Syntax-Guided Termination Analysis*

Grigory Fedyukovich, Yueling Zhang, Aarti Gupta

Princeton University, Princeton, USA
{grigoryf,yuelingz,aartig}@cs.princeton.edu

Abstract. We present new algorithms for proving program termination and non-termination using syntax-guided synthesis. They exploit the symbolic encoding of programs and automatically construct a formal grammar for symbolic constraints that are used to synthesize either a termination argument or a non-terminating program refinement. The constraints are then added back to the program encoding, and an off-the-shelf constraint solver decides on their fitness and on the progress of the algorithms. The evaluation of our implementation, called FreqTerm, shows that although the formal grammar is limited to the syntax of the program, in the majority of cases our algorithms are effective and fast. Importantly, FreqTerm is competitive with state-of-the-art on a wide range of terminating and non-terminating benchmarks, and it significantly outperforms state-of-the-art on proving non-termination of a class of programs arising from large-scale Event-Condition-Action systems.

1 Introduction

Originated from the field of program synthesis, an approach of syntax-guided synthesis (SyGuS) [2] has recently been applied [16,14] to verification of program safety. In general, a SyGuS-based method walks through a set of candidates, restricted by a formal grammar, and searches for a candidate that meets the predetermined specification. The distinguishing insight of [16,14], in which SyGuS discovers inductive invariants, is that a formal grammar need not necessarily be provided by the user (as in applications to program synthesis), but instead it could be automatically constructed on the fly from the symbolic encoding of the program being analyzed. Despite being incomplete, the approach shows remarkable practical success due to its ability to discover various facts about program behaviors whose syntactic representations are compact and look similar to the actual program statements.

Problems of proving and disproving program termination have a known connection to safety verification, e.g., [7,19,41,28,40]. In particular, to prove termination, a program could be augmented by a counter (or a set of counters) that is initially assigned a reasonably large value and monotonically decreases at each iteration [39]. It remains to solve a safety verification task: to prove that the counter never goes negative. On the other hand, to prove that a program has

* This work was supported in part by NSF Grant 1525936.

only infinite traces, one could prove that the negation of a loop guard is never reachable, which boils down to another safety verification task. This knowledge motivates us not only to exploit safety verification as a subroutine in our techniques, but also to adapt successful methods across application domains.

We present a set of SyGuS-based algorithms for proving and disproving termination. For the former, our algorithm LinRank adds a decrementing counter to a loop, iteratively guesses lower bounds on its initial value (using the syntactic patterns obtained from the code), which lead to the safety verification tasks to be solved by an off-the-shelf Horn solver. Existence of an inductive invariant guarantees termination, and the algorithm converges. Otherwise LinRank proceeds to strengthening the lower bounds by adding another guess. Similarly, our algorithm LexRank deals with a system of extra counters ordered lexicographically and thus enables termination analysis for a wider class of programs.

For proving non-termination, we present a novel algorithm NontermRef that iteratively searches for a restriction on the loop guard, that *might lead* to infinite traces. Since safety verification cannot in general answer such queries, we build NontermRef on top of a solver for the validity of $\forall\exists$-formulas. In particular, we prove that if at the beginning of any iteration the desired restriction is fulfilled, then there exists a sequence of states from the beginning to the end of that iteration, and the desired restriction is fulfilled at the end of that iteration as well. Recent symbolic techniques [15] to handle quantifier alternation enabled us to prove non-termination of a large class of programs for which a reduction to safety verification is not effective.

These three algorithms are independent of each other, but they all rely on a generator of constraints that are further applied in different contexts. This distinguishes our work from most of the related approaches [36,7,40,30,18,23,32,20,41]. The key insight, adapted from [16,14], is that the syntactical structures that appear in the program give rise to a formal grammar, from which many candidates could be sampled. Because the grammar is composed from a finite number of numeric constants, operators, and variable combinations, the number of sampled constraints is always finite. Furthermore, since our samples are syntactically close to the actual constructs which appear in the code, they often provide a practical guidance towards the proof of the task. Thus in the majority of cases, the algorithms converge with the successful result.

We have implemented our algorithms in a tool called FreqTerm, which utilizes solvers for Satisfiability Modulo Theory (SMT) [11,15] and satisfiability of constrained Horn clauses [24,26,16]. These automatic provers become more robust and powerful every day, which affects performance of FreqTerm only positively. We have evaluated FreqTerm on a range of terminating and non-terminating programs taken from SVCOMP[1] and on large-scale benchmarks arising from Event-Condition-Action systems[2] (ECA). Compared to state-of-the-art termination analyzers [22,30,18], FreqTerm exhibits a competitive run-

[1] Software Verification Competition, `http://sv-comp.sosy-lab.org/`.
[2] Provided at `http://rers-challenge.org/2012/index.php?page=problems`.

time, and achieves several orders of magnitude performance improvement while proving non-termination of ECAs.

In the rest of the paper, we give background on automated verification (Sect. 2) and on SyGuS (Sect. 3); then we describe the application of SyGuS for proving termination (Sect. 4) and non-termination (Sect. 5). Finally, after reporting experimental results (Sect. 6), we overview related work (Sect. 7) and conclude the paper (Sect. 8).

2 Background and Notation

In this work, we formulate tasks arising in automated program analysis by encoding them to instances of the SMT problem [12]: for a given first-order formula φ and a background theory to decide whether there is an assignment m of values from the theory to variables in φ that makes φ true (denoted $m \models \varphi$). If every assignment to φ is also an assignment to some formula ψ, we write $\varphi \implies \psi$.

Definition 1. *A* transition system *P is a tuple $\langle V \cup V', Init, Tr\rangle$, where V is a vector of variables; V' is its primed copy; formulas $Init$ and Tr encode the* initial states *and the* transition relation *respectively.*

We view *programs* as *transition systems* and throughout the paper use both terms interchangeably. An assignment s of values to all variables in V (or any copy of V such as V') is called a *state*. A trace is a (possibly infinite) sequence of states $s, s', \ldots$, such that (1) $s \models Init$, and (2) for each i, $s^{(i)}, s^{(i+1)} \models Tr$.

We assume, without loss of generality, that the transition-relation formula $Tr(V, V')$ is in Conjunctive Normal Form, and we split $Tr(V, V')$ to a conjunction $Guard(V) \wedge Body(V, V')$, where $Guard(V)$ is the maximal subset of conjuncts of Tr expressed over variables just from V, and every conjunct of $Body(V, V')$ can have appearances of variables from V and V'.

