DynamiTe: Dynamic Termination and Non-termination Proofs

TON CHANH LE, Stevens Institute of Technology, USA
TIMOS ANTONOPOULOS, Yale University, USA
PARISA FATHOLOLUMI, Stevens Institute of Technology, USA
ERIC KOSKINEN, Stevens Institute of Technology, USA
THANHVU NGUYEN, University of Nebraska, Lincoln, USA

There is growing interest in termination reasoning for nonlinear programs and, meanwhile, recent dynamic strategies have shown they are able to infer invariants for such challenging programs. These advances led us to hypothesize that perhaps such dynamic strategies for nonlinear invariants could be adapted to learn recurrent sets (for non-termination) and/or ranking functions (for termination).

In this paper, we exploit dynamic analysis and draw termination and non-termination as well as static and dynamic strategies closer together in order to tackle nonlinear programs. For termination, our algorithm infers ranking functions from concrete transitive closures, and, for non-termination, the algorithm iteratively collects executions and dynamically learns conditions to refine recurrent sets. Finally, we describe an integrated algorithm that allows these algorithms to mutually inform each other, taking counterexamples from a failed validation in one endeavor and crossing both the static/dynamic and termination/non-termination lines, to create new execution samples for the other one.

We have implemented these algorithms in a new tool called DynamiTe. For nonlinear programs, there are currently no SV-COMP termination benchmarks so we created new sets of 38 terminating and 39 non-terminating programs. Our empirical evaluation shows that we can effectively guess (and sometimes even validate) ranking functions and recurrent sets for programs with nonlinear behaviors. Furthermore, we show that counterexamples from one failed validation can be used to generate executions for a dynamic analysis of the opposite property. Although we are focused on nonlinear programs, as a point of comparison, we compare DynamiTe's performance on linear programs with that of the state-of-the-art tool, Ultimate. Although DynamiTe is an order of magnitude slower it is nonetheless somewhat competitive and sometimes finds ranking functions where Ultimate was unable to. Ultimate cannot, however, handle the nonlinear programs in our new benchmark suite.

Supplemental Materials. Our materials are available at https://github.com/letonchanh/dynamite.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Dynamic analysis**; **Formal software verification**; • **Theory of computation** → **Invariants**; **Program analysis**.

Additional Key Words and Phrases: dynamic analysis, termination, non-termination

ACM Reference Format:
Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: Dynamic Termination and Non-termination Proofs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 189 (November 2020), 30 pages. https://doi.org/10.1145/3428257

Authors' addresses: Ton Chanh Le, Stevens Institute of Technology, USA, letonchanh@gmail.com; Timos Antonopoulos, Yale University, USA, timos.antonopoulos@yale.edu; Parisa Fathololumi, Stevens Institute of Technology, USA, pfatholo@stevens.edu; Eric Koskinen, Stevens Institute of Technology, USA, eric.koskinen@stevens.edu; ThanhVu Nguyen, University of Nebraska, Lincoln, USA, tnguyen@cse.unl.edu.

2475-1421/2020/11-ART189
https://doi.org/10.1145/3428257

1 INTRODUCTION

Termination continues to be an important theoretical property that is of practical interest. In recent years, there has been a proliferation of termination and non-termination verification tools, including T2 [Brockschmidt 2020], Ultimate [Ultimate 2020], CPAchecker [Beyer and Keremoglu 2011], AProVE [Giesl et al. 2014], FuncTion [Urban 2015], SeaHorn [Gurfinkel et al. 2015], HipTNT+ [Le et al. 2015], and many others (see the Termination track of SV-COMP [Beyer 2020]). These tools are very effective at proving termination and non-termination, especially for programs with linear arithmetic assignments and loop guards [Podelski and Rybalchenko 2004a].

Meanwhile, researchers are increasingly using techniques based on dynamic execution, to bolster static verification. Static analysis explores all possible program paths but typically has one or more shortcomings: expressivity sacrifices, false positives, simpler invariants or restrictions on kinds of target programs. Dynamic analysis focuses on exploring only a few program executions and, as such, also has its own shortcomings: it is only correct with respect to the explored paths. However, by (initially) sacrificing soundness, dynamic analyses support more expressive invariants and scale well to large and complex programs (see, *e.g.*, [O'Hearn 2020]), often being effective even when the source code is not available. Moreover, false positives for the existence of a bug are not a concern: if any of the explored paths leads to an error, then it is a real error. More recent dynamic analyses have taken a "data-driven" or machine learning approach, *i.e.*, learning based on training data. DIG [Nguyen et al. 2014a] and [Yao et al. 2020] use this form of dynamic analysis to learn invariants of nonlinear programs. Moreover, other recent works combine both dynamic and static analysis techniques in an iterative loop, sometimes for the purpose of termination reasoning [Nori and Sharma 2013]. The dynamic analysis component is used to "guess" some candidate results and the static analysis one is used to verify them. The results of the checker, *e.g.*, counterexamples showing invalidity of the candidate results, are then used to help the dynamic analysis to infer better results.

Landscape. In the context of these recent advances that use dynamic support to learn invariants of nonlinear programs, a natural question is whether they can be used or adapted to empower termination and non-termination reasoning for such challenging programs. We explored in this direction, asking first whether non-termination reasoning can be built from dynamic approaches for nonlinear invariants, and then a similar question for termination. While these endeavors at first seem independent and could potentially be parallelized, we finally explored smarter approaches, where counterexamples from a failed termination proof could be used to generate executions for dynamically learning non-termination, and vice-versa.

Learning ranking functions and recurrent sets. In this paper, we present algorithms that mix static and dynamic strategies in order to prove termination and non-termination of nonlinear programs. Overall, our strategy begins by sampling terminating and potentially non-terminating executions from an instrumented program, with truncated divergence. We first present a novel termination algorithm ProveT, which dynamically samples concrete states from the transitive closure of the loop bodies of sampled traces, fits them to a ranking function template, and uses an SMT solver to generate unknown coefficients in the template. We then attempt to validate these candidate ranking functions (via reachability) and use any possible counterexamples to dynamically generate more sample traces. A second ProveNT algorithm is used for non-termination, and iteratively refines a recurrent set condition, by executing the program and using samples to learn conditions for re-entering versus exiting the loop. We next widen the interfaces of these procedures and show that these algorithms can inform each other. A failed attempt of ProveT to prove termination yields a potentially infinite path, which we use to generate new executions to input to ProveNT, and vice-versa.

We have implemented these algorithms in a new tool called DynamiTe for dynamically proving termination and non-termination. DynamiTe employs the power of many disparate tools: the dynamic invariant inference tool DIG [Nguyen et al. 2014a], the symbolic execution tool CIVL [Siegel et al. 2015], the reachability analysis tools CPAchecker [Beyer and Keremoglu 2011] and Ultimate [Ultimate 2020] (without termination reasoning), and the SMT solvers Z3 [de Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011].

Our main goal is to prove termination and non-termination of nonlinear programs, a domain for which many existing tools struggle. We evaluate DynamiTe on two existing benchmarks consisting of nonlinear programs (`polyrank` [Bradley et al. 2005b], which has nondeterministic terminating programs, and ANANT [Cook et al. 2014], which has nonlinear non-terminating programs) and also create and evaluate DynamiTe on more challenging nonlinear terminating and non-terminating benchmarks (by adapting the SV-COMP benchmark `nla-digbench` [2020] consisting of programs having nonlinear polynomial invariants). We show that DynamiTe is able to learn rich ranking functions and recurrent sets for these programs that cannot be handled by tools like Ultimate. We also show that the integrated algorithm can choose the right algorithm, ProveT or ProveNT or use the counterexample from a failed proof to assist the other.

Although nonlinear programs were our focus, we also compared our algorithms on linear programs against a state-of-the-art termination prover: Ultimate [Ultimate 2020]. We used the 62 benchmarks from the category `termination-crafted-lit` in SV-COMP 2020 and show that, although DynamiTe is typically an order of magnitude slower (owing to the need for program execution), it is nonetheless competitive with Ultimate, which is a much more mature tool. Also, in some cases DynamiTe is faster than Ultimate and in other cases DynamiTe is able to learn ranking functions, where Ultimate is unable to do so.

Contributions. In summary, we present:

(1) A novel termination algorithm, based on sampling concrete states from the transitive closure and fitting to ranking function templates with SMT. (Section 4)
(2) A novel non-termination algorithm, based on refining recurrent sets with conditions learned from dynamic executions of the program (Section 5).
(3) An integrated algorithm for termination and non-termination, that uses counterexamples from one failed static validation attempt to generate executions for dynamic analysis of the other. (Section 6)
(4) A new publicly available tool called DynamiTe [2020] for termination/non-termination of nonlinear programs. (Section 7)
(5) Two new benchmark suites for SV-COMP: one for termination of nonlinear programs, and one for non-termination of nonlinear programs. (Section 8)
(6) An experimental evaluation, demonstrating that DynamiTe is able to learn and sometimes validate ranking functions and recurrent sets for nonlinear programs. (Section 8)

Related work. Similarly to DynamiTe, several existing works support programs with nonlinear properties. Nori and Sharma [2013] show program termination by dynamically inferring (nonlinear, disjunctive) loop bounds from program execution traces. Bradley et al. [2005b,c] use finite difference trees to statically infer lexicographic polynomial ranking functions to prove termination of nonlinear programs. For non-termination analysis, Cook et al. [2014] uses abstract interpretation to over-approximate the nonlinear programs and infer linear recurrent sets to prove program termination. Frohn and Giesl [2019] uses recurrence solvers to generate loop-free transitions so that paths to non-terminating loops can be discovered. In contrast, DynamiTe integrates existing dynamic invariant generation and static verification for dynamically analyze both program termination and

non-termination from their concrete snapshots, and it can analyze many other programs that are not by these works (details in Sections 8 and 9). Also, DynamiTe can analyze termination properties for nondeterministic programs (similarly to [Bradley et al. 2005b,c; Nori and Sharma 2013], but it currently cannot handle non-termination for nondeterministic programs (see Sections 3, 5 and 8.3 for additional discussion and evaluation). In Section 9 we discuss these works and other general termination and non-termination techniques in more details.

2 OVERVIEW

Consider the program to the right. In the loop body of this program, a is incremented by 1, and the loop terminates when the square of a+1 is no longer below n. While the termination of this program is intuitively obvious, existing tools (*e.g.* ULTIMATE, AProVE, SEAHORN) are unable to prove it to be terminating because it requires reasoning about the nonlinear behavior of program variables. Proving termination here involves discovering a ranking function that pertains to variables occurring in a quadratic inequality in the loop condition. As we discuss below, examples like this and more complicated ones with polynomial expressions, foil many existing techniques that are based on linear arithmetic constraints. One major impediment appears to be the lack of static reasoning techniques for programs with such nonlinear behaviors. Some static works have shown static termination reasoning for certain classes of nonlinear programs (*e.g.* "NAW loops" [Babić et al. 2007], loops with finite difference trees [Bradley et al. 2005b,c]), and other works provide static resource bounds [Gulwani 2009; Gulwani et al. 2009; Hoffmann et al. 2011; Hoffmann and Hofmann 2010a,b], but still lack general techniques for termination and non-termination of these challenging programs.

```
int a = 0, n = *;
while ((a+1) * (a+1) <= n):
  a = a + 1
```

Meanwhile, in recent years, a number of works have showed that dynamic analysis can be used to learn rich, nonlinear *invariants*. Nguyen et al. [2012] showed that we can use dynamic analysis to learn expressive (nonlinear) polynomial invariants from a small set of program execution traces. Subsequent works [Nguyen et al. 2017a,b] propose iterative loop algorithms to generate candidate invariants from traces and use symbolic execution to refute spurious results and generate valid counterexamples, which are then used to improve the invariant generation process. Many other works, e.g., [Nguyen et al. 2014b; Sharma et al. 2013], combine inferring nonlinear invariants with static checking. Recently, Yao et al. [2020] proposes using neural networks to learn nonlinear invariants.

2.1 Learning Ranking Functions and Recurrent Sets

In this paper, we explore the question of whether techniques for nonlinear invariants can be extended to reasoning about both termination and non-termination and do so in an integrated way. We begin by adapting earlier dynamic analysis works, to provide a new route for learning ranking functions and recurrent sets. To highlight and illustrate the key features of our work, we will use the following pair of slightly more complicated examples. The following two programs are similar, but one terminates and the other does not:

Termination

```
1 int s = 1, t = 1, k, c = 1
2 while (t*t - 4*s + 2*t + 1 + c <= k):
3   t = t + 2
4   s = s + t
5   c = c + t
```

Non-termination

```
1 int s = 1, t = 1, c = 1
2 while (t*t - 4*s + 2*t + 1 + c >= 0):
3   t = t + 2
4   s = s + t
5   c = c + t
```

These examples are based on `sqrt1.c` from the SV-COMP benchmark `nla-digbench` [2020]. (We discuss how we adapted it in Section 8.) Both of these programs involve loop conditions that

are nonlinear, given the quadratic term `t*t` in them. For illustration purposes, the body of both programs is the same, and it is not difficult to see by induction that the subexpression `t*t - 4*s + 2*t + 1` in both loop conditions is always equal to 0.

In the program on the left, the loop condition is essentially equivalent to `c <= k` for all reachable states in the program. Given that `k` is initialized to a nondeterministic value and unchanged and `t` is always positive, and thus `c` is always increasing up to `k` and the execution will eventually exit the loop. This reasoning is usually captured with ranking functions: a map from every state to an ordered element where a transition in the program between states implies a transition from an element to a strictly smaller in that order element. Moreover, such an order is chosen to not be forever decreasing, and thus there cannot be an infinite sequence of states with valid program transitions between them. In this case such a function would be the one that maps every state to the value of `k-c`. It is the aim of our algorithm to synthesize this function and we describe in the following paragraphs how this is achieved.

In the program on the right, the loop condition is now essentially `c >= 0` for all reachable states of the program, which holds trivially since `c` is initialized to `1` and always increasing. To show non-termination, a recurrent set is usually constructed: by abstracting the loop body in a relation $\mathcal{T}_{\text{loop}}$, a set X of states is collected such that for any state s in that set, and any other state s' we can transition to from s, it is the case that s' is also in X. The existence of such a set that contains reachable states implies that the program is non-terminating at least in some cases.

