Proving Conditional Termination

Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,⋆],
Andrey Rybalchenko[3,⋆⋆], and Mooly Sagiv[2]

[1] Microsoft Research
[2] Tel Aviv University
[3] MPI-SWS

Abstract. We describe a method for synthesizing reasonable underapproximations to weakest preconditions for termination—a long-standing open problem. The paper provides experimental evidence to demonstrate the usefulness of the new procedure.

1 Introduction

Termination analysis is critical to the process of ensuring the stability and usability of software systems, as liveness properties such "Will `Decode()` always return back to its call sites?" or "Is every call to `Acquire()` eventually followed by a call to `Release()`?" can be reduced to a question of program termination [8, 22]. Automatic methods for proving such properties are now well studied in the literature, *e.g.* [1, 4, 6, 9, 16]. But what about the cases in which code only terminates for *some* inputs? What are the preconditions under which the code is still safe to call, and how can we automatically synthesize these conditions? We refer to these questions as the *conditional termination* problem.

This paper describes a method for proving conditional termination. Our method is based on the discovery of *potential ranking functions*—functions over program states that are bounded but not necessarily decreasing—and then finding preconditions that promote the potential ranking functions into valid ranking functions. We describe two procedures based on this idea: PreSynth, which finds preconditions to termination, and PreSynthPhase, which extends PreSynth with the ability to identify the phases necessary to prove the termination of phase-transition programs [3].

The challenge in this area is to find the *right precondition*: the empty precondition is correct but useless, whereas the *weakest precondition* [13] for even very simple programs can often be expressed only in complex domains not supported by today's tools (*e.g.* non-linear arithmetic). In this paper we seek a method that finds useful preconditions. Such preconditions need to be weak enough to allow interesting applications of the code in question, but also expressible in

⋆ Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.

⋆⋆ Supported in part by Microsoft Research through the European Fellowship Programme.

the subset of logic supported by decision procedures, model checking tools, etc. Furthermore, they should be computed quickly (the weakest precondition expressible in the target logic may be too expensive to compute). Since we are not always computing the *weakest* precondition, in this paper we allow the reader to judge the quality of the preconditions computed by our procedure for a number of examples. Several of these examples are drawn from industrial applications.

Limitations. In this paper, we limit ourselves to the termination property and to sequential arithmetic programs. Note that, at the cost of complicating the exposition, we could use known techniques (*e.g.*, [2] and [8]) to extend our approach to programs with heap and ω-regular liveness properties. Our technique could also provide assistance when analyzing concurrent programs via [10], although we suspect that synthesizing *environment* abstractions that guarantee thread-termination is a more important problem for concurrent programs than conditional termination.

Related work. Until now, few papers have directly addressed the problem of automatically underapproximating weakest preconditions. One exception is [14], which yields constraint systems that are non-linear. The constraint-based technique in [5] could also be modified to find preconditions, but again at the cost of non-linear constraints. In contrast to methods for underapproximating weakest preconditions, techniques for weakest *liberal* preconditions are known (*e.g.*, [7, 17]). Note that weakest preconditions are so rarely considered in the literature that weakest *liberal* preconditions are often simply called weakest preconditions, *e.g.*, [17].

2 Example

In this section we informally illustrate our method by applying it to several examples. The procedures proposed by this paper are displayed in Figures 1 and 2. They will be more formally described in Section 3.

We have split our method into two procedures for presentational convenience. The first procedure illustrates our method's key ideas, but fails for the class of phase-transition programs. The second procedure extends the first with support for phase-transition programs. Note that phase-change programs and preconditions are interrelated (allowing us to solve the phase-change problem easily with our tool), as a phase-change program can be thought of as several copies of the same loop composed, but with different preconditions.

2.1 Finding preconditions for programs without phase-change

We consider the following code fragment:

```
1       // @requires true;
2       while(x>0){
3           x=x+y;
4       }
```

We assume that the program variables x and y range over integers. The initially given `requires`-clause is not sufficient to guarantee termination. For example, if x=1 and y=0 at the loop entry then the code will not terminate. The weakest precondition for termination of this program is $\mathtt{x} \leq 0 \vee \mathtt{y} < 0$.