Intuitively, formula $Guard(V)$ encodes a loop guard of the program, whose loop body is encoded in $Body(V, V')$. For example, for a program shown in Fig. 1a, $V = \{x, y, K\}$, the $Guard = y < K \vee y > K$, and the entire encoding of the transition relation is shown in Fig. 1b.

Definition 2. *If each program trace contains a state s, such that $s \models \neg Guard$, then the program is called* terminating *(otherwise, it is called* non-terminating*).*

Tasks of proving termination and non-termination are often reduced to tasks of proving program safety. A *safety verification task* is a pair $\langle P, Err\rangle$, where $P = \langle V \cup V', Init, Tr\rangle$ is a program, and Err is an encoding of the *error states*. It has a solution if there exists a formula, called a *safe inductive invariant*, that implies $Init$, is closed under Tr, and is inconsistent with Err.

Definition 3. *Let $P = \langle V \cup V', Init, Tr\rangle$; a formula Inv is a* safe inductive invariant *if the following conditions hold:* (1) $Init(V) \implies Inv(V)$, (2) $Inv(V) \wedge Tr(V, V') \implies Inv(V')$, *and* (3) $Inv(V) \wedge Err(V) \implies \bot$.

If there exists a trace c (called a *counterexample*) that contains a state s, such that $s \models Err$, then the safety verification task does not have a solution.

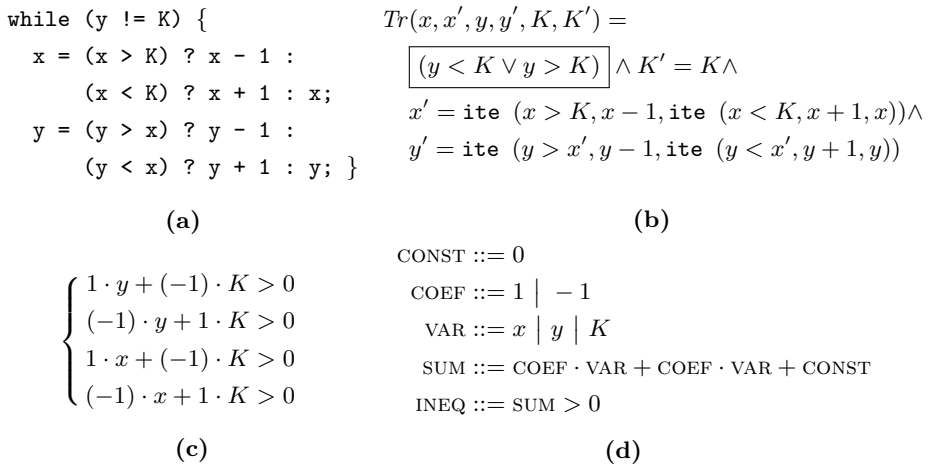


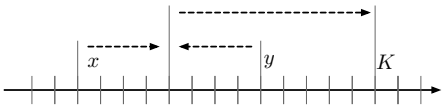
Fig. 1: **(a)**: C-code; **(b)**: transition relation Tr (in the framebox – *Guard*); **(c)**: formulas S extracted from Tr and normalized; **(d)**: grammar that generalizes S.

3 Exploiting Program Syntax

The key driver of our termination and non-termination provers is a generator of constraints that help to analyze the given program in different ways. The source code often gives useful information, e.g., of occurrences of variables, constants, arithmetic and comparison operators, that could bootstrap the formula generator. We rely on the SyGuS-based algorithm [16] introduced for verifying program safety. It automatically constructs the grammar G based on the fixed set of formulas S obtained by traversing parse trees of *Init*, *Tr*, and *Err*. In our case, *Err* is not given, so G is based only on *Init* and *Tr*.

For simplicity, we require formulas in S to have the form of inequalities composed from a linear combination over either V or V' and a constant (e.g., $x' < y' + 1$ is included, but $x' = x + 1$ is excluded). Then, if needed, variables are deprimed (e.g., $x' < y' + 1$ is replaced by $x < y + 1$), and formulas are normalized, such that all terms are moved to the left side (e.g., $x < y + 1$ is replaced by $x - y - 1 < 0$), the subtraction is rewritten as addition, $<$ is rewritten as $>$, and respectively $\leq$ as $\geq$ (e.g., $x - y - 1 < 0$ is replaced by $(-1) \cdot x + y + 1 > 0$).

The entire process of creation of G is exemplified in Fig. 1. Production rules of G are constructed as follows: (1) the production rule for normalized inequalities (denoted INEQ) consists of choices corresponding to distinct types of inequalities in S, (2) the production rule for linear combinations (denoted SUM) consists of choices corresponding to distinct arities of inequalities in S, (3) production rules for variables, coefficients, and constants (denoted respectively VAR, COEF, and CONST) consist of choices corresponding respectively to distinct variables, coefficients, and constants that occur in inequalities in S. Note that the method of creation of G naturally extends to considering disjunctions and nonlinear arithmetic [16].



(a)

① $i > \boxed{x - K} \wedge i > \boxed{K - x} \wedge i > \boxed{y - K} \wedge$
$i > \boxed{K - y} \wedge i > \boxed{x - y} \wedge i > \boxed{y - x} \Longrightarrow \boldsymbol{Inv}(x, y, i, K)$

② $\boldsymbol{Inv}(x, y, i, K) \wedge (y < K \vee y > K) \wedge K' = K \wedge i' = i - 1 \wedge$
$x' = \texttt{ite}\ (x > K, x - 1, \texttt{ite}\ (x < K, x + 1, x)) \wedge$
$y' = \texttt{ite}\ (y > x', y - 1, \texttt{ite}\ (y < x', y + 1, y)) \Longrightarrow \boldsymbol{Inv}(x', y', i', K')$

③ $\boldsymbol{Inv}(x, y, i, K) \wedge (y < K \vee y > K) \wedge i < 0 \Longrightarrow \bot$

(b)

Fig. 2: (a): The worst-case dynamics of program from Fig. 1a; **(b)**: the termination-argument validity check (in the frameboxes – lower bounds $\{\ell_j\}$ for i).

