Satisfiability and Synthesis Modulo Oracles

Elizabeth Polgreen
Andrew Reynolds
Sanjit A. Seshia

Satisfiability and Synthesis Modulo Oracles

Elizabeth Polgreen[1,2] and Andrew J. Reynolds[3] and Sanjit A. Seshia[1]

[1] University of California, Berkeley
[2] University of Edinburgh
[3] University of Iowa

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 oracles*, and present an algorithm for solving this problem. We show that our algorithm for solving synthesis modulo oracles is expressive enough to execute several algorithms for synthesis and verification including CEGIS and ICE learning. We implement the first prototype solver for satisfiability and synthesis modulo oracles and, with an image transformation case study, demonstrate applicability to problems using complex oracles that incorporate compilation and execution of code.

1 Introduction

A common formulation of program synthesis is to find a program, from a specified class of programs, that meets some correctness specification [2]. Classically, this is encoded as the 2nd-order logic formula $\exists f_1 \dots f_m \forall x_1 \dots x_n\ \phi$, where $f_1 \dots f_m$ is a set of target functions to be synthesized, $x_1 \dots x_n$ 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 $f_1^* \dots f_m^*$ satisfies the semantic restrictions if the formula $\forall x_1 \dots x_n\ \phi$ 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 [20] 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 an satisfiability modulo theories (SMT) [5] formula in a fully white-box manner. Put another way, SyGuS problems must be specified with static constraints a-priori to the solving process. The 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. [25]), such a model-checking oracle cannot be directly invoked within a general-purpose SyGuS solver and instead requires creating a custom solver. Typically the solution approach for SyGuS problems is thus some variable of a particular type of *counterexample-guided inductive synthesis* (CEGIS), which is illustrated in Figure 1, wherein a synthesis phase sends queries to a verification oracle in the form of a candidate program, and the verification phase response with a single point counterexample if the program is incorrect. This motivates our introduction of (potentially black-box) oracles to the synthesis problem, and necessarily therefore also to the SMT problem therein. Oracles can be black-box external implementations which 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), external verification engines solving verification queries beyond SMT solving, or even large databases which could be queried using simple interfaces.

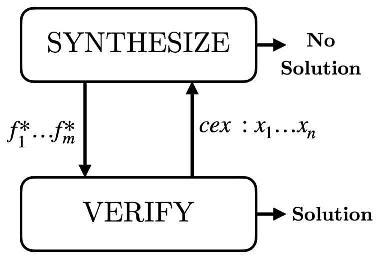
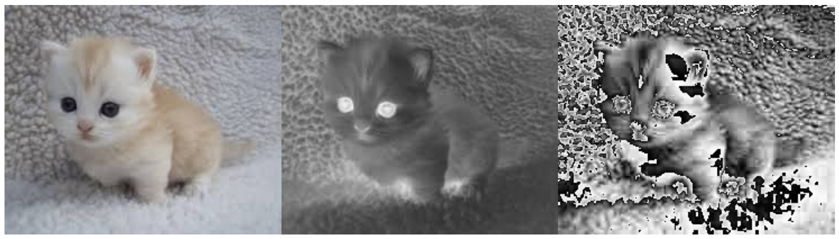


Fig. 1: CEGIS with an SMT-based verification oracle returning counterexamples

Prior work has set out a theoretical framework expressing synthesis algorithms as oracle guided inductive synthesis [18], 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 provide 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. SyMO is an iterative algorithm that alternates between a synthesis phase, and an oracle phase which calls any available oracles. The oracles generate assumptions and constraints, which are used to determine correctness of candidate solutions and to guide the synthesis phase respectively. One can think of constraints as "generalized counterexamples" — at any point in the algorithm, the synthesis phase is searching for a function that satisfies the oracle constraints obtained so far.

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 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



(a) Original image (b) Target image (c) Target image 2

Fig. 2: Image manipulation: transformations synthesized by Delphi in < 15 sec.

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. Determining the correctness of a candidate function in a synthesis problem is an SMTO problem and assumptions generated by oracles are used to determine the correctness of candidate solutions.

As an exemplar of an existing oracle-guided synthesis algorithm that goes beyond the SMT-solver based counterexample oracles, consider ICE-learning [16] for invariant synthesis. ICE-learning is based on the synthesizer having access to three oracles: an oracle which provides positive examples (examples which *should* be contained within the invariant); an oracle which provides negative examples (examples which *should not* be contained within the invariant); and an oracle which provides 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, invariant synthesis, synthesizing approximations for trigonometric functions, and synthesizing image transformations. The latter use case, shown in Fig. 2, is an illustration of the power of being able to incorporate oracles into SyMO that are too complex to be modeled (in this instance, the oracle incorporates a compiler and an image processing library).

To summarize, the main contributions of this paper are:

- A formalization of the problem of satisfiability and synthesis modulo oracles (Sec. 2);
- A unifying algorithmic approach for solving these problems (Sec. 3 and Sec. 4);
- Demonstration of how this approach can capture popular synthesis strategies from the literature (Sec. 5), and

- A prototype solver Delphi, and an experimental demonstration of the broad applicability of this framework (Sec. 6).

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 algorithm used for Syntax-Guided Synthesis [24], and uses a correctness oracle that returns counterexamples. Further developments in synthesis then typically fall into one of two categories: either they develop innovative search algorithms to search the space more efficiently (for instance genetic algorithms [12], or reinforcement learning [23], or partitioning the search space in creative ways [3]; or they focus on extending the communication paradigm permitted between the synthesis and the verification phase. For instance CEGIS modulo theories [1], 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; CVC4 stands out in making full use of the white-box nature of the SMT solver oracle with its single-invocation algorithm [22]. Other work leverages the ability to classify counterexamples as positive or negative examples [19]. There are also notable algorithms in invariant synthesis based on innovative use of different query types; IC3 [8], LoopInvGen [21], and ICE-learning [16] being some specific instances. 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 [17] is an exception to this literature and uses a specific set of three interacting black-box oracles, to solve a 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., [4, 10, 9]), which use counterexample-guided abstraction refinement [11]. Here, components of a system are abstracted and then refined based on whether the abstraction is sufficiently detailed to prove a property. However, in order 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.

2 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.

2.1 Preliminaries and Notation

Satisfiability Modulo Theories (SMT): The input to an SMT problem is a closed first-order logical formula φ. The task is to determine whether φ is satisfiable or unsatisfiable with respect to some background theory T, which restricts the interpretation of symbols in φ. If φ is satisfiable, a solver will usually return a model of T that makes φ true, which will include assignments to all free variables in φ. In this work we consider formula in prenex normal form: $\varphi \approx q_1 x_1 q_2 x_2 \dots q_n x_n . \rho$, where $q_1 \dots q_n \in \{\forall, \exists\}$, all quantifiers range over 0-ary symbols (symbolic constants), and ρ is a quantifier-free formula. Any first-order formula can be rewritten into this prenex normal form so this restriction is not limiting.

