Termination and Non-termination Specification Inference

Ton Chanh Le¹  
Shengchao Qin²  
Wei-Ngan Chin¹

¹Department of Computer Science, National University of Singapore  
²School of Computing, Teesside University

{chanhle,chinwn}@comp.nus.edu.sg, s.qin@tees.ac.uk

Abstract

Techniques for proving termination and non-termination of imperative programs are usually considered as orthogonal mechanisms. In this paper, we propose a novel mechanism that analyzes and proves both program termination and non-termination at the same time. We first introduce the concept of second-order termination constraints and accumulate a set of relational assumptions on them via a Hoare-style verification. We then solve these assumptions with case analysis to determine the (conditional) termination and non-termination scenarios expressed in some specification logic form. In contrast to current approaches, our technique can construct a summary of terminating and non-terminating behaviors for each method. This enables modularity and reuse for our termination and non-termination proving processes. We have tested our tool on sample programs from a recent termination competition, and compared favorably against state-of-the-art termination analyzers.

Categories and Subject Descriptors  
D.2.4 [Software Engineering]: Software/Program Verification;  
F.3.1 [Logics and Meanings of Programs]: Specifying, Verifying and Reasoning about Programs;  
F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis

General Terms  
Languages, Theory, Verification

Keywords  
Program termination and non-termination analysis; Bi-abductive inference; Hoare logic.

1. Introduction

For the last ten years, we have seen a fruitful line of research on proving termination [9, 11, 17, 25, 26, 29, 30, 34, 37, 39] and non-termination [1, 6, 8, 24, 31, 35, 47] of imperative programs. Although these techniques for proving program termination and non-termination are often considered separately, a termination prover might deploy its own non-termination analysis mechanism to search for a feasible counterexample when a termination proof fails. TEx [25] is one of the first works to combine two different termination and non-termination proving techniques to alternatively assist each other in a whole program analysis for non-recursive programs. Nevertheless, current techniques for proving non-termination are mostly standalone techniques to existing termination proving mechanisms.

To capture the termination and non-termination behaviors of each program, Le et al. [33] has recently proposed an integrated specification logic with three temporal predicates $\text{Term}$, $\text{Loop}$ and $\text{MayLoop}$, which denote, respectively, the scenarios for definite program termination (with a lexicographic ranking measure $M$ made of a list of positive integers), definite non-termination (with an unreachable post-condition) and possible (unknown) non-termination. However, this framework currently requires temporal specifications to be given by programmers.

We propose in this paper a modular inference framework that can analyze both the termination and non-termination of each method in a program. This approach is novel in that it guides us to perform suitable case-splits on pre-conditions that lead to definite non-termination or definite termination, where possible. If a definite termination (or non-termination) case is not yet attained, we may perform a further case-split via an abductive inference [36] or decide to finish with a $\text{MayLoop}$ classification to signify an unknown outcome. For each method, our inference mechanism incrementally constructs a summary of its termination, non-termination or unknown behaviors, so that it can be reused in the inference of the remaining methods higher-up in the calling hierarchy.

To support termination and non-termination inference, we introduce unknown temporal pre- and post-predicates in our specification logic to capture termination or non-termination behaviors (that are to be resolved by our inference). For example, in Figure 1, the pair of unknown pre-predicate $U_{\text{pre}}(x, y)$ and post-predicate $U_{\text{po}}(x, y)$ in the specification of method $\text{foo}$ denotes that the termination or non-termination status of $\text{foo}$ is currently unknown. While the pre-predicate $U_{\text{pre}}(x, y)$ in the method’s precondition guides the overall inference process with suitable case-splits, the post-predicate $U_{\text{po}}(x, y)$ in its postcondition is meant to capture the reachability or unreachability of the method’s exits. This post-predicate will be strengthened to $\text{false}$ in scenarios where $\text{foo}$ is definitely non-terminating. Moreover, it can also be used to trivially determine base-case scenarios with immediate termination property. This combined use of unknown pre- and post-predicates is novel, since it allows us to modularly analyze each method (with the help of case-splits where needed) to obtain a comprehensive summary of the method’s termination and non-termination characteristics.

Our main contribution lies in a single modular bi-abductive analysis that automatically infers sufficient preconditions respectively for termination and non-termination of each method in a program. The rest of this paper is organized as follows. Section 2 introduces the novel unknown pre/post predicates and illustrates their use for inferring termination and non-termination properties of the $\text{foo}$ example. Section 3 summarizes the background on the termination

Figure 1. The $\text{foo}$ example

```c
void foo(int x, int y)
requires $U_{\text{pre}}(x, y)$
ensures $U_{\text{po}}(x, y)$:
{  
  if ($x < 0$) return;
  else foo($x + y, y$);
}
```
and non-termination reasoning with the temporal predicates. Section 4 shows how to generate a set of relational assumptions over the unknown temporal predicates. Section 5 introduces the inference algorithm to resolve the unknown predicates from their relational assumptions. Section 6 provides the experimental results. Section 7 discusses related work and Section 8 concludes our paper.

2. Overview of Our Approach

Specification Language. Le et al. [33] proposed three temporal predicates, Term, Loop, and MayLoop, to help reason about program termination and non-termination. For the current evaluation, we adopt these predicates as well as a rich underlying specification language proposed by [9] that is able to express both heap properties with separation logic (π in Figure 2) and pure (non-heap) properties with Presburger arithmetic (π in Figure 2). This logic uses a fragment of separation logic with the separation conjunction * to denote the disjointness of heap portions and the heap predicate hpred (Figure 3) to specify various data structures. Moreover, to simplify the presentation, we express a specification as a pair of pre- and post-condition (see spec in Figure 2). In our example programs, specifications will be written using the usual requires...ensures... form for better readability. Specifically for termination reasoning, we have designed the termination measure M as a (finite) list of arithmetic expressions [π] whose order is based on the lexicographic ordering < (defined below) and c:es denotes a non-empty list with c and es as its head and tail, respectively.

\[
\left\{ e_1 < e_2 \right\} \quad \left( e_1 = e_2 \land e_1 < e_2 \right) \quad e_1 < e_2 < e_3 < e_2 < e_3
\]