Choices in production rules of grammar G can be further assigned probabilities based on frequencies of certain syntactic features (e.g., frequencies of particular constants or combinations of variables) that belong to the program's symbolic encoding. In the interest of saving space, we do not discuss it here and refer the reader to [16]. The generation of formulas from G is performed recursively by sampling from probability distributions assigned to rules. Note that the choice of distributions affects only the order in which formulas are sampled and does not affect which formulas *can* or *cannot* be sampled in principle (because the grammar is fixed). Thus, without loss of generality, it is sound to assume that all distributions are uniform. In the context of termination analysis, we are interested in formulas produced by rules INEQ and SUM.

4 Proving Termination

We start this section with a motivating example and then proceed to presenting the general-purpose algorithms for proving program termination.

Example 1. The program shown in Fig. 1a terminates. It operates on three integer variables, x, y, and K: in each iteration y gets closer to x, and x gets closer to K. Thus, the total number of values taken by y before it equals K is no bigger than the maximal distance among x, y, and K (in the following, denoted Max). The worst-case dynamics happens when initially $x < y < K$ (shown in Fig. 2a), in other cases the program terminates even faster. To formally prove this, the program could be augmented by a so-called *termination argument*. For this example, it is simply a fresh variable i which is initially assigned Max (or any other value greater than Max) and which gets decremented by one in each iteration. The goal now is to prove that i never gets negative. Fig. 2b shows the encoding of this safety verification task (recall Def. 3). The existence of a

Algorithm 1: LinRank(P): proving termination with linear termination argument

Input: $P = \langle V \cup V', \mathit{Init}, \mathit{Tr}\rangle$ where $\mathit{Tr} = \mathit{Guard} \wedge \mathit{Body}$
Output: $\mathit{res} \in \langle$terminates, unknown$\rangle$

1 $V \leftarrow V \cup \{i\}$; $\quad V' \leftarrow V' \cup \{i'\}$;
2 $\mathit{Tr} \leftarrow \mathit{Tr} \wedge i' = i - 1$; $\quad \mathit{Err} \leftarrow \mathit{Guard} \wedge i < 0$;
3 $G \leftarrow$ getGrammarAndDistributions(Init, Tr);
4 **while** canSample(G) **do**
5 $\quad \mathit{cand} \leftarrow$ sample(G, sum);
6 $\quad G \leftarrow$ adjust(G, cand);
7 $\quad$ **if** $\mathit{Init} \implies i > \mathit{cand}$ **then continue**;
8 $\quad \mathit{Init} \leftarrow \mathit{Init} \wedge i > \mathit{cand}$;
9 $\quad$ **if** isSafe(Init, Tr, Err) **then return** terminates;
10 **return** unknown;

solution to this task guarantees the safety of the augmented program, and thus, the termination of the original program. Most state-of-the-art Horn solvers are able to find a solution immediately. □

The main challenge in preparing the termination-argument validity check is the generation of lower bounds $\{\ell_j\}$ for i in *Init* (e.g., conjunctions of the form $i > \ell_j$ in ① in Fig. 2b). We build on the insight that each ℓ_j could be constructed independently from the others, and then an inequality $i > \ell_j$ could be conjoined with *Init*, thus giving rise to a new safety verification task. For a generation of candidate inequalities, we utilize the algorithm from Sect. 3: all $\{\ell_j\}$ can be sampled from grammar G which is obtained in advance from *Init* and *Tr*.

For example, all six formulas in ① in Fig. 2b: $x-K, K-x, y-K, K-y, x-y$, and $y - x$ belong to the grammar shown in Fig. 1d. Note that for proving termination it is not necessary to have the most precise lower bounds. Intuitively, the larger the initial value of i, the more iterations it will stay positive. Thus, it is sound to try formulas which are not even related to actual lower bounds at all and keep them conjoined with *Init*.

4.1 Synthesizing linear termination arguments

Alg. 1 shows an "*enumerate-and-try*" procedure to search for a linear termination argument that proves termination of a program P. To initialize this search, the algorithm introduces an extra counter variable i and adds it to V (respectively, its primed copy i' gets added to V') (line 1).[3] Then the transition-relation formula *Tr* gets augmented by $i' = i-1$, the decrement of the counter in the loop body. To specify a set of error states, Alg. 1 introduces a formula *Err* (line 2): whenever the loop guard is satisfied and the value of counter i is negative. Alg. 1 then starts searching for large enough lower bounds for i (i.e., a set of constraints over $V \cup \{i\}$ to be added to *Init*), such that no error state is ever reachable.

Before the main loop of our synthesis procedure starts, various formulas are extracted from the symbolic encoding of P and generalized to a formal grammar

[3] Assume that initially set V does not contain i.

(line 3). The grammar is used for an iterative probabilistic sampling of candidate formulas (line 5) that are further added to the validity check of the current termination argument (line 8). In particular, each new constraint over i has the form $i > cand$, where *cand* is produced by the SUM production rule described in Sect. 3. Once *Init* is strengthened by this constraint, a new safety verification condition is compiled and checked (line 9) by an off-the-shelf Horn solver.

As a result of each safety check, either a formula satisfying Def. 3 or a counterexample *cex* witnessing reachability of an error state is generated. Existence of an inductive invariant guarantees that the conjunction of all synthesized lower bounds for i is large enough to prove termination, and thus Alg. 1 converges. Otherwise, if grammar G still contains a formula that has not been considered yet, the synthesis loop iterates.

For the progress of the algorithm, it must keep track of the strength of each new candidate *cand*. That is, *cand* should add more restrictions on i in *Init*. Otherwise, the outcome of the validity check (line 9) would be the same as in the previous iteration. For this reason, Alg. 1 includes an important routine [16]: after each sampled candidate *cand*, it adjusts the probability distributions associated with the grammar, such that *cand* could not be sampled again in the future iterations (line 6). Additionally, it checks (line 7) if a new constraint adds some value over the already accepted constraints. Consequently, our algorithm does not require explicit handing of counterexamples: if in each iteration *Init* gets only stronger then current *cex* is invalidated. While in principle the algorithm could explicitly store *cex* and check its consistency with each new *cand*, however in our experiments it did not lead to significant performance gains.

Theorem 1. *If Alg. 1 returns* TERMINATES *for program P, then P terminates.*

Indeed, the verification condition, which is proven safe in the last iteration of Alg. 1, corresponds to some program P' that differs from P by the presence of variable i. The set of traces of P has a one-to-one correspondence with the set of traces of P', such that each state reachable in P could be extended by a valuation of i to become a reachable state in P'. That is, P terminates iff P' terminates, and P' terminates by construction: i is initially assigned a reasonably large value, monotonically decreases at each iteration, and never goes negative.