Syntax-Guided Synthesis: In syntax-guided synthesis, we are given a set of functions $f_1 \dots f_m$ to be synthesized, associated languages of expressions $L_1, \dots, L_m$ (typically generated by grammars), and we seek to solve a formula of the form

$$\exists f_1 \in L_1 \dots f_m \in L_m \; \forall x_1, \dots x_n \; \phi,$$

where $x_1 \dots x_n$ 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 can be dropped from the formula. A tuple of candidate functions $(f_1^*, \dots, f_m^*)$ satisfies the semantic restrictions for functions-to-synthesize $(f_1, \dots, f_m)$ in conjecture $\exists f_1, \dots, f_m . \forall x_1, \dots x_n \, \phi$ in background theory T if $\forall x_1, \dots x_n \, \phi$ is valid in T when $f_1, \dots, f_n$ are defined to be terms whose semantics are given by the functions $(f_1^*, \dots, f_n^*)$ [2, 20].

2.2 Basic Definitions

In this work, we use the term *oracle* to refer to a (black-box) component which can be queried in a pre-defined way by the solver. Here we define the necessary oracle concepts we will need before we can outline satisfiability and synthesis modulo oracles. These concepts are borrowed from [18] but we give our own formal definitions.

Definition 1 (Oracle). *Let Q be a set of queries and R be a set of responses. An* oracle *is a binary relation $\mathcal{O} \subseteq Q \times R$ which associates each query $q \in Q$ to a response $r \in R$. $\mathcal{O}$ may be functional, i.e., $\forall q \in Q, \forall r \in R, \forall p \in Q, ((q, r) \in \mathcal{O} \wedge (p, r) \in \mathcal{O}) \Rightarrow q = p$, or non-functional. It is serial: $\forall q \in Q, \exists r \in R, (q, r) \in \mathcal{O}$.*

A query-response pair (q, r) is *consistent* with oracle $\mathcal{O}$ if $(q, r) \in \mathcal{O}$.

The solver interacts with oracles via queries and responses defined by an *oracle interface*. Specifically, the oracle interface defines the query type and response type, and any additional expressions are either oracle assumption generators or oracle constraint generators. Substitution of the oracle response types into the former generates an expression the solver assumes is true when checking correctness of a solution; the latter generates an expression that holds for any valid solution. We introduce the former to allow black-box components of correctness specifications and the latter to provide guidance to a synthesis solver. Formally:

Definition 2 (Oracle Interface). *An oracle interface $\mathcal{I}$ is a tuple that defines the* domain *of the oracle (the set of queries Q that the oracle accepts), and the* co-domain *of the oracle (the set of responses R that the oracle can return). The domain Q is given as a list of sorted variables $y_1, \ldots, y_j$ where the sorts of the variable correspond to the sorts in the query type. The response R is also given as such a list of sorted variables $z_1), \ldots z_k$. The oracle interface may provide one or more assumption generators α_{gen} and constraint generators β_{gen}.*

Definition 3 (Query type). *A query type is defined as a tuple of sorts $\sigma_1, ..., \sigma_j$ permitted in the background theory. A solver may query the oracle with a tuple of constant literals of sorts $\sigma_1, ..., \sigma_j$.*

Definition 4 (Response type). *A response type is defined as a tuple of sorts $\sigma'_1, ..., \sigma'_k$ permitted in the background theory. The oracle will return a tuple of constant literals of sorts $\sigma'_1, ..., \sigma'_k$.*

We first define oracle assumption generators and oracle function symbols. If e is an expression and x is free in e, let $e(t/x)$ be the formula obtained from the formula e by proper substitution of the variable x by the variable t.

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

Fig. 3: Oracle interface

Definition 5 (Oracle assumption generators). *An oracle assumption generator α_{gen} is a formula using symbols from the background theory, and variables $y_1, \ldots y_j$ from the domain and $z_1, \ldots z_k$ from the co-domain of the oracle. When the oracle is queried with a tuple of constant literals $c_1, ..., c_k$ and responds with a tuple of constant literals $d_1, \ldots, d_j$, a new term α is generated with constant literals $c_1, \ldots c_j$ in place of identifiers $y_1 \ldots y_j$ and constant literals $d_1, \ldots d_k$ in place of $z_1 \ldots z_k$. That is, $\alpha \approx \alpha_{gen}(c_1/y_1, \ldots, c_j/y_j, d_1/z_1, \ldots d_k/z_k)$.*

Given a synthesis conjecture $\exists f_1 \ldots f_m \forall x_1 \ldots x_n,\ \phi$, and a generated assumption α, the new synthesis conjecture becomes $\exists f_1 \ldots f_m \forall x_1 \ldots x_n,\ \alpha \Rightarrow \phi$. Given a first-order satisfiability formula in prenex normal form $\exists x_1 \ldots \exists x_n \rho$, the new satisfiability problem becomes $\exists x_1 \ldots \exists x_n \alpha \Rightarrow \rho$. We comment on the satisfiability of these formulas in the following sections, but a solution to the original first-order formula is not necessarily a valid solution to the formula with the assumption applied.

Next, we define a notion of an oracle constraint. We define oracle constraints only for the purposes of synthesis problems, where these constraints are used to guide the synthesis algorithm.

Definition 6 (Oracle constraint generators). *An oracle constraint generator β_{gen} is a formula that uses symbols from the background theory, and sorted variables $y_1, \dots y_j$ from the domain and $z_1, \dots z_k$ from the co-domain of the oracle. When the oracle is queried with a tuple of constant literals $c_1, ..., c_j$ and responds with a tuple of constant literals $d_1, \dots, d_k$, a new term β is generated with constant literals $c_1, \dots c_j$ in place of identifiers $y_1 \dots y_j$ and constant literals $d_1, \dots d_k$ in place of $z_1 \dots z_k$. That is $\beta \approx \beta_{gen}(c_1/y_1, \dots, c_j/y_j, d_1/z_1, \dots d_k/z_k)$.*

Given a synthesis conjecture $\exists f_1 \dots f_m \forall x_1 \dots x_n \, \phi$, and a generated constraint β, the new synthesis conjecture becomes $\exists f_1 \dots f_m \, \forall x_1 \dots x_n \, .\phi \wedge \beta$.

Finally, we define *oracle function symbols*, syntactic sugar for a specific use case of oracle assumption generators. An oracle function symbol is a functional symbol that may be used in the synthesis or satisfiability conjecture, and is associated to a functional (black-box) external interpretation in the form of an oracle. These symbols allow us to use oracles in constraints and assertions. They introduce a very specific restricted type of higher-order quantification to the problem which we will describe in more detail in Section 3: oracle function symbols are treated as universally quantified uninterpreted functions, and the associated oracle generates assumptions constraining the behavior of the uninterpreted function.