To facilitate termination and non-termination inference, we allow the use of an unknown temporal pre-predicate \( U_p(\pi) \) and a post-predicate \( U_p(\pi) \) in the specification language to indicate the unknown termination status of a program. The solutions of these unknown predicates would be then derived by the inference mechanism, as shown next. Note that the inferred result for each unknown pre-predicate \( U_p(\pi) \) will be of the form \( \sqrt{\pi \land \theta} \) with \( \theta \) ranging over \( \{\text{Term}, \text{Loop}, \text{MayLoop}\} \); while the inferred result for each unknown post-predicate \( U_p(\pi) \) will be in a guarded conjunction \( \left( \pi \Rightarrow \text{post} \right) \) with post being true or false. Such a guarded form is equivalent to a disjunctive form \( \sqrt{\pi \land \theta} \) when the set of guards are complete. These unknown temporal predicates are expressed in pure arithmetic domain. Heap-based properties in our logic are currently handled prior to termination analysis.

Illustrating Example. We now demonstrate how our inference mechanism derives the preconditions for termination and non-termination of method foo in Figure 2. Initially, the termination and non-termination behaviors of method foo are captured by a pair of unknown pre-predicate \( U_p(x, y) \) and unknown post-predicate \( U_p(x, y) \). Like the other known temporal predicates for termination and non-termination reasoning, these unknown predicates are part of the specification logic’s formulas and can therefore be reasoned in a similar way via Hoare-style verification. With the help of an enhanced entailment procedure, we shall prove that the precondition of each method call is always satisfied and the postcondition always holds at the end of the method body.

For example, the verification conditions (VCs) encountered by Hoare-style forward verification of method foo are:

\[
(c_1) \quad x < 0 \land U_p(x, y) \Rightarrow U_p(x, y)
\]

\[
(c_2) \quad x \geq 0 \land U_p(x, y) \Rightarrow U_p(x, y)
\]

\[
(c_3) \quad x \geq 0 \land x' = x + y \land y' = y \land U_p(x, y) \Rightarrow U_p(x', y')
\]

The first VC \( c_1 \) is obtained from the base-case scenario when the post-condition of the foo method is being proven. The second VC \( c_2 \) captures the proving of pre-condition for the recursive call, while the last VC \( c_3 \) captures the entailment proving of the postcondition of method foo in the recursive branch. These VCs capture the unknown termination behaviors of both the caller (i.e. denoted by the pair of predicates \( U_p(x, y) \) and \( U_p(x, y) \)) and the callee (i.e. denoted by \( U_p(x', y') \) and \( U_p(x', y') \)).

For these unknown predicates, we attempt to derive the strongest possible post-predicate, where possible. As we intend to capture the unreachability of each post-predicate, the strongest post-predicate in our analysis is actually false. If our inference for falsity of post-predicates fails, we denote its possible reachability by true instead and then attempt to infer the weakest pre-predicate, where possible.

The temporal pre-predicates are ordered by the following implication hierarchy \( MayLoop \Rightarrow Loop \) and \( Loop \Rightarrow \text{Term} \). Amongst them, MayLoop is considered as the strongest one, which is analogous to false in the domain of logical specification. The intuition is that MayLoop can be used to denote the termination property of any program though such a use would form a rather poor specification, which is similar to how false could be naively (and redundantly) used as the precondition for any program. On the other hand, the Loop and Term \( [\pi] \) predicates are incomparable since they denote disjoint classes of programs (i.e. definitely non-terminating vs. definitely terminating programs, respectively). Our inference thus attempts to discover the weaker Loop and Term \( [\pi] \) for its unknown pre-predicate, where possible.

From the earlier VCs, we infer three relational assumptions where unknown pre-predicate \( U_p(x', y') \) is related inductively to an earlier pre-predicate \( U_p(x, y) \) (see \( (a_1^2) \)), while unknown post-predicate \( U_p(x, y) \) is either expressed in base-case form (see \( (a_1^2) \)) or related inductively to an earlier occurrence of the post-predicate \( U_p(x', y') \) (see \( (a_1^2) \)).

\[
(a_1^2) \quad x < 0 \land \text{true} \Rightarrow U_p(x, y)
\]

\[
(a_2^2) \quad x \geq 0 \land x' = x + y \land y' = y \land U_p(x, y) \Rightarrow U_p(x', y')
\]

\[
(a_3^2) \quad x \geq 0 \land x' = x + y \land y' = y \land U_p(x', y') \Rightarrow U_p(x, y)
\]

We derive inductive definitions for these unknown predicates, in order to give the best possible interpretations to their temporal predicates. In the case of post-predicate, we attempt to determine its reachability or unreachability, so that we can immediately decide on either (base-case scenario for) termination or (inductive-case scenario for) definite non-termination. From the relational assumption \( (a_1^2) \), we can immediately infer a base-case scenario \( x < 0 \) where the foo method would terminate. The other two relational assumptions occur under a different scenario \( x \geq 0 \) which neither indicates definite termination nor definite non-termination. From these partial instantiations on the two unknown temporal predicates, we refine them to the following definitions:

\[
U_p(x, y) \equiv x < 0 \land \text{Term} \lor x \geq 0 \land U_p(x, y)
\]

\[
U_p(x, y) \equiv (x < 0 \Rightarrow \text{true}) \land (x \geq 0 \Rightarrow U_p(x, y))
\]

where two auxiliary unknown predicates are introduced for the input scenario \( x \geq 0 \). Note that Term, short for Term \( [\pi] \), is used to denote base-case termination scenario where its lexicographic ranking...
measure is trivially empty. Our unknown pre-predicate is being expressed as a disjunction on either known or unknown temporal resource constraints, while the post-predicate is being expressed as a guarded conjunction of either reachability (true), unreachability (false) or unknown. That is the two predicates are currently known for the input scenario \(x<0\) but unknown for the scenario \(x\geq0\).

