfiability modulo theories and oracles*, and present an algorithm for solving this problem. We implement a prototype solver for satisfiability and synthesis modulo oracles and demonstrate that, by using oracles that execute functions not easily modeled in SMT-constraints, such as recursive functions or oracles that incorporate compilation and execution of code, SMTO and SyMO are able to solve problems beyond the abilities of standard SMT and synthesis solvers.**

I. Introduction

A common formulation of program synthesis is to find a program, from a specified class of programs, that meets some correctness specification [5]. Classically, this is encoded as the 2nd-order logic formula $\exists \vec{f} \forall \vec{x}\ \phi$, where $\vec{f}$ is a set of target functions to be synthesized, $\vec{x}$ is a set of 0-ary symbols, and ϕ is a quantifier-free formula in a logical theory (or combination of theories) T. A tuple of functions $\vec{f^*}$ satisfies the semantic restrictions if the formula $\forall \vec{x}\ \phi$ is valid in T when the tuple is substituted for $\vec{f}$ in ϕ. Many problems are specified in this form, and the SyGuS-IF format [24] is one way of specifying such syntax-guided synthesis (SyGuS) problems.

Whilst powerful, this format is restrictive in one key way: it requires the correctness condition to be specified as a satisfiability modulo theories (SMT) [9] formula in a fully white-box manner. Put another way, SyGuS problems must be specified with static constraints before the solving process begins. This limits the problems that can be specified, as well as the oracles that can be used to guide the search. For example, if one wants to synthesize (parts of) a protocol whose correctness needs to be checked by a temporal logic model checker (e.g. [30]), such a model-checking oracle cannot be directly invoked within a general-purpose SyGuS solver and instead requires creating a custom solver.

Similarly, SMT solvers, used widely in verification and synthesis, require their input to be encoded as a logical formula prior to the initiation of solving. Whilst the language of SMT-LIB is powerful and expressive, many formulas are challenging for current SMT solvers to reason about; e.g., as in Figure 1, finding a prime factorization of a given number. Here it would be desirable to abstract this reasoning to an external oracle, rather than rely on the SMT solver's ability to reason about recursive functions.

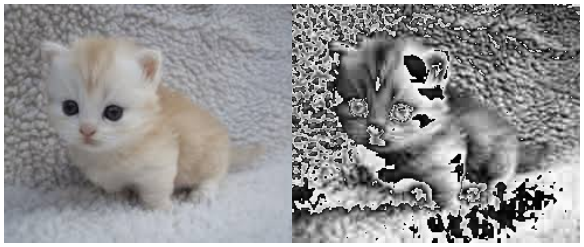
```
(define-fun-rec isPrimeRec ((a Int) (b Int)) Bool
  (ite (> b (div a 2)) true
    (ite (= (mod a b) 0)
      false
      (isPrimeRec a (+ b 1)))))

(define-fun isPrime ((a Int)) Bool
  (ite (<= a 1)
    false
    (isPrimeRec a 2)))

(assert (and (isPrime f1)(isPrime f2)(isPrime f3)))
(assert (= (* f1 f2 f3) 76))
```

Fig. 1: SMT problem fragment: find prime factors of 76. Unsolved by CVC4 v1.9. Solved by SMTO using isPrime oracle in < 1s

This motivates our introduction of oracles to synthesis and SMT solving. Oracles can be black-box implementations that can be queried based on a pre-defined interface of query and response types. Examples of oracles could be components of systems that are too large and complex to analyze (but which can be executed on inputs) or external verification engines solving verification queries beyond SMT solving.

Prior work has set out a theoretical framework expressing synthesis algorithms as oracle-guided inductive synthesis [21], where a learner interacts with an oracle via a pre-defined oracle interface. However, this work does not give a general algorithmic approach to solve oracle-guided synthesis problems or demonstrate the framework on practical applications. An important contribution we make in this work is to give *a unified algorithmic approach to solving oracle-guided synthesis* problems, termed SyMO. The SyMO approach is based on a key insight: that query and response types can be associated with two types of logical formulas: *verification assumptions* and *synthesis constraints*. The former provides a way to encode restrictions on black-box oracle behavior into an SMT formula, whereas the latter provide a way for oracles to guide the search of the synthesizer.

In order to explain the use-case for assumptions, let us first introduce *oracle function symbols* and *Satisfiability Modulo Theories and Oracles (SMTO)*. Oracle function symbols are n-ary symbols whose behavior is associated with some oracle. Consider a quantifier-free formula ρ which contains an oracle function symbol θ. SMTO looks for a satisfying assignment



(a) Original image (b) Target image

Fig. 2: Image manipulation: transformations synthesized by SyMO in < 1 sec.

to the formula based on initially assuming θ is a universally quantified uninterpreted function (i.e., we look for a satisfying assignment that would work for any possible implementation of the oracle): $\forall \theta \rho$. As we make calls to the oracle, we begin to learn more about its behavior, and we encode this behavior as assumptions α, such that the formula becomes $\forall \theta \alpha \Rightarrow \rho$. This is the primary use case for assumptions generated by oracles, they are used to constrain the behavior of oracle function symbols. In SyMO, determining the correctness of a candidate function is an SMTO problem, and assumptions generated by oracles are used in the SMTO solving process.

As an exemplar of an existing oracle-guided synthesis algorithm that goes beyond the SMT-solver-based counterexample oracles, consider ICE-learning [19] for invariant synthesis. ICE-learning uses three oracles: an oracle to provide positive examples (examples which *should* be contained within the invariant); an oracle to provide negative examples (examples which *should not* be contained within the invariant); and an oracle to provide implication examples (an example pair where if the first element is contained within the invariant, both must be contained). Whilst it is possible to build some of these oracles using an SMT solver, it is often more effective to construct these oracles in other ways, for instance, the positive example oracle can simply execute the loop or system for which an invariant is being discovered and return the output.

We implement SyMO in a prototype solver Delphi, and hint at its broad utility by demonstrating several applications including programming by example, synthesis of controllers for LTI systems, synthesizing image transformations (e.g., Figure 2, and satisfiability problems that reason about primes (e.g., Figure 1). The latter use cases illustrate the power of being able to incorporate oracles into SyMO that are too complex to be modeled or for SMT solvers to reason about.

To summarize, the main contributions of this paper are:

- A formalization of the problem of satisfiability and synthesis modulo oracles (Sec. II);
- A unifying algorithmic approach for solving these problems (Sec. III and Sec. IV);
- Demonstration of how this approach can capture popular synthesis strategies from the literature (Sec. V), and
- A prototype solver Delphi, and an experimental demonstration of the broad applicability of this framework (Sec. VI).

Related work: Almost all synthesis algorithms can be framed as some form of oracle-guided synthesis. Counterexample-guided inductive synthesis (CEGIS) is the original synthesis strategy used for Syntax-Guided Synthesis [29], and uses a correctness oracle that returns counterexamples. Further developments in synthesis typically fall into one of two categories. The first comprises innovative search algorithms to search the space more efficiently; for instance, genetic algorithms [16], reinforcement learning [28], or partitioning the search space in creative ways [6]. The second category comprises extensions to the communication paradigm permitted between the synthesis and the verification phase. For instance, CEGIS modulo theories [4], CEGIS(T), extends the oracle interface over standard CEGIS to permit responses in the form of a restricted set of constraints over constants in the candidate program, while CVC4's single-invocation algorithm [27] makes full use of the white-box nature of the SMT solver oracle. Other work leverages the ability to classify counterexamples as positive or negative examples [23]. There are also notable algorithms in invariant synthesis based on innovative use of different query types [23], [19]. Our work has one key stand-out difference over these: in all of these algorithms, the correctness criteria must be specified as a logical formula, whereas in our framework we enable specification of the correctness criteria as a combination of a logical formula and calls to external oracles which may be opaque to the solver. Synthesis with distinguishing inputs [20] is an exception to this pattern and uses a specific set of three interacting black-box oracles, to solve the very specific problem of synthesis of loop-free programs from components. Our work differs from this and the previously-mentioned algorithms in that they are customized to use certain specific types of oracle queries, whereas, we give a "meta-solver" allowing any type of oracle query that can be formulated as either generating a constraint or an assumption in the form of a logical formula.