We note that the loop in Alg. 1 always executes only a finite number of iterations since G is constructed from the finite number of components, and in each iteration it gets adjusted to avoid re-sampling of the same candidates. However, an off-the-shelf Horn solver that checks validity of each candidate might not converge because the safety verification task is undecidable in general. To mitigate this obstacle, our implementation supports several state-of-the-art solvers and provides a flexibility to specify one to use.

4.2 Synthesizing lexicographic termination arguments

There is a wide class of terminating programs for which no linear termination argument exists. A commonly used approach to handle them is via a search for

Algorithm 2: LexRank(P): proving termination with lexicographic termination argument

Input: $P = \langle V \cup V', Init, Tr \rangle$ where $Tr = Guard \wedge Body$
Output: $res \in \langle$ TERMINATES, UNKNOWN $\rangle$

1 $V \leftarrow V \cup \{i, j\}$; $V' \leftarrow V' \cup \{i', j'\}$;
2 $Err \leftarrow Guard \wedge i < 0$; $jBounds \leftarrow \varnothing$;
3 $G, G', G'' \leftarrow$ getGrammarAndDistributions($Init$, Tr);
4 **while** canSample(G) **or** canSample(G') **or** canSample(G'') **do**
5 **if** nondet() **then**
6 $cand \leftarrow$ sample(G, sum); $G \leftarrow$ adjust(G, $cand$);
7 $Init \leftarrow Init \wedge i > cand$;
8 **if** nondet() **then**
9 $cand \leftarrow$ sample(G', sum); $G' \leftarrow$ adjust(G', $cand$);
10 $Init \leftarrow Init \wedge j > cand$;
11 **if** nondet() **then**
12 $cand \leftarrow$ sample(G'', sum); $G'' \leftarrow$ adjust(G'', $cand$);
13 $jBounds \leftarrow jBounds \cup \{j > cand\}$;
14 $Tr' \leftarrow Tr \wedge ite(j > 0, i' = i \wedge j' = j - 1, i' = i - 1 \wedge \bigwedge_{b \in jBounds} b)$;
15 **if** isSafe($Init$, Tr', Err) **then return** TERMINATES;
16 **return** UNKNOWN;

a so-called lexicographic termination argument that requires introducing two or more extra counters. A SyGuS-based instantiation of such a procedure for two counters is shown in Alg. 2 (more counters could be handled similarly). Alg. 2 has a similar structure to Alg. 1: the initial program gets augmented by counters, formula Err is introduced, lower bounds for counters are iteratively sampled and added to $Init$ and Tr, and the verification condition is checked for safety.

The differences in Alg. 2 are in how it handles two counters i and j, between which an implicit order is fixed. In particular, Err is still expressed over i only, but i gets decremented by one only when j equals zero (line 14). At the same time, j gets updated in each iteration: if it was equal to zero, it gets assigned a value satisfying the conjunction of constraints in an auxiliary set $jBounds$; otherwise it gets decremented by one. Alg. 2 synthesizes $jBounds$ as well as lower bounds for initial conditions over i and j. The sampling proceeds separately from three different grammars (lines 6, 9, and 12), and the samples are used in three different contexts (lines 7, 10, and 13 respectively). Optionally, Alg. 2 could be parametrized by a synthesis strategy that gives interpretations for each of the NONDET() calls (lines 5, 8, and 11 respectively). In the simplest case, each NONDET() call is replaced by $\top$, which means that in each iteration Alg. 2 needs to sample from all three grammars. Alternatively, NONDET() could be replaced by a method to identify only one grammar per iteration to be sampled from.

Theorem 2. *If Alg. 2 returns* TERMINATES *for program P, then P terminates.*

The proof sketch for Th. 2 is similar to the one for Th. 1: an augmented program P' terminates by construction (due to a mapping of values of $\langle i, j \rangle$ into ordinals), and its set of traces has a one-to-one correspondence with the set of traces of P.

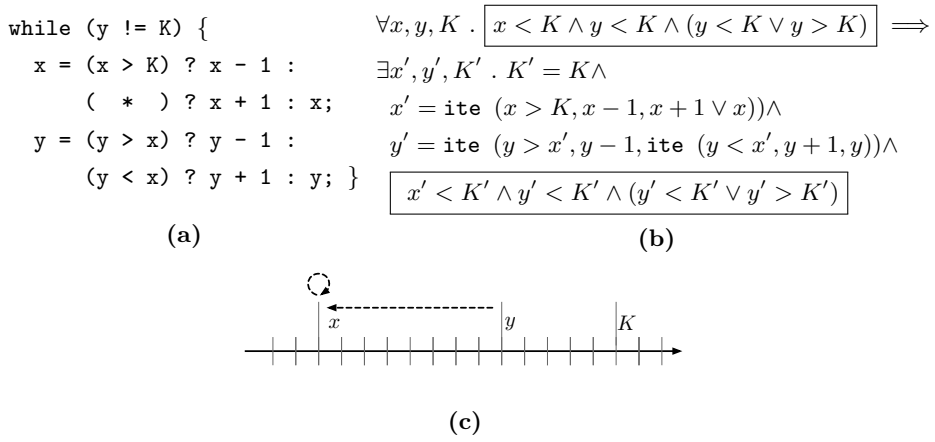


Fig. 3: (a): A variant of program from Fig. 1a; **(b)**: the valid ∀∃-formula for its non-terminating refinement (in frameboxes – refined *Guard*-s); **(c)**: an example of a non-terminating dynamics, when value of `x` (and eventually, `y`) never gets changed.

5 Proving non-termination

In this section, we aim at solving the opposite task to the one in Sect. 4, i.e., we wish to witness infinite program traces and thus, to prove program non-termination. However, in contrast to a traditional search for a single infinite trace, it is often easier to search for groups of infinite traces.

Lemma 1. *Program* $P = \langle V \cup V', Init, Tr \rangle$ *where* $Tr = Guard \wedge Body$ *does not terminate if:*

1. *there exists a state* s*, such that* $s \models Init$ *and* $s \models Guard$*,*
2. *for every state* s*, such that* $s \models Guard$*, there exists a state* s'*, such that* $s, s' \models Tr$ *and* $s' \models Guard$*.*

The lemma distinguishes a class of programs, for which the following holds. First, the loop guard is reachable from the set of initial states. Second, whenever the loop guard is satisfied, there exists a transition to a state in which the loop guard is satisfied again. Therefore, each initial state s, from which the loop guard is reachable, gives rise to at least one infinite trace that starts with s.