As the precondition is now partially known, we could refine each \(a_i\) through a substitution with the partial definition of \(U_p(x, y)\) and \(U_p(x, y)\) to get the new relational assumptions over the unknowns:

\[
\begin{align*}
(a_1') & \quad x \geq 0 \land x'=x+y \land y'=y \land x' \neq 0 \land U_p(x, y) \Rightarrow \text{Term} \\
(a_2') & \quad x \geq 0 \land x'=x+y \land y' \neq y \land x' \neq 0 \land U_p(x, y) \Rightarrow U_p(x', y') \\
(a_3') & \quad (x \geq 0 \Rightarrow U_p(x, y)) \Rightarrow (x \geq 0 \Rightarrow U_p(x, y))
\end{align*}
\]

The relational assumption \(a_1\) describes the reachability of the base-case condition (i.e. \(x<0\)), denoted by Term, under the input scenario \(x<0\). We can thus attempt a termination proof by synthesizing a ranking function for \(x \geq 0\) but this proof fails. We then try a non-termination proof by examining \(a_1\) on unknown post-predicate to determine a condition for unreachability. Such condition must ensure that the base case is not reached in the next recursion, i.e. \(x>0\), and we refer to this as potential non-termination precondition. Since \(x>0\), the condition \(x+y\geq0\) would be a trivial potential non-termination pre-condition. However, our inference engine would discover a better (or weaker) condition, namely \(y \geq 0\), for definite non-termination with the help of abductive inference. With this, a case-split with the condition \(y \geq 0\) and its negation \(y<0\) is used to re-define the definitions for \(U_p(x, y)\) and \(U_p(x, y)\): \(y \geq 0 \land U_p(x, y)\) \(\lor\) \(y<0 \land U_p(x, y)\) \(\lor\) \(y<0 \Rightarrow U_p(x, y)\).

Consequently, the following six specialized assumptions are derived from the earlier ones \(a_2\), \(a_3\) and \(a_4\):

\[
\begin{align*}
(a_1') & \quad x \geq 0 \land x'=x+y \land y'=y \land x' \neq 0 \land U_p(x, y) \Rightarrow \text{Term} \\
(a_2') & \quad x \geq 0 \land x'=x+y \land y' \neq y \land x' \neq 0 \land U_p(x, y) \Rightarrow U_p(x', y') \\
(a_3') & \quad (x \geq 0 \Rightarrow U_p(x, y)) \Rightarrow (x \geq 0 \Rightarrow U_p(x, y))
\end{align*}
\]

The first three relational assumptions, \(a_1\) to \(a_3\), form a group which will be analyzed together for the given input scenario \(x \geq 0\). The next three relational assumptions, \(a_4\) to \(a_6\), form another group for the input scenario \(x \geq 0\). The first group of relational assumptions \(a_1\) to \(a_3\), allows us to confirm a definite non-termination scenario, since we can use \(a_3\) to determine the unreachability of its post-predicate \(U_p(x, y)\). By using the hypothesis \(U_p(x, y)\) \(\equiv\) false for both occurrences of the post-predicate \(U_p(x, y)\) in \(a_3\), we can inductively determine the falsity (or unreachability) of \(U_p(x, y)\). Note our use of inductive reasoning here which assumes the hypothesis that \(U_p(x, y)\) is unreachable under pre-condition \(x \geq 0\), in order to prove it.

The second group of relational assumptions \(a_4\) to \(a_6\), suggests us to prove the method’s termination under the pre-condition \(x \geq 0\) first, since its base case (captured by \(a_4\)) is possibly reachable under this condition. This termination scenario is confirmed, once we have derived a lexicographic ranking measure \(x\) that is bounded and would moreover decrease with each recursive invocation for the pre-predicate \(U_p(x, y)\) using \(a_4\).

As a summary of our combined analyses, we have effectively derived the following definitions for the two unknown predicates:

\[
\begin{align*}
U_p(x, y) & \equiv x<0 \land \text{Term} \lor x \geq 0 \land y<0 \land \text{Term} \\
U_p(x, y) & \equiv (x<0 \Rightarrow \text{Term}) \land (x \geq 0 \land y<0 \Rightarrow \text{Term}) \land (x \geq 0 \land y \geq 0 \Rightarrow \text{false})
\end{align*}
\]

Note how the unknown temporal predicates \(U_p(x, y)\) and \(U_p(x, y)\) are being resolved to be Loop and an unreachable false for input scenario \(y \geq 0\), respectively. In contrast, the unknown predicates \(U_p(x, y)\) and \(U_p(x, y)\) are being resolved to be Term\( [x] \) and a reachable true state for input scenario \(y < 0\), respectively.

Using the inferred predicate definitions, we can finally construct the following case-based specification \(\{20\}\) which fully captures termination and non-termination behaviors for method foo.

\[
\begin{align*}
\text{case}\{ \\
x < 0 & \rightarrow \text{requires Term ensures true;} \\
x \geq 0 & \rightarrow \text{case} \{ \\
y < 0 & \rightarrow \text{requires Term [x] ensures true;} \\
y \geq 0 & \rightarrow \text{requires Loop ensures false;} \}
\end{align*}
\]

### 2.1 Other Examples

Our termination and non-termination inference is completely automated. By allowing unknown temporal predicates into functional correctness specifications, our inference mechanism can freely leverage on prior infrastructures to (i) handle a wider class of programs, and to (ii) improve the accuracy of the inference results. Note that prior safety specifications for the analyzed methods might be manually given or be automatically derived by other inference mechanisms, but they are orthogonal to our current proposal.

We list below some interesting examples to demonstrate how our inference mechanism works with programs that already have some safety specifications.