The idea of satisfiability with black-boxes has been tackled before on work on abstracting functional components as uninterpreted/partially-interpreted functions (see, e.g., [7], [14], [13]), which use counterexample-guided abstraction refinement [15]. Here, components of a system are abstracted and then refined based on whether the abstraction is sufficiently detailed to prove a property. However, to do this, the full system must be provided as a white-box. The key contribution our work makes in this area is a framework allowing the use of black-box components that obey certain query-response interface constraints, where the refinement is dictated by these constraints and the black-box oracle interaction.

II. Oracles

In this section, we introduce basic definitions and terminology for the rest of the paper. We begin with some preliminaries about SMT and synthesis.

A. Preliminaries and Notation

We use the following basic notations throughout the paper. If e is an expression and x is free in e, let $e \cdot \{x \rightarrow t\}$ be the formula obtained from the formula e by proper substitution of the variable x by the variable t.

1) Satisfiability Modulo Theories (SMT): The input to an SMT problem is a first-order logical formula ρ. We use $\approx$ to denote the (infix) equality predicate. The task is to determine whether ρ is T-satisfiable or T-unsatisfiable, that is, satisfied by a model which restricts the interpretation of symbols in ρ based on a background theory T. If ρ is satisfiable, a solver will usually return a model of T that makes ρ true, which will include assignments to all free variables in ρ. We additionally say that a formula is T-valid if it is satisfied by *all* models of T.

2) Syntax-Guided Synthesis: In syntax-guided synthesis, we are given a set of functions $\vec{f}$ to be synthesized, associated languages of expressions $\vec{L} = L_1, \ldots, L_m$ (typically generated by grammars), and we seek to solve a formula of the form

$$\exists \vec{f} \in \vec{L} \forall \vec{x} \ \phi$$

where $\vec{x} \approx x_1 \ldots x_n$ is a set of 0-ary symbols and ϕ is a quantifier-free formula in a background theory T. In some cases, the languages L_i include all well-formed expressions in T of the same sort as f_i, and thus L_i can be dropped from the problem. A tuple of candidate functions $\vec{f^*}$ satisfies the semantic restrictions for functions-to-synthesize $\vec{f}$ in conjecture $\exists \vec{f} \forall \vec{x} \, \phi$ in background theory T if $\forall \vec{x} \, \phi$ is valid in T when $\vec{f}$ are defined to be terms whose semantics are given by the functions $(\vec{f^*})$ [5], [24].

B. Basic Definitions

We use the term *oracle* to refer to a (possibly black-box) component that can be queried in a pre-defined way by the solver. An oracle interface defines how an oracle can be queried. This concept is borrowed from [21]. We extend the definition of oracle interfaces to also provide the solver with information on the *meaning* of the response, in the form of expressions that generate assumptions or constraints.

Definition II.1 (Oracle Interface)**.** *An oracle interface $\mathcal{I}$ is a tuple $(\vec{y}, \vec{z}, \alpha_{gen}, \beta_{gen})$ where:*

- *$\vec{y}$ is a list of sorted variables, which we call the* query domain *of the oracle interface;*
- *$\vec{z}$ is a list of sorted variables, which we call its* response co-domain*;*
- *α_{gen} is a formula whose free variables are a subset of $\vec{y}, \vec{z}$, which we call its* assumption generator*; and*
- *β_{gen} is a formula whose free variables are a subset of $\vec{y}, \vec{z}$, which we call its* constraint generator.

□

We assume that all oracle interfaces have an associated oracle that implements their prescribed interface for *values* of the input sort, and generates concrete values as output. In particular, an oracle for an oracle interface of the above form accepts a tuple of values with sorts matching $\vec{y}$, and returns a tuple of values with sorts matching $\vec{z}$. It is important to note that the notion of a value is specific to a sort, which we intentionally do not specify here. In practice, we assume e.g. the standard values for the integer sort; we assume all closed lambda terms are values for higher-order sorts, and so on.

$$I = \begin{cases} Q & : (y_1 : \sigma_1), \ldots, (y_j : \sigma_j) \\ R & : (z_1 : \sigma'_1), \ldots, (z_k : \sigma'_k) \\ \alpha_{gen} & : \textit{assumption generator} \\ \beta_{gen} & : \textit{constraint generator} \end{cases}$$

Fig. 3: Oracle interface

An oracle interface defines how assumptions and constraints can be given to a solver via calls to black-box oracles, as given by the following definition.

Definition II.2 (Assumptions and Constraints Generated by an Oracle Interface)**.** *Assume $\mathcal{I}$ is an oracle interface of form $(\vec{y}, \vec{z}, \alpha_{gen}, \beta_{gen})$. We say formula $\alpha_{gen} \cdot \{\vec{y} \to \vec{c}, \vec{z} \to \vec{d}\}$ is an assumption generated by $\mathcal{I}$ if calling its associated oracle for input $\vec{c}$ results in output $\vec{d}$. In this case, we also say that $\beta_{gen} \cdot \{\vec{y} \to \vec{c}, \vec{z} \to \vec{d}\}$ is a constraint generated by $\mathcal{I}$.* □

We are now ready to define the main problems introduced by this paper. In the following definition, we distinguish two kinds of function symbols: *oracle function symbols*, which are given special semantics in the following definition; all others we call *ordinary function symbols*. As we describe in more detail in Section III, oracle function symbols allow us to incorporate function symbols that correspond directly to oracles in specifications and assertions.

Definition II.3 (Satisfiability Modulo Theories and Oracles)**.** *A satisfiability modulo oracles (SMTO) problem is a tuple $(\vec{f}, \vec{\theta}, \rho, \vec{\mathcal{I}})$, where $\vec{f}$ is a set of ordinary function symbols, $\vec{\theta}$ is a set of oracle function symbols, ρ is a formula in a background theory T whose free function symbols are $\vec{f} \uplus \vec{\theta}$, and $\vec{\mathcal{I}}$ is a set of oracle interfaces. We say this input is:*

- *unsatisfiable if $\exists \vec{f} . \exists \vec{\theta} . A \wedge \rho \wedge B$ is T-unsatisfiable,*
- *satisfiable if $\exists \vec{f} . \forall \vec{\theta} . A \Rightarrow (\rho \wedge B)$ is T-satisfiable,*

where, in each case, A (resp. B) is a conjunction of assumptions (resp. constraints) generated by $\vec{\mathcal{I}}$. □

According to the above semantics, constraints are simply formulas that we conjoin together with the input formula. Assumptions play a different role. In particular, they restrict the possible interpretations of $\vec{\theta}$ that are relevant. As they appear in the antecedent in our satisfiability criteria, values of $\vec{\theta}$ that do not satisfy our assumptions need not be considered when determining whether an SMTO input is satisfiable. As a consequence of the quantification of $\vec{\theta}$, by convention we will say a model M for an SMTO problem contains interpretations for function symbols in $\vec{f}$ only; the values for $\vec{\theta}$ need not be given.

It is important to note the role of the quantification for oracle symbols $\vec{\theta}$ in the above definition. An SMTO problem is unsatisfiable if the conjunction of assumptions, input formula, and constraints are unsatisfiable when treating $\vec{\theta}$ existentially, i.e. as uninterpreted functions. Conversely, an SMTO problem is satisfiable only if there exists a model satisfying $(\rho \wedge B)$ for *all* interpretations of $\vec{\theta}$ for which our assumptions A hold.

In the absence of restrictions on oracle interfaces $\vec{\mathcal{I}}$, an SMTO problem can be both satisfiable and unsatisfiable, depending on the constraints and assumptions generated. For instance, when A becomes equivalent to false, the input is trivially both unsatisfiable and satisfiable. However, in practice, we define a restricted fragment of SMTO, for which this is not the case, and we present a dedicated procedure for this fragment in Section III. To define this fragment, we introduce the following definition.

