# DRYADSYNTH : A Concolic SyGuS Solver

Kangjing Huang Xiaokang Qiu Yanjun Wang
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana 47907–2035
Email: {kangjinghuang,xkqiu,wang3204}@purdue.edu

Qi Tian
Nanjing University
Nanjing, Jiangsu, China
Email: 141250126@smail.nju.edu.cn

***Abstract*—this paper presents DRYADSYNTH , a concolic SyGuS solver. The synthesis algorithm is CEGIS-based and combines enumerative search and symbolic search.**

DRYADSYNTH is a Syntax-Guided Synthesis (SyGuS) synthesizer that combines explicit search and symbolic search. The solver currently supports synthesis for Conditional Linear Integer Arithmetic (CLIA) and Invariants (INV). For both the two tracks, a conditional arithmetic function or predicate is represented using a decision-tree data structure.

*Decision Tree Representation*

To represent a $n$-ary function $f(x_1, \ldots, x_n)$ in LIA, we first normalize to the *decision tree normal form* described in Figure 1. It is not hard to see that every LIA function can be converted to this normal form. The proof relies on the fact that every atomic LIA equation or inequation can be rewritten to the form of $e \geq 0$, where $e$ is a linear expression. Then the normal form expression can be represented as a binary tree in which every node contains $n+1$ integer fields $c_0, \ldots, c_n$, representing the expression $c_0 + \sum_{1 \leq i \leq n} c_i \cdot x_i$. Each decision node (non-leaf node) tests whether the associated expression is nonnegative and proceeds to the "true" child or "false" child. Each leaf node determines the value of the function using its associated expression. For example, the binary max function

$$max2(x_1, x_2) \stackrel{def}{=} \text{ if } x_1 \geq x_2 \text{ then } x_1 \text{ else } x_2$$

can be represented as the tree shown in Figure 2.

Representing invariants is similar. The only difference is that the leaf node will return true or false, depending on whether the associated expression is nonnegative or not.

We assume that the decision-tree is a full binary tree – if it is not full, one can extend the tree with extra nodes with a tautology condition, e.g., $1 \geq 0$. This allows us to reduce the SyGuS synthesis problem to the problem of searching for:

a) the height of the decision tree;
b) the field values $c_0, \ldots, c_n$ for each node.

*Concolic Search*

Overall, DRYADSYNTH implements the standard CEGIS framework [2]. The verifier is quite straightforward: let the synthesis specification be $\varphi(f, \vec{x})$, then whenever the synthesizer proposes a candidate solution $f_0$, the specification $\forall \vec{x}.\varphi(f_0, \vec{x})$ is encoded as a SMT query and solved by Z3 [1], a state-of-the-art SMT solver. If the verification fails, the counterexample (assignments to $x_1$ through $x_n$) will be added to the accumulated set of counterexamples.

The synthesizer in the CEGIS framework adopts an enumeration strategy: it enumerates all possible heights of the decision tree, starting from 1. In each iteration of the CEGIS loop, the synthesizer attempts to synthesize a solution with the current height $h$. If there is no solution, the synthesizer increases the height to $h+1$ and retry.

For each synthesis attempt, the height $h$ is fixed and a set of counterexamples $\Gamma$ is available. The synthesis task is to find all field values of the full decision tree of height $h$, representing a function $f_0$, such that for any counterexample $\vec{d}$ in $\Gamma$, the synthesis specification is valid: $\bigwedge_{\vec{d} \in \Gamma} \varphi(f_0, \vec{d})$.

$$\begin{array}{rrcl} \text{Int Const: } & c_0, c_1, c_2 \ldots & & \\ \text{Expr: } & e, e_1, e_2 & ::= & c_0 + \sum\limits_{1 \leq i \leq n} c_i \cdot x_i \\ \text{Atom Cond: } & \alpha & ::= & e \geq 0 \\ \text{Cond Expr: } & E, E_1, E_2 & ::= & \alpha \mid \text{if } \alpha \text{ then } E_1 \text{ else } E_2 \\ \text{Cond: } & \varphi & ::= & \alpha \mid \text{if } \alpha \text{ then } \varphi_1 \text{ else } \varphi_2 \end{array}$$