If we apply an existing termination prover, *e.g.*, TERMINATOR [9] or ARMC [21], on this code fragment then it will compute a counterexample to termination. The counterexample consists of 1) a stem η, which allows for manipulating the values before the loop is reached, and 2) a repeatable cycle ρ, which is a relation on program states that represents an arbitrary number of iterations of the loop.

To simplify the presentation, we represent the stem η as an initial condition θ on the variables of the loop part. (Section 4 describes this step in more detail.) In our example, the initial condition θ is *true* and the transition relation of the loop is defined by

$$\rho(\{\mathtt{x},\mathtt{y}\},\{\mathtt{x}',\mathtt{y}'\}) \;\equiv\; \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} + \mathtt{y} \wedge \mathtt{y}' = \mathtt{y}\ .$$

In order to try and prove this counterexample spurious (*i.e.* to prove it well-founded, as explained in [9]), we need to find a ranking function f such that $\rho(X, X') \Rightarrow \mathcal{R}_f(X, X')$, where $\mathcal{R}_f$ is the *ranking relation* defined by f:

$$\mathcal{R}_f(X, X') \;\equiv\; f(X) \geq 0 \wedge f(X') \leq f(X) - 1\ .$$

As the termination prover has returned the above relation ρ as a counterexample, we can assume that no linear ranking function f exists (note that there could exist a non-linear ranking function, depending on the completeness of the termination prover).

Due to the absence of a linear ranking function for ρ, we find a *potential ranking function*, *i.e.*, a function b such that one of the conjuncts defining $\mathcal{R}_b(X, X')$ holds for ρ. We compute a potential ranking function for ρ by finding an expression on the variables $\{\mathtt{x},\mathtt{y}\}$ that is bound from below. One method for finding such candiate functions ito consider only the domain (and not the range) of ρ, *i.e.*, find functions that are bounded when there is a successor. In other words, consider $\exists \mathtt{x}', \mathtt{y}'.\, \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} + \mathtt{y} \wedge \mathtt{y}' = \mathtt{y}$. In practice we achieve this via a the application of a quantifier elimination procedure, *i.e.*, we have

$$\mathrm{QELIM}(\exists \mathtt{x}', \mathtt{y}'.\, \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} + \mathtt{y} \wedge \mathtt{y}' = \mathtt{y}) \;\equiv\; \mathtt{x} > 0\ .$$

We can normalize the condition $\mathtt{x} > 0$ as $\mathtt{x} - 1 \geq 0$, and thus use the function $b = \mathtt{x} - 1$. Because $\rho(\{\mathtt{x},\mathtt{y}\},\{\mathtt{x}',\mathtt{y}'\}) \Rightarrow b(\{\mathtt{x},\mathtt{y}\}) \geq 0$, which is the first conjunction required by $\mathcal{R}_b$, we can use b as our potential ranking function.[4]

Enforcing ranking with a strengthening. The function $b = \mathtt{x} - 1$ that we found only satisfies part of the requirements for proving termination with $\mathcal{R}_b$ (*i.e.*,

[4] In this simple example the result was exactly the loop condition. However, when translating the cycle returned from the termination prover to a formula, some of the conditions are not on the initial variables.

$b(X) \geq 0$ but not $b(X') \leq b(X) - 1$). We need a strengthening $s(\{\mathtt{x}, \mathtt{y}\})$ such that

$$s(\{\mathtt{x}, \mathtt{y}\}) \wedge \rho(\{\mathtt{x}, \mathtt{y}\}, \{\mathtt{x}', \mathtt{y}'\}) \Rightarrow \mathcal{R}_b(\{\mathtt{x}, \mathtt{y}\}, \{\mathtt{x}', \mathtt{y}'\}) \ .$$

Since b is bounded, we find $s(\{\mathtt{x}, \mathtt{y}\})$ as follows:

$$s(\{\mathtt{x}, \mathtt{y}\}) \ \equiv \ \text{QELIM}(\forall \mathtt{x}', \mathtt{y}'. \, \rho(\mathtt{x}, \mathtt{y}, \mathtt{x}', \mathtt{y}') \Rightarrow b(\{\mathtt{x}', \mathtt{y}'\}) \leq b(\{\mathtt{x}, \mathtt{y}\}) - 1) \ .$$

We obtain $s(\{\mathtt{x}, \mathtt{y}\}) = \mathtt{x} \leq 0 \vee \mathtt{y} < 0$. That is, if s were an invariant (and usually it is not), then ρ would be provably well-founded using b.