The above examples reiterate the point that, even for simple nonlinear programs, the necessary reasoning evades existing termination and non-termination tools, many of which are based on linear arithmetic constraint solving [Gurfinkel et al. 2015; Podelski and Rybalchenko 2004a].

Dynamic snapshots for termination/non-termination. In this paper, we work in a direction that is based on learning from dynamically generated program traces, for guessing (and possibly validating) ranking functions and recurrent sets for examples such as those above. To this end, we begin by describing a simple mechanism for collecting traces which may or may not terminate. Tools for dynamic analysis typically take "snapshots" of the state of the program to record trace information. These snapshots may record the values of some/all variables, as well as the program location and there are many techniques for injecting snapshots.

In the case of termination, though, there is the additional challenge of recording snapshots of a loop that may run forever. Our solution is to break the loop so that potentially infinite executions are truncated, and then we can learn about those prefixes and try to characterize the ones that would have terminated versus those that would not have. We can truncate loops by introducing a counter. Counter instrumentation is common in other kinds of static analysis [Gulwani 2009; Gulwani et al. 2009; Hoffmann et al. 2011; Hoffmann and Hofmann 2010a,b], but here we use it to truncate executions.

Using the above non-termination example, our transformation generates the program on the right, where changes are indicated in the gray boxed regions. Technically, these changes are made to every loop of the program, but for illustrative purposes in the section we will just use one loop. Our transformation begins by introducing a pair of variables for the loop. The variable `_ctr` is used to count the loop's iterations, and `_bnd` is an input to the dynamic analysis, whose value is pragmatically chosen to determine a useful prefix of potentially non-terminating traces. As such, `_ctr` is incremented inside the loop body and

```
1  int s = 1, t = 1, c = 1
2  int _ctr = 0, _bnd = 500
3  vtrace_pre(s,t,c)
4  while (t*t - 4*s + 2*t + 1 + c >= 0):
5    if (_ctr >= _bnd) abort() else _ctr++
6    vtrace_body(s,t,c)
7    t = t + 2
8    s = s + t
9    c = c + t
10 vtrace_post(s,t,c)
```

when it reaches _bnd, control exits the loop if it hasn't already. Next, we inject function calls to a method vtrace in three places: before ($\text{vtrace}_{\texttt{pre}}$), during ($\text{vtrace}_{\texttt{body}}$), and after ($\text{vtrace}_{\texttt{post}}$) the loop. In each location, vtrace is used to record the values of the variables in scope as a tuple such as $(\texttt{body}, \texttt{s} = 1, \texttt{t} = 1, \texttt{c} = 1)$, which we call a "snapshot". It should be noted that $\text{vtrace}_{\texttt{post}}$ only captures values for states that exited the loop due to natural causes.

This truncation and instrumentation permits us to distinguish between three classes of traces:

$$\bar{\pi}_{\texttt{base}} = \{(\texttt{pre}, \_)\cdot(\texttt{post}, \_)\},\ \bar{\pi}_{\texttt{term}} = \{(\texttt{pre}, \_)\cdot(\texttt{body}, \_)^{+}\cdot(\texttt{post}, \_)\},\ \bar{\pi}_{\texttt{mayloop}} = \{(\texttt{pre}, \_)\cdot(\texttt{body}, \_)^{+}\}$$

Above we have described the classes of traces using simple regular expressions, matching the first component of the tuple, and ignoring the values of variables. Technically, by these expressions, we mean the set of all traces that match the regular expression. The first set of traces $\bar{\pi}_{\texttt{base}}$ are those that have a snapshot before the loop, skip the loop entirely and then have a snapshot immediately after the loop. The second set $\bar{\pi}_{\texttt{term}}$ is similar but has at least one snapshot from inside the body of the loop. The third set $\bar{\pi}_{\texttt{mayloop}}$ includes traces that entered the loop but for which there is no post-loop snapshot. (Of course the union of these languages covers the language of the program.) This transformation is unsound because (i) it does not account for loop body states beyond _bnd, and (ii) executions may be forced to exit the loop before their day has come to do so. However, as we will see, this strategy collects *rich* data that enables us to start making guesses for ranking functions and recurrent sets, even in nonlinear contexts such as this example.

2.2 Algorithms

Learning ranking functions for termination. In Section 4, we present an algorithm beginning with:

(1) Instrumenting the program as discussed above.
(2) Generating random inputs to the program and collect traces (as in [Nguyen et al. 2012, 2014a]).
(3) Partitioning traces into $(\bar{\pi}_{\texttt{base}}, \bar{\pi}_{\texttt{term}}, \bar{\pi}_{\texttt{mayloop}})$.
(4) Using $\bar{\pi}_{\texttt{term}}$ as an input to subprocedure InferRF($\bar{\pi}_{\texttt{term}}$), discussed below, to infer a ranking function from the data.

For the above Termination example, such a possible trace is: $\{(\texttt{pre}, 1, 1, 42, 1), (\texttt{body}, 1, 1, 42, 1), (\texttt{body}, 4, 3, 42, 4), (\texttt{body}, 9, 5, 42, 9), \ldots\} \in \bar{\pi}_{\texttt{term}}$, where each tuple represents variables (_,s,t,k,c). Running DynamiTe on this example takes 7.55 seconds to produce an answer. By comparison, existing tools for termination [AProVE 2020; CPAChecker 2020; Gurfinkel et al. 2015; Ultimate 2020] typically perform well on linear programs, but fail to produce an answer on this program. The output of InferRF is the ranking function expression k-c. In some cases this guess may already be useful, even though it has not been verified. A user may wish to examine it and, in this case, it appears to be correct. If a stronger guarantee is needed, this ranking function can be given to a *safety* reachability prover such as (the reachability analyses of) CPAChecker [2020] or Ultimate [2020]. For example, using a standard translation [Cook et al. 2006], we can use an encoding that reduces ranking function validity checking to reachability. In this example, Ultimate's reachability reasoning can verify that k-c is a valid ranking function after 167 seconds.

In some cases, however, InferRF guesses incorrectly and returns a ranking function that is invalid. In these cases, a reachability solver may return a counterexample, which contains valuable information: a stem (path to the loop) and lasso (cycle through the loop body) that potentially non-terminates. In Section 4 we describe a sub-procedure GuessInput(cex) that uses this counterexample to guide the generation of new program inputs that can lead to a trace corresponding to this stem and lasso.

Our algorithm is parametric over the procedure InferRF for inferring ranking functions from trace samples. In Section 4 we give one strategy that is based on taking samples from the transitive closure over the trace snapshots and then fitting them to a template, using models of the well-foundedness constraints from an SMT solver to generate unknown coefficients of the template. The output ranking functions can be seen in Section 8.

Learning recurrent sets for non-termination. For non-termination, the goal is to guess a *recurrent set*, which is a set of states X, such that once X is reached, every subsequent transition will return to X. (We will define this formally later.) We begin by instrumenting the program but, unlike the termination algorithm, we instead use a form of iterative refinement to learn recurrent sets. For each loop, our overall algorithm keeps a work-list of candidate recurrent sets, starting by using the loop *condition* itself as the first candidate recurrent set. On each iteration, we select such a candidate recurrent set and check whether it is a valid recurrent set and reachable. If so, the algorithm has proven non-termination and returns. Otherwise, we have a model witness to the invalidity of the recurrent set which can be used for its refinement.

When R is not a recurrent set, our procedure $\texttt{RefineRS}(R, P, \mathcal{T}_{\texttt{stem}}, \mathcal{T}_{\texttt{loop}})$ attempts to learn a refined set from traces of the program. RefineRS uses the reason for the invalidity of R to generate a set of traces Π of the instrumented/truncated program. These traces are then used to learn conditions that lead to *refined* candidate recurrent sets. These refined candidate recurrent sets are then returned to the outer algorithm for further validity checking.

From failed validation to dynamic learning. A failed proof of static termination can be used to inform a dynamic non-termination proof and vice-versa. We discuss how our algorithms can be integrated together in an algorithm called ProveTNT, described in Section 6. The key idea is to additionally parameterize ProveT (respectively, ProveNT) by a set of input traces, which are derived from a failed ProveNT (resp., ProveT) attempt. These concrete traces contain useful data: examples of where the program appears to terminate or appears to diverge, and can immediately be used to guess ranking functions or recurrent sets.

2.3 The DynamiTe Tool

We have developed DynamiTe for dynamically guessing (and statically validating) rank functions and recurrent sets. The tool is publicly available at [DynamiTe 2020]. DynamiTe is written in a combination of Python and OCaml, the latter used mostly for program transformations (instrumentation and ranking function validity checking) with CIL [Necula et al. 2002].

DynamiTe takes advantage of several existing dynamic, symbolic, and static analysis techniques and tools as shown in Fig. 1. The two main algorithms to check for termination and non-termination mentioned above are the two blocks labeled ProveT and ProveNT, respectively. As shown in the figure, ProveT uses the two tools Ultimate and CPAchecker to verify the inferred ranking functions and obtain counterexamples to improve the inference process. On the other hand, ProveNT uses the CIVL symbolic execution tool to obtain program information such as loop conditions and transition relations, the DIG

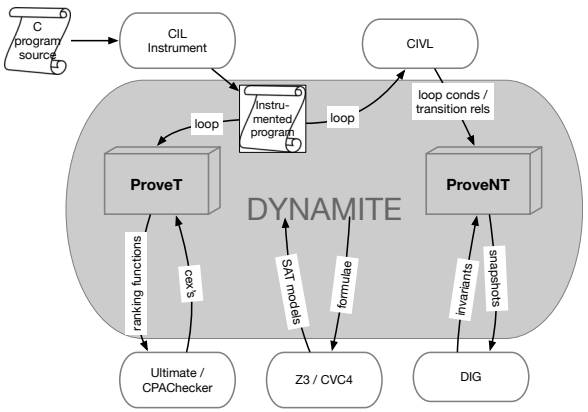

Fig. 1. Tools used in DynamiTe.

dynamic invariant generation tool to infer invariants from snapshot traces in order to represent and refine recurrent sets. DynamiTe also uses the Z3 and CVC4 SMT solvers to check if the candidate recurrent sets are valid and if not obtains counterexamples to refine DIG's inference process.

Our algorithms can work with programs containing *sequential and/or nested loops*. Our program transformation puts each loop into a separate method and replaces the loop by a call to that method. We then build a call graph of those methods and extract a *postorder* call sequence from it. We analyze each loop at a time in that order, i.e., the top-down innermost loop will be examined first. We proceed to the next loop in the sequence only when we have learned that the current loop is terminating. Otherwise, we conclude that the whole program does not terminate due to the non-termination of the current loop.

Our main goal in this paper is to develop integrated termination/nontermination algorithms that exploit dynamic analysis to support nonlinear programs. However, we also evaluate how DynamiTe performs on *linear* examples. To this end, we evaluated DynamiTe on the 66 benchmarks from the SV-COMP `termination-crafted-lit` set of linear arithmetic termination and non-termination problems collected from literature. We compared DynamiTe to Ultimate, because it is one of the most successful termination reasoning tools available. We report Ultimate's proving time as compared with DynamiTe's guessing time and DynamiTe's time to validate guesses. Details are in Section 8. Overall, DynamiTe is roughly an order of magnitude slower than Ultimate on linear benchmarks, owing to the fact that DynamiTe must repeatedly execute the program to collect data. Nonetheless, it's worth noting that Ultimate is a much more mature tool. In one case, DynamiTe was able to learn a rank function that Ultimate could not. We also show that DynamiTe is competitive for proving *non*-termination of linear programs, as compared to Ultimate's ability to generate lasso counterexamples to termination.

For the nonlinear case, there are two existing benchmarks: `polyrank` for termination and ANANT for non-termination. However, they only have at-most-quadratic polynomial programs and 10/11 programs in the `polyrank` benchmark are linear. To make (non)termination reasoning more challenging, we adapted the closely related SV-COMP `digbench` set of programs for nonlinear invariant generation problems to create two new sets of benchmarks called `termination-nla` and `nontermination-nla` which we are submitting to SV-COMP. The set `termination-nla` consists of 37 terminating programs and `nontermination-nla` consists of 38 non-terminating programs, which were created by adapting (up to sextic degree) nonlinear invariants in their loop conditions. Our empirical evaluation shows that DynamiTe can discover and sometimes validate rank functions (in 35 of 37 cases) and recurrent sets (in 33 of 38 cases) for nonlinear programs, that are not supported by Ultimate. (Ultimate returns an unsupported error message.)

3 PRELIMINARIES

We denote a program by P. We assume, for simplicity, that it has a single set V of variables. We will sometimes use the notation V' to mean a second set of primed versions of the same variables, *i.e.* $V' = \{v' \mid v \in V\}$, to describe transition relations. We denote by Σ set of states which we treat as valuations of the variables V, *i.e.* $\Sigma : V \to Val$. To represent conditions, we use logical formulae for states, denoted C, where $[\![C]\!] : \Sigma \to \mathbb{B}$. We also work with logical state *transition relations* denoted $\mathcal{T}$, where $[\![\mathcal{T}]\!] \subseteq \Sigma \times \Sigma$. $\mathcal{T}$ can also be presented in the form of logical formulae. As we describe below, program loops can be summarized using these conditions and relations, in a standard way.

Definition 3.1 (Ranking functions). For a state space S, a ranking function f is a total map from S to a well-ordered set with ordering relation $\prec$. A relation $\mathcal{T} \subseteq S \times S$ is *well-founded* if and only if there exists a ranking function f such that $\forall (s, s') \in \mathcal{T}.\ f(s') \prec f(s)$.

The existence of a ranking function over the transition relation $\mathcal{T}$ of a loop implies the termination of that loop, as there can be no infinite sequence of states $s_0, s_1, \ldots$ such that $\mathcal{T}(s_i, s_{i+1})$ holds for every $i \geq 0$. This is because for all $i \geq 0$, $f(s_{i+1}) \prec f(s_i)$ and the sequence of states mapped under f cannot be decreasing forever as the image of f is a well-order.