Definition II.4 (Oracle Interface Defines Oracle Function Symbol)**.** *An oracle interface $\mathcal{J}$* defines *an oracle function symbol θ if it is of the form $((y_1, \dots y_j), (z), \theta(y_1, \dots y_j) \approx z, \emptyset)$, and its associated oracle $\mathcal{O}$ is functional. In other words, calling the oracle interface generates an equality assumption of the form $\theta(y_1, \dots y_j) \approx z$ only.* □

From here on, as a convention, we use $\mathcal{J}$ to refer to an oracle interface that specifically defines an oracle function symbol, and $\mathcal{I}$ to refer to a free oracle interface, i.e., an oracle interface which may not define an oracle function symbol.

Definition II.5 (Definition Fragment of SMTO)**.** *An SMTO problem $(\vec{f}, \vec{\theta}, \rho, \vec{\mathcal{J}})$ is in Definitional SMTO if and only if $\vec{\theta} = (\theta_1, \dots, \theta_n)$, $\vec{\mathcal{J}} = (\mathcal{J}_1, \dots, \mathcal{J}_n)$, and $\mathcal{J}_i$ is an oracle interface that defines θ_i for $i = 1, \dots, n$.* □

Note that each oracle function symbol is defined by one and only one oracle interface.

We are also interested in the problem of synthesis in the presence of oracle function symbols, which we give in the following definition.

Definition II.6 (Synthesis Modulo Oracles)**.** *A synthesis modulo oracles (SyMO) problem is a tuple $(\vec{f}, \vec{\theta}, \forall \vec{x}.\ \phi, \vec{\mathcal{I}})$, where $\vec{f}$ is a tuple of functions (which we refer to as the functions to synthesize), $\vec{\theta}$ is a tuple of oracle function symbols, $\forall \vec{x}.\ \phi$ is a formula is some background theory T where ϕ is quantifier-free, and $\vec{\mathcal{I}}$ is a set of oracle interfaces. A tuple of functions $\vec{f^*}$ is a solution for synthesis conjecture if $(\vec{x}, \vec{\theta}, \neg\phi \cdot \{\vec{f} \to \vec{f^*}\}, \vec{\mathcal{I}})$ is unsatisfiable modulo theories and oracles.* □

Although not mentioned in the above definition, the synthesis modulo oracles problem may be combined with paradigms for synthesis that give additional constraints for $\vec{f}$ that are not captured by the specification, such as syntactic constraints in syntax-guided synthesis. In Section IV, we present an algorithm for a restricted form of SyMO problems where the verification of candidate solutions $\vec{f^*}$ reduces to Definitional SMTO.

III. Satisfiability Modulo Theories and Oracles

In this section, we describe our approach to solving inputs in the definition fragment of SMTO, according to Definition II.5. First, we note a subtlety with respect to satisfiability of SMTO problems in the definition fragment vs. the general problem. Namely that a problem cannot be both satisfiable and unsatisfiable, and once a result is obtained for Definitional SMTO, the result will not change regardless of subsequent calls to the oracles. This is not true for the general SMTO problem. In particular, note the following scenarios:

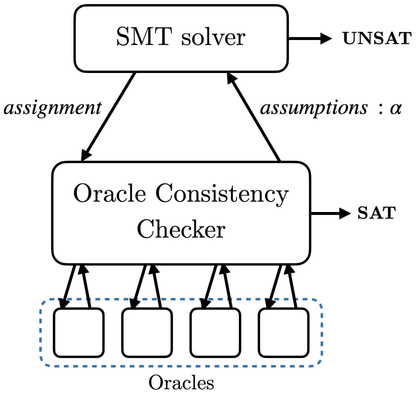


Fig. 4: Satisfiability Modulo Oracle Solver

a) Conflicting Results: Assume that $\exists \vec{f}.\exists\vec{\theta}.A_i \wedge \rho \wedge B_i$ is T-unsatisfiable, where A_i (resp. B_i be the conjunction of assumptions (resp. constraints) obtained after i calls to the oracles. In unrestricted SMTO, it is possible that A_i alone is T-unsatisfiable, thus $\forall\vec{\theta}A_i \Rightarrow (\rho \wedge B_i)$ is T-satisfiable and the problem is both satisfiable and unsatisfiable. However, in Definitional SMTO, it is impossible for A_i alone to be unsatisfiable, since all oracle interfaces defining oracle function symbols, which generate assumptions only of the form $\theta(\vec{y}) \approx z$ and the associated oracles are functional.

b) Non-fixed Results: Assume that $\exists \vec{f}.\forall\vec{\theta}.A_i \Rightarrow (\rho \wedge B_i)$ is T-satisfiable, where A_i (resp. B_i be the conjunction of assumptions (resp. constraints) obtained after i calls to the oracles. Thus, by Definition II.3, our input is satisfiable. In unrestricted SMTO, it is possible for an oracle to later generate an additional constraint β such that $\forall\vec{\theta}A_i \Rightarrow (\rho \wedge B_i \wedge \beta)$ is T-unsatisfiable, thus invalidating our previous result of "satisfiable". However, in Definitional SMTO, this cannot occur, since oracles that generate non-trivial constraints are not permitted. It is trivial that once any SMTO is unsatisfiable, it remains unsatisfiable. Thus the satisfiability results for Definitional SMTO, once obtained, are fixed.

A. Algorithm for Definitional SMTO

Our algorithm for Definitional SMTO is illustrated in Figure 4 and given as Algorithm 1. The algorithm maintains a dynamic set of assumptions A generated by oracles. In its main loop, we invoke an off-the-shelf SMT solver (which we denote SMT) on the conjunction of ρ and our current assumptions A. If this returns UNSAT, then we return UNSAT along with the set of assumptions A we have collected. Otherwise, we obtain the model M generated by the SMT solver from the previous call.

The rest of the algorithm (lines 8 to 20) invokes what we call the *oracle consistency checker*. Intuitively, this part of the algorithm checks whether our assumptions A about $\vec{\theta}$ are consistent with the external implementation the oracle function symbols are associated with.

We use the following notation: we write $e[t]$ to denote an expression e having a subterm t, and $e[s]$ to denote replacing that subterm with s. We write $t{\downarrow}$ to denote the result of *partially evaluating* term t. For example, $(\theta(1+1)+1){\downarrow} = \theta(2)+1$.

In the oracle consistency checker, we first construct the formula μ which replaces in ρ all occurrences of ordinary function symbols f with their value in the model M, and partially evaluate the result. Thus, initially, μ is a formula whose free symbols are $\vec{\theta}$ only. The inner loop (lines 9 to 17) incrementally simplifies this formula by calling external oracles to evaluate (concrete) applications of functions from $\vec{\theta}$. In particular, while μ contains at least one application of a function from $\vec{\theta}$, that is, it is of the form $\mu[\theta_i(\vec{c})]$ where $\vec{c}$ is a vector of values. We know that such a term exists by induction, noting that an innermost application of a function from $\vec{\theta}$ must be applied to values. We replace this term with the output d obtained from the appropriate oracle. The call to the oracle for input values $\vec{c}$ may already exist in A; otherwise, we call the oracle $\mathcal{J}_i$ for this input and add this assumption to A. After replacing the application with d, we partially evaluate the result and proceed. In the end, if our formula μ is the formula $true$, the consistency check succeeds and we return SAT, along with the current set of assumptions and the model M. We restrict the returned model so that it contains only interpretations for $\vec{f}$ and not $\vec{\theta}$, which we denote $M\mid_{\vec{f}}$. This process repeats until a model is found that is consistent with the oracles, or until the problem is shown to be unsatisfiable.

We will now show that this intuitive approach is consistent with the previously defined semantics for SMTO.