#### Nested Recursion

For examples with nested recursion, such as the Ackermann function and the McCarthy 91 function in Figure 3, some knowledge on its output may be crucial for the inference of their termination and non-termination properties. Without any specification, our inference mechanism returns incomplete summaries on the terminating and non-terminating behaviors of these two functions. The result for the Ackermann function is:

\[
\begin{align*}
\text{case}\{ \\
m=0 & \rightarrow \text{requires Term ensures true;} \\
m<0 \land n<0 & \rightarrow \text{requires Loop ensures false;} \\
m>0 \land n \geq 0 & \rightarrow \text{requires MayLoop ensures true;}
\end{align*}
\]

While the inference shows that this function is terminating when \(m=0\) (base case) or non-terminating when \(m<0 \lor n<0\), it cannot prove the termination of the function under the input scenario \(m>0 \land n \geq 0\) since the value of the second argument in the last recursive call is unknown (or unbounded). However, with the stronger specification given in Figure 3a, with a lower bound \(\text{res} \geq n+1\) on the function’s returned value, denoted by \(\text{res}\), our inference mechanism can replace \(\text{MayLoop}\) in scenario \(m>0 \land n \geq 0\) by \(\text{Term}\) \([m, n]\) where \(\text{[m, n]}\) is a valid lexicographic ranking function. Similarly, without specification, the inference only shows that the McCarthy 91 function terminates in its base case when \(n \geq 100\). However, with the specification given in Figure 3b, our inference can prove that the function terminates for all inputs.

While our termination inference mechanism does not directly infer postconditions, it has been made to work with other automated postcondition inference sub-systems, such as.\[23, 40\]. Such post-condition inference sub-systems are orthogonal to our proposal, and can be leveraged to provide a more comprehensive solution for fully automated termination and non-termination inference.
Heap-Manipulating Programs. Our inference mechanism can be readily integrated into existing verification frameworks (such as [9], or even shape inference system [32]) that reason about safety properties of heap programs via separation logic [41]. Such an extension can be applied to help prove the termination and non-termination of heap-manipulating programs.

Figure 4 shows the specification and implementation (for the method append) of the method of two linked lists x and y. With the separation conjunction + and the points-to operator → of separation logic, the heap predicate \( \text{lsseg}(\text{root} \cdot q, n) \) represents a list segment from root to q with n elements. This predicate can then be used in the declarations of other predicates, such as \( \text{cll}(\text{root}, n) \) for circular lists. Using these two predicates, we can capture two safety specifications of append in Figure 4.

In the first scenario when the input x is a null-terminating list with size n, our inference mechanism is able to show that the method append always terminates with the ranking function \([n]\). In the second scenario where x is a circular linked list, our inference can show that append is definitely non-terminating, after confirming (by induction) that its postcondition can be strengthened to \( \text{false} \). These examples highlight the modular nature of our non-termination and termination inference mechanism, which can be built on top of other inference mechanisms.

3. Technical Background

So far we have illustrated a unified specification logic with three known temporal predicates: Term \([\rho] \), Loop and MayLoop. Semantically, these predicates can be defined using resource capacities (on lower and upper bounds) of execution length, i.e. Term \([\rho] =_df \text{RC}(0, f(\rho)) \), Loop \( =_df \text{RC}(\infty, \infty) \), and MayLoop \( =_df \text{RC}(0, \infty) \). The resource predicate RC\((L, U)\) specifies a resource capacity with a lower bound \(L\) and an upper bound \(U\). It is satisfied by each program state whose resource capacity \((t, \mathbb{U})\) is subsumed by \((L, U)\), i.e. \(L \leq t\) and \(u \leq \mathbb{U}\). Note that the function \(f(\rho)\) obtains a finite bound through an order-embedding of \(\mathbb{P}\) into naturals.

Verification conditions involving these temporal predicates can be discharged by a resource consumption entailment \(\vdash\), that is used to account for (lower and upper bound) resources that are utilized by each code fragment. Such entailment can be used to analyze termination or non-termination property for some given method via resource reasoning. Given the temporal constraint \(\theta_t\) associated with the current program state \(p\) and the temporal resource constraint \(\theta_r\) (of some code fragment that must be executed), the entailment \(\rho \land \theta_t \vdash_\rho \theta_r\) checks whether the execution resource required by constraint \(\theta_r\) can be met by the execution resource of constraint \(\theta_t\) or not. If it succeeds, the entailment will return the remaining execution resource that is denoted by residue \(\theta_\rho\).

In terms of the actual execution capacity, this consumption entailment can be formalized by the following rule:
A Core Imperative Language

**Definition 1.** The temporal relational assumption $\mathcal{R}$ in the residue of a temporal entailment $\rho \land \bigwedge_i \psi_i \ni \theta_i \vdash \theta_i \circ \mathcal{R}$ can be defined as follows:

(i) $\mathcal{R} \equiv \text{true}$, if both $\theta_0$ and $\theta_i$ are known predicates from $\{\text{Term}[\cdot], \text{Loop}, \text{MayLoop}\}$.

(ii) $\mathcal{R} \equiv \rho \land \bigwedge_i \psi_i \ni \theta_i$, if $\theta_i$ is an unknown post-predicate.

(iii) $\mathcal{R} \equiv \rho \land \theta_0 \rightarrow \theta_i$, otherwise.

This temporal entailment can be integrated into an entailment system with frame $\Psi \vdash \Phi \rightarrow \Psi_r$, to obtain a new entailment procedure of the form $\Psi \vdash \Phi \rightarrow (\Psi_r, \mathcal{S})$, that also captures in its residue the set of relational assumptions $\mathcal{S}$ generated by the temporal sub-entailments. The rules to discharge entailments of logic formulas with disjunctions are:

$$
\Psi = \exists i. (\rho_i \land \bigwedge_j \theta_j) \rightarrow \bigwedge_i (\rho_i \land \bigwedge_j \theta_j) \vdash \Phi \rightarrow (\Psi_r, \mathcal{S})
$$

4. Generating Temporal Assumptions

In this section, we show how our new entailment procedure is incorporated into Hoare logic to generate a set of relation assumptions over the unknown temporal constraints.

**Language.** To formalize this task, we provide a core language (Figure 5) with usual constructs, such as data structure declaration $\text{tdcl}$, method declaration $\text{meth}$, method call, assignment. This core language does not include the while-loop construct, as it assumes an automatic translation of loops into tail-recursive methods. For int type, we assume the use of arbitrary precision integers.

A method declaration consists of a specification with pre- and post-condition and its body. Primitive/library methods do not have a body and must have their specifications (including termination) pre-declared. For termination and non-termination inference, a pair of unknown pre- and post-predicate are automatically added into the specification of each method whose termination status is unknown.