The termination of a loop with a transition relation $\mathcal{T}$ can also be proved by a finite set of ranking functions (or measures) $\mathcal{M} = \{f_1, \ldots, f_m\}$ by showing that the transitive closure of $\mathcal{T}$ is contained in the *disjunctively well-founded* relation defined from $\mathcal{M}$ [Podelski and Rybalchenko 2004b]. That is, $\mathcal{T}^+ \subseteq \{(s, s') \mid f_1(s') \prec f_1(s) \vee \ldots \vee f_m(s') \prec f_m(s)\}$.

This validity of the finite set of ranking functions $\mathcal{M}$ for the loop's termination can be checked via proving safety of the following instrumented loop (i.e., the error is unreachable) [Cook et al. 2006]. This check can be performed by a reachability prover such as Ultimate [2020] or CPAchecker [Beyer and Keremoglu 2011]. Below is an illustration for a while loop, whose instrumentation code are put in gray boxes. In this instrumented program, a state $\hat{s}$ of the loop is arbitrarily recorded and then for any subsequent state s, we check if the transition $(\hat{s}, s)$ satisfies at least one ranking function in $\mathcal{M}$. A transition $(\hat{s}, s)$ that does not satisfy any ranking function in $\mathcal{M}$, indicates that the transitive closure transition $\mathcal{T}^+$ is not a subset of the disjunctively well-founded relation of $\mathcal{M}$. In this case, the error is reached and the termination proof fails. Otherwise, a safe program in which the error is unreachable implies the loop's termination.

```
_dup = False
while C:
  if _dup:
    if not (f_1(x̂_1, ..., x̂_n) > f_1(x_1, ..., x_n) and f_1(x̂_1, ..., x̂_n) ≥ 0):
      ...
      if not (f_m(x̂_1, ..., x̂_n) > f_m(x_1, ..., x_n) and f_m(x̂_1, ..., x̂_n) ≥ 0):
        ERROR: skip
  if not _dup and *:
    x̂_1 = x_1; ...; x̂_n = x_n
    _dup = True
  B
```

Definition 3.2 (Recurrent set). For sets of states X and transition relation $\mathcal{T}$, X is a *recurrent set* if

$$(1)\ X \neq \emptyset, \quad (2)\ \mathcal{T} \text{ is total on } X, \quad (3)\ \text{the image of } \mathcal{T} \text{ on } X \text{ is contained within } X$$

The above notion of recurrent sets (*i.e.* "closed recurrent sets" in [Chen et al. 2014]) can help to avoid the difficulty and inefficiency of reasoning the $\forall\exists$ alternation in "open recurrent sets" [Gupta et al. 2008], but it cannot support nondeterminism without under-approximation. Therefore, our non-termination proofs are restricted to only deterministic programs. Finding under-approximation of nondeterminism from concrete possibly-nonterminating snapshots to support non-termination proofs of nondeterministic programs will be our future work. Note that such restriction does not apply to our termination proofs.

Definition 3.3 (Loop summary). As is typical [Cook et al. 2006], we will describe loops in terms of a triple $(\mathcal{T}_{\texttt{stem}}, \mathcal{C}_{\texttt{loop}}, \mathcal{T}_{\texttt{loop}})$, where $\mathcal{T}_{\texttt{stem}}$ over-approximates the transition from the entrypoint of the program up to the loop header, $\mathcal{C}_{\texttt{loop}}$ over-approximates the condition for entering the loop, and $\mathcal{T}_{\texttt{loop}}$ over-approximates the transition through the entire body of the loop back to the header.

4 INFERRING RANKING FUNCTIONS FOR TERMINATION

The algorithm for proving termination is summarized as follows, and two of the main subprocedures involved, ProveT and InferRF, are shown in Fig. 2.

ProveT. The procedure ProveT aims to prove the termination of a loop L in an instrumented program $P_{\texttt{instr}}$ by inferring a set of ranking functions from a given set of terminating traces $\pi_{\texttt{term}}$.

```
ProveT(P, P_instr, L, π_term):
  rfset = {}
  π̄_mayloop = {}
  while (True):
    new_rfset = InferRF(π_term, L)
    rfset = rfset ∪ new_rfset
    if (IsUnchanged(rfset):
      return (Unk, π̄_mayloop)
    else:
      cex = ValidateRFs(P, rfset)
      if (not cex):
        return (, {})
      else:
        inps = GuessInputs(P_instr, cex)
        π = Execute(P_instr, inps)
        π_L = Project(π, L)
        π_base, π_term, π_mayloop = Partition(π_L, L)
        π̄_mayloop = π̄_mayloop ∪ π_mayloop
```

```
InferRF(π_term, L):
  tcTrans = {}
  for τ_L in π_term:
    tcTrans = tcTrans ∪ GenTCTrans(τ_L)
  rfTemplate = GenRFTemplate(L)
  rfset = {}
  while not IsEmpty(tcTrans):
    (s_1, s_2) = RandPop(tcTrans)
    t_1 = rfTemplate(s_1)
    t_2 = rfTemplate(s_2)
    rf = Solve(rfTemplate, {t_1 > t_2, t_1 ≥ 0})
    rfset = rfset ∪ {rf}
    tcTrans.filter(t: NotSatisfied(t, rf))
  return rfset
```

Fig. 2. Algorithm ProveT for proving Termination, aided by dynamic inference of candidate ranking functions.

(We discuss how $P_{\texttt{instr}}$ is built from P in Section 2 and formalize it in Section 7.) The procedure returns either the result `Term` when the termination proof is successful or otherwise, returns `Unk` with a set of "possibly non-terminating" traces $\bar{\pi}_{\texttt{mayloop}}$ as a counterexample. Initially, the counterexample $\bar{\pi}_{\texttt{mayloop}}$ and the set of ranking functions rfset are initialised to be empty. The procedure then enters a loop until a valid set of ranking functions is found or until no progress is made when updating the set of ranking functions. Starting with the set of terminating traces $\pi_{\texttt{term}}$, the subprocedure InferRF is called to produce a set of ranking functions that attempts to cover those traces in $\pi_{\texttt{term}}$. The details for this subprocedure is given in the next paragraph. The current set of ranking functions rfset is updated to include the resulting set of ranking functions (new_rfset) from InferRF. The loop in ProveT exits if no new ranking functions were added. Otherwise, the updated set of ranking functions rfset is validated against the original program P via a reachability prover (as is standard [Cook et al. 2006]). If the prover returns no counterexample, which means the validation is successful, ProveT returns `Term` indicating that the loop L is terminating (via the set of ranking functions rfset). On the other hand, if a counterexample to the set of ranking functions is found, then a new set of inputs is generated. The given program is executed on those new inputs and a set of concrete traces from these executions (π) is collected. These traces are then projected into the locations of interest in the loop L. That is, for each trace $\tau \in \pi$, the projection returns a sequence of states τ_L, comprising the state right before the loop, the states reached inside the loop, right after the loop header, and the state at the loop's exit. Subsequently, the set of these sub-traces (π_L) are partitioned on whether they never enter the loop's body ($\pi_{\texttt{base}}$), whether they terminate ($\pi_{\texttt{term}}$), and whether they reach the instrumented bound of iterations before terminating, and as such are classified as "possibly non-terminating" ($\pi_{\texttt{mayloop}}$). Finally, the traces in $\pi_{\texttt{mayloop}}$ are added into the counterexample $\bar{\pi}_{\texttt{mayloop}}$ and the procedure repeats the above steps with the new set of terminating traces $\pi_{\texttt{term}}$.

InferRF. This sub-procedure first generates a random sample of pairs of snapshots from the transitive closure of the concrete transition relation as follows. For each terminating trace $\tau_L \in \pi_{\texttt{term}}$, with an implicit order of appearance in the trace τ_L present, all combinations (σ_1, σ_2) of the states $\sigma_1, \sigma_2 \in \tau_L$ are generated, restricted so that σ_1 appears before σ_2 in τ_L. The set of these combinations is randomly shuffled into a list, and the first K pairs are selected, with K being a predefined value for the desired size of the sample. All these samples from each trace τ_L are aggregated into the set

```
1  ProveNT(P_instr, L, π_mayloop):
2    (T_stem, C_loop, T_loop) = GetLoopInfo(P_instr, L)
3    C_mayloop = DynInfer(π_mayloop)
4    # stack of candidate recurrent sets
5    stack S = {(0, C_loop), (0, C_mayloop)}
6    π̄_term = {}
7
8    while not IsEmpty(S):
9      (depth, R) = Pop(S)
10     if (depth>UPPERBOUND or R(V̄) ⇏ C_loop(V̄)):
11       continue;
12     if IsValid(R(V̄) ∧ T_loop(V̄, V̄′) ⟹ R(V̄′)):
13       if IsSat(T_stem(V̄_0, V̄) ∧ R(V̄)):
14         return (NonTerm, {})
15     else:
16       RS, π_term = RefineRS(R, P_instr, L, T_stem, T_loop)
17       π̄_term = π̄_term ∪ π_term
18       for R′ in RS:
19         S.push((depth + 1, R′))
20   return (Unk, π̄_term)
```

```
1  RefineRS(R, P_instr, L, T_stem, T_loop):
2    R as ⋀_i R_i
3    RS = {}
4    π̄_term = {}
5    for R_i in R:
6      r_i = (R(V̄) ∧ T_loop(V̄, V̄′) ⟹ R_i(V̄′))
7      if IsSat(¬r_i):
8        inps = GuessInputs(T_stem(V̄_0, V̄) ∧ ¬r_i(V̄, V̄′)
9        π = Execute(P_instr, inps)
10       π_L = Project(π, L)
11       π_base, π_term, π_mayloop = Partition(π_L, L)
12       C_term = DynInfer(π_term)
13       C_mayloop = DynInfer(π_mayloop)
14       π̄_term = π̄_term ∪ π_term
15       C_term as ⋀_i C_i
16       for C_i in C_term:
17         RS = RS ∪ {R ∧ ¬C_i}
18       RS = RS ∪ {C_mayloop}
19   return (RS, π̄_term)
```

Fig. 3. Algorithm ProveNT for proving Non-termination, aided by dynamic inference of recurrent sets.

tcTrans. The subprocedure InferRF also generated a ranking function template, which is of the form $u_0 + u_1 \cdot v_1 + u_2 \cdot v_2 + \ldots u_n \cdot v_n$ for the set of variables $\{v_1, \ldots, v_n\}$ in the loop L and the unknown coefficients $u_0, u_1 \ldots, u_n$.

While the set tcTrans is non-empty, an element (s_1, s_2) is randomly popped, and two instances t_1, t_2 of the template are produced for the two respective states s_1 and s_2. Given the valuation $\{h_1^i, \ldots, h_n^i\}$ of the set of variables $\{v_1, \ldots, v_n\}$ in the state s_i, for $i \in \{1, 2\}$, the instance t_i is of the form $u_0 + u_1 \cdot h_1^i + u_2 \cdot h_2^i + \ldots u_n \cdot h_n^i$. The solver from Z3 is then asked to return values for $u_0, \ldots, u_n$ that satisfy the constraints

$$u_0 + \textstyle\sum_{1 \le j \le n} u_j \cdot h_j^1 > u_0 + \sum_{1 \le j \le n} u_j \cdot h_j^2, \qquad \text{and} \qquad u_0 + \sum_{1 \le j \le n} u_j \cdot h_j^1 \ge 0,$$

while minimizing the value of $\sum_{0 \le j \le n} |u_j|$. The resulting solution of values for $u_0, \ldots, u_n$ is added as a candidate ranking function to the set rfset of accumulated ranking functions. Any pair of states (s_1, s_2) from the random sample of trasitive closure of the transition relation that was constructed earlier, that satisfies the latter candidate ranking function is removed from that sample, and the procedure continues with the remaining ones.

Correctness. Sub-procedure InferRF terminates since at each iteration of the loop we remove at least one of the pairs (s_1, s_2) from tcTrans (line 8 of InferRF), but possibly more (line 13 of InferRF). By construction, each ranking function returned by InferRF handles at least one of the pairs (s_1, s_2) in tcTrans. Such a candidate ranking function is only returned by ProveT if it is validated on P by a reachability solver. On the other hand, it is not guaranteed that ProveT will terminate. Because the traces are dynamically generated, and because the transitive closure is sampled randomly, a newly inferred candidate ranking function could potentially only handle few of the possible pairs of states in the actual transitive closure of the loop body. As a result, a new ranking function may be added to the set of possible ranking functions continuously (see line 6 of ProveT).

5 INFERRING RECURRENT SETS FOR NON-TERMINATION

The algorithm ProveNT for proving non-termination is given in Fig. 3. The input is an instrumented program $P_{\texttt{instr}}$, the loop L currently being analysed, and a set $\pi_{\texttt{mayloop}}$ of traces that may be non-terminating. The procedure outputs either that a recurrent set was found (`NonTerm`), or that such a recurrent set was not found (`Unk`) together with a set of traces that were found to be terminating. The algorithm is aided by a dynamic sub-procedure `RefineRS` for guessing candidate recurrent sets, which are then validated.

ProveNT begins by collecting summaries for the loop L in P. We use standard techniques [Cook et al. 2006] to represent L in terms of three entities:

- $\mathcal{T}_{\texttt{stem}}$ is a state relation that over-approximates the transition from the entry point of the program up to the entry point of loop L.
- $C_{\texttt{loop}}$ is a state predicate that over-approximates the condition for entering the loop.
- $\mathcal{T}_{\texttt{loop}}$ is a state relation that over-approximates all transitions from the beginning of the body of the loop, back to the loop header.

An illustration of these entities is given in Fig. 4. The algorithm is structured using a stack S as a work list, tracking candidate recurrent sets that will later be examined and possibly refined. To begin with, we ambitiously select $C_{\texttt{loop}}$ to be the first candidate recurrent set. The stack element also includes an integer 0, to track the exploration depth, so that we can later bound the search. We also add the condition $C_{\texttt{mayloop}}$ into the work list S, which is dynamically inferred from the set $\pi_{\texttt{mayloop}}$ of possibly non-terminating traces received from a failed termination proof.