Theorem III.1 (Correctness of SMTO algorithm)**.** *Algorithm 1 returns UNSAT (resp. SAT) iff the SMTO problem $(\vec{f}, \vec{\theta}, \rho, \vec{\mathcal{J}})$ is unsatisfiable (resp. satisfiable) according to Definition II.3.*

Proof. UNSAT case: By definition, an SMTO problem is unsatisfiable if $\exists \vec{f}.\exists \vec{\theta}.A \wedge \rho$ is T-unsatisfiable, noting that for the definitional fragment of SMTO, B is empty. Algorithm 1 returns UNSAT when the underlying SMT solver returns UNSAT on the formula $\rho \wedge A_0$ for some A_0. Since A_0 is generated by oracles $\vec{\mathcal{J}}$, it follows that our input is unsatisfiable.

SAT case: By definition, an SMTO problem is SAT *iff* $\exists \vec{f}.\forall \vec{\theta}.A \Rightarrow \rho$ is T-satisfiable for some A. Algorithm 1 returns SAT when $\rho \wedge A_0$ is SAT with model M for some A_0, and when the oracle consistency check subsequently succeeds. Assume that the inner loop (lines 9 to 17) for this check ran n times and that a superset A_n of A_0 is returned as the set of assumptions on line 19. We claim that $M\mid_{\vec{f}}$ is a model for $\forall \vec{\theta}.A_n \Rightarrow \rho$. Let M' be an arbitrary extension of $M\mid_{\vec{f}}$ that satisfies A_n. Note that such an extension exists, since by definition of Definitional SMTO, A_n is a conjunction of equalities over distinct applications of $\vec{\theta}$. Let $\mu_0, \mu_1, \ldots, \mu_n$ be the sequence of formulas such that μ_i corresponds to the value of μ after i iterations of the loop on lines 9 to 17. We show by induction on i, that M' satisfies each μ_i. When $i = n$, μ_i is $true$ and the statement holds trivially. For each $0 \leq i < n$, we have that μ_i is the result of replacing an occurrence of $\theta(\vec{c})$ with d in μ_{i-1} and partially evaluating the result, where $\theta(\vec{c}) \approx d \in A_n$. Since M' satisfies $\theta(\vec{c}) \approx d \in A_n$ and by the induction hypothesis satisfies μ_i, it satisfies μ_{i-1} as well. Thus, M' satisfies μ_0, which is $(\rho \cdot \{\vec{f} \rightarrow \vec{f}^M\}){\downarrow}$. Thus, since M' is an arbitrary extension of $M\mid_{\vec{f}}$ satisfying A_n, we have that $M\mid_{\vec{f}}$ satisfies $\forall \vec{\theta}.A_n \Rightarrow \rho$ and thus the input is indeed satisfiable. □

Algorithm 1: Satisfiability Modulo Oracles (SMTO)

```
   input  : $(\vec{f}, \vec{\theta}, \rho, \vec{\mathcal{J}})$
   output : UNSAT/SAT + assumptions $A$ + (model $M$)?
1  Algorithm SMTO
2      $A \leftarrow true$
3      while true do
4          if SMT($\rho \wedge A$)=UNSAT then
5              return UNSAT, A
6          else
7              Let $M$ be model for $\rho \wedge A$ from $SMT$
8              Let $\mu$ be $(\rho \cdot \{\vec{f} \rightarrow \vec{f}^M\}){\downarrow}$
9              while $\mu$ is of the form $\mu[\theta_i(\vec{c})]$ do
10                 if $(\theta_i(\vec{c}) \approx d) \in A$ for some $d$ then
11                     $\mu \leftarrow \mu[d]{\downarrow}$
12                 else
13                     Let $d = call\_oracle(\mathcal{J}_i, \vec{c})$
14                     $A \leftarrow A \cup (\theta_i(\vec{c}) \approx d)$
15                     $\mu \leftarrow \mu[d]{\downarrow}$
16                 end
17             end
18             if $\mu$ is $true$ then
19                 return SAT, A, $M\mid_{\vec{f}}$
20             end
21         end
22     end
```

Theorem III.2 (Completeness for Decidable T and Finite Oracle Domains)**.** *Let background theory T be decidable, and let the domain of all oracle function symbols be finite. In this case, Algorithm 1 terminates.*

Proof sketch: Termination is guaranteed since the algorithm never repeats the same assignment to oracle inputs, and therefore, all input-output pairs for each oracle will be exhausted eventually. At this point, the oracle function symbols can be replaced by interpreted functions (lookup tables), and the formula reduces to one in the (decidable) background theory T.

Termination is not guaranteed in all background theories since it may be possible to write formulas where the number of input valuations to the oracle function symbols that must be enumerated is infinite. For example, this is possible in the theory of Linear Integer Arithmetic with an oracle function symbol with integer arguments.

Algorithm 2: Synthesis Modulo Oracles

```
input  : $(\vec{f}, \vec{\theta}, \forall\vec{x}\phi, \vec{\mathcal{J}} \uplus \vec{\mathcal{I}})$
output : solution $\vec{f}^*$ or no solution
$A \leftarrow true$ ;                              // conjunction of assumptions
$S \leftarrow true$ ;                              // synthesis formula
while true do
    $\vec{f}^* \leftarrow$Synthesize( $\exists\vec{f}.S$) ;
    if $\vec{f}^* = \emptyset$ then
        return no solution;
    else
        $V \leftarrow A \wedge \neg\phi$ ;                  // verification formula
        $(r, \alpha, M) \leftarrow$ SMTO$(\vec{x}, \vec{\theta}, V{\cdot}\{\vec{f} \rightarrow \vec{f}^*\}, \vec{\mathcal{J}})$ ;
        if $r$=UNSAT then
            return $\vec{f}^*$
        else
            $\beta \leftarrow$ call_additional_oracles$(\vec{\mathcal{I}}, \phi, M)$ ;
            $A \leftarrow A \cup \alpha$ ;
            $S \leftarrow S \cup \phi{\cdot}\{\vec{x} \rightarrow \vec{x}^M\} \cup \beta$;
        end
    end
end
```

IV. Synthesis Modulo Oracles

A SyMO problem consists of: a tuple of functions to synthesize $\vec{f}$; a tuple of oracle function symbols $\vec{\theta}$; a specification in the form $\forall\vec{x}.\ \phi$, where ϕ is a quantifier-free formula in some background theory T, and a set of oracle interfaces $\vec{\mathcal{I}} \uplus \vec{\mathcal{J}}$. We present an algorithm for a fragment of SyMO, where the verification condition reduces to a Definitional SMTO problem. To that end, we require that $\vec{\mathcal{J}}$ is a set of oracle interfaces that define $\vec{\theta}$, and $\vec{\mathcal{I}}$ is a set of oracle interfaces that only generate constraints, i.e., α_{gen} is empty. We will show that these restrictions permit us to use the algorithm for Definitional SMTO to check the correctness of a tuple of candidate functions in Theorem IV.1.

A. Algorithm for Synthesis with Oracles

We now proceed to describe an algorithm for solving synthesis problems using oracles, illustrated in Figure 5. Within each iteration of the main loop, the algorithm is broken down into two phases: a *synthesis phase* and an *oracle phase*. The former takes as input a synthesis formula S which is incrementally updated over the course of the algorithm and returns a (tuple of) candidate solutions $\vec{f}^*$. The latter makes a call to an underlying SMTO solver for the verification formula V, which is a conjunction of the current set of assumptions A we have accumulated via calls to oracles, and the negated conjecture $\neg\phi$. In detail:

- **Synthesis Phase:** The algorithm first determines if there exists a set of candidate functions $\vec{f}^*$ that satisfy the current synthesis formula S. If so, the candidate functions are passed to the oracle phase.