Synthesizing a precondition guaranteeing the strengthening. Recall that the original problem statement is to find a precondition that guarantees termination of the presented code fragment. As the strengthening s guarantees termination, we now need to find a precondition that guarantees the validity of s on every iteration of ρ. The required assertion is the weakest *liberal* precondition of s wrt. the loop statement. We use known techniques for computing underapproximations of weakest liberal preconditions to find the precondition that ensures that after any number of iterations of the loop s must hold in the next iteration. Using a tool for abstract interpretation of arithmetic programs [15], we obtain $r(\{\mathtt{x}, \mathtt{y}\}) = \mathtt{x} \leq 0 \vee \mathtt{y} < 0$. In summary, our procedure has discovered the precondition proposed above.

Note that we can alternate executions of our procedure together with successive applications of a termination prover to find a precondition that is strong enough to prove the termination of the entire program. The interaction between the tools is based on counterexamples for termination, which are discovered by the termination prover and are eliminated by the precondition synthesis procedure.

2.2 Finding preconditions for phase-change programs

Consider the following code fragment:

```
1       // @requires true;
2       while(x>0){
3           x=x+y;
4           y=y+z;
5       }
```

Again, the given `requires`-clause is not sufficient to ensure termination. For example, if x=1, y=0, and z=0 at the loop entry then the code will not terminate. However, this time, the weakest precondition is given by a non-linear assertion, which is difficult to construct automatically.

Note that the precondition $\mathtt{z} < 0$ guarantees that the loop terminates, but the termination may take place after the computation passed through two phases. The first phase is characterized by the assertion $\mathtt{y} \geq 0$. In fact, during this phase the value of x may not decrease towards zero. Nevertheless, *eventually* the value

of $\mathtt{y}$ will decrease below zero, *i.e.*, a phase transition takes place. At this point $\mathtt{x}$ will start decreasing towards (and eventually reaching) zero.

In this example, the termination prover returns a stem which is the identity relation and cycle relation

$$\rho(\{\mathtt{x},\mathtt{y},\mathtt{z}\},\{\mathtt{x}',\mathtt{y}',\mathtt{z}'\}) \;\equiv\; \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} + \mathtt{y} \wedge \mathtt{y}' = \mathtt{y} + \mathtt{z} \wedge \mathtt{z}' = \mathtt{z}\ .$$

The first step of the precondition inference repeats the procedure presented in the previous subsection. On this example, similarly to the previous one, the procedure computes $b = \mathtt{x} - 1$ and $s = \mathtt{x} \leq 0 \vee \mathtt{y} < 0$. However, when computing a precondition that ensures that s in an invariant, we obtain a linear underapproximation[5]

$$\begin{aligned}&\mathtt{x} \leq 0 \vee \mathtt{x} + \mathtt{y} \leq 0 \vee (\mathtt{y} < 0 \wedge \mathtt{x} + 2 * \mathtt{y} + \mathtt{z} \leq 0) \vee \\ &(\mathtt{y} < 0 \wedge \mathtt{x} + 3 * \mathtt{y} + 3 * \mathtt{z} \leq 0) \vee (\mathtt{y} < 0 \wedge \mathtt{z} \leq 0)\ .\end{aligned}$$

The first four disjuncts correspond to the cases when the loop terminates after 0, 1, 2, and 3 iterations, respectively, and in which $\mathtt{y} < 0$ holds until possibly the last iteration. The last disjunct is more interesting, as it states that if $\mathtt{y} < 0$ and $\mathtt{z} \leq 0$ then the loop terminates. The expected precondition $\mathtt{z} < 0$ is not included in the disjunction, since it does not guarantee that $\mathtt{y} < 0$ from the start.