The main loop iterates as long as S is non-empty and no valid recurrent set was found. Popping a candidate R off the stack, if we have gone beyond some upper bound, then we simply ignore R rather than exploring further refinements of R. R is also ignored if it doesn't even imply the loop condition: it could not be a recurrent set. We next use an SMT query `IsValid` to check whether R is indeed a recurrent set, *i.e.*, Definition 3.2: if R holds of variables $\bar{V}$, and a loop body transition to $\bar{V}'$ is possible, then R must hold of $\bar{V}'$. If R is a recurrent set, then we check that at least some state in R is reachable from an initial state, using $\mathcal{T}_{\texttt{stem}}$, and if it is we have succeeded in proving the program to be non-terminating. Alternatively, if R is not a recurrent set, we explore further by refining R, with respect to this loop, using subprocedure `RefineRS` discussed below. This subprocedure also collects any terminating traces found during the refinement. Such terminating traces are evidence that the program is terminating, and thus useful for the case where non-termination fails to be proved and the algorithm switches to proving termination.

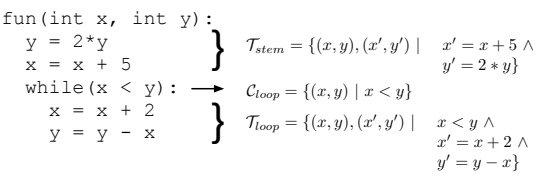

Fig. 4. Illustration of $\mathcal{T}_{\texttt{stem}}$, $C_{\texttt{loop}}$ and $\mathcal{T}_{\texttt{loop}}$

`RefineRS`. This subprocedure, shown in Fig. 3, takes as input the current candidate recurrent set R with respect to the loop L and returns a set of new candidate recurrent sets RS. The input recurrent set R is assumed to be a conjunction of $\{R_i\}_{i \leq k}$, for some $k \in \mathbb{N}$, and it is known that R is not a recurrent set for the transition relation $\mathcal{T}_{\texttt{loop}}$. As such there are two states σ and σ' (*i.e.* two valuations of the set of variables $\bar{V}$), for which the formula on the left does not hold:

$$\bigwedge_{i \leq k} R_i(\sigma) \wedge \mathcal{T}_{\texttt{loop}}(\sigma, \sigma') \implies \bigwedge_{i \leq k} R_i(\sigma') \qquad r_i = \bigwedge_{i \leq k} R_i(\sigma) \wedge \mathcal{T}_{\texttt{loop}}(\sigma, \sigma') \implies R_i(\sigma')$$

Therefore, for at least one of the $i \leq k$, the formula on the right does not hold. The candidate recurrent set R is updated for each such R_i as described next. The algorithm proceeds via `GuessInputs` using SMT to generate solutions $\bar{V}_0$ to the formula $\exists \bar{V}, \bar{V}'\ \mathcal{T}_{\texttt{stem}}(\bar{V}_0, \bar{V}) \wedge \neg r_i(\bar{V}, \bar{V}')$, which are used

as inputs to execute $P_{\texttt{instr}}$. Informally, these inputs are witnesses to a path, via $\mathcal{T}_{\texttt{stem}}$, from the initial state to a state on which the recurrent set fails. After the program is executed using the resulting inputs `inps` and a set of traces is produced as a result. The traces are first projected—to include the instrumented information regarding only the loop L being analyzed—and then partitioned into: the traces $\pi_{\texttt{base}}$ that never enter the loop, traces $\pi_{\texttt{term}}$ that definitely terminate, and traces $\pi_{\texttt{mayloop}}$ that may be non-terminating, as the execution for the latter reached the imposed loop bound. It should be noted that, since any candidate R implies $C_{\texttt{loop}}$, if the program is deterministic, and assuming soundness of the preceding subprocedures, then the inputs `inps` will not cause any traces that never enter the loop to be generated, and thus $\pi_{\texttt{base}}$ will be empty. Traces $\pi_{\texttt{term}}$ are used to dynamically infer a condition $C_{\texttt{term}}$ that captures the set of states reached right after the loop header by those terminating traces and a similar condition $C_{\texttt{mayloop}}$ is inferred using $\pi_{\texttt{mayloop}}$. The accumulating set $\bar{\pi}_{\texttt{term}}$ is updated to include $\pi_{\texttt{term}}$ and the recurrent set RS is then updated as follows. For every conjunct C_i of $C_{\texttt{term}}$, RS is updated to include the strengthened candidate recurrent set $R \wedge \neg C_i$ in which any states in C_i that is possibly in a terminating trace is excluded from the candidate. The condition $C_{\texttt{mayloop}}$ is also included as a new candidate recurrent set since it captures all states that are in possibly non-terminating traces. At the end, the procedure `RefineRS` returns the set of candidate recurrent sets constructed, together with any terminating traces accumulated in $\bar{\pi}_{\texttt{term}}$.

Consider the example to the right with nonlinear expressions to illustrate how ProveNT and `RefineRS` works. The summary of this loop is: $\mathcal{T}_{\texttt{stem}} = \text{true}$, $C_{\texttt{loop}} = t \leq n^2 + 1$, and $\mathcal{T}_{\texttt{loop}} = t \leq n^2 + 1 \wedge t' = t + 2m \wedge n' = n + 1 \wedge m' = m$. The procedure ProveNT first uses the loop condition $t \leq n^2 + 1$ as a candidate recurrent set and checks if the implication $t \leq n^2 + 1 \wedge \mathcal{T}_{\texttt{loop}} \implies t' \leq n'^2 + 1$ is valid. As this is not the case, ProveNT invokes `RefineRS` on this invalid recurrent set to refine it. Then `RefineRS` finds a set of inputs over the variables (t, n, m) that invalidate the implication, such as $\{(29, -6, -1), (13, -4, 0), (1, 0, 1), (1, -1, 2), (0, 0, 3), (1, 1, 4), \ldots\}$. The program execution over these inputs produces only terminating traces and a dynamic invariant inference tool, like DIG [Nguyen et al. 2012, 2014a], can generate the condition $m \geq -1$ from the snapshots at the beginning of the loop's body in those traces. This possibly terminating condition (see $C_{\texttt{term}}$ in `RefineRS`) is used to refine the current candidate into a new one $t \leq n^2 + 1 \wedge \neg(m \geq -1)$. Unfortunately, this new candidate recurrent set is still invalid and `RefineRS` generates a new set of inputs from its validity check, that is $\{\ldots, (78, -9, -5), (26, -5, -4), (24, -5, -3), (15, -4, -2)\}$. Again, all these inputs lead to terminating traces, from which a new condition $n \leq m - 1 \wedge m \leq -2$ can be dynamically inferred. From this new condition, `RefineRS` returns two new candidate recurrent sets by strengthening the current candidate $t \leq n^2 + 1 \wedge \neg(m \geq -1)$:

```
int t, n, m
while (t <= n*n + 1):
  t = t + 2*m
  n = n + 1
```

(1) $t \leq n^2 + 1 \wedge \neg(m \geq -1) \wedge \neg(n \leq m - 1)$, and
(2) $t \leq n^2 + 1 \wedge \neg(m \geq -1) \wedge \neg(m \leq -2)$.

Finally, the procedure ProveNT determines that the new candidate (1) is a valid recurrent set and returns the result `NonTerm`.

`DynInfer`. We use dynamic invariant inference to guess conditions from terminating traces and potentially non-terminating traces which are then used to refine the invalid candidate recurrent sets. Dynamic invariant generation works, pioneered by the tool Daikon [Ernst et al. 2001, 2007], learns candidate invariants from program execution traces and templates (e.g., equalities, inequalities). Recent works in dynamic invariant generation are capable of generating very expressive invariants (e.g., nonlinear invariants). In addition, many works integrate dynamic inference with static checking to remove spurious invariants. We use the DIG [Nguyen et al. 2012, 2014a] tool to infer numerical invariants from traces. DIG supports nonlinear equations as well as several other

forms of inequalities such as octagonal invariants and max/min-plus invariants. DIG reduces the problem of nonlinear equation solving to linear equation and ussing terms to represent nonlinear polynomials and uses linear constraint solving to find octagonal and max/min-plus invariants. In addition, DIG implements a counterexample-guided invariant generation technique that iteratively infer candidate invariants from program traces and check them using symbolic execution, which, for incorrect invariants, returns counterexamples that are used as traces to help DIG infer better results in the next iteration.

5.1 Correctness

ProveNT. For correctness of the algorithm ProveNT we aim to show that if its output is (NonTerm, {}) then there is at least one execution of the program P, that leads to a non-terminating execution of the loop L at hand. For what follows, we assume that

(1) for any state that satisfies $C_{\texttt{loop}}$, there is a valid transition from that state in the program P,
(2) $\mathcal{T}_{\texttt{stem}}$ is an exact representation, or at worst an under-approximation, of the transition from the entrypoint to the loop header, and
(3) $\mathcal{T}_{\texttt{loop}}$ is an exact representation, or at worst an over-approximation, of the loop body transition.

The procedure ProveT will declare that the input loop is non-terminating only when both $\texttt{IsValid}(R(\bar{V}) \wedge \mathcal{T}_{\texttt{loop}}(\bar{V}, \bar{V}') \implies R(\bar{V}'))$ and $\texttt{IsSat}(\mathcal{T}_{\texttt{stem}}(\bar{V}_0, \bar{V}) \wedge R(\bar{V}))$ hold (see lines 12 and 13 of ProveNT). In other words,

$$(i)\ \exists \bar{V}_0, \bar{V}\ \mathcal{T}_{\texttt{stem}}(\bar{V}_0, \bar{V}) \wedge R(\bar{V}) \qquad \text{and} \qquad (ii)\ \forall \bar{V}, \bar{V}' R(\bar{V}) \wedge \mathcal{T}_{\texttt{loop}}(\bar{V}, \bar{V}') \implies R(\bar{V}').$$

Formula (ii), together with the assumption (3) above implies the condition (3) of Definition 3.2. From formula (i) and the assumption (2) above, there is a state S' at the loop header that can be reached from S, such that $R(S')$ holds which implies that condition (1) of Definition 3.2 holds for a reachable state in P from S. Finally, given that R implies $C_{\texttt{loop}}$ (see line 10 of ProveNT), and given the assumption (1) above, it follows that the real transition relation for the loop is total on R. Therefore there is a non-terminating execution of P starting from the state S. We should note that, given that $\mathcal{T}_{\texttt{stem}}$ is an over-approximation in reality, our implementation could simply check if a witness path exists.

Further, the algorithm terminates, since whenever a new candidate recurrent set R is added the variable depth is increased and the recurrent sets with an accompanying depth of value higher than UPPERBOUND are ignored (see line 10 of ProveNT).

6 AN INTEGRATED ALGORITHM

We now describe ProveTNT, an algorithm supported with dynamic analysis, that mixes termination and non-termination reasoning, allowing the failed outcomes of one endeavor to provide feedback to the other. In this algorithm, ProveNT consumes the previously ignored argument $\bar{\pi}_{\texttt{mayloop}}$ (a set of potentially non-terminating traces returned by ProveT) and ProveT consumes the previously ignored argument $\bar{\pi}_{\texttt{term}}$ (a set of terminating traces from ProveNT).

The procedure ProveTNT is given in Fig. 5 and begins by instrumenting the input program P, generating random initial inputs, and executing the instrumented program on those inputs to get a set π of concrete traces. These executions may be used for reasoning about multiple loops in the program, and avoid the need for re-execution. We then iterate over the loops in the program in a post-order fashion, in which the top-down innermost loop will be analyzed first. If that loop is proved to be non-terminating, the procedure returns the result NonTerm immediately. Otherwise, it continues to analyze the next loop in the post-order sequence. At the end, the procedure returns the result Term when all loops in the program are proved to be terminating.

```
1  ProveTNT(P):
2    P_instr = Instrument(P)
3    inps = GenRandomInputs(P_instr)
4    π = Execute(P_instr, inps)
5    ℒ = GetLoopSeq(P_instr)
6
7    for L in ℒ:
8      π_L = Project(π, L)
9      π_base, π_term, π_mayloop = Partition(π_L, L)
10     if card(π_mayloop) >> card(π_base ∪ π_term):
11       r_nt, π̄_term = ProveNT(P_instr, L, π_mayloop)
12       if r_nt is NonTerm:
13         return NonTerm
14       else:
15         r_t, _ = ProveT(P, P_instr, L, π_term ∪ π̄_term)
16         if r_t is Unk:
17           return Unk
18     else:
19       r_t, π̄_mayloop = ProveT(P, P_instr, L, π_term)
20       if r_t is Unk:
21         r_nt, _ = ProveNT(P_instr, L, π_mayloop∪π̄_mayloop)
22         if r_nt is NonTerm:
23           return NonTerm
24         else:
25           return Unk
26   return Term
```

Fig. 5. The integrated algorithm for approving termination and non-termination, via mutual feedback.

Within each loop L we project on the set of traces, focusing on only those that reach L's header and keeping only the relevant snapshots from the instrumentation on that loop in π_L (see line 8 in ProveTNT). We then partition π_L into the three classes of traces $\bar{\pi}_{\texttt{base}}, \bar{\pi}_{\texttt{term}}, \bar{\pi}_{\texttt{mayloop}}$, similarly to what was described in previous sections. We next make a decision as to whether we should attempt non-termination or termination reasoning first. Our algorithm heuristically chooses which action to perform after comparing the sizes of terminating trace sets ($\bar{\pi}_{\texttt{base}}$ and $\bar{\pi}_{\texttt{term}}$) and the potentially non-terminating one ($\bar{\pi}_{\texttt{mayloop}}$). In our implementation, we decide to prove non-termination first when the number of the potentially non-terminating traces is four times larger than the total size of terminating traces. In the case the algorithm succeeds in proving the chosen analysis, it moves to the next step as described above (*i.e.* returning `NonTerm` immediately if ProveNT is chosen or analyzing the next loop if ProveT is chosen). Otherwise, the chosen sub-procedure will return counterexamples in the form of new traces, that can be used, together with the traces collected from running the random inputs, as input to the alternative analysis.