**Hoare rules.** To support inference, Hoare judgment is formalized as $\vdash (\Psi_{\text{pre}}, \mathcal{S}, \emptyset) \ni e \rightarrow (\Psi_{\text{post}}, \mathcal{S})$, where $\mathcal{S}$ is a generated set of temporal assumptions. For illustration, we show the rule for method call:

$$
\text{TNT-CALL} \quad \text{t0 mm}(\tau, \{e\}; \{\Psi_{\text{pre}}, \mathcal{S}_{\text{pre}}\} \ni e \rightarrow \text{Prog} \\
\Psi \vdash \Psi_{\text{pre}} \rightarrow (\Phi, \mathcal{S}_1) \quad \Psi_r = \Phi \circ \mathcal{S}_{\text{post}} \quad \mathcal{S}_2 = \text{filter} (\mathcal{S}_1)
$$

To facilitate the termination inference, at method calls, we collect only nontrivial assumptions of unknown temporal constraints. We list below trivial relational assumptions, which will be removed by the function $\text{filter}$ as shown in the rule $\text{TNT-CALL}$.

Firstly, the relational assumption $\rho \land \theta_0 \rightarrow \theta_i$ is trivial for any $\theta_0$ and $\theta_i$ if the context $\rho$ is unsatisfiable. Secondly, the assumptions $\rho \land \bigwedge_i \psi_i \ni \theta_i \rightarrow \theta_i \circ \mathcal{R}$ are trivially valid for any program state $\rho$ because the constraints $\text{Loop}$ and $\text{MayLoop}$ can accept any temporal constraints in the RHS [3]. Finally, $\rho \land \theta_0 \rightarrow \text{Term}\mathcal{M}$ is trivial if the call $n$, whose termination is denoted by the temporal constraint $\text{Term}\mathcal{M}$, and the call $m$ are not mutually recursive.

Note that assumptions of the form $\rho \land \psi_i \rightarrow \psi_j (\tau) \ni \tau \ni \psi_i \circ \mathcal{R}$ are not trivial for any caller $m$ and callee $n$. However, when $m$ and $n$ are in two different SCC groups, this kind of assumptions can be avoided if we do a bottom-up verification and inference in which the (non-)termination of the callee $n$ is inferred and the unknown $\psi_j (\tau) \ni \psi_i \circ \mathcal{R}$ is instantiated before the caller $m$ is processed.

For each method declaration, we collect a set of relational assumptions $\mathcal{S}$ during the verification of its body, and another set of relational assumptions $\mathcal{T}$ at the method’s exit points via the entailment for proving the post-condition, as shown below:

$$
\vdash (\Psi_{\text{pre}}, \emptyset) \ni e \rightarrow (\Psi_{\text{post}}, \emptyset) \ni \text{TNT-METH} \quad \text{t0 mm}(\tau; \{e\}; \{\Psi_{\text{pre}}, \mathcal{S}_{\text{pre}}\} \ni e \rightarrow \text{Prog} \\
\Psi \vdash \Psi_{\text{pre}} \rightarrow (\Phi, \mathcal{S}_1) \quad \Psi_r = \Phi \circ \mathcal{S}_{\text{post}} \quad \mathcal{S}_2 = \text{filter} (\mathcal{S}_1)
$$

The termination and non-termination inference engine is invoked when a whole group of mutually recursive methods are verified and their sets of relational assumptions are collected, as shown in the rule $\text{TNT-INF}$ below.

$$
\vdash (\Psi_{\text{pre}}, \emptyset) \ni e \rightarrow (\Psi_{\text{post}}, \emptyset) \ni \text{TNT-METH} \quad \text{t0 mm}(\tau; \{e\}; \{\Psi_{\text{pre}}, \mathcal{S}_{\text{pre}}\} \ni e \rightarrow \text{Prog} \\
\Psi \vdash \Psi_{\text{pre}} \rightarrow (\Phi, \mathcal{S}_1) \quad \Psi_r = \Phi \circ \mathcal{S}_{\text{post}} \quad \mathcal{S}_2 = \text{filter} (\mathcal{S}_1)
$$

The $\text{solve}(\mathcal{M})$ procedure infers definitions for unknown temporal predicates and will be depicted in detail next.

5. Inferring Termination and Non-Termination

This section is devoted to the $\text{solve}$ procedure used to infer the definitions for the unknown pre/post-predicates, based on the set of relational assumptions generated by Hoare-style verification. The overall algorithm is shown in Figure 6.

In this algorithm, $\Theta$ is used to store the set of definitions inferred thus far for the unknown temporal predicates. Since a key idea of our inference mechanism is case analysis that incrementally separates the terminating and non-terminating behaviors of the analyzed methods, the definition for each unknown predicate might be split into multiple scenarios, for which termination is either known or unknown.

**Definition 2 (Unknown Temporal Predicates).** During the inference process, the definitions for a pair of unknown pre-predicate
After updating the set of relational assumptions (line 7), our algorithm refines the definitions of unknown predicates collected by rule [TNT-METH], namely

1. Pre-assumptions \( \mathcal{S} \) collected when proving preconditions at method calls. They can be used to infer (i) ranking functions for termination proving, and (ii) temporal reachability graph that guides our search for proving termination vs. non-termination.

2. Post-assumptions \( \mathcal{T} \) collected when proving postconditions contain information about unknown post-predicates. They can be used to infer (i) termination base cases, (ii) inductive unreachability to prove non-termination or (iii) new conditions for the case analysis.

The algorithm in Figure 6 first derives the base case of each analyzed method (line 3), and then refines the definitions of unknown temporal predicates in \( \Theta \) with these newly inferred cases (line 5). After updating the set of relational assumptions (line 7), our algorithm (re-)builds the temporal reachability graph \( \mathcal{G} \) from the latest \( \mathcal{S} \) (line 8).