Definition 7 (Oracle Function Symbol). *An oracle function symbol θ is a function symbol of sort $\sigma_1 \times \dots \times \sigma_j \to \sigma'$ which is associated to a* functional *oracle $\mathcal{O}$ with an oracle interface which defines the domain as $(y_1, \sigma_1), \dots, (y_j, \sigma_j)$, and the co-domain as (z, σ'), and an oracle assumption generator $\alpha_{gen} \approx \theta(y_1, \dots y_j) = z$.*

We are now ready to define the two main problems introduced by this paper.

Definition 8 (Satisfiability Modulo Oracles). *A satisfiability modulo oracles (SMTO) problem consists of a formula ζ, which may be quantified, in a background theory T, and a set of oracle interfaces $\mathcal{I}_1 \dots \mathcal{I}_p$. The formula is satisfiable if there exists a satisfying assignment to $A \implies (\zeta \wedge B)$ in T, where A is a conjunction of all assumptions and B is a conjunction of all constraints generated by the oracle interfaces.*

Definition 9 (Restricted SMTO). *We defined a restricted SMTO problem where ζ is of the form $\exists x_1 \dots x_n, \forall \theta_1 \dots \theta_p \, \rho$, where $\theta_1 \dots \theta_p$ are oracle function symbols and $\mathcal{I}_1 \dots \mathcal{I}_p$ is the set of oracle interfaces corresponding to these symbols. The formula is satisfiable if there exists a satisfying assignment to ρ in T that includes only query-response pairs consistent with the oracles for $\theta_1 \dots \theta_p$.*

The alternation of quantifiers in the above definition is important. We are looking for an assignment to the x_i variables that will satisfy ρ no matter what (black-box) oracles implement the interfaces $\mathcal{I}_i$ corresponding to the θ_i symbols.

Definition 10 (Synthesis Modulo Oracles). *A synthesis modulo oracles (SyMO) conjecture consists of a formula $\exists f_1 \dots f_m, \forall \theta_1 \dots \theta_p, x_1 \dots x_n \; \phi$, and a set of oracle interfaces $\mathcal{I}_1 \dots \mathcal{I}_k$, where $k \geq p$, and interfaces $\mathcal{I}_1 \dots \mathcal{I}_p$ correspond to the oracle function symbols.*

A tuple of functions $\vec{f^*}$ satisfies the synthesis conjecture if $\phi[\vec{f^*}/\vec{f}]$ is true for all $x_1 \dots x_n$ and for all (black-box) oracles implementing the oracle interfaces for $\theta_1 \dots \theta_p$.

3 Satisfiability Modulo Oracles

Recall we defined a general SMTO problem and a restricted SMTO problem in section 2. In this section, we describe our approach to solving the restricted SMTO problem. First, we discuss some of the subtleties that arise when reasoning about the satisfiability and unsatisfiability of formulas the general SMTO problem.

Oracle assumption generators: Suppose we have a formula, $Q_1 x_1 \dots Q_n x_n \rho$, where $Q_1, \dots, Q_n$ are quantifiers, $x_1, \dots, x_n$ are 0-ary symbols, and there is additionally an external oracle that generates assumptions. Calling the oracle i times generates a sequence of formulas, where α_i is the assumption generated by the i^{th} call to the oracle:

$$\begin{aligned}
\gamma_0 \approx & Q_1 x_1 \dots Q_n x_n \quad \rho \\
\gamma_1 \approx & Q_1 x_1 \dots Q_n x_n \, (\alpha_1 \Rightarrow \rho) \\
& \dots \\
\gamma_i \approx & Q_1 x_1 \dots Q_n x_n \, (\alpha_i \Rightarrow \dots (\alpha_2 \Rightarrow (\alpha_1 \Rightarrow \rho)))
\end{aligned}$$

The problem is T-satisfiable after i calls to the oracle *iff* $\neg\gamma_i$ is unsatisfiable. The problem is unsatisfiable *iff* γ_i is unsatisfiable. If the assumptions generated do not contradict each other, a formula that is *satisfiable* after i calls to the solver will remain satisfiable, no matter how many further calls to the oracle are made.

Oracle constraint generators: Extending this correctness definition to constraints, suppose we have the same formula $Q_1 x_1 \dots Q_n x_n \rho$ but we now have an external oracle that generates *constraints*. Calling the oracle i times generates the following sequence of formula, where β_i is the constraint generated by the i^{th} call to the oracle:

$$\begin{aligned}
\gamma_0 \approx & Q_1 x_1 \dots Q_n x_n \quad \rho \\
\gamma_1 \approx & Q_1 x_1 \dots Q_n x_n \, \rho \wedge \beta_1 \\
& \dots \\
\gamma_i \approx & Q_1 x_1 \dots Q_n x_n \, \rho \wedge \beta_1 \wedge \dots \wedge \beta_i
\end{aligned}$$

Again, the problem is satisfiable after i calls to the oracle, iff any formula $\neg\gamma_i$ is unsatisfiable, and unsatisfiable after i calls to the oracle iff the formula γ_i is unsatisfiable. If a formula is *unsatisfiable* after i calls, it will remain unsatisfiable, no matter how many more calls to the oracles are made.

Correctness of general SMTO: The general SMTO problem permits both assumption and constraint generators, giving the formula $\gamma_i \approx Q_1 x_1 \ldots Q_n x_n(\alpha_1 \wedge \ldots \wedge \alpha_i) \implies (\rho \wedge \beta_1 \wedge \ldots \wedge \beta_i$. The problem is satisfiable after i calls to the oracle if $\neg\gamma_i$ is unsatisfiable, and it is unsatisfiable if γ_i is unsatisfiable. Given that satisfiablility changes depending on the sequence of oracle calls, a solver must return both a "satisfiable/unsatisfiable" result and the sequence of calls used to reach that result.