Consider the simple program: `while(x>=0): x = x + y`. This example conditionally terminates, depending on the initial values of `x` and `y`. That is, the loop does not terminate when $\texttt{x} \geq 0$ and $\texttt{y} \geq 0$ and terminates otherwise. Given that the random inputs are evenly-distributed then `x` is negative in roughly half of the random inputs, on which the loop terminates. From the heuristic for choosing the sub-procedure, ProveTNT may decide to attempt proving termination first. More specifically, in our implementation, we decide to prove non-termination first when the number of the potentially non-terminating traces is four times larger than the total size of terminating traces. The

sub-procedure ProveT may find a ranking function such as x from the terminating traces. However, it is not a valid ranking function for all inputs and the validation in ProveT returns counterexamples whose corresponding inputs create potentially non-terminating execution traces, such as [(x=0, y=0), (x=0, y=0), . . .], [(x=3, y=1), (x=4, y=1), . . .], etc., in which the ranking function x is not decreasing. Since there is no terminating trace generated from those inputs, ProveT gives up and returns such potentially non-terminating execution traces as its counterexample traces. At this point, our ProveTNT algorithm switches gears and uses these counterexample traces as inputs to ProveNT. Finally, ProveNT proceeds on these traces and finds a recurrent set $x \geq 0 \wedge y \geq 0$ from them to confirm the loop's non-termination.

The proving strategy in ProveTNT also helps to overcome the scenario when execution traces from a terminating program, like the simple loop in this program: `while(x<1000): x = x + 1`. This is wrongly categorized as potentially non-terminating due to the predefined instrumented execution bound (*e.g.* `500`) being reached before the loop terminates. In this example, the execution traces with inputs where $x < 500$ are considered as potentially non-terminating. Note that on those inputs, the collected traces from that loop are identical to traces collected from the non-terminating loop `while True: x = x + 1`. If those inputs dominate the set of random inputs then the procedure ProveTNT may attempt proving non-termination first. The sub-procedure ProveNT then starts with the first candidate recurrent set $x<1000$ and performs a check on it with the implication $x<1000 \wedge x' = x+1 \implies x'<1000$. The implication does not hold and there is only one input of $x=999$ and the corresponding terminating trace $[x=999, x=1000]$ are generated from it as counterexample. Due to the lack of data, the dynamic inference is not triggered and there is no new candidate recurrent set generated. The procedure ProveTNT then passes that terminating counterexample trace to the sub-procedure ProveT for proving termination. From that trace, ProveT can easily find the ranking function $999 - x$ to prove the loop's termination. Interestingly, the ProveT alone cannot prove the termination of this loop due to the lack of terminating traces. This happens since we usually prefer to generate small random inputs and limit the number of them, which may help to reduce the program execution time, for efficiency. In this example, we can try inputs larger than the predefined instrumented bound (*i.e.* $x \geq 500$) but the same problem may occur on other examples if the generated inputs are not large enough. For example, for the same program but with the loop condition replaced with `x < 10000`, the algorithm would require some inputs where x would be larger than or equal to 9500.

7 THE DYNAMITE TOOL

We have realized our learning-based algorithms in a new tool called DynamiTe. DynamiTe employs the power of several major existing tools, yet our particular combination of them allow DynamiTe to do things that none of these tools can achieve individually (see Fig. 1). We found that we were able to use these tools with few modifications and, consequently, our framework allows us to benefit from improvements in those tools or substitute alternatives. For SMT, we use SMTlib and for reachability, we follow the SV-COMP [Beyer 2020] format. We now discuss some of the key components of the implementation.

We perform two transformations on the input program. For validating our guesses in termination reasoning, we use a standard transformation [Cook et al. 2006] that lets us input candidate rank functions and apply a reachability prover. This is a common technique and we have briefly described it in Sec. 3. The second transformation (described in Section 2) involves (i) instrumenting the program to collect states and traces and (ii) truncating potentially infinite loops. For a formal description of this transformation, see our technical report [Le et al. 2020].

DynamiTe and Ultimate on LIN Termination Programs from SV-COMP

Benchmark	DynamiTe Learned Rank Functions	Learning T(s)	Res	Validate T(s)	Res	Total T(s)	σ_5	UAutomizer T(s)	Res
AlDaFeGo-SAS2010-easy2-2.c	z	7.4	✓	8.0	✓	20.7	1.2	0.4	✓
AlDaFeGo-SAS2010-random2d.c	$N+r$, $-r$, $r-i$, $x-r$, $i-r$, $N-i$	23.8	✓	13.2	✓	52.38	7.9	0.6	✓
AlDaFeGo-SAS2010-wcet2.c	$-i$, $1-i$, j, $j-i$, $-j$, $2-i$	11.5	✓	121.5	?	148.14	75.3	1.3	✓
BrCoFu-CAV2013-Fig1.c	$n-j$, j, $-i$, $n-i$	28.9	✓	14.2	✓	66.22	22.0	1.5	✓
ChCoGuSaYa-ESOP2008-easy2.c	z	3.9	✓	2.8	✓	9.4	0.8	0.4	✓
ChFlMu-SAS2012-Ex3.01.c	$-x+z$, y, $-x$	5.9	✓	4.0	✓	16.34	2.4	2.8	✓
CoSeZu-TACAS2013-Fig8a-mod.c	$-x$, x, $-K+x$, $K-x$	11.7	✓	4.0	✓	18.62	0.9	4.6	✓
HaLaNoRa-SAS2010-Fig1.c		0.0	?	4.9	?	8.94	1.8	29.4	✓
HeHoLePo-ATVA2013-Fig5.c	x	2.3	✓	5.8	✓	9.56	0.5	3.8	?
KrShTsWi-CAV2010-Fig1.c	$28-x$, $82-x$, $88-x$, (see below)	8.9	✓	6.0	?	14.66	1.5	3.9	✓
LeHe-TACAS2014-Ex1.c	q	2.2	✓	2.9	✓	8.78	1.8	0.5	✓
PoRy-TACAS2011-Fig1.c	y	2.4	✓	4.6	✓	8.76	0.9	0.3	✓
Ur-WST2013-Fig2.c	$-x1$, $-x1 + 5{\cdot}x2$, (see below)	14.2	✓	12.5	✓	32.78	5.7	1.6	✓
cstrcspn.c		–	–	–	–	2	0.2	90.3	✓
genady.c	i	0.1	✓	7.4	✓	9.78	1.6	0.4	✓

...

(Results of for the other 46 benchmarks in [Le et al. 2020].

KrShTsWi-CAV2010-Fig1.c: $28-x$, $82-x$, $88-x$, $90-x$, $104-x$, $118-x$, $144-x$, $156-x$, $212-x$, $214-x$, $228-x$, $234-x$, $246-x$

Urban-WST2013-Fig2.c: $-x1$, $-x1 + 5{\cdot}x2$, $-x1 + 6{\cdot}x2$, $-x1 + 7{\cdot}x2$, $-x1 + 8{\cdot}x2$, $-x1 + 9{\cdot}x2$, $-x1 + 10{\cdot}x2$, $-x2$

Fig. 6. Results of applying UAutomizer and DynamiTe on the 61 termination benchmarks from SV-COMP `termination-crafted-lit`. For lack of space, we only show 15 rows (every 4th row) with abbreviated names. The full result can be found in our technical report [Le et al. 2020]. ? indicates unknown results.

8 EVALUATION

Our main goal of DynamiTe is to improve the state-of-the-art in termination and non-termination reasoning to better support nonlinear (NLA) programs. To this end, Sections 8.2 and 8.3 report experimental results on those programs. However, in Section 8.1 we first evaluated DynamiTe to see how it performs on linear programs, particularly in comparison with the state-of-the-art tool UAutomizer from Ultimate Ultimate [2020], which is the winner of the Termination category in the recent Competition on Software Verification (SV-COMP) [Beyer 2020].

Our experiments were all run on a 20-core Intel(R) Core(TM) i7-6950X CPU @ 3.00GHz. DynamiTe in general take advantages of parallel processing when possible. For example, in termination reasoning, multiple instances of Ultimate's variants (Automizer and Taipan) and CPAchecker are invoked to validate the termination results. In non-termination reasoning, we use these CPU cores to run the symbolic execution tool CIVL to obtain program information at multiple depths. The dynamic inference tool DIG also computes invariants simultaneously. The timeout for each benchmark program is 400s.

8.1 Linear Programs

Although our main goal was to support NLA programs (*i.e.* expressivity) we nonetheless compared our work against the state-of-the-art termination tool UAutomizer. We ran both UAutomizer and DynamiTe on the 61 termination benchmarks and 5 non-termination benchmarks from the SV-COMP suite `termination-crafted-lit`, which were used in SV-COMP 2020. Note that this folder contains other benchmarks that are for other properties like overflow.

Terminating Linear Programs. The results of the experiments on these programs are shown in Table 8. Since DynamiTe is nondeterministic, we ran our experiments 5 times. We depict the ranking

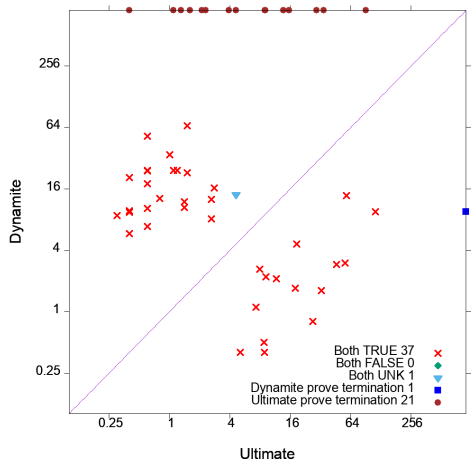

Fig. 7. Visual comparison between DynamiTe and UAutomizer on linear termination benchmarks.

functions learned by DynamiTe in the second column, taken from the first iteration of DynamiTe. If a benchmark program has more than one loop, we report the ranking functions learned from the last analyzed loop. We also break down the overall time (and result) of DynamiTe into time spent to learn ranking functions versus validate them. Finally, we report the **Total** time, averaged over 5 runs, as well as the standard deviation σ_5.

In the last two columns, we show the time and result of UAutomizer. These results are also visualized in the plot in Fig. 7. The results show that UAutomizer often performs much faster than DynamiTe on these linear examples, owing largely to the fact that DynamiTe must execute the program many times (on the newly generated inputs), as is typically the case for data-driven strategies [Nguyen et al. 2017a]. Nonetheless, the results show that DynamiTe is competitive. In most benchmark programs, DynamiTe can learn useful ranking functions from their terminating traces. For ranking functions that cannot be validated by the reachability provers, we manually checked if they are valid with respect to the observed terminating traces. We found that some of them are the desired ranking functions to prove the programs' termination while the others are still in good progress so that we could infer the desired ranking functions from their validation's counterexamples. There are no ranking functions learned from unsupported recursive programs and string-manipulating programs. We also found that DynamiTe was able to infer a ranking function for the program `HeHoLePo-ATVA2013-Fig5.c`, which can be validated successfully, while UAutomizer cannot. It is worth noting that UAutomizer is a very mature tool, with contributions from multiple researchers/developers, has been applied in industrial settings, and has consistently performed well in the SV-COMP Termination categories. By contrast, DynamiTe is still in its infancy. Furthermore, we will soon discuss nonlinear programs, a class of programs not currently supported by UAutomizer.

Non-terminating Linear Programs. Of the `termination-crafted-lit` SV-COMP suite, only 5 benchmarks were for non-termination. We ran DynamiTe and Ultimate on them; the results are given in Fig. 8. Again we report the mean and standard deviation over 5 runs. In all cases, UAutomizer was able to generate a lasso counterexample to termination. DynamiTe was able to produce a validated recurrent set in 3 programs with `UPPERBOUND=3` in ProveNT. The program `HeJhMaSu-POPL2002-LockEx.c` has a nondeterministic loop condition so that *True*is its trivially valid recurrent set. Therefore,

DynamiTe and Ultimate on LIN Non-termination Programs from SV-COMP

Benchmark	DynamiTe Learned Rec. Sets	Learning T(s)	Res	Validate T(s)	Res	Total T(s)	σ_5	UAutomizer T(s)	Res
BrMaSi-CAV2005-Fig1-mod.c	$y1{\neq}y2 \wedge y2 = 0$	11.7	✓	8.9	✓	44.3	1.7	0.1	χ
ChCoFuNiHe-TACAS2014-Intro.c		5.8	?	2.7	?	38.68	5.4	0.2	χ
HeJhMaSu-POPL2002-LockEx.c	$True$	0.0	✓	0.1	✓	17.4	0.1	0.1	χ
Ur-WST2013-Fig1.c		6.8	?	0.8	?	17.7	1.8	0.4	χ
Velroyen.c	$x{\neq}0$	0.0	✓	0.1	✓	12.1	0.7	2.2	χ

Fig. 8. Results of applying ultimate and DynamiTe on the 5 *non*-termination benchmarks from SV-COMP `termination-crafted-lit`. Ultimate returned "incorrect," indicating that it had found a non-terminating lasso to disprove termination.

there is no cost for learning recurrent sets in these programs. On the other hand, the program `ChCoFuNiHe-TACAS2014-Intro.c` has a nondeterministic assignment in its loop body so that while the loop condition is a (closed) recurrent set, it cannot be validated without an underapproximation to restrict the choice of nondeterministic values in that assignment. The two programs `Ur-WST2013-Fig1.c` and `Velroyen.c` have many branches in their loop bodies but only some of them were taken by the symbolic execution tool to build the loop summaries. Unfortunately, in `Ur-WST2013-Fig1.c`, the non-terminating branch was missing so that DynamiTe cannot find any valid recurrent set from the other (terminating) branches in the summary.

8.2 Termination of NLA Programs

Currently, we lack challenging benchmark suites for termination of nonlinear programs. The existing `polyrank` benchmark [Bradley et al. 2005b] has only one (quadratic) polynomial program. The other programs in `polyrank` are linear and many of them were included into the SV-COMP `termination-crafted-lit` suite. DynamiTe can prove the termination of 8/11 benchmarks (see Fig. 9) by inferring multiple linear ranking functions, instead of a single nonlinear ranking function, and successfully validating them with Ultimate or CPAchecker. For the remaining 3 examples, DynamiTe was able to infer the correct ranking functions, but the validators could not validate them before timeout. In order to better evaluate the tool, we adapted an existing nonlinear testsuite from SV-COMP called `nla-digbench` which consists of 28 programs implementing mathematical functions such as `intdiv`, `gcd`, `lcm`, `power`. Although these programs are relatively small (under 50 LoCs) they contain nontrivial structures such as nested loops and nonlinear invariant properties. To the best of our knowledge, `nla-digbench` contains the largest number of programs containing nonlinear arithmetic. These programs have also been used to evaluate other numerical invariant systems [Rodríguez-Carbonell and Kapur 2007b; Yao et al. 2020].