For each \( \text{scc} \) of the graph \( \mathcal{G} \) in the bottom-up topological order, the analysis attempts to prove either termination or non-termination or to infer new cases for case-splitting and then updates the set \( \Theta \) with the inferred result (line 10). If every unknown temporal predicate corresponding to the current \( \text{scc} \) is resolved into known predicates, the inference continues with the next \( \text{sccs} \) after updating the post-assumptions in \( \mathcal{T} \) (line 13) and the graph \( \mathcal{G} \) (line 14) with the new inferred known predicates. Otherwise, it restarts the core algorithm (line 11) with the updated \( \Theta \), whose elements have been refined into new sub-cases.

The algorithm halts when every unknown predicate has been resolved or the number of iterations reaches the maximum \( \text{MAXITER} \) pre-set by users. In the latter case, the remaining unknown predicates in \( \Theta \) will be marked as MayLoop by an auxiliary procedure finalize. Next we will explain each inference step in some detail.

5.1 Inferring Base Case Termination

Identifying the conditions for base-case termination is an important first step before any other analyses. Formally:

**Definition 3** (Base Case Pre-Condition). Each base case termination precondition of a method must satisfy the following three conditions:

(i) Its method’s exit is reachable from it.

(ii) No mutually recursive method call is met in executions starting from this pre-condition.

(iii) All other method calls encountered from this pre-condition must have been proven to terminate.

While a syntactic-based approach that identifies base-case termination from its control-flow may be sufficient, we propose a semantics-based approach which infers a method’s base case precondition from the two sets of assumptions \( \mathcal{S} \) and \( \mathcal{T} \) collected from the method, as follows:

\[
\begin{align*}
\rho & = \bigvee \{ (\rho) \} \mid \rho \land \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \Rightarrow \theta_1 \in \mathcal{S} \\

\rho & = \bigvee \{ (\rho) \} \mid \beta \land \text{true} \Rightarrow \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \in \mathcal{T} \\

\text{_syn_base}(\mathcal{S}, \mathcal{T}) & = \rho \land \neg \rho
\end{align*}
\]

where \( \rho \{ \mathcal{T} \} \equiv \exists \{ \mathcal{FV}(\rho) \} \cdot \rho \). Using our running example, we have \( \mathcal{S} = \{ a_2^0 \} \) and \( \mathcal{T} = \{ a_1^0, a_2^0 \} \):

\[
\begin{align*}
(a_1^0) \ & x < 0 \land \text{true} \Rightarrow \mu_{\mathcal{S}}(x, y) \\
(a_2^0) \ & x \geq 0 \land x' = x + y \land y' = y \land \beta \Rightarrow \mathcal{U}_{\mathcal{S}}(x', y') \\
(a_3^0) \ & x \geq 0 \land x' = x + y \land y' = y \land \text{true} \land \mathcal{U}_{\mathcal{S}}(x', y') \Rightarrow \mu_{\mathcal{S}}(x, y).
\end{align*}
\]

Each post-assumption \( \beta \land \text{true} \Rightarrow \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \in \mathcal{T} \), whose antecedent does not contain any unknown post-predicate, capture a potential base-case termination condition. Due to over-approximation, the actual base-case condition (over the method’s parameters \( \mathcal{T} \)) must be formed by such conditions (\( \mathcal{V} \beta \)), conjointed with the negation of contexts (\( \neg \rho \)) for the recursive calls. By identifying the base-case condition in \( \{ a_1^0 \} \) and conditions for recursive pre-approximation in \( \{ a_2^0 \} \), we can precisely infer \( \text{syn_base}(\mathcal{S}, \mathcal{T}) \equiv x < 0 \land \neg (x \geq 0) \).

With the inferred base case \( \beta \equiv \text{syn_base}(\mathcal{S}, \mathcal{T}) \) (line 4), we can now invoke the procedure refine_base (line 5) to refine (or specialize) the unknown predicates \( \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \) and \( \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \), before updating their definitions in \( \Theta \) (via the operator \( \oplus \)) as shown below.

\[
\begin{align*}
\bigvee \mu_i & \equiv \neg \beta \\
\Delta_{\mathcal{S}} & = \{ \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \equiv \bigvee (\mu_i \land \mathcal{U}_{\mathcal{S}}(\mathcal{T})) \} \land \text{Term} \\
\Delta_{\mathcal{P}} & = \{ \mathcal{U}_{\mathcal{P}}(\mathcal{T}) \equiv \bigvee (\mu_i \Rightarrow \mathcal{U}_{\mathcal{P}}(\mathcal{T})) \} \\
\Omega & = \bigcup \{ \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \equiv \mathcal{U}_{\mathcal{P}}(\mathcal{T}) \}
\end{align*}
\]

**5.2 Specializing Relational Assumptions**

Whenever some unknown predicates in \( \Theta \) receive new definitions, our inference algorithm will update its sets of relational assumptions with the procedure \text{spec_relass}. Its first parameter is a set of relational assumptions. Its second parameter \( \Theta \) contains the definitions of unknown predicates.

For each relational assumption with unknown predicates, the procedure \text{spec_relass} finds the current definitions of these unknown predicates in \( \Theta \) and substitutes them directly into the assumption. As the definition of each unknown predicate consists of exclusive and complete guards, we can further split each substituted assumptions into multiple specialized assumptions. We show below just one example where \text{spec_relass} is called with a new pre-approximation with two unknown predicates.

\[
\begin{align*}
\mathcal{U}_{\mathcal{S}}(\mathcal{T}) & \equiv \bigvee_{i=1}^{m} (\rho_{\mathcal{S}} \land \theta_{\mathcal{S}}^i) \in \Theta \\
\mathcal{U}_{\mathcal{P}}(\mathcal{T}) & \equiv \bigvee_{i=1}^{m} (\theta_{\mathcal{S}} \land \theta_{\mathcal{P}}^i) \in \Theta \\
\mathcal{C} & = \{ \rho_{\mathcal{S}} \land \theta_{\mathcal{S}}^i \land \theta_{\mathcal{P}}^i \} \mid 1 \leq i \leq m \\
\text{spec_relass}(\rho \cup \mathcal{U}_{\mathcal{S}}(\mathcal{T}) \cup \mathcal{U}_{\mathcal{P}}(\mathcal{T}) \cup \Theta, \Theta) & = \mathcal{C} \cup \text{spec_relass}(\Theta, \Theta)
\end{align*}
\]