Note that each of these conditions guarantees termination when they are satisfied at any iteration of the loop, not necessarily at the first one. These conditions identify phase transition points. Once they are met, $\mathtt{x}$ will start decreasing until the loop terminates. The solution is to constrain the transition relation with the negated condition. Termination of the constrained loop ensures that these conditions are eventually met, and thus the original loop will terminate. Thus, we call the procedure recursively with the constrained transition relation, and disjoin the returned preconditions with the existing ones.

For example, constraining the loop with $\neg(\mathtt{y} < 0 \wedge \mathtt{z} \leq 0)$ and calling the procedure recursively yields the additional preconditions $\mathtt{x} + 2 * \mathtt{y} + \mathtt{z} \leq 0 \vee \mathtt{x} + 3 * \mathtt{y} + 3 * \mathtt{z} \leq 0 \vee \mathtt{z} < 0$. The first two disjuncts are weakenings of the previous preconditions for the cases in which the loop terminates after 2 or 3 iteration, removing the condition $\mathtt{y} < 0$. The last disjunct $\mathtt{z} < 0$ is the more interesting, since it ensures eventual termination.

After simplification the precondition computed by the procedure is

$$\begin{aligned}&\mathtt{x} \leq 0 \vee \mathtt{x} + \mathtt{y} \leq 0 \vee \mathtt{x} + 2 * \mathtt{y} + \mathtt{z} \leq 0 \vee \\ &\qquad \mathtt{x} + 3 * \mathtt{y} + 3 * \mathtt{z} \leq 0 \vee (\mathtt{y} < 0 \wedge \mathtt{z} \leq 0) \vee \mathtt{z} < 0\ .\end{aligned}$$

3 Computing preconditions for termination

This section formally describes the two methods for computing preconditions for termination, PreSynth and PreSynthPhase. We first define basic concepts.

[5] Any linear precondition must be an underapproximation, since the weakest liberal precondition is non-linear.

3.1 Preliminaries

We assume a program $P = (X, \theta, \rho)$ over a finite set of variables X. For simplicity, we do not single out the program counter variable that ranges over control locations of the program, and assume that it is included in X. The assertion θ represents the initial condition of the program. The transition relation ρ is represented by an assertion over the program variables X and their primed versions X'. The primed variables refer to the values after the transition is executed. For practicality, we assume that the initial condition and the transition relation are represented using a logical theory for which practical quantifier elimination procedures exist.[6]

We refer to valuations of program variables as program states. A program computation is a finite or infinite sequence of program states $s_1, s_2, \ldots$ such that s_1 is an initial state and each pair of consecutive states s_i and s_{i+1} follows the transition relation. Formally, $s_1 \models \theta$ and for each but final s_i in the computation we have $(s_i, s_{i+1}) \models \rho$. The program terminates from a state s if there is no infinite computation that starts at s. An assertion r is a precondition for termination if the program P terminates from each state s that satisfies r and the initial condition, *i.e.*, $s \models \theta \wedge r$.

Our procedure uses (an under-approximation of) the weakest liberal precondition operator WLP, which we define as usual. Given a transition relation $\rho(X, X')$ and an assertion over program variables $\varphi(X)$, this transformer yields the following conjunction when applied on the transitive closure of ρ:

$$\mathsf{WLP}(\rho^*(X, X'), \varphi(X)) \;\equiv\; \bigwedge_{n \geq 0} \forall X' .\, \rho^n(X, X') \Rightarrow \varphi(X') \;.$$

3.2 The Procedure PreSynth

Figure 1 shows the basic precondition synthesis procedure PreSynth. The input of PreSynth consists of an initial condition θ and a transition relation ρ. Note that in the most likely usage scenario, θ and ρ will represent a counterexample to termination reported by a termination prover. In this case, PreSynth is applied on a compact code fragment and not the full program, which allows one to apply precise, automated reasoning-based techniques.