However, these benchmarks are for invariant generation rather than termination and most of them are linear programs (with nonlinear invariant properties). We therefore adapted these benchmarks to make them suitable nonlinear examples for termination. For each benchmark, we manually examined the behavior of the program. The benchmarks contain commented nonlinear assertions, that illustrate the need for nonlinear reasoning. For example, `bresenham1` contains the assertion `2*Y*x - 2*X*y - X + 2*Y - v == 0`. We adapted these assertions to be loop conditions in various ways, creating one or more termination challenge programs. We typically geared the loop condition to the assertion. In this case, we used the invariant that the LHS is 0 and added an additional term `+c` that increased on each iteration, and made the loop condition bounded by a variable `k`. For example, from the `bresenham1` program, we created a new program in which the loop condition is `2*Y*x - 2*X*y - X + 2*Y - v + c <= k`. In other cases, we introduced new variables and had them be integrated by other expressions that we knew to be monotonically increasing or

DynamiTe on NLA Termination Programs adapted from SV-COMP

Benchmark	DynamiTe Learned Ranking Functions	Learning Time	Learning Res	Validation Time	Validation Res
loop1.c	$-i, -j + bn, -i + an$	16.3	✓	7.6	✓
loop2.c	$x, x + -y$	3.5	✓	3.8	✓
loop3.c	$x, -y + z, z, x + -z, x + -y$	8.2	✓	17.4	✓
loop4.c	$-i, -i + an, -j, -k + bn, -k, -j + bn$	17.4	✓	179.7	✓
loop6.c	$-x, N + -x, -x + -y$	1.2	✓	4.2	✓
loop7.c					
loop8.c	$-1 \cdot y1 + 12 \cdot y2, y2, -1 \cdot y1 + 14 \cdot y2, -1 \cdot y1 + 29 \cdot y2, -1 \cdot y1 + 34 \cdot y2, -1 \cdot y1 + 42 \cdot y2, -1 \cdot y1 + 49 \cdot y2, -1 \cdot y1 + 50 \cdot y2$	5.0	✓	15.9	✓
loop9.c	$y2, y1$	6.9	✓	4.9	✓
loop10.c	$y, x, w + -1 \cdot y, w, z$	0.7	✓	682.3	?
loop11.c	$-e, e, -n, n$	1.0	✓	296.3	?
loop12.c	$-1 \cdot y, -1 \cdot y + -1 \cdot z, x + -1 \cdot y, -1 \cdot x, z, 1 + -1 \cdot y, 1 + z$	7.7	✓	725.2	?

Fig. 9. Results of applying DynamiTe to the benchmark suite polyrank.

replace variables and numbers with nonlinear expressions that are equal to them. We have made these 38 benchmarks available in the supplemental materials [DynamiTe 2020] and will issue a pull request to submit them to the SV-COMP benchmark repository.

The results of applying DynamiTe to these benchmarks is given in Fig. 10. For each benchmark, we give a brief description of the mathematical behaviors of the program in the second column. Based on the results, we display the output list of inferred ranking functions, as well as a breakdown of the time it took to learn versus validate them. As mentioned in the previous section, we use Ultimate and CPAchecker for validation. In 34 of the 38 benchmarks, DynamiTe was able to guess ranking functions. The ranking functions derived from 8 of those 32 benchmarks can be validated. For those ranking functions that cannot be validated by the existing safety provers, we manually inspected them and confirmed they were correct. In the remaining 4 cases, the program cohencu4 is non-terminating because the increment statement c++ was unintentionally not added. Therefore, there are no terminating traces to learn ranking functions. After fixing that problem, DynamiTe can infer the desired ranking functions for this program successfully. The 2 programs freire1 and knuth-nosqrt are originally floating-point programs but were intentionally transformed to integer programs. However, the invariant assertions in the original benchmarks is no longer valid in our adapted benchmark programs. Therefore, the desired ranking functions cannot be found from them. The last program knuth still has the use of sqrt function which is not supported by our CIL instrumentation. It is worth noting that UAutomizer cannot handle these benchmarks. In addition, since the validation get stuck on most benchmarks, we do not report the total time in Fig. 10.

8.3 Non-termination of NLA Programs

We first apply DynamiTe to the existing nonlinear non-termination benchmark Anant [Cook et al. 2014]. The benchmark is a set of quadratic polynomial programs and some of them have nondeterminism and divisions. The result of applying DynamiTe on this benchmark is given in Fig. 11. DynamiTe can prove the non-termination of 4 benchmarks that [Cook et al. 2014] cannot handle. However, there are 10 benchmarks that DynamiTe cannot handle, due to nondeterminism (4), overfitting invariants (1), overflow (1), and problems in SMT solvers (2) or in symbolic execution (2). This result shows that our dynamic approach is orthogonal to the existing static techniques for proving non-termination of (nonlinear) programs.

In addition to the benchmark Anant, we also adapted SV-COMP nla-digbench suite to create NLA *non*-termination challenge programs (*e.g.* up-to-sextic polynomial programs). (These are

DynamiTe on NLA Termination Programs adapted from SV-COMP

Benchmark	Desc.	DynamiTe Learned Ranking Fns.	Learning Time	Learning Res	Validation Time	Validation Res
bresenham1		$-c+k$	33.3	✓	17.5	✓
cohencu1	cubic sum	$-x$, $k+-z$	11.9	✓	8.2	✓
cohencu2	cubic sum	$k+-z$, $-x$, $k+-y$	7.5	✓	172.9	?
cohencu3	cubic sum	$k+-z$, $-y+z$, $k+-x$	8.6	✓	39.4	?
cohencu4	cubic sum		0.0	?	16.5	?
cohencu5	cubic sum	$-c+k$	14.4	✓	147.0	?
cohencu6	cubic sum	$a+-n$	21.4	✓	197.7	?
cohencu7	cubic sum	$a+-n$	12.0	✓	38.6	?
dijkstra1	square root	$n+-q$	4.9	✓	15.8	✓
dijkstra2	square root	$-c+k$	13.7	✓	45.7	?
dijkstra3	square root	$n+-q$	1.3	✓	6.5	?
dijkstra4	square root	$-c+k$, h	7.3	✓	199.9	?
dijkstra5	square root	$-c$, $-c+k$	12.6	✓	16.5	?
dijkstra6	square root	$-c$, $-c+k$	11.1	✓	15.4	?
divbin1	int div	$r-b$	6.2	✓	192.4	?
egcd	gcd	b, a	21.4	✓	13.2	?
egcd2	gcd	c, $-s$, s, $-q$	13.9	✓	34.8	?
egcd3	gcd	$-v$, $c+-v$, $b+-v$, v, $c+-2\cdot v$, $c+-d$, d, $-c$	23.4	✓	12.1	?
fermat1	product	$-c+k$	8.4	✓	17.5	?
freire1	square root	$-1\cdot r+12\cdot a+3\cdot k$	19.6	?	4.4	?
geo1	geo series	y, $-x$, $-y$, $-x+y$, x, $k-c$	18.4	✓	13.5	?
geo2	geo series	$-y$, y, $x+-y$, $k+-c$	16.5	✓	4.5	?
geo3	geo series	$-x$, $-x+y$, y, $-y$, $k+-c$	690.1	✓	7.9	?
hard	int div	$-q$, p	5.0	✓	60.6	?
hard2	int div	$r+-p$	6.8	✓	7.1	?
knuth	product		–	–	–	–
knuth-nosqrt	product	q, t, $-t$, $-r+t$	2.9	?	10.3	?
lcm1	divisor	x, v, $u-y$	22.6	✓	6.6	?
lcm2	divisor	y, x	21.3	✓	13.2	?
mannadiv	divisor	$y3$	13.8	✓	4.6	?
prod4br	gcd, lcm	b, $-q$, a	15.0	✓	15.4	?
prodbin	gcd, lcm	y	9.8	✓	10.4	?
ps2	pow sum	$k+-c$	5.2	✓	5.0	?
ps3	pow sum	$k+-c$	4.1	✓	8.3	✓
ps4	pow sum	$k+-c$	4.9	✓	14.1	✓
ps5	pow sum	$-x$, $k+-c$	8.2	✓	12.7	✓
ps6	pow sum	x, $k+-c$	6.4	✓	176.8	✓
sqrt1	square root	$k+-c$	6.3	✓	205.4	✓

Full RF for egcd3: $c+-v$, v, $-37\cdot b+-d$, $-293\cdot b+-d$, $-2341\cdot b+-d$, $-18725\cdot b+-d$, $-37449\cdot b+c+-d$, $10083\cdot c+-d$, $322639\cdot c+-d$, $2581111\cdot c+-d$, $-1677722\cdot b+-d$

Full RF for freire1: $-1\cdot r+12\cdot a+3\cdot k$, $-1\cdot r+19\cdot a$, $-1\cdot r+28\cdot a+3\cdot k$, $-1\cdot r+2\cdot k$, $-1\cdot r+-1\cdot a$, $-1\cdot r+-2\cdot a$, $-1\cdot r+-16\cdot a$, $-1\cdot r+25\cdot k$, $-1\cdot r+34\cdot k$, $-1\cdot r+38\cdot k$, $-1\cdot r+40\cdot k$, $-1\cdot r+44\cdot k$, $-1\cdot r+-19\cdot a$, $-1\cdot r+89\cdot k$, $-1\cdot r+105\cdot k$, $-1\cdot r+154\cdot k$, $-1\cdot r+186\cdot a$

Fig. 10. Results of applying DynamiTe to our new benchmark suite of NLA termination challenge problems. For egcd3 and geo1 the full set of ranking functions are below the table.

also available in the supplementary materials and will be submitted to SV-COMP.) The results of applying DynamiTe on this benchmark are given in Fig. 12. Out of the 39 benchmarks, DynamiTe was able to generate a recurrence set for 37 programs.

Interestingly, we found that, for non-termination, our algorithm's semantic extraction of the first candidate recurrent set from the loop condition already provides a good guess to start with. Consequently, in 34 cases, dynamic analysis was actually unnecessary because our algorithm could already use the loop condition to guide guessing for a recurrent set. This is in contrast with termination, where we don't have have any semantic information to provide a starting guess for

DynamiTe on NLA Non-termination Programs from Anant

	DynamiTe	Learning		Validation	
Benchmark	**Learned Recurrent Sets**	Time	Res	Time	Res
p1.c	$-1 >= x + -y$	?	✓	0.1	✓
p2.c	$And(y >= 2, 5 >= y, y + -z <= -5, 0 >= -x + y)$	1.8	✓	2.1	✓
p2a.c	$And(y >= 3, 3 >= -y + z, 8 >= z, x >= 6, y + -z <= -3)$	3.0	✓	30.2	✓
p3.c	$And(y + -1 \cdot z <= -7, y >= 1, 0 <= x)$	1.5	✓	1.4	✓
p4.c	$And(w + -z <= -8, 0 >= w + -x)$	2.8	✓	2.2	✓
p5.c		?	?	0.1	?
p6.c	$-5 <= z$	?	✓	0.2	✓
p7.c	$2 <= w$	0.0	✓	1.4	✓
p8.c		2.3	?	0.4	?
p9.c		1.7	?	1.6	?
p10.c		?	?	?	?
p11.c		8.0	?	3.9	?
p12.c	$And(y + -z <= -3, 6 >= z, -x + -y <= -3, y >= 3)$	19.5	✓	81.3	✓
p13.c	$And(Not(w >= 2), 0 >= w + -x)$	1.3	✓	37.8	✓
p14.c	$0 <= x$	?	✓	0.1	✓
p15.c	$And(-20 >= y, 1 <= x)$	?	✓	0.1	✓
p16.c	$And(1 <= x, Not(y <= -1))$	2.5	✓	14.7	✓
p17.c	$-1 >= x + -y$	?	✓	0.1	✓
p18.c	$-1 >= x + -y$	?	✓	0.1	✓
p19.c	$And(y >= 6, z >= 1, 0 <= x)$	3.0	✓	1.1	✓
p20.c	$And(1 == y \cdot z + -x + -z, -x + z <= -8, 2 >= -x + y, y >= 4)$	27.7	✓	25.1	✓
pfactorial.c		64.9	?	14.0	?
pinteger_log.c		?	?	?	?
pinteger_log_by_mul.c		?	?	?	?
plasso_example1.c	$And(j >= 2, -i + -j <= -6, k >= 3, -j + -k <= -9, -i + -k <= -6, i >= 0)$	4.2	✓	6.5	✓
plasso_example2.c		?	?	?	?
plasso_example3.c	$And(0 <= i, k >= 1, j >= 1)$	6.1	✓	17.7	✓
pnCr_combination.c	$1 <= nCr$	?	✓	0.4	✓
ppower.c		?	?	?	?

Fig. 11. Results of applying DynamiTe to the benchmark suite Anant of NLA non-termination problems.

rank functions. However, in 3 cases, dynamic refinement was necessary where the loop conditions are not the existing invariant assertions in the original benchmark programs and the refinement can find non-trivial conditions to construct valid recurrent sets.

8.4 Integrated Algorithm: Discriminating between Termination and Non-termination

We experimented with ProveTNT to evaluate (a) whether the algorithm is able to discriminate programs that terminate from those that non-terminate and (b) whether feedback from a failed attempt to prove termination can inform a proof of non-termination and vice-versa. For (a), we jumbled together all of the NLA benchmarks and ran the integrated algorithm on them. The results are given in Fig. 13. For these benchmarks we note the number of loops (#L). We also indicate whether a guess was made of either a recurrent set (rcr) or a ranking function (rf). ProveTNT makes an initial guess whether to pursue termination or non-termination and if the choice fails, "switches" to the opposite tack. We report the number of switches (#Sw), as well as the final validated conclusion and the total time.