Note that for programs with deterministic transition relations (like, e.g., in Fig. 1a), the check of the second condition of Lemma 1 reduces to deciding the satisfiability of a quantifier-free formula since each state can be transitioned to exactly one state. But if the transition relation is non-deterministic, the check reduces to deciding validity of a ∀∃-formula. Although handling quantifiers is in general hard, some recent approaches [15] are particularly tailored to solve this type of queries efficiently.

In practice, the conditions of Lemma 1 are too strict to be fulfilled for an arbitrary program. However, to prove non-termination, it is sufficient to constrain the transition relation as long as it preserves at least one original transition and only then to apply Lemma 1.

Definition 4. *Given programs* $P = \langle V \cup V', \mathit{Init}, \mathit{Tr} \rangle$, *and* $P' = \langle V \cup V', \mathit{Init}, \mathit{Tr}' \rangle$, *we say that* P' *is a* refinement *of* P *if* $\mathit{Tr}' \implies \mathit{Tr}$.

Intuitively, Def. 4 requires P and P' to operate over the same sets of variables and to start from the same initial states. Furthermore, each transition allowed by Tr' is also allowed by Tr. One way to refine P is to restrict $\mathit{Tr} = \mathit{Guard} \wedge \mathit{Body}$ by conjoining either *Guard*, or *Body*, or both with some extra constraints (called *refinement constraints*). In this work, we propose to sample them from our automatically constructed formal grammar (recall Sect. 3).

Example 2. Consider a program shown in Fig. 3a. It differs from the one shown in Fig. 1a by a non-deterministic choice in the second `ite`-statement. That is, y still moves towards x; but x moves towards K only when $x > K$, and otherwise x may always keep the initial value. The formal grammar generated for this program is the same as shown in Fig. 1d, and it contains constraints $x < K$ and $y < K$. Lemma 1 does not apply for the program as is, but it does after refining *Guard* with those constraints. In particular, the $\forall\exists$-formula in Fig. 3b is valid, and a witness to its validity is depicted in Fig. 3c: eventually both x and y become equal and always remain smaller than K. Thus, the program does not terminate. □

5.1 Synthesizing non-terminating refinements

The algorithm for proving program's non-termination is shown in Alg. 3. It starts with a simple satisfiability check (line 1) which filters out programs that never reach the loop body (thus they immediately terminate). Then, the transition relation *Tr* gets strengthened by auxiliary inductive invariants obtained with the help of the initial states *Init* (line 2). The algorithm does not impose any specific requirements on the invariants (and it is sound even for a trivial invariant $\top$) and on a method that detects them. In many cases, auxiliary invariants make the algorithm converge faster. Similar to Algs. 1-2, Alg. 3 splits *Init* and *Tr* to a set of formulas and generalizes them to a grammar. The difference lies in the type of formulas sampled from the grammar (INEQ vs SUM) and their use in the synthesis loop: Alg. 3 treats sampled candidates as *refinement constraints* and attempts to apply Lemma 1 (line 6).

The algorithm maintains a stack of refinement constraints *Refs*. At the first iteration, *Refs* is empty, and thus the algorithm tries to apply Lemma 1 to the original program. For that application, a $\forall\exists$-formula is constructed and checked for validity. Intuitively the formula expresses the ability of *Body* to transition each state which satisfies *Guard* to a state which satisfies *Guard* as well. If the validity of $\forall\exists$-formula is proven, the algorithm converges (line 7). Otherwise, a

Algorithm 3: NONTERMREF(P): proving non-termination

Input: $P = \langle V \cup V', \mathit{Init}, \mathit{Tr} \rangle$ where $\mathit{Tr} = \mathit{Guard} \wedge \mathit{Body}$
Output: $\mathit{res} \in \langle$TERMINATES, DOES NOT TERMINATE, UNKNOWN$\rangle$

1 **if** $\mathit{Init}(V) \wedge \mathit{Guard}(V) \implies \bot$ **then return** TERMINATES;
2 $\mathit{Tr} \leftarrow \mathit{Tr} \wedge$ GETINVS($\mathit{Init}, \mathit{Tr}$);
3 $G \leftarrow$ GETGRAMMARANDDISTRIBUTIONS($\mathit{Init}, \mathit{Tr}$);
4 $\mathit{Refs} \leftarrow \varnothing$; $\mathit{Gramms} \leftarrow \varnothing$; Gramms.PUSH(G);
5 **while** *true* **do**
6 **if** $\forall V \,.\, \mathit{Guard}(V) \wedge \bigwedge_{r \in \mathit{Refs}} r(V) \implies \exists V' \,.\, \mathit{Body}(V, V') \wedge \mathit{Guard}(V') \wedge \bigwedge_{r \in \mathit{Refs}} r(V')$ **then**
7 **return** DOES NOT TERMINATE;
8 $\mathit{cand} \leftarrow \top$;
9 **while** $\mathit{Guard}(V) \wedge \bigwedge_{r \in \mathit{Refs}} r(V) \implies \mathit{cand}(V)$ **or** $\mathit{Init}(V) \wedge \mathit{Guard}(V) \wedge \mathit{cand}(V) \wedge \bigwedge_{r \in \mathit{Refs}} r(V) \implies \bot$ **do**
10 **if** $\mathit{Refs} = \varnothing$ **and** $\neg$CANSAMPLE(G) **then return** UNKNOWN;
11 **if** $\mathit{Refs} \neq \varnothing$ **and** $\neg$CANSAMPLE(G) **then**
12 Refs.POP();
13 Gramms.POP();
14 $\mathit{cand} \leftarrow \top$; $G \leftarrow \mathit{Gramms}$.TOP();
15 **continue**;
16 $\mathit{cand} \leftarrow$ SAMPLE(G, INEQ);
17 $G \leftarrow$ ADJUST(G, cand);
18 Refs.PUSH(cand);
19 Gramms.PUSH(G);

refinement of P needs to be guessed. Thus, the algorithm samples a new formula (line 16) using the production rule INEQ, which is described in Sect. 3, pushes it to *Refs*, and iterates. Note that G permits formulas over V only (i.e., to restrict *Guard*), however, in principle it can be extended for sampling formulas over $V \cup V'$ (thus, to restrict *Body* as well).