For our running example, the relational assumption \( \{ a_2^0 \} \) was specialized by its earlier partial definition into two more specialized assumptions: \( \{ a_2^0 \} \) and \( \{ a_2^0 \} \).
5.3 Resolving Temporal Reachability Graph

The core of our inference algorithm (in Figure 7) incrementally solves the unknown predicates present in the (specialized) relational pre-assumptions. If its attempt fails, it would also derive conditions for the next case analysis. This core algorithm uses a reachability graph \( G \), constructed from pre-predicates in \( \mathcal{S} \), to guide its proof search. Formally:

**Definition 4** (Temporal Reachability Graph). Given a set of pre-assumptions \( \mathcal{S} \), a temporal reachability graph \( G = (V, E) \) is constructed from a set of vertices \( V \) and a set of labeled edges \( E \), as follows. For each pre-assumption \( \rho \in \mathcal{S} \), we add two vertices \( \rho_0 \) and \( \rho_1 \) into \( V \) and an edge \( \rho_0 \prec \rho_1 \) labeled by \( \rho \) into \( E \).

\[
G = (V, E)
\]

For example, the two graphs \( G_1 \) and \( G_2 \) are built for the inference of the running example. \( G_1 \) is constructed from pre-assumptions \((a_{2b}), (a_{2a})\) and \((a_{1a}), (a_{1b})\) obtained after base case inference. The edges of \( G_1 \) are labeled by \( \rho(a_{2b}) \) and \( \rho(a_{2a}) \), respectively. Similarly, the graph \( G_2 \) is constructed from pre-assumptions \((a_{2a}), (a_{1a})\) and \((a_{1b})\).

Our core formulation first partitions \( G \) into strongly connected components \( \mathcal{SCC} \), e.g., dashed boxes in \( G_1 \) and \( G_2 \), where each unknown temporal predicate denotes an unknown behavior. Moreover, this unknown predicate is mutually dependent on the other predicates in the same \( \mathcal{SCC} \). Using a bottom-up approach, the inference mechanism processes each \( \mathcal{SCC} \) in a topologically sorted order. With this approach, termination and non-termination proofs for phase-change programs [12] and that for mutual recursion are easily supported.

**Definition 5** (\( \mathcal{SCC} \)’s successors). Given a graph \( G \), the outside successors of a \( \mathcal{SCC} \) in \( G \) is the set of all successors of any vertex in this \( \mathcal{SCC} \) but excluding the itself.

\[
\text{succ}(v, G) = \bigcup \{\text{succ}(v, G) \mid v \in \mathcal{SCC}\} \setminus \text{succ}(v, G)
\]

where \( \text{succ}(v, G) \) returns all successors of the vertex \( v \).

This constraint indicates that the ranking function \( r_i(T_i) \) is bounded and decreasing across a (mutually) recursive method call under the call context \( \rho \). For example, the constraint generated from the middle \( \mathcal{SCC} \) in \( G_2 \) is

\[
\forall x, y. x \geq 0 \land y > 0 \land x + y \geq 0 \land y < 0 \Rightarrow r(x, y) > r(x', y') \land r(x, y) > y \geq 0
\]

which is then solved by \( \text{syn.rank} \) to obtain the ranking function \( r(x, y) = x \). The method \( \text{syn.rank} \) [line 31] solves the generated constraints by applying Farkas’ lemma on them to obtain another set of constraints over their unknown coefficients, which can be solved by a nonlinear solver, such as [28], to get the actual values of these.
If the proof succeeds for all pre-predicates in \( \text{scc} \) (signified by \( r \) in line 40), we mark the unknown termination status as definitely non-terminating. This procedure thus refines, where possible, each unknown pre-predicate as \( U_p(\varpi) \equiv \text{Loop} \) and its post-predicate as \( U_p(\varpi) \equiv \text{false} \) (line 41) and updates \( \Theta \) before returning (true, \( \Theta \)).

For our running example, \( (a'_2) \) and \( (a'_3) \) from \( T \) would cause prove_nonterm(scc, \( T \), \( \Theta \)) to return false, but provide an additive condition \( \gamma \geq 0 \) that facilitates case-splitting (see next subsection). In contrast, \( (a'_3) \) would be used to show that \( U_p^{\prime}(x, y) \) is inductively false (or unreachable).

5.6 Case-Splitting

If non-termination proving fails, the method abd_inf abductively infers new sub-conditions from the failed proof to refine the potential non-termination condition by case-split.

In the case \( t \equiv \rho \land \text{false} \Rightarrow (\mu \Rightarrow U_p(\varpi)) \), if the proof fails, i.e. \( \rho \land \mu \Rightarrow \text{false} \), \( \text{abd} \_\text{inf}(t) \) simply returns \( \{ \text{true} \} \) as any condition that makes the entailment to hold would contradict with the antecedent \( \rho \land \mu \).

If \( t \equiv \rho \land (\eta_i \Rightarrow \text{false}) \land (\mu_j \Rightarrow U_p(\varpi)(\eta_j)) \Rightarrow (\mu \Rightarrow U_p(\varpi)) \), and the proof fails, i.e. \( \rho \land (\eta_i \Rightarrow \text{false}) \land (\mu_j \Rightarrow U_p(\varpi)(\eta_j)) \Rightarrow (\mu \Rightarrow U_p(\varpi)) \), that is, if the potential non-termination condition \( \mu \) of the caller is strengthened by \( \eta_i \), then (the potential) non-termination condition \( \beta_k \) of a callee is satisfied.

For each condition \( \beta_k \), the solution \( \alpha_k \equiv \beta_k \) is a trivial but the weakest solution for \( \alpha_k \). For a more effective case-split, we aim to derive a stronger additive condition \( \alpha_k \). By the same constraint-based approach used for the ranking function synthesis, we assume the template \( \alpha_k \equiv c_0 + \sum_{i=1}^{m} c_i v_i \geq 0 \), where \( v_1, \ldots, v_n \equiv \text{true} \) and \( \alpha_k \) from unknown coefficients. We might solve these unknown coefficients with additional optional constraints, e.g. the number of zero-coefficients is maximum, so that we can obtain a better solution with minimum number of program variables.