Fig. 1. Decision Tree Normal Form

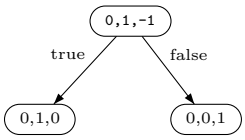


Fig. 2. Representation of the $max2$ function

DRYADSYNTH symbolically solves the synthesis task by encoding it to a SMT query. Notice that the full decision tree of height $h$ consists of $2^h - 1$ nodes. We first encode the location of each node to a distinct integer between 0 and $2^h - 2$, and represent the field $c_i$ of node $j$ as $c_i^j$. Then each occurrence of $f(\vec{d})$ in the synthesis condition with concrete $\vec{d}$ can be encoded using these field variables. For example, if an assignment $(x_1 = 1, x_2 = 2)$ is a counterexample in $\Gamma$ and the current height is 2, then the expression $f(1, -2)$ can be represented as a LIA expression

$$\text{if } c_0^0 + c_1^0 - 2c_2^0 \geq 0 \text{ then } c_0^1 + c_1^1 - 2c_2^1 \text{ else } c_0^2 + c_1^2 - 2c_2^2$$

*Parallelization and Optimization*

While the naive enumeration algorithm incrementally searches all possible height of the decision tree, starting from 1, the algorithm can be naturally parallelized to leverage the multi-cores, if available. If there are $n$ cores available on the machine, the parallel version DRYADSYNTH solver runs the same CEGIS algorithm with $n$ different heights on the $n$ cores independently, i.e., each maintains a separate set of counterexamples. The algorithm starts with the $n$ smallest heights, $\{1, \ldots, n\}$. It also maintains a variable $p$ as the next height to search, starting from $n + 1$. Whenever a core concludes that there is no solution at the current height, it starts a new CEGIS loop at height $p$, and the value of $p$ gets increased. The whole algorithm stops whenever a core finds a solution.

We observed several things from the synthesis results for SyGuS benchmarks:

1) most of the $c_i$'s are very small. In fact, a lot of benchmarks don't need any coefficient beyond 0, 1, or $-1$.
2) when a benchmark's solution contains large coefficients, there is usually a large coefficient in the specification as well.
3) if the search space to restricted to solutions with small coefficients only, the performance can be significantly improved.

These observations allow us to optimize the synthesis algorithm by leveraging a multi-stage strategy. For each height $h$, we split the CEGIS loop into three stages:

1) in each synthesis iteration, we impose extra constraint $c_i^j \leq 1 \wedge c_i^j \geq -1$ for every $i$ and $j$. Once the synthesis query can't find any solution, move to the next stage –
2) in each synthesis iteration, we impose extra constraint $c_i^j \leq D \wedge c_i^j \geq -D$ where $D$ is a bound such that all coefficients in $\varphi(f, \vec{x})$. Once the synthesis query can't find any solution, move to the next stage –
3) resume the regular synthesis setting, i.e., remove all constraints on coefficients.

We also implemented several simplification strategies. For example, a variable $x$ is irrelevant in an invariant synthesis problem if: a) $x$ does not appear in the $pre$ function or $x'$ does not appear in the $trans$ function; and b) $x$ does not appear in the $post$ function.

*Decidable fragments*

In addition to our Concolic algorithm as a general-purpose algorithm, we also identified two decidable fragments as subclasses of SyGuS benchmarks, and developed two *pure symbolic and decidable procedures*, namely *Strong Single Invocation*(SSI) and *Acyclic Translational* invariant synthesis in DRYADSYNTH . Although these two decidable fragments are not general enough to serve as standalone synthesiszers, we found they cover a nontrivial portion of SyGuS benchmarks and could work as a complement to our concolic algorithm.

*Strong Single Invocation:* This fragment is defined based on a *strong single invocation* property:

A CLIA synthesis problem with synthesis specification $\varphi(f, \vec{x})$ has the Strong Single Invocation property if there is a single vector of terms $t$ such that

1) $f$ only occurs in $\varphi$ as $f(\vec{t})$;
2) in each atomic formula, $f(\vec{t})$ occurs at most once.

Due to the first property, for such a problem, the formula $\exists f \forall \vec{x} \varphi(f; \vec{x})$ is logically equivalent to $\forall \vec{x} \exists z \varphi(z; \vec{x})$, where $\varphi(z; \vec{x})$ replaces all occurrences of $f(\vec{t})$ in $\varphi(z; \vec{x})$ with $z$, a CLIA term.