Restricted SMTO: In restricted SMTO, we permit only oracle function symbols and their associated assumption generating oracles, and no other oracles. This introduces a specific form of higher order quantification: $\exists x_1 \ldots \exists x_n \forall\theta\ \rho$, where θ is an oracle function symbol and $x_1 \ldots x_n$ are 0-ary symbols. For simplicity we consider a formula with only one oracle function symbol, but the following applies for a formula with any finite positive number of oracle function symbols. Intuitively, we wish to find a model for the free variables $x_1, \ldots x_n$ and for θ such that ρ is true and the model for θ is consistent with the external implementation the oracle function symbol is associated to. Recall that, for an oracle function symbol θ with sort $\sigma_1 \times \ldots \times \sigma_j \to \sigma'$, calling the corresponding oracle with a set of constant literals $c_1, \ldots, c_j$ and receiving a response d will generate an assumption $\alpha \approx \theta(c_1, \ldots c_j) = d$. Now suppose we generate the following sequence of formula:

$$
\begin{aligned}
\gamma_0 \approx & \exists x_1 \ldots \exists x_n \forall\theta \quad \rho \\
\gamma_1 \approx & \exists x_1 \ldots \exists x_n \forall\theta\, (\alpha_1 \Rightarrow \rho) \\
& \ldots \\
\gamma_n \approx & \exists x_1 \ldots \exists x_n \forall\theta\, (\alpha_n \Rightarrow \ldots (\alpha_2 \Rightarrow (\alpha_1 \Rightarrow \rho)))
\end{aligned}
$$

As before, the problem is satisfiable after i calls to the oracle, and for any number of subsequent calls to the oracles, if $\neg\gamma_i$ is unsatisfiable. The problem is unsatisfiable if γ_i is unsatisfiable. However, we know that the oracles will not generate assumptions that contradict each other, since they only generate constraints based on equality, and the oracles are functional. Thus $\alpha_1 \wedge \ldots \wedge \alpha_i$ is always satisfiable, and the problem is unsatisfiable *iff* $\alpha_1 \wedge \ldots \wedge \alpha_i \wedge \rho$ is unsatisfiable. Once a satisfiable/unsatisfiable result is obtained for a restricted SMTO problem, the result cannot be changed by making more calls to the oracles.

3.1 Algorithm for restricted Satisfiability Modulo Oracles

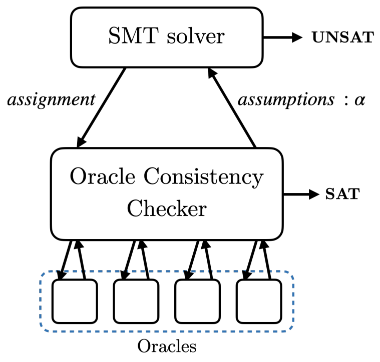


Fig. 4: Satisfiability Modulo Oracle Solver

Our algorithm for restricted SMTO builds on existing work in SMT solving, first looking for satisfying assignments to the SMT formula while over-approximating oracle function symbols as uninterpreted functions, and then checking whether this assignment is consistent with the oracles. That is, an SMT solver finds a satisfying assignment to the following formula, assuming the oracle function symbols are uninterpreted functions: $\exists x_1 \ldots x_n, \exists \theta_1 \ldots \theta_p \; \rho$.

The corresponding oracles are called to check whether the satisfying assignment to the oracle function symbols $\vec{\theta}$ is consistent with them. If the satisfying assignment was not consistent with the oracles, an oracle assumption is added to the formula and it is passed back to the SMT solver, which attempts to find a new satisfying assignment. A sequence of i calls to oracles generates a sequence of assumptions $\alpha_1 \ldots \alpha_i$. These assumptions are conjoined to yield the formula:

$$\exists x_1 \ldots x_n, \exists \theta_1 \ldots \theta_p \; \rho \wedge \alpha_1 \wedge \ldots \wedge \alpha_i.$$

This process repeats until a satisfying assignment is found that is consistent with the oracles. If $\exists x_1 \ldots x_n, \exists \theta_1 \ldots \theta_p \; \rho \wedge \alpha_1 \wedge \ldots \wedge \alpha_i$ is T-satisfiable and consistent with the behavior of the oracles, then it follows the formula $\exists x_1 \ldots x_n, \forall \theta_1 \ldots \theta_p \; (\alpha_1 \wedge \ldots \wedge \alpha_i) \Rightarrow \rho$ is also T-satisfiable since the behavior of the oracle function symbols must be restricted by an assumption in $\alpha_1 \ldots \alpha_i$ for every argument used in application of an oracle function symbol in the satisfying assignment. This algorithm is illustrated in Figure 4 and given as Algorithm 1.

Theorem 1 (Soundness of SMTO algorithm). *Algorithm 1 returns UNSAT iff the SMTO problem $\exists x_1 \ldots x_m \forall \theta_1 \ldots \theta_p \, \rho$ is unsatisfiable with respect to oracles for $\theta_1, \ldots, \theta_p$ implementing oracle interfaces $\mathcal{I}_1, \ldots, \mathcal{I}_p$.*

Proof sketch. Algorithm 1 returns UNSAT when the underlying SMT solver returns UNSAT on the formula $\exists x_1 \ldots x_n, \exists \theta_1 \ldots \theta_p \; \rho \wedge \alpha_1 \wedge \ldots \wedge \alpha_i$ for some $i \geq 0$. This indicates there are no values for the x_is and no oracle implementations consistent with the assumptions generated from the oracle interfaces on which ρ evaluates to true, viz., the SMTO problem is unsatisfiable. Similarly, Algorithm 1 returns SAT only when the SMT solver returns a model that is consistent with all oracles: i.e., each oracle, when evaluated on an input query generated from the model, produces a response matching the interpretation of its corresponding oracle function symbol.

Algorithm 1: Satisfiability Modulo Oracles (SMTO)

```
input  : $\exists x_1 \dots x_n \forall \theta_i \dots \theta_p\, \rho$, Oracle Interfaces $\mathcal{I}_1 \dots \mathcal{I}_p$
output : UNSAT/SAT + assumptions $\alpha$ + optional(model)
Algorithm SMTO
    $\alpha \leftarrow true$
    while true do
        success=true if SMTSolver($\rho \wedge \alpha$)=UNSAT then
            return UNSAT, $\alpha$
        else
            model←model from SMT solver
            for $\theta_i \in \{\theta_1 \dots \theta_p\}$ do
                is_consistent, $\alpha_{new}$ ← consistent($\theta_i$, $\mathcal{I}_i$, $\rho$, model)
                $\alpha \leftarrow \alpha \wedge \alpha_{new}$
                if !is_consistent then
                    success=false
                    break
                end
            end
            if success then
                return SAT, $\alpha$, model
            end
        end
    end
Procedure consistent($\theta_i$, $\mathcal{I}_i$, $\rho$, model)
    for app in applications of $\theta_i$ in $\rho$ do
        inputs ← evaluate(app, model)
        response ← call_oracle($\mathcal{I}_i$, inputs)        // Call oracle interface
        $\alpha_{new} \leftarrow \theta_i(inputs) = response$
        if !(response=evaluate($\theta_i$(inputs),model)) then
            return False, $\alpha_{new}$
        end
    end
    return True, $\alpha_{new}$
```

Theorem 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 formula 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.