For the vast majority of the examples, there are no switches, which means the initial choice (based on dynamic execution sampling the instrumented program) was was a good one, or that ProveTNT timed out before it could validate a guess. In 16 of the 77 benchmarks, a switch was made at least once. As compared with Fig. 10 (NLA Termination) and Fig. 12 (NLA Non-termination),

DynamiTe on NLA Non-termination Programs adapted from SV-COMP

Benchmark	Desc.	DynamiTe Learned Rec. Sets	Learning Time	Learning Res	Validation Time	Validation Res
bresenham1		$And(0 == (2 \cdot Y \cdot x + -X + 2 \cdot Y + -v)\%2, 0 == X \cdot y + -((2 \cdot Y \cdot x + -X + 2 \cdot Y + -v)/2))$	?	✓	0.2	✓
cohencu1	cubic sum	$-6 <= 6 \cdot n + -z$	?	✓	0.1	✓
cohencu2	cubic sum	$And(-6 == 6 \cdot n + -z, 0 == n^2 + -((-1 + -3 \cdot n + y)/3), 0 == (2 + -3 \cdot n + y)\%3)$	4.1	✓	21.5	✓
cohencu3	cubic sum	$And(-6 == 6 \cdot n + -z, 6 == y \cdot z + -18 \cdot x + -12 \cdot y + 2 \cdot z, -12 == z^2 + -12 \cdot y + -6 \cdot z)$	3.9	✓	1.3	✓
cohencu4	cubic sum	$And(6 == y \cdot z + -18 \cdot x + -12 \cdot y + 2 \cdot z, -12 == z^2 + -12 \cdot y + -6 \cdot z)$	4.0	✓	8.6	✓
cohencu5	cubic sum	$-12 == z^2 + -12 \cdot y + -6 \cdot z$	?	✓	0.2	✓
dijkstra1	square root	$1 <= 2 \cdot p + q + -r$	?	✓	0.1	✓
dijkstra2	square root	$0 == n \cdot q + -p^2 + -q \cdot r$	?	✓	6.2	✓
dijkstra3	square root	$0 == h^3 + -q \cdot (h \cdot (12 \cdot n + q + -12 \cdot r) + -16 \cdot n \cdot p + 4 \cdot p \cdot q + 16 \cdot p \cdot r)$	?	✓	6.3	✓
dijkstra4	square root	$0 == n \cdot h^2 + 4 \cdot q \cdot n^2 + -n \cdot q^2 + -8 \cdot n \cdot q \cdot r + r \cdot q^2 + 4 \cdot q \cdot r^2 + -h \cdot (h \cdot r + 4 \cdot n \cdot p + -4 \cdot p \cdot r)$	?	✓	0.4	✓
dijkstra5	square root	$0 == p \cdot h^2 + -q \cdot (h \cdot (4 \cdot n + -4 \cdot r) + -4 \cdot n \cdot p + p \cdot q + 4 \cdot p \cdot r)$	?	✓	0.3	✓
dijkstra6	square root	$0 == n \cdot q + -p^2 + -q \cdot r$	?	✓	6.2	✓
divbin1	int div	$0 == b \cdot q + -A + r$	?	✓	0.1	✓
egcd	gcd	$0 == q \cdot x + s \cdot y + -b$	?	✓	0.2	✓
egcd2	gcd	$0 == p \cdot x + r \cdot y + -a$	?	✓	0.2	✓
egcd3	gcd	$0 == q \cdot x + s \cdot y + -b$	?	✓	0.1	✓
fermat1	product	$0 == u^2 + -v^2 + -4 \cdot A + -4 \cdot r + -2 \cdot u + 2 \cdot v$	?	✓	0.2	✓
fermat2	divisor	$0 == u^2 + -v^2 + -4 \cdot A + -4 \cdot r + -2 \cdot u + 2 \cdot v$	?	✓	0.2	✓
fermat3		$0 == u^2 + -v^2 + -4 \cdot A + -4 \cdot r + -2 \cdot u + 2 \cdot v$	?	✓	0.2	✓
freire1	square root	$0 == r^2 + -a + -r + 2 \cdot x$	?	✓	0.2	✓
geo1	geo series	$-1 == x \cdot z + -x + -y$	?	✓	0.1	✓
geo2	geo series	$-1 == x \cdot z + -y \cdot z + -x$	?	✓	0.1	✓
geo3	geo series	$0 == a \cdot y \cdot z + -x \cdot z + -a + x$	?	✓	0.2	✓
hard	int div	$0 == B \cdot p + -d$	?	✓	0.1	✓
hard2	int div	$0 == B \cdot p + -d$	?	✓	0.1	✓
knuth	product		–	–	–	–
knuth-nosqrt		$0 == a \cdot k + -a \cdot t + -d \cdot k + d \cdot t$	?	✓	0.2	✓
lcm1	divisor	$0 == a \cdot b + -u \cdot x + -v \cdot y$	?	✓	0.2	✓
lcm2	divisor	$And(0 == (u \cdot x + v \cdot y)\%2, 0 == a \cdot b + -((u \cdot x + v \cdot y)/2))$	?	✓	0.2	✓
mannadiv		$0 == x2 \cdot y1 + -x1 + y2 + y3$	?	✓	0.1	✓
prod4br	gcd, lcm	$0 == a \cdot b \cdot p + -x \cdot y + q$	?	✓	0.1	✓
prodbin	gcd, lcm	$0 == a \cdot b + -x \cdot y + -z$	?	✓	0.2	✓
ps2	pow sum	$0 == y^2 + -2 \cdot x + y$	?	✓	0.1	✓
ps3	pow sum	$And(0 == (-3 \cdot y^2 + 6 \cdot x + -y)\%2, 0 == y^3 + -((-3 \cdot y^2 + 6 \cdot x + -y)/2))$	?	✓	0.5	✓
ps4	pow sum	$0 == y^4 + y^2 \cdot (1 + 2 \cdot y) + -4 \cdot x$	?	✓	0.2	✓
ps5	pow sum	$And(0 == y^5 + -((-y^3 \cdot (10 + 15 \cdot y) + 30 \cdot x + y)/6), 0 == (-y^3 \cdot (10 + 15 \cdot y) + 30 \cdot x + y)\%6)$	?	✓	0.6	✓
ps6	pow sum	$And(0 == y^6 + -((-y^2 \cdot (-1 + y^2 \cdot (5 + 6 \cdot y)) + 12 \cdot x)/2), 0 == (-y^2 \cdot (-1 + y^2 \cdot (5 + 6 \cdot y)) + 12 \cdot x)\%2)$	?	✓	0.7	✓
sqrt1	square root	$-1 == t^2 + -4 \cdot s + 2 \cdot t$	?	✓	0.2	✓
sqrt2			?	?	?	?

Fig. 12. Results of applying DynamiTe to our new benchmark suite of NLA termination challenge problems.

more timeouts occur here. However, the comparison is a little unfair: in the earlier experiments we already knew the conclusion (T versus NT) so we aimed DynamiTe toward the prize.

For ProveTNT, one pitfall is that a wrong initial choice could lead to time spent attempting to validate a ranking function, when it should be spent pursing recurrent sets (or vice-versa). A first

DynamiTe's ProveTNT Algorithm on NLA Term. & Non-term. Programs adapted from SV-COMP

Benchmark	#L	Exp.	DynamiTe Out	#Sw.	Res	Time
bresenham1	1	NT	rcr.	–	NT	15.4
bresenham1	1	T	rf.	–	T	24
cohencu1	1	NT	rcr.	–	NT	10.1
cohencu1	1	T	rf.	–	T	9.6
cohencu2	1	NT	–	–	?	24.8
cohencu2	1	T	rf.	–	?	0
cohencu3	1	NT	–	1	?	35.9
cohencu3	1	T	rf.	–	?	0
cohencu4	1	NT	–	1	?	39
cohencu4	1	T	–	1	?	105.9
cohencu5	1	NT	rcr.	–	NT	10.6
cohencu5	1	T	rf.	–	T	166.4
cohencu6	1	T	rf.	–	?	0
cohencu7	1	T	rf.	–	?	0
dijkstra1	2	NT	rf.	1	?	59.4
dijkstra1	2	T	rf.	–	T	15.2
dijkstra2	2	NT	rf.	1	?	353.1
dijkstra2	2	T	rf.	–	?	0
dijkstra3	2	NT	rf.	–	?	**T.O.**
dijkstra3	2	T	rf.	1	?	0
dijkstra4	2	NT	rf.	2	?	**T.O.**
dijkstra4	2	T	rf.	–	?	0
dijkstra5	2	NT	rf.	–	?	**T.O.**
dijkstra5	2	T	rf.	1	?	0
dijkstra6	2	NT	rf.	2	?	**T.O.**
dijkstra6	2	T	rf.	1	?	0
divbin1		T	–	–	?	2
divbin1	2	NT	–	–	?	**T.O.**
egcd	1	NT	rcr.	–	NT	85.6
egcd	1	T	rf.	–	?	0
egcd2		T	–	–	?	2
egcd2	2	NT	rcr.	–	NT	86.1
egcd3	3	NT	rcr.	–	NT	139.2
egcd3	3	T	rf.	1	NT	744.9
fermat1	3	NT	rf.	–	?	**T.O.**
fermat1	3	T	rf.	–	NT	396.2
fermat2	3	NT	rcr.	–	NT	138.3
fermat3	3	NT	rf.	–	NT	**T.O.**
freire1	1	NT	–	–	?	**T.O.**

Benchmark	#L	Exp.	DynamiTe Out	#Sw.	Res	Time
freire1	1	T	–	1	?	0
geo1	1	NT	rcr.	–	NT	48.5
geo1	1	T	rf.	1	?	0
geo2	1	NT	rcr.	–	NT	48.6
geo2	1	T	rf.	1	?	0
geo3	1	NT	rcr.	–	NT	53.2
geo3	1	T	rf.	1	?	0
hard	2	NT	rcr.	–	NT	49.2
hard	2	T	rf.	1	NT	121.1
hard2	2	NT	rcr.	–	NT	48.9
hard2	2	T	rf.	1	?	57.3
knuth		NT	–	–	?	10.6
knuth		T	–	–	?	3.4
knuth-nosqrt	1	NT	rcr.	–	NT	75.9
knuth-nosqrt	1	T	–	1	?	0
lcm1	3	NT	rcr.	–	NT	82.9
lcm1	3	T	rcr.	1	NT	172.7
lcm2	1	NT	rcr.	–	NT	86.4
lcm2	1	T	rf.	1	?	0
mannadiv	1	NT	–	1	?	197.2
mannadiv	1	T	rf.	1	?	0
prod4br	1	NT	–	1	?	198.4
prod4br	1	T	rf.	1	?	0
prodbin	1	NT	–	1	?	337
prodbin	1	T	rf.	1	?	0
ps2	1	NT	rcr.	–	NT	50.3
ps2	1	T	rf.	–	T	19
ps3	1	NT	–	–	?	**T.O.**
ps3	1	T	rf.	–	T	20.6
ps4	1	NT	rcr.	–	NT	51.4
ps4	1	T	rf.	1	?	56.5
ps5	1	NT	–	–	?	61.1
ps5	1	T	rf.	1	?	58.2
ps6	1	NT	–	–	?	61.7
ps6	1	T	rf.	1	?	29.1
sqrt1	1	NT	rcr.	–	NT	51.1
sqrt1	1	T	rf.	1	?	0
sqrt2	1	NT	rf.	1	?	**T.O.**

Fig. 13. Results of applying DynamiTe to a mix of terminating and non-terminating examples.

attack against this problem is to improve the first guess: the better the initial guess, the closer the results come to those in Fig. 10 and Fig. 12. A naïve strategy could be to add a timeout to validation. Another could be to parallelize, pursuing termination and non-termination concurrently. The downside of parallelization, is that one cannot use the output of one failed endeavor to inform the other. ProveTNT takes an alternate strategy, as dicussed in Section 6: pursue one and, if it fails, exploit the information from the counter example to expedite the alternative. In the worst case, this at least improves over running the two strategies sequentially. In Section 6 we gave two examples that demonstrate where the integrated strategy helps.

8.5 Discussion

Unlike static analysis techniques, our dynamic analysis technique executes the programs to collect data in the form of snapshots at several program locations. In some cases, the time to execute the programs and process their raw output is significant, especially on programs with high-complexity

or programs with a large number of parameters which require a large number of random inputs to maintain the data's diversity. On the other hand, when there is not enough data, overfitting may occur. In proving non-termination, overfitting can make the dynamically inferred conditions too strong to refine a recurrent set. In proving termination, overfitting may create a large number of ranking functions and overwhelm the validation tools. We also have a problem with branching in the loop body where the loop summary returned by the symbolic execution is imprecise since some branches are not taken. That imprecision affects the refinement of candidate recurrent sets.

On the other hand, our dynamic analysis has some advantages that static analysis does not have. For example, we can find a reliable set of ranking functions from known terminating traces at the beginning so we can avoid many expensive validation steps whereas static analysis techniques require many of them to refine the ranking function set from scratch.

9 RELATED WORK

Inference of nonlinear invariants. Nonlinear polynomial relations arise in many safety-and security-critical applications. For example, the Astrée analyzer, which has been applied to verify the absence of errors in the Airbus A340/A380 avionic systems [Blanchet et al. 2003], implements the ellipsoid abstract domain [Feret 2004] to represent and analyze a class of quadratic inequalities.

Rodríguez-Carbonell and Kapur [2007a,b] used abstract interpretation to infer nonlinear equalities. They first observe that a set of polynomial invariants form the algebraic structure of an ideal, then compute the polynomial invariants using Grobner basis and operations over ideals based on the structure of the program until a fixed point is reached. The approach is restricted to non-nested loops and programs with assignments and loop guards expressible as polynomial equalities. The SPEED project [Gulwani 2009; Gulwani et al. 2009] uses a numerical abstract domain [Gulavani and Gulwani 2008] to compute disjunctive and nonlinear invariants representing runtime complexity bounds. The numerical domain uses operators such as `max` to represent disjunction and constraints over various operators using inference rules to represent nonlinear operators.