We discuss the major steps of the procedure in more detail. The procedure takes a formula representing a set of initial states, together with a relation representing the transitions. Line 1 in Figure 1 strengthens the transition relation ρ with (possibly an over-approximation of) the states that are reachable from θ. This step provides necessary precision for the subsequent computations by taking into account reachability invariants. Here, we rely on an efficient abstract interpretation tool based on *e.g.,* the Octagon domain [18]. Thus, we can implement line 1 using a standard abstract reachability procedure [12] as follows (we

[6] If necessary, we can over-approximate the initial condition and the transition relation using assertions from such a theory.

```
        function PreSynth
        input
           θ(X) : initial condition
           ρ(X, X′) : transition relation
        begin
0          r(X) := false
1          ρ(X, X′) := ρ(X, X′) ∧ QELIM(∃X_0. θ(X_0) ∧ ρ*(X_0, X))
2          B := Finite({b(X) | ∀X. (∃X′. ρ(X, X′)) ⇒ b(X) ≥ 0})
3          foreach b(X) ∈ B do
4             s(X) := QELIM(∀X′.ρ(X, X′) ⇒ b(X) ≥ 0 ∧ b(X′) ≤ b(X) − 1)
5             r(X) := r(X) ∨ WLP(ρ*(X, X′), s(X))
6          done
7          return "precondition for termination r(X) "
        end.
```

Fig. 1. The procedure PreSynth synthesizes a precondition for the termination of a transition relation ρ from initial states θ. The procedure Finite returns a selected finite subset of its input (*i.e.* $\text{Finite}(S) \subseteq_{\text{fin}} S$). Finite may choose, for example, to return only the linear elements of S.

assume that $\theta \Rightarrow \theta^{\#}$, $\rho \Rightarrow \rho^{\#}$, and lfp is the least fixpoint operator):

$$\rho(X, X') := \rho(X, X') \wedge \mathsf{lfp}(\rho^{\#}, \theta^{\#})$$

Line 2 of PreSynth computes a finite set of expressions B that are bounded by the (strengthened) transition relation ρ. In theory we could generalize the set B to include non-linear or lexicographic ranking functions, though in practice it will be limited to linear ranking functions. We will assume that each b is a function ranging over simple arithmetic types (*i.e.*, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$) though in principle we could generalize the procedure to support any well-order. In practice, to compute B we apply an existential quantifier elimination procedure to eliminate the primed variables. Then, we consider the linear inequalities that appear in the result. Each bound expression $b(X) \in B$ is treated as a potential ranking function, and is used to guide the search for a strengthening $s(X)$ on the domain of the transition relation ρ that makes ρ well-founded. PreSynth only considers the well-foundedness arguments constructed using $b(X)$. Line 4 uses a quantifier elimination procedure to construct the strengthening $s(X)$ by imposing the *bounded* and *decrease* conditions

$$s(X) \wedge \rho(X, X') \Rightarrow b(X) \geq 0 \wedge b(X') \leq b(X) - 1 \ .$$

The assertion $s(X)$ guarantees that $b(X)$ is a ranking function for the given transition relation.

The strengthening imposed by $s(X)$ is effective if $s(X)$ holds for all states that are reachable from θ by applying ρ. Line 5 in Figure 1 computes the necessary precondition that guarantees the invariance of $s(X)$. All preconditions that are found using the potential ranking functions are accumulated in an assertion $r(X)$, and reported to the programmer in line 7.

Line 5 of PreSynth uses known abstract interpretation-based techniques for under-approximating sets and relations for the computation of $\mathsf{WLP}(\rho^*(X, X'), s(X))$. We pass the following program to the Interproc analyzer [15]:

```
assume θ(X′);
while (∗) do
   X=X′;
   X′=∗;
   assume ρ(X, X′);
od
assume ¬s(X);
```

Using Interproc we first compute an abstract fixpoint using backwards analysis starting from top element in the abstract domain. Then, the abstract element at the loop entry location represents an over-approximation of the states that fail the assertion $s(X)$. The complement of this element, which we obtain by negating the corresponding assertion, provides an under-approximation of the initial set of states of the program that guarantees the invariance of $s(X)$.