4 Synthesis Modulo Oracles

A synthesis modulo oracles problem consists of: a set of universally quantified 0-ary symbols $x_1, \ldots x_n$; a set of oracle function symbols $\theta_0, \ldots \theta_t$; a synthesis conjecture
$\exists f_1 \ldots f_m \; \forall x_1 \ldots x_m, \theta_1 \ldots \theta_p, .\phi$ where ϕ is a quantifier-free formula; and a set of oracle interfaces $\vec{\mathcal{I}}$ that refer to oracles corresponding to oracle function symbols and any additional external oracles. We now explain how the oracle assumptions, oracle constraints and oracle function symbols affect the synthesis conjecture.

Oracle Symbols and Assumptions: Suppose we have a formula
$\exists f_1 \ldots f_m \; \forall x_1 \ldots x_n, \forall \theta \phi$ and a set of external oracles corresponding to the oracle function symbols. Initially, before we call any oracles, a set of candidate functions $f_1^* \ldots f_m^*$ is correct *iff* the formula $\gamma_0 \approx \forall x_1 \ldots x_n, \forall \theta \phi$ is T-valid when $f_1 \ldots f_m$ are replaced by $f_1^* \ldots f_m^*$. Each call we make to an oracle generates an assumption α (which may be an assumption related to an oracle function symbol, or any other assumption). Making i calls to the oracle gives us this sequence of formula, where α_i is the assumption generated by the i^{th} call to the oracle:

$$\begin{aligned}
&\gamma_0 \approx \forall x_1 \ldots x_n, \forall \theta \phi \\
&\gamma_1 \approx \forall x_1 \ldots x_n, \forall \theta (\alpha_1 \Rightarrow \phi) \\
&\quad \ldots \\
&\gamma_0 \approx \forall x_1 \ldots x_n, \forall \theta (\alpha_n \Rightarrow \ldots (\alpha_2 \Rightarrow (\alpha_1 \Rightarrow \phi)))
\end{aligned}$$

Rewriting the implications as disjuctions (e.g., $\neg \alpha_1 \vee \phi$), makes it clear this sequence of formula is monotone: if a set of candidate functions is valid according to γ_0, it is valid according to all $\gamma_{i>0}$. Thus, if we find a set of valid solutions at any iteration of the synthesis process, we know the solutions will remain valid no matter how many times we call the oracles that generate assumptions.

Oracle Constraint Generators: As previously mentioned, we require oracles will not generate constraints that remove solutions to the synthesis conjecture, i.e., for any constraint β, $(\forall \vec{x} \phi) \Rightarrow \beta$ should be true . These constraints are used to guide the solver. If a sequence of oracles are called, and a sequence of constraints $\beta_1, \ldots \beta_n$ are generated, the constraints could be conjoined together and this new synthesis conjecture, $\exists \vec{f} \; \forall \vec{x}, \vec{\theta} \; \phi \wedge \beta_1 \wedge \ldots \wedge \beta_n$, will be equisatisfiable with the original synthesis conjecture. We will show how oracle constraints are used to guide the solver in the following section.

4.1 Algorithm for Synthesis with Oracles

We now proceed to describe an algorithm for solving synthesis problems using oracles, illustrated in Figure 5. The algorithm is broken down into two phases: a *synthesis phase* and an *oracle phase*, and maintains two formulas, a verification formula V and a synthesis formula S. These are initialised as $S = \exists f_1 \dots f_m.\ .true$ and $V = \exists x_1 \dots x_n\ .\theta_1 \dots \theta_p\ .\neg\phi$. The algorithm proceeds as follows, updating these two formulas:

Synthesis Phase: When the synthesis phase receives a new constraint β, it updates the synthesis formula such that $S' = S \wedge \beta$. It looks for a set of candidate functions that satisfy the synthesis formula. The candidate functions $\vec{f^*}$ are passed to the oracle phase.

Oracle Phase I: The oracle phase calls the SMTO solver to solve the verification formula, as described in section 3. In the process of solving the formula, any oracle assumption α that is generated is added to the verification formula, which becomes $\exists x_1 \dots x_n\, \forall \theta_1 \dots \theta_p (\alpha \Rightarrow \neg\phi)$. If the SMTO solver returns UNSAT, then $\vec{f^*}$ is a valid solution to the synthesis problem. If the SMTO solver returns SAT, then a counterexample constraint β is generated, which is returned to the synthesis phase.

Oracle Phase II: Additionally, in the second half of the oracle phase, the solver may call any further oracles which are available (not corresponding to oracle function symbols) and the assumptions are added to the verification formula and constraints are passed to the synthesis formula.

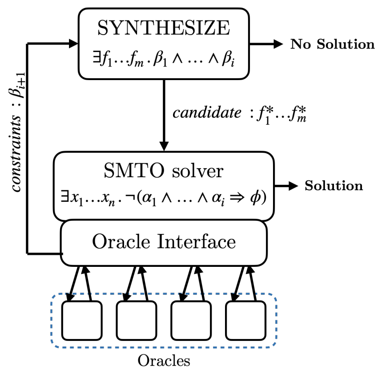

SYNTHESIZE
$\exists f_1 \dots f_m . \beta_1 \wedge \dots \wedge \beta_i$
No Solution
candidate : $f_1^* \dots f_m^*$
SMTO solver
$\exists x_1 \dots x_n . \neg(\alpha_1 \wedge \dots \wedge \alpha_i \Rightarrow \phi)$
Solution
Oracle Interface
constraints : β_{i+1}
Oracles

Fig. 5: SyMO Algorithm Illustration

Generating counterexample constraints: In order to generate a counterexample constraint β from the SMTO solver, take the formula $\exists f_1 \dots f_m .(\vec{\alpha} \Rightarrow \phi)$, where $\vec{\alpha}$ is a conjunction of any assumptions generated by the SMTO solver. We substitute any 0-ary symbols $x_1 \dots x_n$ for their concrete values on the counterexample model. We use $e(t/x)$ to indicate the formula obtained by substitution of x with t in formula e. If we obtain a counterexample which assigns $c_1 \dots c_n$ to 0-ary variables $x_1 \dots x_n$, we obtain the following counterexample constraint:

$$(\vec{\alpha} \Rightarrow \phi)(c_1/x_1, \dots, c_n/x_n)$$

All applications of functional oracle function symbols accepting only 0-ary parameters now have constant input arguments. Use $\theta_{i,j}$ to indicate the j^{th} application of oracle i in the formula $\alpha \Rightarrow \phi$. For any oracle application, we obtain the result of that oracle call from the satisfiability modulo oracles solver, which we denote