Now notice that its negation, $\forall z \neg \varphi(z; \vec{x})$, is a first-order CLIA formula, and the satisfiability can be decided. Obviously, if the formula is satisfiable, the satisfying assignment of $\vec{x}$ witnesses the original synthesis problem is unsynthesizable. Hence what remains is to show that if $\forall z \neg \varphi(z; \vec{x})$ is unsatisfiable, i.e., $\forall \vec{x} \exists z \varphi(z; \vec{x})$ is valid, the function $f$ can be implemented: there is a CLIA term $t(\vec{x})$ such that $\varphi(t(\vec{x}); \vec{x})$ is valid.

Now due to the second property, $\varphi(z; x)$ could be normalized such that every atomic formula is in the form of $z \geq t(\vec{x})$ or $z \leq t(\vec{x})$. Now as we assume $\forall \vec{x} \exists z \varphi(z; \vec{x})$ is valid, and let the set of all $t(\vec{x})$ in these forms to be $T$. For each $t_i(\vec{x}) \in T$, $z = t_i(\vec{x})$ must be a valid solution if $\varphi(t_i(\vec{x}); \vec{x})$ is valid. As specification $\varphi(z; x)$ only has atomic formulae in such forms, by enumerating every $t_i(\vec{x}) \in T$ from $\varphi(z; x)$ as the value of $f(\vec{t})$ and check if $\varphi(t_i(\vec{x}); x)$ is valid as the branch condition to use that term, we could build a solution to the problem:

$$\begin{aligned} f(\vec{x}) = z \quad = \quad & \mathrm{ite}\Big(\varphi(t_1(\vec{x}); \vec{x}),\ t_1(\vec{x}), \\ & \quad \mathrm{ite}\Big(\varphi(t_2(\vec{x}); \vec{x}),\ t_2(\vec{x}), \\ & \quad\quad \mathrm{ite}\Big(\cdots,\ t_3(\vec{x}), \\ & \quad\quad\quad \cdots, 0\Big)\cdots\Big)\Big) \end{aligned}$$

*Acyclic Translational Invariant Synthesis:* Another decidable fragment is for invariant synthesis. First we define a invariant synthesis problem

$$\begin{aligned} \varphi(\mathbf{inv}; \vec{x}) \quad \equiv \quad & \Big(\mathbf{pre}(\vec{x}) \rightarrow \mathbf{inv}(\vec{x})\Big) \\ \wedge \quad & \Big(\mathbf{inv}(\vec{x}) \rightarrow \mathbf{inv}(\mathbf{trans}(\vec{x}))\Big) \\ \wedge \quad & \Big(\mathbf{inv}(\vec{x}) \rightarrow \mathbf{post}(\vec{x})\Big) \end{aligned}$$

to be *translational* if the corresponding $\mathbf{trans}(\vec{x})$ term is defined in a way such that all atomic terms occurring in it

are of the form $\vec{x}+\vec{c}$. We call this term $\mathbf{trans}$ as translational CLIA term as well.

Then, we could extend the translational CLIA term $\mathbf{trans}(\vec{x})$ to a translation on any CLIA condition $\varphi(\vec{x})$, denoted as $\mathbf{trans}(\varphi)$ so that $\forall\vec{x}\Big(\varphi(\vec{x}) \leftrightarrow \mathbf{trans}(\varphi)(\mathbf{trans}(\vec{x}))\Big)$.

At this point, intuitively, the problem of finding invariant for the translational invariant synthesis problem could be converted to finding the *transitive closure* of $\mathbf{pre}$ in $\mathbf{trans}$: $\mathbf{trans}^*(\mathbf{pre})$. Actually it could be proved that, the translational invariant synthesis problem has a solution if and only if $\mathbf{trans}^*(\mathbf{pre}) \rightarrow \mathbf{post}$ is valid, and $\mathbf{trans}^*(\mathbf{pre})$ should be the strongest invariant possible.