For the progress of the algorithm, it must keep track of how each new candidate *cand* corresponds to constraints already belonging to *Refs*. That is, *cand* should not be implied by $\mathit{Guard} \wedge \bigwedge_{r \in \mathit{Refs}} r$ since otherwise the $\forall\exists$-formula in the next iteration would not change. Also, *cand* should not over-constrain the loop guard, and thus it is important to check that after adding *cand* to constraints from *Guard* and *Refs*, the loop guard is still reachable from the initial states. Both these checks are performed before the sampling (line 9). After the sampling, necessary adjustments on the probability distributions, assigned to the production rules of the grammar [16], are applied to ensure the same refinement candidates are not re-sampled again (line 17).

Because by construction G cannot generate conjunctions of constraints, the algorithm handles conjunctions externally. It is useful in case when a single constraint is not enough for application of Lemma 1, and it should be strengthened by another constraint. On the other hand, it also might be needed to withdraw some sampled candidates before converging. For this reason, Alg. 3 maintains

a stack *Gramms* of grammars and handles it synchronously with stack *Refs* (lines 12-14 and 18-19). When all candidates from a grammar were considered and were unsuccessful, the algorithm pops the latest candidate from *Refs* and rolls back to the grammar used in the previous iteration. Additionally, a maximum size of *Refs* can be specified to avoid considering too deep refinements.

Theorem 3. *If Alg. 3 returns* DOES NOT TERMINATE *for program P, then P does not terminate.*

Indeed, constraints that belong to *Refs* in the last iteration of the algorithm give rise to a refinement P' of P, such that $P' = \langle V \cup V', \mathit{Init}, \mathit{Tr} \wedge \bigwedge_{r \in \mathit{Refs}} r \rangle$. The satisfiability check (line 9) and the validity check (line 6) passed, which correspond to the conditions of Lemma 1. Thus, P' does not terminate, and consequently it has an infinite trace. Finally, since P' refines P then all traces (including infinite ones) of P' belong to P, and P does not terminate as well.

5.2 Integrating algorithms together

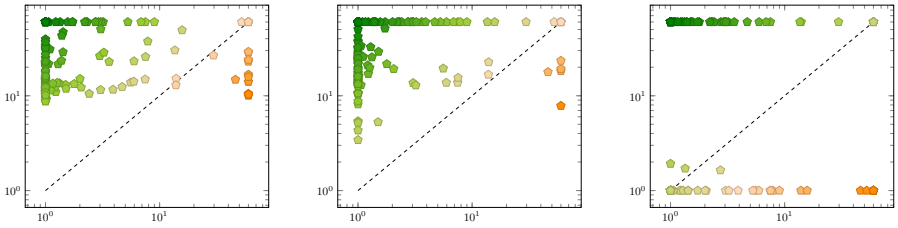
With a few exceptions [30,40], existing algorithms address either the task of proving, or the task of disproving termination. The goal of this paper is to show that both tasks benefit from syntax-guided techniques. While an algorithmic integration of several orthogonal techniques is itself a challenging problem, it is not the focus of our paper. Still, we use a straightforward idea here. Since each presented algorithm has one big loop, an iteration of Alg. 1 could be followed by an iteration of Alg. 2 and in turn, by an iteration of Alg. 3 (i.e., in a lockstep fashion). A positive result obtained by any algorithm forces all remaining algorithms to terminate. Based on our experiments, provided in detail in Sect. 6, the majority of benchmarks were proven either terminating or non-terminating by one of the algorithms within seconds. This justifies why the lockstep execution of all algorithms in practice would not bring a significant overhead.

6 Evaluation

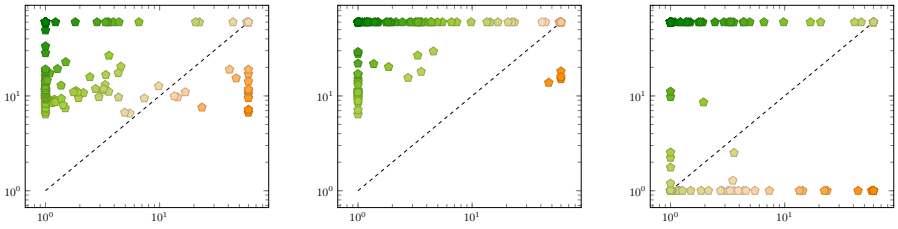
We have implemented algorithms for proving termination and non-termination in a tool called FreqTerm[4]. It is developed on top of FreqHorn [16], uses it for Horn solving, and supports other Horn solvers, Spacer3 [26] and μZ [24], as well. To solve $\forall\exists$-formulas, FreqTerm uses the AE-VAL tool [15]. All the symbolic reasoning in the end is performed by the Z3 SMT solver [11].

FreqTerm takes as input a program encoded as a system of linear constrained Horn clauses (CHC). It supports any programming language, as long as a translator from it to CHCs exists. For encoding benchmarks to CHCs, we used SeaHorn v.0.1.0-rc3. To the best of our knowledge, FreqTerm is the only (non)-termination prover that supports a selection of Horn solvers in the backend. This allows the prover to leverage advancements in Horn solving easily.

[4] The source code of the tool is publicly available at `https://goo.gl/HecBWc`.



(a) terminating examples (171)



(b) non-terminating examples (176)

Fig. 4: FreqTerm vs respectively Ultimate Automizer, AProVE, and HipTNT+.

We have compared FreqTerm against AProVE rev. c181f40 [18], Ultimate Automizer v.0.1.23 [22], and HipTNT+ v.1.0 [30]. The rest of the section summarizes three sets of experiments. Sect. 6.1 and 6.2 discuss the comparison on small but tricky programs, respectively terminating and non-terminating, which shows that our approach is applicable to a wide range of conceptually challenging problems. In Sect. 6.3, we target several large-scale benchmarks and show that FreqTerm is capable of significant pushing the boundaries of termination and non-termination proving. In total, we considered 856 benchmarks of various size and complexity. All experiments were conducted on a Linux SMP machine, Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 56 CPUs, 377 GB RAM.

6.1 Performance on terminating benchmarks

We considered **171** terminating programs[5] from the Termination category of SVCOMP and programs crafted by ourselves. Altogether, four tools in our experiment were able to prove termination of 168 of them within a timeout of 60 seconds and left only three programs without a verdict. AProVE verified 76 benchmarks, HipTNT+ 90 (including 3 that no other tool solved), Ultimate Automizer 105 (including 4 that no other tool solved). FreqTerm, implementing Algs. 1-2 and relying on different solvers verified in total **155** (including **30** that no other tool solved). In particular, Alg. 1 instantiated with Spacer3, proved termination of 88 programs, with μZ 79, and with FreqHorn 80. Alg. 2 instantiated with Spacer3, proved termination of 92 programs, with μZ 109, and with FreqHorn 74.