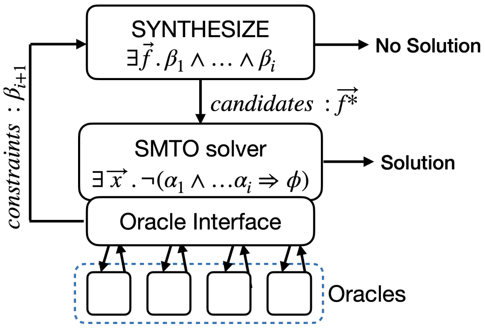

SYNTHESIZE
$\exists\vec{f}.\beta_1 \wedge \ldots \wedge \beta_i$
No Solution
candidates : $\vec{f*}$
SMTO solver
$\exists\vec{x}.\neg(\alpha_1 \wedge \ldots\alpha_i \Rightarrow \phi)$
Solution
Oracle Interface
Oracles
constraints : β_{i+1}

Fig. 5: SyMO Algorithm Illustration

- **Oracle Phase I:** The oracle phase calls the SMTO solver as described in section III on the following Definitional SMTO problem: $(\vec{x}, \vec{\theta}, V{\cdot}\{\vec{f} \rightarrow \vec{f}^*\}, \vec{\mathcal{J}}$. If the SMTO solver returns UNSAT, then $\vec{f}^*$ is a solution to the synthesis problem. Otherwise, the SMTO solver returns SAT, along with a set of new assumptions α and a model M. The assumptions α are appended to the set of overall assumptions A. Furthermore, an additional the constraint $\phi{\cdot}\{\vec{x} \rightarrow \vec{x}^M\}$ is added to the current synthesis formula S. This formula can be seen as a counterexample-guided refinement, i.e. future candidate solutions must satisfy the overall specification for the values of x in the model M returned by the SMTO solver.
- **Oracle Phase II:** As an additional step in the oracle phase, the solver may call any further oracles $\vec{\mathcal{I}}$ and the constraints β are passed to the synthesis formula. Note that the oracles in $\vec{\mathcal{I}}$ generate constraints only and not assumptions.

Theorem IV.1 (Soundness)**.** *If Algorithm 2 returns $\vec{f}^*$, then $\vec{f}^*$ is a valid solution for the SyMO problem $(\vec{f}, \vec{\theta}, \forall\vec{x}\phi, \vec{\mathcal{J}} \uplus \vec{\mathcal{I}})$.*

Proof. According to Definition II.6, a solution $\vec{f}^*$ is valid for our synthesis problem *iff* $(\vec{x}, \vec{\theta}, \neg\phi{\cdot}\{\vec{f} \rightarrow \vec{f}^*\}, \vec{\mathcal{J}} \uplus \vec{\mathcal{I}})$ is unsatisfiable modulo theories and oracles, i.e. when $\exists\vec{\theta}A \wedge (\neg\phi{\cdot}\{\vec{f} \rightarrow \vec{f}^*\} \wedge B)$ is T-unsatisfiable for assumptions A and constraints B generated by oracle interfaces $\vec{\mathcal{J}} \uplus \vec{\mathcal{I}}$. By definition, Algorithm 2 returns a solution if the underlying SMTO solver finds that $(\vec{x}, \vec{\theta}, \neg\phi{\cdot}\{\vec{f} \rightarrow \vec{f}^*\}, \vec{\mathcal{J}})$ is unsatisfiable modulo theories and oracles, i.e. $\exists\vec{\theta}A \wedge (\neg\phi{\cdot}\{\vec{f} \rightarrow \vec{f}^*\})$ is T-unsatisfiable, which trivially implies that the above statement holds. Thus, and since the SMTO solver is correct for UNSAT responses due to Theorem III.1, any solution returned by Alg. 2 is a valid solution. □

Inferring inputs for additional oracles: Although not described in detail in Algorithm 2, we remark that an implementation may infer additional calls to oracles based on occurrences of terms in constraints from $\vec{\mathcal{I}}$ and ground terms in ϕ under the current counterexample from M. For example, if $f(7)$ appears in $\phi{\cdot}\{\vec{x} \rightarrow \vec{x}^M\}$, and there exists an oracle interface with a single input z and the generator $\beta_{gen} : f(z) \approx y$, we will call that oracle with the value 7. In general, inferring such inputs amounts to matching terms from constraint generators

Query Type	Oracle Interface	Example synthesis algorithms
Constraint generating oracles		
Membership	$\mathcal{I}_{mem}(y_1, y_2, y, z_b, \emptyset, z_b \Leftrightarrow f(y_1, y_2) = y)$	Angluin's L^* [8]
Input-Output	$\mathcal{I}_{io}(y_1, y_2, y, z_b, \emptyset, z_b \Leftrightarrow f(y_1, y_2) \neq y)$	Classic PBE
Negative witness	$\mathcal{I}_{neg}(\emptyset, z_1, z_2, z, \emptyset, f(z_1, z_2) \neq z)$	ICE-learning [19]
Positive witness	$\mathcal{I}_{pos}(\emptyset, z_1, z_2, , z, \emptyset, f(z_1, z_2) = z)$	ICE-learning [19]
Implication	$\mathcal{I}_{imp}(f^*, z_1, z_2, z'_1, z'_2, \emptyset, f(z_1, z_2) \Rightarrow f(z'_1, z'_2)$	ICE-learning [19]
Counterexample	$\mathcal{I}_{cex}(f^*, \vec{z}, \emptyset, \phi\{\vec{x} \to \vec{z}\})$	Synthesis with validators [23]
Distinguishing-input	$\mathcal{I}_{di}(f^*, z_1, z_2, z, \emptyset, f(z_1, z_2) = z)$	Synthesis with distinguishing inputs [20]
Constraint and assumption generating oracles		
Correctness	$\mathcal{J}_{corr}(f^*, z_b, \theta(f^*) = z_b, \emptyset)$	ICE-learning [19]
Correctness with cex	$\mathcal{J}_{ccex}(f^*, z_b, \vec{z}, \theta(f^*) = z_b, \phi\{\vec{x} \to \vec{z}\})$	classic CEGIS [29]

TABLE I: Common oracle interfaces, illustrated for synthesizing a single function which takes a two inputs $f(x_1, x_2)$. y indicates query variables, except where they are the candidate function, in which case we use f^*, and z indicates response variables, where z_b is a Boolean.

to concrete terms from $\phi \cdot \{\vec{x} \to \vec{x}^M\}$. Our implementation in Section VI follows this principle.

V. Instances of Synthesis Modulo Oracles

A number of different queries are categorized in work by Jha and Seshia [21]. Briefly, these query types are

- membership queries: the oracle returns true *iff* a given input-output pair is permitted by the specification
- input-output queries: the oracle returns the correct output for a given input
- positive/negative witness queries: the oracle returns a correct/incorrect input-output pair
- implication queries: given a candidate function which the specification demands is inductive, the oracle returns a counterexample-to-induction [12], [19].
- Counterexample queries: given a candidate function, the oracle returns an input on which the function behaves incorrectly if it is able to find one
- Correctness queries: the oracle returns true *iff* the candidate is correct
- Correctness with counterexample: the oracle returns true *iff* the candidate is correct and a counterexample otherwise
- Distinguishing inputs: given a candidate function, the oracle checks if there exists another function that behaves the same on the set of inputs seen so far, but differently on a new input. If one exists, it returns the new input and its correct output.

We show the oracle interfaces for each of the classic query types in Table I. Full details are given in Appendix A. To give an intuition for how these interfaces are *used* in a synthesizer: oracle function symbols, $\vec{\theta}$ are used to encapsulate correctness criteria (in full or in part) of $\vec{f}$, when correctness depends on external oracles. For instance, we could write a specification that states that $\theta(\vec{f})$ should return true, and assumptions generated by the oracle are used to determine when that specification is satisfied. Constraints generated by oracles associated with $\vec{\mathcal{I}}$ are used to guide the synthesizer. Synthesis algorithms typically alternate between a synthesis phase, which solves an approximation of the specification, and a verification phase which verifies the solution to the approximation against the full specification. For example, CEGIS [29] uses the counterexamples to construct this approximation in the form of constraints specifying the behavior of the function must be correct on the counterexamples obtained so far.