To calculate the transitive closure, we normalize $\mathbf{trans}$ so that in this term, for each occurrence of conditional term $\mathrm{ite}(\varphi,\vec{t_1},\vec{t_2})$, $\varphi$ is free of disjunction, and $\vec{t_1}$ is an atomic term of the form $\vec{x}+\vec{c}$; and for each two occurrences $\mathrm{ite}(\varphi,\vec{t_1},\vec{t_2})$ and $\mathrm{ite}(\psi,\vec{t_3},\vec{t_4})$, $\varphi$ and $\psi$ are disjoint, i.e. $\varphi\wedge\psi=\bot$;

Then we could obtain two other translational terms from $\mathbf{trans}$, $\mathbf{trans}_l$ and $\mathbf{trans}_c$, such that for any $\vec{x}$ and $\vec{y}$, $\mathbf{trans}_l(\vec{x})=\vec{y}$ if and only if $\mathbf{trans}(\vec{x})=\vec{y}$ and $\vec{x}$ and $\vec{y}$ belong to the same branch, $\mathbf{trans}_c(\vec{x})=\vec{y}$ if and only if $\mathbf{trans}(\vec{x})=\vec{y}$ and $\vec{x}$ and $\vec{y}$ belong to two different branches.

By dividing the translational terms to local translational terms ($\mathbf{trans}_l$) and cross-branch translational terms ($\mathbf{trans}_c$). The symbolic calculation of the transitive closure could be converted as well. It could be proven that for any translational CLIA term $\mathbf{trans}$ and any CLIA condition $\varphi$, $\mathbf{trans}^*(\varphi)\equiv\mathbf{trans}_l^*\circ(\mathbf{trans}_c\circ\mathbf{trans}_l^*)^*$

For the local translational terms, we use a quantifier elimination based fast transformation to calculate its transitive closure $\mathbf{trans}_l^*$. $\mathbf{trans}_l^*$ perserves the branch so to calculate $\mathbf{trans}_l^*(\varphi)$ we could divide $\varphi$ into each branch and calculate the transitive closure separately. It could be proven that given the branch condition is $\psi$ and translational term is $\vec{x}+\vec{c}$, the local transitive closure could be expressed as

$$\mathbf{trans}_l^*(\varphi)\equiv\exists k\,(k\geq 0\wedge\varphi(\vec{x}-k\cdot\vec{c})\wedge\psi(\vec{x}-k\cdot\vec{c}))$$

Intuitively, this comes from the observations that if a vector $\vec{x}$ is in the local transitive closure $\mathbf{trans}_l^*(\varphi)$, it must be able to be reached from another vector in $\varphi$ in a finite number of translations without crossing the branches.

Next, we introduce the notion of transition graph for translational terms. The transition graph $(V,E)$ of $\mathbf{trans}$ is a directed graph with $V$ being the set of all branches $V=\{v_{\psi,\vec{c}}\}$ as vertices and $E$ denoting reachability between branches $E=\{(v_{\psi,\vec{c}},v_{\psi',\vec{c'}})\mid\psi(\vec{x})\wedge\psi'(\vec{x}+\vec{c})\text{ is satisfiable}\}$ as directed edges. We denote the translational term $\mathbf{trans}$ and its related translational invariant synthesis problem to be acyclic translational if its corresponding transition graph is acyclic.

For acyclic translational terms, the calculation of $(\mathbf{trans}_c\circ\mathbf{trans}_l^*)^*$ could be converted to a finite series of the underlying translations. As we have already discussed about the calculation of $\mathbf{trans}_l^*$ by quantifier elimination. The calculation of $\mathbf{trans}^*(\varphi)\equiv\mathbf{trans}_l^*\circ(\mathbf{trans}_c\circ\mathbf{trans}_l^*)^*$ could thus be done in such a decidable procedure. And then we can check it against $\mathbf{post}$ to determine if it is a solution to the problem or conclude that there's no solution.

One thing needs to be noted is that the invariant generated by the above procedure could contain modulo expressions generated from quantifier eliminations, which would make the solution non-CLIA. This could be solved by dropping all the modulo assertions (replace them with $\top$). It could be proven that if the term with the modulo assertions is a solution to the synthesis problem, the one with the assertions dropped is a solution as well.

## REFERENCES


[1] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[2] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS'06*. ACM, 2006, pp. 404–415.