Theorem 1 (PreSynth correctness). *Let $r(X)$ be an assertion computed by the procedure* PreSynth*. Then, r is a precondition for termination of a program with the initial condition θ and the transition relation ρ.*

Proof. Let $r(X)$ be computed by PreSynth. For a proof by contradiction, we assume that there is an infinite computation $s_1, s_2, \ldots$ from an initial state that satisfies $r(X)$. Let $b(X)$ be a bound expression that contributed a disjunct in $r(X)$ that holds for the state s_1, and $s(X)$ be a corresponding strengthening. From the definition of WLP, we have that $s(X)$ holds for each state s_i, where $i \geq 0$. Thus, the value of $b(X)$ decreases after each program step, while being bounded from below. We reached a contradiction to the assumption that the computation is infinite. □

3.3 The Procedure PreSynthPhase

See Figure 2. The procedure PreSynthPhase extends the applicability of the basic procedure from Figure 1 to phase-transition programs (as described in Section 2). It removes the condition that the computed strengthening needs to apply immediately. Instead, we only require that the strengthening applies *eventually*. PreSynthPhase implements this eventuality requirement by finding a precondition for termination of an augmented transition relation that avoids visiting states satisfying the strengthening constraint. The inferred precondition can be enforced eventually by applying PreSynthPhase recursively.

We discuss the major steps of the procedure in more detail. Line 1 computes an initial precondition $r(X)$ by applying the procedure PreSynthPhase. If the precondition $r(X)$ is non-empty, as checked in line 2, then PreSynthPhase weakens it by a precondition that ensures the eventuality of $r(X)$. Here, we follow a standard technique for the verification of temporal liveness properties [22],

```
        function PreSynthPhase
        input
           θ(X) : initial condition
           ρ(X, X′) : transition relation
        begin
1          r(X) := PreSynth(θ, ρ)
2          if ∃X. r(X) then
3             ρ(X, X′) := ρ(X, X′) ∧ ¬r(X)
4             r(X) := r(X) ∨ PreSynthPhase(θ, ρ)
5          endif
6          return "precondition for phase termination r(X) "
        end.
```

Fig. 2. The procedure PreSynthPhase synthesizes a precondition for the phase termination of a transition relation ρ from initial states θ. It applies the procedure PreSynth from Figure 1 when recursively identifying computation phases. PreSynthPhase can be stopped at any time, e.g., by reaching a user-provided upper bound, and it will yield a sound precondition for phase-transition termination.

and apply a translation to a termination problem. Effectively, line 3 constructs a new transition relation from ρ that avoids $r(X)$. Thus, we can apply PreSynthPhase recursively on the new transition relation, see line 4.

Theorem 2 (PreSynthPhase correctness). *Let $r(X)$ be an assertion computed by the procedure* PreSynthPhase*. Then, r is a precondition for termination of a program with the initial condition θ and the transition relation ρ.*

Proof. Let $r_1(X), \ldots, r_n(X)$ be a sequence of preconditions that are computed by applying PreSynth during the execution of PreSynthPhase. We observe that every computation that starts in an initial state that satisfies $r_i(X)$ for $i > 1$ either terminates or eventually reaches $r_{i-1}(X)$. Hence, eventually every computation either stops or reaches $r_1(X)$. From Theorem 1 follows that the program terminates on each state reachable from an initial state satisfying $r_1(X)$. □

4 Implementation and experiments

We have built a prototype implementation of our method based on the following collection of tools: for termination proving we use ARMC [20,21], for quantifier elimination we use Cooper's procedure [11], and for abstract interpretation we use the Interproc analyzer [15].

See Figures 3 and 4, which contain example programs (both hand written and drawn from industrial examples) together with the results of our tool. We leave it to the reader to judge the usefulness of the synthesized preconditions. The running times in the figures include the times of iterating the procedure of Figure 2 and the termination prover. In the remainder of this section we highlight relevant implementation details.

Program fragment	Precondition & notes

`i = 0;`
`if (l_var >= 0) {`
`  while (l_var < 1073741824) {`
`  i++;`
`  l_var = l_var << 1;`
`  }`
`}`
	$\texttt{l\_var} > 0 \vee \texttt{l\_var} < 0 \vee \texttt{l\_var} \geq 1073741824$
	Example from an audio compression module: We model shift by multiplication and checked for overflow with an extra check and subtract.
	Time: 22 seconds.