[5] These benchmarks are available at `https://goo.gl/MPimXE`.

A scatterplot with logarithmic scale on the axes in Fig. 4 (**a**) shows comparisons of best running times of FreqTerm vs the running times of competing tools. Each point in a plot represents a pair of the FreqTerm run (x-axis) and the competing tool run (y-axis). Intuitively, green points represent cases when FreqTerm outperforms the competitor. On average, for programs solved by both FreqTerm and Ultimate Automizer, FreqTerm is 29 times faster (speedup calculated as a ratio of geometric means of the corresponding runs). In a similar setting, FreqTerm is 32 times faster than AProVE. However, FreqTerm is 2 times slower than HipTNT+. The evaluation further revealed (in Sect. 6.3) that the latter tool is efficient only on small programs (around 10 lines of code each), and for large-scale benchmarks it exceeds the timeout.

6.2 Performance on non-terminating benchmarks

We considered **176** terminating programs[6] from the Termination category of SVCOMP and programs crafted by ourselves. Altogether, four tools proved non-termination of 172 of them: AProVE 35, HipTNT+ 92, Ultimate Automizer 123, and Alg. 3 implemented in FreqTerm **152**. Additionally, we evaluated the effect of $\forall\exists$-solving in FreqTerm. For that reason, we implemented a version of Alg. 3 in which non-termination is reduced to safety, but the conceptual SyGuS-based refinement generator remained the same. This implementation used Spacer3 for proving that the candidate refinement *can never* exit the loop. Among 176 benchmarks, such routine solved only 105, which is 30% fewer than Alg. 3. However, it managed to verify 8 benchmarks that Alg. 3 could not verify (we believe, because Spacer3 was able to add an auxiliary inductive invariant).

Logarithmic scatterplot in Fig. 4 (**b**) shows comparisons of FreqTerm vs the running times of competing tools. On average, FreqTerm is 41 times faster than Ultimate Automizer, 73 times faster than AProVE, and exhibits roughly similar runtimes to HipTNT+ (again, here we considered only programs solved by both tools). Based on these experiments, we conclude that currently FreqTerm is more effective and more efficient at synthesizing non-terminating program refinements than at synthesizing terminating arguments.

6.3 Large-scale benchmarks

We considered some large-scale benchmarks for evaluation arising from Event-Condition-Action (ECA) systems that describe reactive behavior [1]. We considered various modifications of five challenging ECAs[7]. Each ECA consists of one large loop, where each iteration reads an input and modifies its internal state. If an unexpected input is read, the ECA terminates.

In our first case study, we aimed to prove non-termination of the given ECAs, i.e., that for any reachable internal state there exists an input value that would

[6] These benchmarks are available at `https://goo.gl/bZbuA2`.
[7] These benchmarks are available at `https://goo.gl/7mc2Ww`.

Table 1: FreqTerm vs Ultimate Automizer on non-terminating ECAs (302).

Benchmarks			FreqTerm		Ultimate Automizer	
class	# of tasks	avg # of LoC	# solved	avg time	# solved	avg time
1 & 2	122	500	122	5 sec	3	27 min
3	60	1600	60	56 sec	0	∞
4	60	4700	60	9 min	6	82 min
5	60	10000	59	52 min	0	∞

Table 2: FreqTerm vs Ultimate Automizer on terminating ECAs (207).

Benchmarks			FreqTerm		Ultimate Automizer	
class	# of tasks	avg # of LoC	# solved	avg time	# solved	avg time
1 & 2	97	500	97	8 sec	96	73 sec
3	40	1600	40	3 min	12	56 min
4	35	4700	35	10 min	27	19 min
5	35	10000	34	65 min	19	99 min

keep the ECA alive. The main challenge appeared to be in the size of benchmarks (up to 10000 lines of C code per loop) and reliance on an auxiliary inductive invariant. With the extra support of Spacer3 to provide the invariant, FreqTerm was able to prove non-termination of a wide range of programs. Among all the competing tools, only Ultimate Automizer was able to handle these benchmarks, but it verified only a small fraction of them within a 2 hours timeout. In contrast, FreqTerm solved 301 out of 302 tasks and outperformed Ultimate Automizer by up to several orders of magnitude (i.e., from seconds to hours). Table 1 contains a brief summary of our experimental evaluation.[8]

In our second case study, we instrumented the ECAs by adding extra conditions to the loop guards, thus imposing an implicit upper bound on the number of loop iterations, and applied tools to prove termination[9] (shown in Table 2). Again, only Ultimate Automizer was able to compete with FreqTerm, and interestingly it was more successful here than in the first case study. Encouragingly, FreqTerm solved all but one instance and was consistently faster.

7 Related Work

Proving termination A wide range of state-of-the-art methods are based on iterative reasoning driven by counterexamples [4,9,19,21,27,36,5,10,29,23] whose goal is to show that transitions cannot be executed forever. These approaches typically combine termination arguments, proven independently, but none of them leverages the syntax of programs during the analysis.

A minor range of tools of termination analyzers are based on various types of learning. In particular, [40] discovers a terminating argument from attempts

[8] To calculate average timings, we excluded cases when the tool exceeded timeout.

[9] The task of adding interesting guards appeared to be non-trivial, so we were able to instrument only a part of all non-terminating benchmarks.

to prove that no program state is terminating; [34] exploits information derived from tests, [38] guesses and checks transition invariants (over-approximations to the reachable transitive closure of the transition relation) from libraries of templates. The closest to our approach, [31] guesses and checks transition invariants using loop guards and branch conditions. In contrast, our algorithms guess lower bounds for auxiliary program counters and extensively use all available source code for guessing candidates.

Proving non-termination Traditional algorithms, e.g. [20,6,8,3,22], are based on a search for lasso-shaped traces and a discovery of *recurrence sets*, i.e., states that are visited infinitely often. For instance, [32] searches for a geometric series in lasso-shaped traces. Our algorithm discovers *existential* recurrence sets and does not deal with traces at all: it handles their abstraction via a $\forall\exists$-formula.