Algorithm 2: Synthesis Modulo Oracles

```
input  : $\exists f_1 \dots f_m \forall x_1 \dots x_n \forall \theta_1 \dots \theta_p \, \phi$
input  : Oracle Interfaces: $\mathcal{I}_1, \dots \mathcal{I}_k$, where $p \leq k$.
output : solution $f_1 \dots f_m$ / no solution
$A \leftarrow true$ ;                                   // conjunction of assumptions
$S \leftarrow true$ ;                                   // synthesis formula
while true do
    /* verification formula */
    $V \leftarrow \exists x_1 \dots x_n \forall \theta_i \dots \theta_p \, (A \wedge \neg\phi)$
    $\mu \leftarrow$Synthesize( $\exists f_1 \dots f_m .S$) ;
    if $\mu = \emptyset$ then
        return no solution;
    else
        $f_1^* \dots f_m^* \leftarrow \mu$ ;           // extract candidate solutions from model
        Result, $\alpha$, model $\leftarrow$ SMTO(V($f_1^* \dots f_m^*$), $\mathcal{I}_1 \dots \mathcal{I}_p$) ;
        if Result=UNSAT then
            return $f_1^* \dots f_m^*$
        else
            $A \leftarrow A \wedge \alpha$ ;
            /* Generate constraint from model */
            $S \leftarrow S \wedge$ generate_constraint(model);
            /* call all oracles that were not called by SMTO solver */
            for $i \in \mathcal{I}_p \dots \mathcal{I}_k$ do
                if * then
                    $\alpha, \beta \leftarrow$ call_oracle($i$) ;   // call oracle based on interface
                    $A \leftarrow A \wedge \alpha$ ;                 // Update assumptions
                    $S \leftarrow S \wedge \beta$ ;                  // Update synthesis formula
                end
            end
        end
    end
end
```

$z_{i,j}$.

$$(\vec{\alpha} \Rightarrow \phi)(c_1/x_1, \dots, c_n/x_n, z_{1,1}/\theta_{1,1}, \dots, z_{i,j}/\theta_{i,j})$$

Any applications of functional oracle function symbols in ϕ accepting functional parameters, i.e.,., oracles that receive the candidate function as an argument, still have a non-constant argument. The results of these oracles are therefore replaced with fresh free variables. This weakening is introduce since state-of-the-art SMT solvers do not currently support uninterpreted functions that accept n-ary parameters as arguments.

Inferring inputs for additional oracles: The inputs for oracle function symbols are inferred by the satisfiability modulo oracles solver. For other oracles, the input values are inferred by mapping concrete values from the counterexample to applications of the synthesis function in ϕ and mapping oracle query inputs to synthesis function applications within constraint generators. For example, if $f(x)$ appears in ϕ, and the counterexample for x is 7, and there exists an oracle interface with a single input z and the generator $\beta_{gen} : f(z) = y$, we will call that oracle with the value 7. To make this inference more straightforward, given an oracle interface with query inputs $y_1 \dots y_j$ and responses $z_1 \dots z_k$, we limit α_{gen} and β_{gen} to expressions in the following grammar, which notably prohibits arbitrary nesting of oracle function symbol calls:

$$
\begin{array}{l}
\alpha_{gen} ::= \mathrm{P} \\
\beta_{gen} ::= \mathrm{P} \\
\mathrm{P} ::= \neg \mathrm{P} \,|\, \mathrm{P} \vee \mathrm{P} \,|\, \mathrm{P} \wedge \mathrm{P} \,|\, \mathrm{P} \Rightarrow \mathrm{P} \,|\, \mathrm{F}(\mathrm{V}, \dots, \mathrm{V}) \,|\, \mathrm{V} < \mathrm{V} \;\; |\, \mathrm{V} > \mathrm{V} \,|\, \mathrm{V} = \mathrm{V} \,|\, \theta(\mathrm{V}, \dots, \mathrm{V}) \,| \\
\qquad \theta(F) \\
\mathrm{V} \; ::= \; y_1 \,|\, \dots \,|\, y_j \,|\, z_1 \,|\, \dots \,|\, z_k \,|\, \mathrm{F}(\mathrm{V}, \dots \mathrm{V}) \,|\, \mathtt{constant} \\
\mathrm{F} \; ::= \; f_1 \,|\, \dots \,|\, f_m \\
\theta \; ::= \; \theta_1 \,|\, \dots \,|\, \theta_m
\end{array}
$$

Theorem 3 (Soundness). *Suppose Algorithm 2 returns a solution f. This solution satisfies the synthesis conjecture.*

Proof sketch. Solutions are returned if the SMTO solver is unable to find any counterexample that shows the solution violates ϕ. Since the SMTO solver is sound (Theorem 1), it follows that the solution returned by Alg. 2 satisfies the synthesis conjecture.

5 Instances of Synthesis Modulo Oracles

5.1 Query Classes and Corresponding Interfaces

A number of different queries are categorized in work by Jha and Seshia [18], including queries such as membership queries, where the learner selects an example set of input-output pair(s) and asks if they are permitted by the specification ϕ. We describe the oracle interfaces for each of these classic query types. Suppose we are synthesising a single function $f : \sigma_1 \times \ldots \times \sigma_j \rightarrow \sigma'$ to satisfy a specification $\exists f \forall x_1, \dots x_n \forall \theta_1, \dots \theta_p \,.\phi$. For consistency with the definitions in Section 2, we use $y_1, y_2 \dots$ for oracle query identifiers and $z_1, z_2, \dots$ for oracle response identifiers, while the sorts are consistent with the sorts in the function definition. The solver infers from the specification ϕ which constant literals it should call the oracles with in place of the symbols $y_1, y_2, \dots$. The following oracles generate oracle constraints, and are not associated to oracle function symbols. Their interfaces are shown in Figure 6:

Membership queries: the learner selects a concrete set of constants of sorts $\sigma_1, \dots, \sigma_j, \sigma'$. This is an input-output pair $\{c_1, \dots c_j\}, c$ and the learner asks whether $f(c_1, \dots, c_j) = c$ is permitted by the specification ϕ. The oracle responds with a boolean variable b which indicates whether the input-output pair is permitted by the specification, and a constraint that specifies $b \iff f(c_1, \dots, c_j) = c$.

Input-output query: the learner selects a possible set of inputs $c_1, \ldots, c_j$ and requests the correct output from the oracle. This is a type of membership query, and the oracle returns a concrete value c and a constraint $(f(c_1, \ldots, c_j) = c)$.

Negative example queries: the learner asks for a negative input-output example which is not permitted by ϕ. The response from the oracle is a set of constants $c_1, \ldots, c_j, c$ of sort $\sigma_1, \ldots, \sigma_j, \sigma'$ and a constraint that $(f(c_1, \ldots, c_j) \neq c)$.