`while (cbSrcLength >= cbHeader) {`
`  DWORD dwHeader;`
`  UINT cbBlockLength;`
`  cbBlockLength = (UINT)min(cbSrcLength, nBlockAlignment);`
`  cbSrcLength -= cbBlockLength;`
`  cbBlockLength -= cbHeader;`
`  dwHeader = *(DWORD HUGE_T *)pbSrc;`
`  pbSrc += sizeof(DWORD);`
`  nPredSample = (int)(short)LOWORD(dwHeader);`
`  nStepIndex = (int)(BYTE)HIWORD(dwHeader);`
`  if( !imaadpcmValidStepIndex(nStepIndex) ) return 0;`
`  *pbDst++ = (BYTE)((nPredSample >> 8) + 128);`
`  while (cbBlockLength--) {`
`  bSample = *pbSrc++;`
`  nEncSample = (bSample & (BYTE)0x0F);`
`  nSz = step[nStepIndex];`
`  nPredSample =`
`  imaadpcmSampleDecode(nEncSample, nPredSample, nSz);`
`  nStepIndex =`
`  imaadpcmNextStepIndex(nEncSample, nStepIndex);`
`  *pbDst++ = (BYTE)((nPredSample >> 8) + 128);`
`  nEncSample = (bSample >> 4);`
`  nSz = step[nStepIndex];`
`  nPredSample =`
`  imaadpcmSampleDecode(nEncSample, nPredSample, nSz);`
`  nStepIndex =`
`  imaadpcmNextStepIndex(nEncSample, nStepIndex);`
`  *pbDst++ = (BYTE)((nPredSample >> 8) + 128);`
`  }`
`}`
	$\texttt{cbSrcLength} < \texttt{cbHeader} \vee$
	$(\texttt{nBlockAlignment} > 0 \wedge \texttt{cbHeader} > 0)$
	Example from another audio application.
	Time: 106 seconds.

Fig. 3. Programs drawn from industrial examples, runtimes, and synthesized preconditions. The runtimes include the iteration of the procedure in Figure 2 together with the termination prover.

Simplification. Both the quantifier elimination and the abstract interpretation tools do not give minimal formulas. In many cases the formulas have redundant conjuncts or several disjuncts that imply each other. Simplification of these formulas was important for reducing the size of the result preconditions.

Loop termination after a bounded number of iterations. The recursive call in PreSynthPhase is problematic in the case that the precondition is the result of a loop termination in a bounded number of iteration, say 3. The problem is that the next call will find precondition that come from the loop terminating after 6 iteration and so on. To solve this problem, before the recursive call, we check whether the precondition ensures termination in a fixed number of steps, and if it does then we avoid the recursive call. That is, we unroll the relation k times (for some threshold k) and then check to see if $\rho^k = \emptyset$, which implies termination.

Program fragment	Precondition & notes

`// @requires true;`
`while(x>0){`
`  x=x+y;`
`  y=y+z;`
`}`
	$\mathtt{x} \leq 0 \vee \mathtt{x}+\mathtt{y} \leq 0 \vee \mathtt{x}+2\mathtt{y}+\mathtt{z} \leq 0 \vee \mathtt{x}+3\mathtt{y}+3\mathtt{z} \leq 0 \vee \mathtt{z} < 0 \vee (\mathtt{z} \leq 0 \wedge \mathtt{y} < 0)$
	This is the example from §2. The first four disjuncts cover the case where the loop terminates after 0, 1, 2, or 3 iterations respectively. The last two disjuncts ensure that the loop eventually terminates. The condition $z \leq 0 \wedge y < 0$ is the case in which the loop starts with $y < 0$ and z does not interfere.
	Time: 24 seconds.

`// @requires true;`
`while(x<=N){`
`  if (*) {`
`  x=2*x+y;`
`  y=y+1;`
`  } else {`
`  x++;`
`  }`
`}`
	$\mathtt{x} > \mathtt{N} \vee \mathtt{x}+\mathtt{y} \geq 0$
	An example from [3]. We find preconditions that ensure that the loop is executed 0 times ($x > N$) and the precondition that ensures termination as it appears in the paper ($x + y \geq 0$).
	Time: 4 seconds.

`// @requires true;`
`while(x>=0){`
`  x= -2*x + 10;`
`}`
	$\mathtt{x} > 5 \vee \mathtt{x} < 0$
	Example (from [19]) of a terminating linear program with no linear ranking function. Note that the program always terminates after at least 5 iterations. Our synthesized precondition provides conditions that guarantee termination after 0 or 1 iterations.
	Time: 2 seconds.