Thus the synthesis modulo oracles framework we present is a flexible and general framework for program synthesis. SyMO can implement any inductive synthesis algorithm, i.e., any synthesis algorithm where the synthesis phase of the algorithm iteratively increases the constraints over the synthesis function. Common synthesis algorithms are composed of combinations of the queries described in Table I. For instance, CEGIS [29] is SyMO with a single counterexample-with-correctness interface $\mathcal{J}_{ccex}$; ICE-learning [19] uses $\mathcal{J}_{corr}, \mathcal{I}_{imp}, \mathcal{I}_{pos}, \mathcal{I}_{neg}$. For full details on these equivalences, see Appendix A.

VI. Delphi: a Satisfiability and Synthesis Modulo Oracles Solver

We implement the algorithms described above in a prototype solver Delphi[1]. The solver can use any SMT-lib compliant SMT solver as the sub-solver in the SMTO algorithm, or bitblast to MiniSAT version 2.2 [17], and it can use any SyGuS-IF compliant synthesis solver in the synthesis phase of the SyMO algorithm, or a symbolic synthesis encoding based on bitblasting. In the evaluation we report results using CVC4 [11] v1.9 in the synthesis phase and as the sub-solver for the SMTO algorithm. The input format accepted by the solver is an extension of SMT-lib [10] and SyGuS-IF [24].

A. Case Studies

We aim to answer the following research questions: RQ1 – when implementing a logical specification as an oracle executable, what is the overhead added compared to the oracle-free encoding? RQ2 – can SMTO solve satisfiability problems beyond state-of-the-art SMT solvers? RQ3 – can SyMO solve synthesis problems beyond state-of-the-art SyGuS solvers? To that end, we evaluate Delphi on the following case studies.

a) Reasoning about primes: We convert a set of 12 educational mathematics problems [22] (Math) that reason about prime numbers, square numbers, and triangle numbers into SMT and SMTO problems. We demonstrate that using an oracle to determine whether a number is a prime, a square or a triangle number is more efficient than the pure SMT encoding.

[1] link: https://github.com/polgreen/delphi

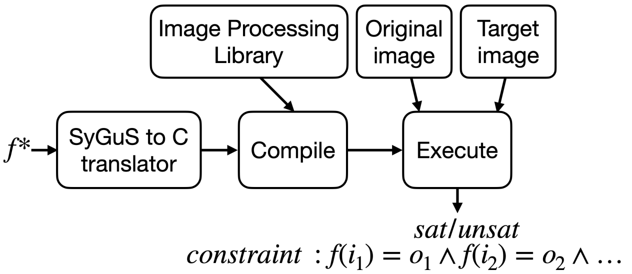


Fig. 6: Oracle for image transformations

b) Image Processing: Given two images, we encode a synthesis problem to synthesize a pixel-by-pixel transformation between the two. Figure 2 shows an example transformation. The SyMO problem uses an oracle, shown in Figure 6, which loads two JPEG images of up to 256×256 pixels: the original image, and the target image. Given a candidate transformation function, it translates the function into C code, executes the compiled code on the original image and compares the result with the target image, and returns "true" if the two are identical. If the transformation is not correct, it selects a range of the incorrect pixels and returns constraints to the synthesizer that give the correct input-output behavior on those pixels. The goal of the synthesis engine is to generalize from few examples to the full image. The oracle-free encoding consists of an equality constraint per pixel. This is a simplification of the problem which assumes the image is given as a raw matrix and omits the JPEG file format decoder.

c) Digital Controller Synthesis: We encode the task of synthesizing floating-point controllers which guarantee bounded safety and stability of Linear Time Invariant systems [3], using a range of time discretizations as both a SyMO and an oracle-free synthesis problem. We use two oracles in the SyMO problem: a correctness oracle to check the stability of the system using Eigen [1] (which avoids the need for the solver to reason about complex non-linear arithmetic). We note that these benchmarks do not necessarily have solutions for all time discretizations.

d) Programming by example: We encode PBE [2] benchmarks as SyMO problems using oracles that demonstrate the desired behavior of the function to be synthesized. These examples show that PBE benchmarks have a simple encoding in our framework.

B. Observations

We report a summary of the results for these case-studies in Table II and make the following observations:

a) RQ1: The overhead incurred by using oracles is small: performance on PBE problems encoded with oracles is similar to PBE problems encoded without oracles, with a small overhead incurred by calling external binaries. Given this low overhead, SyMO would be amenable to integration with many more sophisticated synthesis search approaches [18], [25], [6].

b) RQ2: Delphi solves more educational mathematics questions than CVC4, demonstrating that SMTO does enable SMT solvers to solve problems beyond the state-of-the-art by delegating challenging reasoning to an external oracle.

	Delphi		Delphi		CVC4	
(encoding)	(oracles)		(no oracles)		(no oracles)	
Benchmarks (#)	#	t	#	t	#	t
Images(10)	9	21.6s	0	–	0	–
Math(12)	9	<0.5s	1	<0.2s	5	2.2s
Control-stability(112)	104	29.3s	–	–	16	19.4s
Control-safety(112)	31	59.9s	0	–	0	–
PBE(150)	148	0.5s	150	1.6s	150	<0.2s

TABLE II: Comparison of Delphi and CVC4. # is the number of benchmarks solved within the $600s$ timeout, and t is the average run-time for solved benchmarks. The first column shows results on SyMO and SMTO problems, the second two columns show results on the equivalent oracle-free encodings.

c) RQ3: Delphi solves control synthesis problems and image transformation problems that cannot be easily expressed as SyGuS and elude CVC4, demonstrating that SyMO can solve synthesis problems beyond state-of-the-art solvers. When tackling the image transformation problems, SyMO dynamically generates small numbers of informative constraints, rather than handling the full image at once.

We also note that in many cases the encodings for SyMO and SMTO problems are more compact and (we believe) easier to write in comparison to pure SMT/SyGuS encodings. For instance, Figure 1 reduces to two assertions and a declaration of a single oracle function symbol.

Future work: We see a lot of scope for future work on SyMO. In particular, we plan to embed SMTO solving into software verification tools; allowing the user to replace functions that are tricky to model with oracle function symbols. The key algorithmic developments we plan to explore in future work include developing more sophisticated synthesis strategies that decide when to call oracles based on the learned utility and cost of the oracles. An interesting part of future work will be to explore interfaces to oracles that provide *syntactic* constraints, such as those used in [4], [18], which will require the use of context-sensitive grammars in the synthesis phase.

VII. Conclusion

We have presented a unifying framework for synthesis modulo oracles, identifying two key types of oracle query-response patterns: those that return constraints that can guide the synthesis phase and those that assert correctness. We proposed an algorithm for a meta-solver for solving synthesis modulo oracles, and, as a necessary part of this framework, we have formalized the problem of satisfiability modulo oracles. Our case studies demonstrate the flexibility of a reasoning engine that can incorporate oracles based on complex systems, which enables SMTO and SyMO to tackle problems beyond the abilities of state-of-the-art SMT and Synthesis solvers, and allows users to specify complex problems without building custom reasoning engines.

References

[1] Eigen, C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. https://eigen.tuxfamily.org. Accessed: 2021-05-19.

[2] Sygus competition. https://sygus.org/. Accessed: 2021-05-19.

[3] Alessandro Abate, Iury Bessa, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica*, 57(1-2):223–244, 2020.

[4] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, pages 270–288. Springer, 2018.

[5] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

[6] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *TACAS (1)*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[7] Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 218–223. ACM, 2004.

[8] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[9] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, 2009.

[10] Clark Barrett, Cesare Tinelli, et al. The SMT-LIB standard: Version 2.0.

[11] Clark W. Barrett, Haniel Barbosa, Martin Brain, Duligur Ibeling, Tim King, Paul Meng, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, and Cesare Tinelli. CVC4 at the SMT competition 2018. *CoRR*, abs/1806.08775, 2018.

[12] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[13] Bryan A. Brady, Randal E. Bryant, and Sanjit A. Seshia. Learning conditional abstractions. In *FMCAD*, pages 116–124. FMCAD Inc., 2011.

[14] Bryan A. Brady, Randal E. Bryant, Sanjit A. Seshia, and John W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 31–40, July 2010.