Positive example queries: the learner asks for a negative input-output example which is permitted by ϕ. The response from the oracle is a set of constants $c_1, \ldots, c_j, c$ of sort $\sigma_1, \ldots, \sigma_j, \sigma'$ and a constraint that $(f(c_1, \ldots, c_j) = c)$.

$$I_{mem} = \begin{cases} Q & : (y_1\, \sigma_1), \ldots, (y_j, \sigma_j)(y_{j+1}\ \sigma') \\ R & : (z_1\ \mathit{bool}) \\ \beta_{gen} & : z_1 \iff (f(y_1, \ldots, y_j) = y_{j+1}) \end{cases}$$

Membership query interface

$$I_{i/o} = \begin{cases} Q & : (y_1\, \sigma_1), \ldots, (y_j\, \sigma_j) \\ R & : (z_1\, \sigma') \\ \beta_{gen} & : (f(y_1, \ldots, y_j) = z_1) \end{cases}$$

Input-output example interface

$$I_{neg} = \begin{cases} Q & : \emptyset \\ R & : (z_1\, \sigma_1), \ldots, (z_j\, \sigma_j), (z\, \sigma') \\ \beta_{gen} & : (f(z_1, \ldots, z_j) \neq z_{j+1}) \end{cases}$$

Negative example interface

$$I_{pos} = \begin{cases} Q & : \emptyset \\ R & : (z_1\, \sigma_1), \ldots, (z_j\, \sigma_j), (z_{j+1}\, \sigma') \\ \beta_{gen} & : (f(z_1, \ldots, z_j) = z_{j+1}) \end{cases}$$

Positive example interface

Fig. 6: Oracle interfaces for constraint generating oracles. *bool* indicates a boolean sort.

The following oracles are associated to oracle function symbols and generate oracle constraints and assumptions:

Correctness queries: Given a candidate program y, the oracle returns *true* if y is correct and *false* otherwise, along with counterexample values $c_1, \ldots c_n$ corresponding to the variables $x_1, \ldots x_n$ if y is not correct. Suppose that $x_1, \ldots x_n$ have sorts $\sigma_1, \ldots \sigma_n$. The constraint generator is the formula ϕ with the oracle response symbols $z_1, \ldots, z_n$ in place of $x_1, \ldots x_n$. The constraint is generated by substituting the counterexample values in place of the oracle response symbols, giving $\phi(c_1/x_1, \ldots, c_n/x_n)$. A correctness oracle is associated to an oracle function symbol θ which accepts a function as input, which is asserted to be *true* in the verification formula ϕ. When the oracle is called and the function is correct, it generates an oracle assumption $\theta(y) = \mathit{true}$. If the oracle is called and the function is incorrect, it generates the assumption $\theta(y) = \mathit{false}$. The interface is as below, and correctness oracles without counterexamples are handled in the same way but without β_{gen}.:

$$I_{corr} = \begin{cases} Q & : (y\ (\sigma_1 \times \ldots \times \sigma_j \to \sigma')) \\ R & : (z_1\, \sigma_1), \ldots, (z_n\, \sigma_n), (z_{n+1}\ bool), \\ \alpha_{gen} & : \theta(f^*) = z_{n+1} \\ \beta_{gen} & : \phi(z_1/x_1, \ldots, z_n/x_n) \end{cases}$$

5.2 Reduction from Existing Synthesis Algorithms

The synthesis modulo oracles framework we present is a flexible and general framework for program synthesis. It is possible for this framework and the corresponding algorithm to 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 [24], ICE-learning [16] and Synthesis with distinguishing inputs [17].

CounterExample Guided Inductive Synthesis: Suppose we are solving a synthesis formula with a single variable x and a single synthesis function f, where $f : \sigma \to \sigma'$. 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 way 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 synthesising a function f, and receiving a candidate synthesis function $y : \sigma \to \sigma'$:

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

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	$X_{cex} = c_1$	$\beta_1 = \phi(k_1/x)$
	$.\ \exists f.\forall x \in X_{cex}\ .\phi(x)$	$\exists f.\beta_1$
3	$X_{cex} = c_1, c_2$	$\beta_2 = \phi(k_2/x)$
	$.\ \exists f.\forall x \in X_{cex}\ .\phi(x)$	$\exists f.\beta_1 \wedge \beta_2$
...	...	...

Table 1: 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.

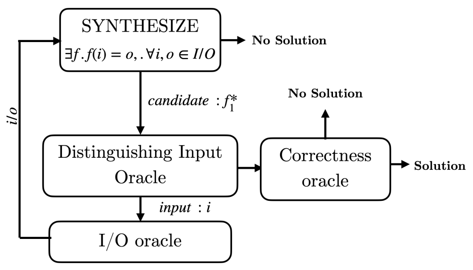


Fig. 7: Synthesis with distinguishing inputs

Synthesis with distinguishing inputs: This algorithm [17], 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.

If such a function exists, the distinguishing input is passed to an input/output oracle, and the input/output pair is passed to the synthesiser. If a distinguishing input does not exist, 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 implement this algorithm in our framework observing that synthesiser 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 synthesising f, and a candidate synthesis function $y : \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma'$:

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

ICE learning: ICE learning [16] 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

$$(\forall x \in E.inv(x)) \;\wedge\; (\forall x \in C.\neg inv(x)) \;\wedge\; (\forall (x, x') \in I.inv(x) \Rightarrow inv(x')).$$

The synthesis modulo oracles 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 $x_1 \dots x_n$, 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.

$$I_{corr} = \begin{cases} Q & : (y\,(\sigma_1 \times \ldots \times \sigma_n \to bool)) \\ R & : (z\; bool) \\ \alpha gen & : \theta_{corr}(y) = z \end{cases} \qquad I_{neg} = \begin{cases} Q & : (y\,(\sigma_1 \times \ldots \times \sigma_n \to bool)) \\ R & : (z_1\,\sigma_1), \ldots, (z_n\,\sigma_n) \\ \beta gen & : \neg inv(z_1, \ldots z_n) \end{cases}$$

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

Fig. 8: Oracle interfaces for ICE learning, receiving a candidate invariant y.

6 Delphi: a Satisfiability and Synthesis Modulo Oracles Solver

We implement the algorithms described above in a prototype solver Delphi[4]. The solver is configured as follows: the synthesis phase uses a symbolic synthesis

[4] link: `https://drive.google.com/file/d/1IW6EcLcJKF7TU_F09WHisTqz_dpyqfRU`

encoding, which uses either an SMT solver(Z3 version 4.4.8 [13] or CVC4 version 1.9 [7]) or MiniSAT version 2.2 [14] as a base solver. Alternatively the synthesis phase may be delegated to any existing synthesis solver that accepts SyGuS-IF. The default is to use CVC4 as a synthesis solver for the synthesis phase, and MiniSAT as a bitblaster for the SMTO solver. The solver supports linear integer arithmetic and bitvector theories and accepts input files expressed in an extension of SMT-lib [6] and SyGuS-IF [20]. We provide a number of utilities for modeling

Benchmarks	constraints	time(s)	Benchmarks	constraints	time(s)
Image invert/zombie	10/20	1/1	Image attenuate	16	14
Image crop1/2/3	22/27/33	14/27/33	Image brighter	18	11
Image darker1/2	34/11	256/18			

Table 2: 9 image transformation examples.

oracles of different types, and present case studies as an illustration of generality (although we remark that SyMO is likely less efficient than a specialized algorithm for each domain due to overhead incurred calling external oracles).