Given a set of collective additive conditions \( C \), the procedure \( \text{abd} \_\text{inf} \) (line 32) refines the pair of \( \{U_p(\varpi), U_p(\varpi)\} \) with these new sub-cases for the update of \( \Theta \).

\[
\text{split}(C) = \{ (\mu_j)^{m+1}, \mu_{m+1} = \neg \mu_1 \land \ldots \land \neg \mu_m \}
\]

As the conditions in \( C \) might be overlapping, we use the function \( \text{split} \) defined below to partition these conditions into the new set of mutually exclusive conditions \( \{ (\mu_j)^{m+1} : \forall \gamma \equiv \bigvee (\mu_j) \} \). We also add into the new set the condition \( \mu_{m+1} = \neg \mu_1 \land \ldots \land \neg \mu_m \), if it is satisfiable, to cover the missing case, so that \( (\mu_j)^{m+1} \) is complete. Note \( \text{split}(\{} \) = \{ \}.

- For \( \alpha \equiv \rho \land (\eta_i \Rightarrow \text{true}) \land (\mu_j \Rightarrow U_p(\varpi)(\eta_j)) \Rightarrow (\mu \Rightarrow U_p(\varpi)) \), we have \( U_p(\varpi) \equiv \text{false} \) if and only if \( \rho \land \mu \) is unsatisfiable. The proof succeeds and \( \text{abd} \_\text{inf}(t) \) returns \{true\} if \( \rho \land \mu \)

6. Experiments

We have implemented the proposed inference mechanism on top of HipTNT\footnote{https://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/plus/}, an existing verification system that can verify both termination and non-termination specifications given by users. The new inference system HipTNT+ and this paper’s artifacts are available for both online use and download at
7. Related Work

Over the last decade, there has been a large body of work on proving program termination. Most of these termination provers, such as TERMINATOR \cite{12} and its successor T2 \cite{15}, ARMC \cite{69}, TAN \cite{29} and ULTIMATE \cite{26}, either show that a program terminates for all (given) inputs or return a counterexample to termination upon the failure of termination proofs. However, due to the incompleteness of termination-based techniques, these provers cannot guarantee that every returned counterexample (from failed termination proofs) leads to a definitely non-terminating execution. Thus, each tool might deploy a separate non-termination proving technique to prove that the counterexample is feasible. Also, each such counterexample is only an under-approximation of its program execution, so that it does not capture the wider scenarios for non-terminating behaviors of the analyzed program.

We have also seen much related work on proving program non-termination, e.g. \cite{1,8,23,31,35,47}. Non-termination provers, such as TNT \cite{24} and INVEL \cite{47}, attempt to disprove program termination by searching for some initial configurations that act as witnesses for non-termination. To find a wider class of non-termination bugs, these approaches attempt to discover sufficient pre-conditions for non-termination. Nevertheless, since non-termination proving techniques are also incomplete, the analyzed program is not guaranteed to terminate under the complement of the inferred pre-condition for non-termination.

The dual problem of conditional termination, first addressed in \cite{14}, identifies initial configurations that ensure termination. In \cite{14}, such termination preconditions are derived from potential ranking functions, which are bounded but not decreasing. Later, the tools of FLATA \cite{3} and ACABAR \cite{19} infer the sufficient precondition for termination from (the negation on over-approximation of) the set of initial states from which the program might not terminate. However, FLATA differs from ACABAR by limiting itself to classes of loops with restricted forms in which the precise non-termination conditions are definable.

8. Conclusion

We have proposed a modular inference framework for program termination and non-termination. By incorporating unknown pre/post-predicates into specification logic for termination reasoning, our proposed framework employs a Hoare-style forward verification to collect a set of relational assumptions to help soundly discover termination and non-termination properties. Our technique analyzes program termination and non-termination at the same time, and constructs a summary of these behaviors for each method. This enables better modularity and reuse for our proving processes. Furthermore, it is integrated with a verification system allowing us to use partial specification and to re-check our inference outcome. As seen in the experiments over SV-COMP 15 benchmarks, our tool compares favorably against current state-of-the-art termination analyzers.

We shall now discuss two current limitations of our tool. Firstly, our tool handles non-deterministic inputs by identifying conditional statements that directly depend on non-deterministic values. Each such non-deterministic conditional is marked as non-terminating if either of its two branches is non-terminating. This works for most examples, but is less general than the proposal in \cite{8}. Secondly, while we support lexicographic linear ranking functions (LLRF) in

---

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Benchmark & crafted & crafted-lit & numeric & memory-alloca & Total \\
\hline
\hline
APRoeVE & 20 & 15 & 14 & 37.3 & 106 \\
ULTIMATE & 21 & 15 & 1 & 231.0 & 115 \\
HITN+ & 19 & 13 & 7 & 0 & 35.1 \\
\hline
Total & 61 & 30 & 28 & 2 & 729.0 \\
\hline
\end{tabular}
\caption{Figure 10. Termination Outcomes on SV-COMP 15’s Benchmarks}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Benchmark & Total & Y & N & U & T/O \\
\hline
T2 (rev. 051de80) & 161 & 30 & 28 & 2 & 729.0 \\
HITN+ & 170 & 30 & 21 & 2 & 204.3 \\
\hline
\end{tabular}
\caption{Figure 11. Comparison on Loop-based Integer Programs}
\end{table}
our termination reasoning, we cannot handle programs that critically depend on Ramsey’s theorem \cite{29} or those that are based on size-change principles \cite{36} but do not have LLRF counterpart. It would be interesting to explore future extensions to our specification logic to better support non-determinism and these other kinds of termination proofs.

Acknowledgements: We thank the reviewers for their insightful suggestions. We are grateful to Long H. Pham, Quang-Trung Ta, Quang Loc Le, Andreea Costea and Zhiuhong Cai for their help in preparing HirTNT+ for the SV-COMP 2015 competition. This work is supported by the MoE Tier-2 grant MOE2013-T2-2-146.

References