[15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[16] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. Program synthesis for program analysis. *ACM Trans. Program. Lang. Syst.*, 40(2):5:1–5:45, 2018.

[17] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435. ACM, 2018.

[19] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer, 2014.

[20] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering (ICSE)*, pages 215–224. ACM, 2010.

[21] Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.

[22] Michael Kent. *Gcse maths edexcel higher student book*. Harpercollins Publishers, 2015.

[23] Anders Miltner, Saswat Padhi, Todd D. Millstein, and David Walker. Data-driven inference of representation invariants. In *PLDI*, pages 1–15. ACM, 2020.

[24] Abhishek Udupa Mukund Raghothaman, Andrew Reynolds. The SyGuS language standard version 2.0. https://sygus.org/language/, 2019.

[25] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

[26] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model finding for recursive functions in SMT. In *IJCAR*, volume 9706 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2016.

[27] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Clark W. Barrett, and Cesare Tinelli. On counterexample guided quantifier instantiation for synthesis in CVC4. *CoRR*, abs/1502.04464, 2015.

[28] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[30] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296. ACM, 2013.

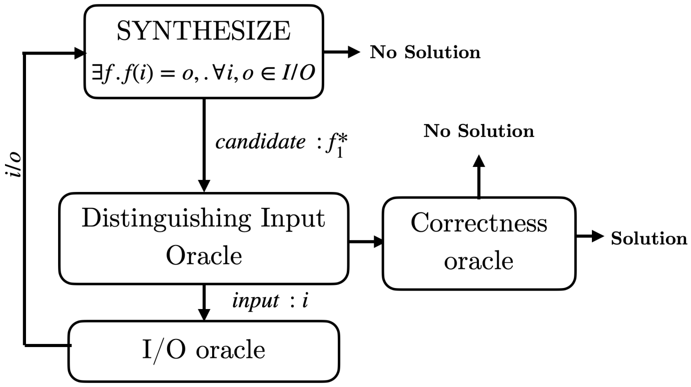


Fig. 7: Synthesis with distinguishing inputs

Appendix

A. Instances of Synthesis Modulo Oracles

The SyMO framework we present is a flexible and general framework for program synthesis. SyMO can implement any inductive synthesis algorithm, i.e., any synthesis algorithm where the synthesis phase of the algorithm iteratively increases the constraints over the synthesis function. Here we describe how, by providing specific oracles the algorithm describes will implement standard synthesis algorithms such as CEGIS [29], ICE-learning [19], Synthesis with distinguishing inputs [20] and CEGIS(T) [4].

1) CounterExample Guided Inductive Synthesis: : Suppose we are solving a synthesis formula with a single variable x and a single synthesis function f, where $f : \sigma \rightarrow \sigma'$. CEGIS consists of two phases, a synthesis phase that solves the formula $S = \exists f \forall x \in X_{cex}, \phi$, where X_{cex} is a subset of all possible values of x, and a verification phase which solves the formula $V = \exists x \neg \phi$. There are two ways of implementing CEGIS in our framework. The first is simply to pass the full SMT-formula ϕ to the algorithm as is, without providing external oracles. The second method is to replace the specification given to the oracle guided synthesis algorithm with $\exists f \forall \theta\, .\theta(f)$ and use an external correctness oracle with counterexamples, illustrated here for a task of synthesizing a function f, and receiving a candidate synthesis function $y : \sigma \rightarrow \sigma'$:

$$I_{corr} = \begin{cases} Q & : (y\ (\sigma \rightarrow \sigma')) \\ R & : (z_1\, \sigma), (z_2\ bool) \\ \alpha_{gen} & : \theta(y) = z_2 \\ \beta_{gen} & : \phi(z_1/x) \end{cases}$$

By inspecting the formula solved by the synthesis phase at each iteration, we can see that, after the first iteration, the synthesis formula are equisatisfiable if the sequence of counterexamples obtained is the same for both algorithms.

iter.	CEGIS	SyMO with correctness oracle
1	$X_{cex} = \emptyset$ $\exists f.\exists x.\phi$	$\exists f.true$
2	$X_{cex} = c_1$ $. \exists f.\forall x \in X_{cex}\, .\phi(x)$	$\beta_1 = \phi(k_1/x)$ $\exists f.\beta_1$
3	$X_{cex} = c_1, c_2$ $. \exists f.\forall x \in X_{cex}\, .\phi(x)$	$\beta_2 = \phi(k_2/x)$ $\exists f.\beta_1 \wedge \beta_2$
...	...	...

TABLE III: Comparison of the synthesis formula at each iteration, showing that, if the same sequence of counterexamples is obtained, the synthesis formula are equisatisfiable at each step, i.e., SyMO reduces to CEGIS.

2) ICE learning: ICE learning [19] is an algorithm for learning invariants based on using examples, counterexamples and implications. Recall the classic invariant synthesis problem is to find an invariant inv such that:

$$\begin{aligned} \forall x, x' \in X.init(x) &\implies inv(x) \wedge \\ inv(x) \wedge trans(x, x') &\implies inv(x') \wedge \\ inv(x') &\implies \phi \end{aligned}$$

where $init$ defines some initial conditions, $trans$ defines a transition relation and ϕ is some property that should hold. ICE is an oracle guided synthesis algorithm, where, given a candidate inv^*, if the candidate is incorrect (i.e., violates the constraints listed above) the oracle can provide: positive examples $E \subseteq X$, which are values for x where $inv(x)$ should be $true$; negative examples $C \subseteq X$, which are values for x where $inv(x)$ should be $false$; and implications $I \subseteq X \times X$, which are values for x and x' such that $inv(x) \Rightarrow inv(x')$. The learner then finds a candidate inv, using a symbolic encoding, such that

$$\begin{aligned} (\forall x \in E.inv(x)) \wedge \\ (\forall x \in C.\neg inv(x)) \wedge \\ (\forall (x, x') \in I.inv(x) \Rightarrow inv(x')). \end{aligned}$$

The SyMO algorithm described in this work will implement ICE learning when given a correctly defined set of oracles and oracle interface and a constraint $\theta_{corr}(inv) = true$. Interfaces for these oracles, in a system with variables $x_1 \ldots x_n$, are shown in Figure 8. Note that implication queries generate constraints enforcing that if the pre-state of the implication pair lies in the invariant, so must the post-state, which also allows the learner to exclude the pre-state in its next round of synthesis.

3) Synthesis with distinguishing inputs: This algorithm [20], illustrated in Figure 7, uses several oracles which interact with each other. The synthesis phase searches for a function that satisfies a list of input-output examples. If one is found, it is passed to a *distinguishing-input oracle*, which looks for another, different, function that behaves the same as the existing function on the list of input-output examples, but behaves differently on another distinguishing input.

$$I_{corr} = \begin{cases} Q & : (f^* \, (\sigma_1 \times \ldots \times \sigma_n \to \mathit{bool})) \\ R & : (z_b \; \mathit{bool}) \\ \alpha_{gen} & : \theta_{corr}(f^*) = z_b \end{cases} \qquad I_{neg} \quad = \begin{cases} Q & : (f^* \, (\sigma_1 \times \ldots \times \sigma_n \to \mathit{bool})) \\ R & : (z_1 \, \sigma_1), \ldots, (z_n \, \sigma_n) \\ \beta_{gen} & : \neg f(z_1, \ldots z_n) \end{cases}$$

$$I_{impl} = \begin{cases} Q & : (f^* \, (\sigma_1 \times \ldots \times \sigma_n \to \mathit{bool})) \\ R & : (z_1 \, \sigma_1), \ldots, (z_n \, \sigma_n), \\ & \quad (z'_1 \, \sigma_1), \ldots, (z'_n \, \sigma_n) \\ \beta_{gen} & : f(z_1, \ldots z_n) \implies \\ & \quad f(z'_1, \ldots z'_n) \end{cases} \qquad I_{pos} = \quad \begin{cases} Q & : (f^* \, (\sigma_1 \times \ldots \times \sigma_n \to \mathit{bool})) \\ R & : (z_1 \, \sigma_1), \ldots, (z_n \, \sigma_n) \\ \beta_{gen} & : f(z_1, \ldots z_n) \end{cases}$$

Fig. 8: Oracle interfaces for ICE learning, synthesizing an invariant f, and receiving a candidate invariant f^*.