6.1 Oracle templates and utilities

The key benefit of our framework is that a user may express and solve a broad range of synthesis problems without building their own custom synthesis engine. Instead, they need only provide oracles and a specification. We provide a number of oracle utilities demonstrating a few such cases:

Programming by Example: We provide template files for synthesising summaries of executable black-boxes based on input-output examples, which we call PBE-exe. The user associates the black-box to an oracle function symbol θ_{ref} and specifies N, the number of input-output pairs the summary must satisfy. These examples are selected using another black-box which generates pseudo-random numbers based off a seed, and is associated to an oracle function symbol θ_{rand}. The specification thus is to satisfy the query $\exists f\, \forall x.\theta_{rand}, \theta_{ref}, (0 \leq x \leq N) \implies f(\theta_{rand}(x)) = \theta_{ref}(\theta_{rand}(x))$. The oracles for θ_{ref} and θ_{rand} have interfaces of the form $\mathcal{I}_{i/o}$, and the SyMO solver is used as correctness oracle.

ICE-learning: We provide example oracles for the oracles for ICE-learning with interfaces shown in Section 5. The negative and implication oracles are based on using an SMT solver to find counterexamples. The positive oracle uses both an SMT solver to find counterexamples, and executes an unrolling of the loop in order to generate positive examples.

Compilation and execution oracles: We provide a utility for compiling candidate functions expressed in SyGuS-IF into C code, which oracles can then compile and execute.

6.2 Case Studies

Image Processing: To illustrate the flexibility of the framework consider an image processing challenge. Given two images, we wish to synthesize a transformation between the two. Figure 2 shows two such example transformations. The oracle is a program that 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, and then executes the compiled code with the original image as input. It compares the result with the target image. If all the pixels are identical, the oracle returns "true". 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 contains a C compiler and the STB image processing library[5], and the workflow is illustrated in Figure 9.

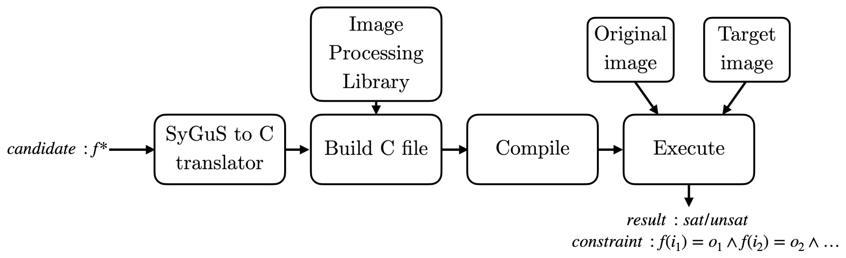


Fig. 9: Oracle for image transformations

SyGuS benchmarks: We apply CEGIS, PBE and ICE instantiations of SyMO on a set of randomly selected SyGuS benchmarks [2]. We compare executable oracles with using CEGIS with SMT-lib constraints for the PBE benchmarks. Note in PBE-cegis, the concrete input-output pairs are included in ϕ and a single call to the SMT solver oracle validates satisfaction of these, thus typically only 2 oracle calls are required.

Our observations are that the performance of different SyMO configurations depends on two factors: both the number and quality of constraints needed to accurately represent the target function, as well as the difficulty the synthesis phase has in solving the constraints. The image crop transformations are particularly affected by the sampling of the counterexample constraints, since there must be sufficient constraints to accurately place the location of the crop boundary. We also observe that our instantiation of ICE typically generates fewer constraints than CEGIS, and the failure mode of the respective algorithms differs: CEGIS typically generates increasingly many constraints, enumerating through constant

[5] `https://github.com/nothings/stb`

values; whilst ICE generates sets of constraints that become tricky to solve and the synthesis phase absorbs all the solving time. This hints at the benefits of a framework that can combine different oracle types. We observe that, whilst the SyMO framework enables these image transformation problems to be solved with ease, it would be very difficult to obtain a pure oracle-free logical encoding of this problem which incorporates both a compiler and an image manipulation library. Without this framework, a user wishing to solve these problems with program synthesis would have to build their own custom synthesis tool.

Benchmark	constraints	time	Benchmark	constraints	time
inv-CEGIS (avg)	40	163s	inv-ICE (avg)	16	121s
Sin-PBE	17	3s	Cos-PBE	17	3s
PBE-exe(avg)	18.8	91s	PBE-cegis(avg)	2	136s

Table 3: Case studies of SyMO. PBE results are an average of 25 benchmarks, and invariant comparisons are an average of 5 examples. Time is reported in seconds to nearest second and timeouts are awarded a penalty of 300s. Number of calls is averaged across all results (solved and unsolved).

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 (for instance trigonometric functions) 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 learnt 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 [1, 15], which will require use of context-sensitive grammars in the synthesis phase. There are a number of future work developments that are implementation focused: we will develop more seamless integration with oracles, permitting communication via APIs as well as the existing command line access to binaries; and implement existing successful synthesis algorithms in the synthesis phase.

7 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. Delphi is the first implemented solver for $SyMO$ problems and our case studies demonstrate the flexibility of a reasoning engine that can incorporate oracles based on complex systems. By making use of this framework, a user is able to solve a broad range of synthesis problems without building their own custom synthesis tool.

Acknowledgments

We thank Susmit Jha for his feedback on an earlier version of this paper. This work was supported in part by NSF grants CNS-1739816 and CCF-1837132, by the DARPA LOGiCS project under contract FA8750-20-C-0156, by the iCyPhy center, and by gifts from Intel, Amazon, and Microsoft.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. Clark Barrett, Cesare Tinelli, et al. The SMT-LIB standard: Version 2.0.
7. 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.
8. Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
9. Bryan A. Brady, Randal E. Bryant, and Sanjit A. Seshia. Learning conditional abstractions. In *FMCAD*, pages 116–124. FMCAD Inc., 2011.
10. 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.
11. 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.
12. 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.
13. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

14. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
15. Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435. ACM, 2018.
16. 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.
17. 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.
18. Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.
19. Anders Miltner, Saswat Padhi, Todd D. Millstein, and David Walker. Data-driven inference of representation invariants. In *PLDI*, pages 1–15. ACM, 2020.
20. Abhishek Udupa Mukund Raghothaman, Andrew Reynolds. The SyGuS language standard version 2.0. `https://sygus.org/language/`, 2019.
21. Saswat Padhi, Rahul Sharma, and Todd Millstein. LoopInvGen: A loop invariant generator based on precondition inference. *ArXiv e-prints*, 2019.
22. 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.
23. 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.
24. 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.
25. 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.