`// @requires n>200 and y<9;`
`x = 0;`
`while (1) {`
`  if (x<n) {`
`  x=x+y;`
`  if (x>=200) break;`
`  }`
`}`
	$\mathtt{y} > 0$
	Example (from [14]) after translation that allows us to disprove a safety property.
	Time: 6 seconds.

`while (x!=y) {`
`  if (x>y) {`
`  x = x - y;`
`  } else`
`  y = y - x;`
`  }`
`}`
	$\mathtt{x} = \mathtt{y} \vee (\mathtt{x} > 0 \wedge \mathtt{y} > 0)$
	The Euclidean algorithm for GCD. Each case in the `if` requires a different ranking function. Demonstrates the interaction with disjunctive well foundedness.
	Time: 17 seconds.

Fig. 4. Handwritten programs, runtimes, and synthesized preconditions. The runtimes include the iteration of the procedure in Figure 2 together with the termination prover.

Stems vs. initial conditions. The termination prover returns a cycle and a stem as a counterexample. The stem is not only an initial condition, but rather a sequence of primitive statements that manipulate variables. We use existential quantifier elimination to convert the stem into the initial condition for the procedure. Because there are no control-flow constructs in the stem, a straightforward application of WLP translates the precondition discovered by the procedure to a precondition applicable at the beginning of the stem.

Translating paths to formulas. The cycles returned by the termination prover often contain multiple assignments and conditions. The translation first converts the path to static single assignment form and then to a path formula (see [9] for details). The extra temporaries generated are handled in the quantifier elimination as extra variables to eliminate.

Handling disjunctions. Effective dealing with disjunctions that appear in intermediate formulas manipulated by our algorithm is crucial for its applicability. Even when applied on conjunctive inputs, the procedure PreSynth creates additional disjunctions. Its line 1 may split the transition relation ρ when restricting it to reachable states. Although line 5 creates disjunction by iterating over the set of candidate bound expressions B, they are tamed by the subsequent negation in PreSynthPhase. On the other hand, line 5 creates a conjunction representing the weakest liberal precondition that, after its negation in PreSynthPhase, may create disjuncts. Disjunctions in the preconditions are handled by calling the algorithm separately on each disjunct and accumulating the results. Disjunctions in the transition relation are handled by iterating the termination prover as explained below.

Iterating the termination prover. The constrained transition relation generated in PreSynthPhase can be complicated. We found that calling the termination prover on this relation and letting the PreSynthPhase only deal with the counterexamples returned improves the preconditions we compute. Thus, there is a mutual recursion between calling the termination prover and PreSynthPhase.

5 Conclusion

This paper has described an automatic method for finding sound underapproximations to weakest preconditions to termination. Using illustrative examples we have shown that the method can be used to find *useful* underapproximations (*i.e.* something larger than `false`, but in cases smaller than the often complex *weakest* precondition). Beyond the direct use of proving conditional termination, we believe that our method can also be used in several areas of program verification, including the disproving of safety properties, interprocedural analysis, interprocedural termination proving, and distributed termination proving.

References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Modular ranking abstraction. *Int. J. Found. Comput. Sci.*, 2007.
2. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.
3. A. Bradley, Z. Manna, and H. Sipma. The polyranking principle. In *ICALP*, 2005.
4. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
5. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, 2005.
6. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *TOPLAS*, 29(2), 2007.
7. C. Calcagno, D. Distefano, P. OHearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS*, 2007.
8. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do something good. In *POPL*, 2007.
9. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
10. B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, 2007.
11. D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
13. E. W. Dikstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1989.
14. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.
15. G. Lalire, M. Argoud, and B. Jeannet. Interproc analyzer. http://bjeannet.gforge.inria.fr/interproc/, 2008.
16. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.
17. R. Leino. Effecient weakest preconditions. *Information Processing Letters*, 93(6), 2005.
18. A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.
19. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
20. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
21. A. Rybalchenko. ARMC. http://www.mpi-sws.org/∼rybal/armc/, 2008.
22. M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *LICS*, 1987.