A reduction to safety attracts significant attention here as well. In particular, [41] relies only on invariant generation to show that the loop guard is also satisfied, [19] infers weakest preconditions over inputs, under which program is non-terminating; and [7,28] iteratively eliminate terminating traces through a loop by adding extra assumptions. In contrast, our approach does not reduce to safety, and thus does not necessarily require invariants. However, we observed that if provided, in practice they often accelerate our verification process.

Syntax-Guided Synthesis SyGuS [2] is applied to various tasks related to program synthesis, e.g., [25,17,35,13,33,42]. However, the formal grammar in those applications is typically given or constructed from user-provided examples. To the best of our knowledge, the only application of SyGuS to automatic program analysis was proposed by [16,14], and it inspired our approach. Originally, the formal grammar, constructed from the verification condition, was iteratively used to guess and check only inductive invariants. In this paper, we showed that a similar reasoning is practical and easily transferable across applications.

8 Conclusion

We have presented new algorithms for synthesis of termination arguments and non-terminating program refinements. Driven by SyGuS, they iteratively generate candidate formulas which tend to follow syntactic patterns obtained from the source code. By construction, the number of possible candidates is always finite, thus the search space is always relatively small. The algorithms rely on recent advances in constraint solving, they do not depend on a particular backend engine, and thus performance of checking validity of a candidate can be improved by advancements in solvers. Our implementation FreqTerm is evaluated on a wide range of terminating and non-terminating benchmarks. It is competitive with state-of-the-art and it significantly outperforms other tools when proving non-termination of large-scale Event-Condition-Action systems.

In future work, it would be interesting to investigate synergetic ways of integrating the proposed algorithms together, as well as exploiting strengths of different backend Horn solvers for different verification tasks.

References

1. E. E. Almeida, J. E. Luntz, and D. M. Tilbury. Event-Condition-Action Systems for Reconfigurable Logic Control. *IEEE Trans. Automation Science and Engineering*, 4(2):167–181, 2007.
2. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
3. A. Bakhirkin and N. Piterman. Finding Recurrent Sets with Backward Analysis and Trace Partitioning. In *TACAS*, volume 9636 of *LNCS*, pages 17–35. Springer, 2016.
4. I. Balaban, A. Pnueli, and L. D. Zuck. Ranking Abstraction as Companion to Predicate Abstraction. In *FORTE*, volume 3731 of *LNCS*, pages 1–12. Springer, 2005.
5. M. Brockschmidt, B. Cook, and C. Fuhs. Better Termination Proving through Cooperation. In *CAV*, volume 8044 of *LNCS*, pages 413–429. Springer, 2013.
6. M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *FoVeOOS*, volume 7421 of *LNCS*, pages 123–141. Springer, 2011.
7. H. Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Proving nontermination via safety. In *TACAS*, volume 8413 of *LNCS*, pages 156–171. Springer, 2014.
8. B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Disproving termination with overapproximation. In *FMCAD*, pages 67–74. IEEE, 2014.
9. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426. ACM, 2006.
10. B. Cook, A. See, and F. Zuleger. Ramsey vs. Lexicographic Termination Proving. In *TACAS*, volume 7795 of *LNCS*, pages 47–61. Springer, 2013.
11. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
12. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
13. G. Fedyukovich, M. B. S. Ahmad, and R. Bodík. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*, pages 572–585. ACM, 2017.
14. G. Fedyukovich and R. Bodík. Accelerating Syntax-Guided Invariant Synthesis. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 251–269. Springer, 2018.
15. G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated discovery of simulation between programs. In *LPAR*, volume 9450 of *LNCS*, pages 606–621. Springer, 2015.
16. G. Fedyukovich, S. Kaufman, and R. Bodík. Sampling Invariants from Frequency Distributions. In *FMCAD*, pages 100–107. IEEE, 2017.
17. J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. CodeHint: dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663. ACM, 2014.
18. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *IJCAR*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.
19. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.

20. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL*, pages 147–158. ACM, 2008.
21. W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *SAS*, volume 6337 of *LNCS*, pages 304–319. Springer, 2010.
22. M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata - (Competition Contribution). In *TACAS, Part II*, volume 10206 of *LNCS*, pages 394–398. Springer, 2017.
23. M. Heizmann, J. Hoenicke, and A. Podelski. Termination Analysis by Learning Terminating Programs. In *CAV*, volume 8559 of *LNCS*, pages 797–813. Springer, 2014.
24. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.
25. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224. ACM, 2010.
26. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014. `https://bitbucket.org/spacer/code/branch/spacer3`.
27. D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV*, volume 6174 of *LNCS*, pages 89–103. Springer, 2010.
28. D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving Non-termination Using Max-SMT. In *CAV*, volume 8559 of *LNCS*, pages 779–796. Springer, 2014.
29. D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD*, pages 218–225. IEEE, 2013.
30. T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *PLDI*, pages 489–498. ACM, 2015.
31. W. Lee, B. Wang, and K. Yi. Termination Analysis with Algorithmic Learning. In *CAV*, volume 7358 of *LNCS*, pages 88–104. Springer, 2012.
32. J. Leike and M. Heizmann. Geometric Nontermination Arguments. In *TACAS, Part II*, volume 10806 of *LNCS*, pages 266–283. Springer, 2018.
33. A. Miltner, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic. Synthesizing bijective lenses. *PACMPL*, 2(POPL):1:1–1:30, 2018.
34. A. V. Nori and R. Sharma. Termination proofs from tests. In *ESEC/FSE*, pages 246–256. ACM, 2013.
35. P. Panchekha and E. Torlak. Automated Reasoning for Web Page Layout. In *OOPSLA*, pages 181–194. ACM, 2016.
36. A. Podelski and A. Rybalchenko. Transition Invariants and Transition Predicate Abstraction for Program Termination. In *TACAS*, volume 6605 of *LNCS*, pages 3–10. Springer, 2011.
37. P. Schrammel and D. Kroening. 2LS for Program Analysis - (Competition Contribution). In *TACAS*, volume 9636 of *LNCS*, pages 905–907. Springer, 2016.
38. A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *TACAS*, volume 6605 of *LNCS*, pages 81–95. Springer, 2011.
39. A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.
40. C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing Ranking Functions from Bits and Pieces. In *TACAS*, volume 9636 of *LNCS*, pages 54–70. Springer, 2016.

41. H. Velroyen and P. Rümmer. Non-termination Checking for Imperative Programs. In *TAP*, volume 4966 of *LNCS*, pages 154–170. Springer, 2008.
42. X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.