The well-known dynamic invariant tool Daikon [Ernst et al. 2001, 2007] infers candidate invariants under various templates over concrete program states. The tool comes with a large set of templates which it tests against observed traces, removing those that fail, and return the remaining ones as candidate invariants. DIG [Nguyen et al. 2014a], which is used by DynamiTe focuses on numerical invariants and therefore can compute more expressive (e.g., nonlinear) numerical relations than those supported by Daikon's templates.

More recently, Yao et al. [2020] described a method for inferring invariants through a form of neural networks. The technique uses a Continous Logic Network to learn SMT formulas directly from program traces. The authors show that this approach can learn more general nonlinear invariants (equalities, inequalities, and disjunction).

There are several hybrid works in the form of guessing and checking invariants. In Sharma et al. [2013], "guess" component infers nonlinear equalities using the similar equation solving technique in DIG and the "check" component uses the Z3 SMT solver. Counterexamples from the checker are used to produce more traces to infer better invariants. The works from NumInv [Nguyen et al. 2017a] and SymInfer [Nguyen et al. 2017c] combined the dynamic analyis from DIG to infer nonlinear invariants with symbolic execution to remove spurious results.

PIE [Padhi et al. 2016] and ICE [Garg et al. 2014] also use an guess and check approach to infer invariants to prove a given specification. To prove a property, PIE iteratively infers and refined invariants by constructing necessary predicates to separate (good) states satisfying the property and (bad) states violating that property. ICE uses a decision learning algorithm to guess inductive invariants over predicates sepa- rating good and bad states. The checker produces good, bad, and "implication" counterexamples to help learn more precise invariants.

Termination. Today, numerous theories, techniques, and tools exist for proving termination and non-termination [Cook et al. 2006, 2011; Cousot and Cousot 2012; Dietsch et al. 2015; Giesl et al. 2004; Podelski and Rybalchenko 2004a]. Tools include Terminator [Cook et al. 2006], Ultimate Automizer [Ultimate 2020], HipTNT+ [Le et al. 2015], FuncTion [2020], CPAChecker [2020], and AProVE [2020]. There is even a category on termination in the Software Verification Competition (SV-COMP) [Beyer 2020]. Along the way, some have shown methods for *conditional* termination, whereby preconditions are found that specify the portion of traces that terminate [Cook et al. 2008; Le et al. 2015]. Another active line of research has focused on flavors of ranking functions, including piece-wise [Urban 2013], ordinals [Urban and Miné 2014], size-change [Lee et al. 2001], and lexicographic [Bradley et al. 2005a]. Babić et al. [2007] focused on proving termination of a restricted class of nonlinear loops, called NAW loops, which have special properties to allow their termination to be proved via analyzing the divergence of variables influencing the loop conditions.

Bradley et al. [2005a,b] focused on the class of polynomial loops from which finite different trees can be derived. However, the techniques could not work on examples with infinite difference trees. For example, to prove the termination of the program on the right, those techniques construct a difference tree whose root is the expression `100 - x * x` in the loop condition. Since the tree is infinite, they could not prove the program's termination. DynamiTe can derive the ranking function `10 - x` from concrete snapshots of that example, which is sufficient to prove its termination.

```
if x >= 0:
  while x * x <= 100:
    x = 2 * x + 1
```

A number of works have exploited dynamic information to inform termination reasoning. Nori and Sharma [2013] showed that linear regression can be used to dynamically infer bounds of program loops from test suites and these bounds imply termination. They then attempt to validate those bounds and use counterexamples to improve the precision of inference. By using the disjunctive well-foundedness in the termination proofs, DynamiTe can prove the termination of examples in [Nori and Sharma 2013] which have a disjunctive or nonlinear bound with only simple linear ranking functions. Nguyen et al. [2019] describe runtime contracts for enforcing termination, using the size-change strategy for termination.

Several static techniques are able to infer polynomial *resource bounds* [Hoffmann et al. 2011; Hoffmann and Hofmann 2010a,b]. The TiML functional language [Wang et al. 2017] allows a user to specify time complexity as types and then uses type checking to verify the specified complexity. The WISE tool [Burnim et al. 2009] uses concolic execution to search for a path policy that leads to an execution path with high resource usage.

Non-termination. Along the line of research on proving non-termination, Gupta et al. [2008] introduced a constraint solving technique to find recurrent sets of non-terminating loops. Later, Chen et al. [2014] strengthened the concept of recurrent sets to "closed" recurrent sets so that they can reduce the non-termination problem to safety proving and support more nondeterministic programs.

Cook et al. [2014] proved non-termination of nonlinear programs by soundly over-approximating the programs to nondeterministic linear programs and then using Chen et al. [2014] approach to disprove their termination. However, since the technique searches for linear recurrent sets via Farkas' lemma on the abstract linear programs, it cannot generate recurrent sets described by nonlinear equations. For example, in the benchmarks from Figure 12, there were only 5 cases where DynamiTe learned a linear recurrent set and in roughly half of the cases, DynamiTe learned a nonlinear recurrent set, which could not be found using the Cook et al. [2014] approach. Therefore, while we are able to leverage ongoing advances in nonlinear invariant generation techniques (a growing area of research), the Cook et al. [2014] approach cannot. In addition, Cook et al.

[2014] build over-approximation by using an abstract interpreter, such as Interproc, which usually does not perform well on nonlinear programs. As shown in Section 8.3, DynamiTe can prove the non-termination of all 4 ANANT benchmarks in [Cook et al. 2014] that they cannot handle.

Frohn and Giesl [2019] utilized recurrence relation solvers to replace loops whose non-termination cannot be proved by loop-free transitions in finding feasible paths to a non-terminating loop. The technique relies on recurrence relation solvers, whose supporting forms of recurrence relations are restricted. For example, the approach cannot prove the non-termination of the `p3` program in the aforementioned nonlinear ANANT benchmarks while DynamiTe can.

There are some other approaches that attempt to reason program termination and non-termination at the same time. Harris et al. [2010] introduced a technique that maintains an over- and under-approximation for alternatively proving termination or non-termination of a program. Le et al. [2014] proposed a resource logic which can uniformly specify and verify preconditions of program termination and non-termination. Later, Le et al. [2015] introduced a second-order constraint-based technique to derive termination summary in the form of that logic automatically. However, they cannot handle nonlinear programs.

10 CONCLUSION

We have shown that dynamic strategies for discovering invariants and sampling transitive closure can be incorporated with static refinement into an overall framework for proving termination or non-termination of nonlinear programs. DynamiTe [2020] is publicly available and the new benchmark suites `nla-term` and `nla-nonterm` will soon be submitted to SV-COMP. While DynamiTe already exploits concurrency by simultaneously attempting validation with CPAchecker and Ultimate, as well as within DIG, one avenue for improvement is to parallelize ProveTNT. Another direction is to explore how a dynamic invariant inference tool for heap-manipulating programs, like SLING [Le et al. 2019], can be incorporated into DynamiTe to dynamically construct termination and non-termination proofs for those programs.

ACKNOWLEDGMENT

We thank the anonymous reviewers for the helpful feedback. Timos Antonopoulos and Eric Koskinen are supported by the Office of Naval Research under Grant N00014-17-1-2787. ThanhVu Nguyen is supported by the National Science Foundation under Grant CCF-1948536 and the Army Research Office under Grant W911NF-19-1-0054.

REFERENCES

AProVE. 2020. AProVE: Automated Program Verification Environment. http://aprove.informatik.rwth-aachen.de/.

Domagoj Babić, Alan J. Hu, Zvonimir Rakamaric, and Byron Cook. 2007. Proving Termination by Divergence. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*. IEEE Computer Society, 93–102. https://doi.org/10.1109/SEFM.2007.32

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. Snowbird, Utah.

Dirk Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II (Lecture Notes in Computer Science)*, Armin Biere and David Parker (Eds.), Vol. 12079. Springer, 347–367. https://doi.org/10.1007/978-3-030-45237-7_21

Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*. Springer, 184–190.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 196–207.

Aaron R Bradley, Zohar Manna, and Henny B Sipma. 2005a. Linear ranking with reachability. In *International Conference on Computer Aided Verification*. Springer, 491–504.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005b. The Polyranking Principle. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings (Lecture Notes in Computer Science)*, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.), Vol. 3580. Springer, 1349–1361. https://doi.org/10.1007/11523468_109

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005c. Termination of Polynomial Programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.), Vol. 3385. Springer, 113–129. https://doi.org/10.1007/978-3-540-30579-8_8

Marc Brockschmidt. 2020. T2: TEMPORAL LOGIC PROVER. https://github.com/mmjb/T2.

Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 463–473.

Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 156–171. https://doi.org/10.1007/978-3-642-54862-8_11

Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Disproving termination with overapproximation. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 67–74. https://doi.org/10.1109/FMCAD.2014.6987597

Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science)*, Aarti Gupta and Sharad Malik (Eds.), Vol. 5123. Springer, 328–340. https://doi.org/10.1007/978-3-540-70545-1_32

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 415–426. https://doi.org/10.1145/1133981.1134029

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving program termination. *Commun. ACM* 54, 5 (2011), 88–98. https://doi.org/10.1145/1941487.1941509

Patrick Cousot and Radhia Cousot. 2012. An abstract interpretation framework for termination. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 245–258. https://doi.org/10.1145/2103656.2103687

CPAChecker. 2020. CPAchecker: The Configurable Software-Verification Platform. https://cpachecker.sosy-lab.org/.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. 2015. Fairness Modulo Theory: A New Approach to LTL Software Model Checking. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 49–66. https://doi.org/10.1007/978-3-319-21690-4_4

DynamiTe. 2020. Supplemental Materials. https://github.com/letonchanh/dynamite.

Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

Jérôme Feret. 2004. Static analysis of digital filters. In *European Symposium on Programming*. Springer, 33–48.

Florian Frohn and Jürgen Giesl. 2019. Proving Non-Termination via Loop Acceleration. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, Clark W. Barrett and Jin Yang (Eds.). IEEE, 221–230. https://doi.org/10.23919/FMCAD.2019.8894271

FuncTion. 2020. FuncTion: An Abstract Domain Functor for Termination. https://www.di.ens.fr/~urban/FuncTion.html.

Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*. Springer, 69–87.

Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et al. 2014. Proving termination of programs automatically with AProVE. In *International Joint Conference on Automated Reasoning*. Springer, 184–191.

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2004. Automated Termination Proofs with AProVE. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings (Lecture Notes in Computer Science)*, Vincent van Oostrom (Ed.), Vol. 3091. Springer, 210–220. https://doi.org/10.1007/978-3-540-25979-4_15

Bhargav S Gulavani and Sumit Gulwani. 2008. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *International Conference on Computer Aided Verification*. Springer, 370–384.

Sumit Gulwani. 2009. Speed: Symbolic complexity bound analysis. In *International Conference on Computer Aided Verification*. Springer, 51–62.

Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. 2009. Speed: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices* 44, 1 (2009), 127–139.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 147–158. https://doi.org/10.1145/1328438.1328459

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.

William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot and Matthieu Martel (Eds.), Vol. 6337. Springer, 304–319. https://doi.org/10.1007/978-3-642-15769-1_19

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 357–370. https://doi.org/10.1145/1926385.1926427

Jan Hoffmann and Martin Hofmann. 2010a. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science)*, Kazunori Ueda (Ed.), Vol. 6461. Springer, 172–187. https://doi.org/10.1007/978-3-642-17164-2_13

Jan Hoffmann and Martin Hofmann. 2010b. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 287–306. https://doi.org/10.1007/978-3-642-11957-6_16

Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: Dynamic Termination and Non-termination Proofs. arXiv:2010.05747 [cs.PL]

Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. 2014. A Resource-Based Logic for Termination and Non-termination Proofs. In *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings (Lecture Notes in Computer Science)*, Stephan Merz and Jun Pang (Eds.), Vol. 8829. Springer, 267–283. https://doi.org/10.1007/978-3-319-11737-9_18

Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and non-termination specification inference. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 489–498. https://doi.org/10.1145/2737924.2737993

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 788–801. https://doi.org/10.1145/3314221.3314634

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 81–92. https://doi.org/10.1145/360204.360210

George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*. Springer, 213–228.

Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 845–859. https://doi.org/10.1145/3314221.3314643

ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017a. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 605–615.

ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. 2017b. Symlnfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 804–814.

ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. 2017c. Symlnfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 804–814.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 683–693.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014a. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Transactions on Software Engineering and Methodology, to appear* (2014).

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014b. Using dynamic analysis to generate disjunctive invariants. In *Proceedings of the 36th International Conference on Software Engineering*. 608–619.

Aditya V Nori and Rahul Sharma. 2013. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 246–256.

Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

Andreas Podelski and Andrey Rybalchenko. 2004a. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings (Lecture Notes in Computer Science)*, Bernhard Steffen and Giorgio Levi (Eds.), Vol. 2937. Springer, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

Andreas Podelski and Andrey Rybalchenko. 2004b. Transition Invariants. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 32–41. https://doi.org/10.1109/LICS.2004.1319598

Enric Rodríguez-Carbonell and Deepak Kapur. 2007a. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming* 64, 1 (2007), 54–75.

Enric Rodríguez-Carbonell and Deepak Kapur. 2007b. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation* 42, 4 (2007), 443–476.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. 2013. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*. Springer, 574–592.

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: the concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 61:1–61:12. https://doi.org/10.1145/2807591.2807635

SV-COMP benchmark `nla-digbench`. 2020. SV-COMP benchmark `nla-digbench`. https://github.com/sosy-lab/sv-benchmarks/tree/master/c/nla-digbench.

Ultimate. 2020. Ultimate Automizer. https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=ltl_automizer.

Caterina Urban. 2013. Piecewise-Defined Ranking Functions. In *13th International Workshop on Termination (WST 2013)*. 69.

Caterina Urban. 2015. FuncTion: an abstract domain functor for termination. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 464–466.

Caterina Urban and Antoine Miné. 2014. An abstract domain to infer ordinal-valued ranking functions. In *European Symposium on Programming Languages and Systems*. Springer, 412–431.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 106–120. https://doi.org/10.1145/3385412.3385986