If such a function exists, the distinguishing input is passed to an input/output oracle, and the input/output pair is passed to the synthesizer. If a distinguishing input does not exist, the correctness of the function is checked and the algorithm terminates (if this function is not correct, then there is no solution to the synthesis problem).

We can implement this algorithm in SyMO by observing that the synthesizer needs only to query the correctness oracle and the distinguishing input oracle, which maintains a list of inputs it has returned so far and receives a response from only one oracle, the input/output oracle. The interface for the distinguishing input-oracle is as follows for a specification synthesizing f, and a candidate synthesis function $y : \sigma_1 \times \ldots \times \sigma_n \to \sigma'$:

$$I_{synthDI} = \begin{cases} Q & : (y \, (\sigma_1 \times \ldots \times \sigma_n \to \sigma')) \\ R & : (z_1 \, \sigma_1), \ldots, (z_n \, \sigma_n), (z_{n+1} \, \sigma') \\ \beta_{gen} & : f(z_1, \ldots z_n) = z_{n+1} \end{cases}$$

4) CEGIS(T): This algorithm [4] extends a CEGIS loop to incorporate a theory solver that, given a candidate solution, generalizes the candidate to a template solution (removing any constant literals in the solution and replacing them with symbolic values), and searches for solutions within that template. If a solution is found, the oracle returns that valid solution. If no such solution exists, a constraint is passed back that blocks the candidate solution. Our framework can be used to implement an approximation of CEGIS(T): one in which the oracle generalizes the candidate and searches for a solution within that template, but if no solution is found, a single point counterexample is returned instead of the constraint over the syntactic space. This highlights a limitation of the framework we propose: the oracles may only place *semantic* constraints over the search space. That is, it may place constraints that reason about the *behavior* of the synthesis functions, and not the syntactic space from which the synthesis functions are to be constructed. The oracle interface for our modified CEGIS(T) is as follows:

$$I_{cegist} = \begin{cases} Q & : (y \; (\sigma_1 \times \ldots \times \sigma_n \to \sigma')) \\ R & : (z_1 \, \sigma_1), \ldots, (z_n \, \sigma_n), (z_{n+1} \; b), \\ & \quad (z_{n+2} \; (\sigma_1 \times \ldots \times \sigma_n \to \sigma')) \\ \beta_{gen} & : z_{n+1} \, ? \, (f = z_2) : \phi(z_1/x_1, \ldots z_n/x_n) \end{cases}$$

B. Evaluation Details

In this section, we give more details on the control, mathematics, and image processing case studies. We also provide a brief practical comparison of invariant synthesis with CEGIS and with ICE learning.

1) Further details on case studies :

a) Control benchmarks: The control benchmarks [3] synthesize single- and double-point precision floating-point controllers that guarantee stability and bounded safety for Linear Time Invariant systems. We use a state-space representation, which is discretized in time with 6 different constant sampling intervals T_s, generating 6 benchmarks per system:

$$\dot{x}_{t+1} = A\vec{x}_t + B\vec{u}_t,$$

where $\vec{x} \in \mathbb{R}^n$, $\vec{u} \in \mathbb{R}^p$ is the input to the system, calculated as $K\vec{x}$ where K is the controller to be synthesized, $A \in \mathbb{R}^{n \times n}$ is the system matrix, $B \in \mathbb{R}^{n \times p}$ is the input matrix, and subscript t indicates the discrete time step.

For stability benchmarks, we aim to find a stabilizing controller, such that absolute values of the (potentially complex) eigenvalues of the closed-loop matrix $A - BK$ are less than one. For bounded safety benchmarks, we aim to find a controller that is both stable as before and guarantees the states remain within a safe region of the state space up to a given number of time steps.

The SyMO encoding uses an oracle to determine the stability of the closed-loop matrix. The encoding without oracles requires the SMT solver to find roots of the characteristic polynomial.

b) Educational mathematics bencharks: These benchmarks are taken from Edexcel mathematics questions [22]. The questions require the SMT solver to find numbers that are (some combination of) factors, prime-factors, square and triangle numbers. The encodings without oracles used recursive functions to determine whether a number is a prime or a triangle number. We note the oracle used alongside the benchmark number in Table IV. We enable the techniques described by Reynolds et. al. [26] when running CVC4 on problems using recursive functions.

Benchmarks	config.	# solved	t
Inv-det(5)	CEGIS	5	163s
Inv-det(5)	ICE	5	121s
Inv-all(127)	CEGIS	102	0.8s
Inv-all(127)	ICE	86	2.1s

TABLE VI: Comparison of ICE learning with CEGIS in Delphi. Inv-det have deterministic transition relations and ICE learning uses a positive witness oracle that can unroll the transition relation. In the Inv-all category, all transition relations are treated as non-deterministic and the positive witness oracle is SMT-based only. t is the average solving time per benchmark (excluding timeouts, where the timeout is 600s).

	Delphi	Delphi	CVC4
benchmark	(oracles)	(no oracles)	(no oracles)
1b-square	<0.2s	<0.2	–
1d-prime	<0.2s	–	<0.2s
1f-prime	3.1s	–	<0.2s
1h-triangle	<0.2s	–	<0.2s
1j-square,prime	<0.2s	–	–
1l-triangle	<0.2s	–	<0.2s
1m-triangle	<0.2s	–	<0.2s
ex7-prime	2.3s	–	–
ex8-prime	–	–	–
ex9-prime	3.2s	–	–
ex10-prime	–	–	–
ex11-prime	<0.2s	–	–

TABLE IV: Solving times for Delphi and CVC4 on math examples using oracle and recursive function encodings. “ – ” indicates the timeout of 600s was exceeded.

c) Image transformations: We run 9 image transformations on images up to 256x256 pixels. The oracle-free encodings consist of an equality constraint for each pixel, and so are more than $50,000$ constraints long. For the oracle-free encoding we simplify the problem by assuming that the images are given as a raw matrix, and this encoding, unlike the oracle-based encoding, does not include the JPEG file format decoder. The results are shown in Table V.

	Delphi	Delphi	CVC4
Benchmarks	(oracles)	(no oracles)	(no oracles)
Image invert	0.4s	–	–
Image zombie	0.4s	–	–
Image attenuate	0.9s	–	–
Image crop1	14s	–	–
Image crop2	11.8s	–	–
Image crop3	12.0s	–	–
Image brighter	35.0s	–	–
Image darker	120.0s	–	–
Image round	0.2s	–	–
Image blur	–	–	–

TABLE V: Solving time for image transformation examples, comparing the oracle vs oracle-free encoding. “ – ” indicates the timeout of 600s was exceeded. The oracle-free encodings are >50,000 lines long.

2) A practical comparison of CEGIS vs ICE: As an illustration of SyMO performing ICE-learning, we implement positive, negative, and implication oracles for a small set of invariant benchmarks taken from the SyGuS competition [2]. We note that, for positive witness oracles to be notably informative beyond a pure specification, the positive witness oracle should unroll the transition relation. This is easier with deterministic transition relations. Table VI shows a comparison of CEGIS against ICE-learning with positive witness oracles that unroll the transition relation (in Inv-det). We observe that using such oracles, ICE-learning can solve these examples faster and using fewer generated synthesis constraints. Note that we disable all heuristics for solving invariants for these examples. We also note that the failure mode of the respective algorithms differs: CEGIS typically generates increasingly many constraints, enumerating through constant values; whilst ICE generates sets of constraints that become tricky to solve and the synthesis phase absorbs all the solving time. Unsurprisingly, in the case where invariant heuristics are enabled and the positive witness oracles are not able to unroll the transition relation (Inv-all), the specifically tailored oracles do not provide a performance gain as they aren’t able to provide information that isn’t already contained within the specification used by CEGIS.