Proving Program Termination with Transition Invariants and Size-Change Analysis

Chris Allsman

May 6, 2019

1 Introduction

This paper serves as a survey of techniques used to prove program termination. Although determining if a general program will terminate on all inputs is undecidable as per the halting problem, methods for proving termination of specific programs have a long history, dating back to work by Turing in 1949. [12]

More recently, two general methods of proving program termination have emerged. Size-change analysis, proposed by Lee et al. [11] provides a general technique for using local changes in program values to create and analyze a set of graphs which can be used to prove program termination. Another technique, developed by Podelski and Rybalchenko [14], attempts to identify a transition invariant using existing methods for theorem provers and safety checkers.

Although much of the work of the past two decades has focused on these techniques in isolation, the theoretical underpinnings of the two approaches have much in common. Both rely on identifying a path through the program for which some value decreases according to a well-founded relation. Furthermore, many techniques for identifying transition invariants mirror the properties of size-change graphs of size-change graphs.

This paper will begin by describing the general approaches of these two methods in isolation, as well as methods for applying them in practice. It will then compare the two approaches and identify which areas each are used in, as well as what areas of overlap exist between the two. Finally, there will be a discussion of ways in which these techniques may be extended, how they have been extended already, and what open problems still exist.

All major approaches for showing program termination involve identifying some value in the program which, overall, strictly decreases according to some well-founded relation. For the purposes of this paper, we will assume the existence of a well-founded ordering $<$ on all program values in the space $A \times A$ such that any non-empty subset S of A has a minimal element. That is, $\exists m \in S \,.\, \forall x \in S \,.\, \neg(m < x)$.

If a value $v \in A$ strictly decreases across the course of program execution, then an infinitely executing program will cause this value to decrease infinitely. However, because of the existence of a well-founded relation on $A \times A$, we know that v cannot decrease infinitely. Therefore, different methods for proving program termination ultimately provide different formalizations for what it means for a value to be "strictly decreasing" across the course of program execution.

In reality, the existence of a well-founded relation for program values are usually provided as a size ordering [16] or generated via term rewriting systems [7]. As this concern is tangential to the methods presented in this survey, we will assume all program values have a natural well-founded relation defined (for example, natural numbers and linked lists).

2 Size-Change Termination

2.1 Methodology

A natural approach for showing that values in a functional program are decreasing is to show that the arguments of a function call are smaller than the corresponding input variables. For example, in the following program, the first input to the function at line 3 strictly decreases with respect to the input `n`.

```
fact(n)  = 1:f1(n, 1)
f1(n, t) = if n=0 then t
              else 2:f1(n-1, t*n)
```

Although this general methodology had been applied prior to the results of size-change termination [7], it was not robust for all cases including permuted arguments to recursive calls and indirect/mutual recursion [11].

The methodology of size-change termination, proposed by Lee et al., attempts to generalize this idea so it does require user provided annotations or theorem proving methods. This method encapsulates the idea of descending input values in a *size-change graph.* While the original paper on size-change termination provides methods for analyzing size-change graphs to prove program termination, which will be detailed here, it does not provide methods for producing these graphs in the first place.

Size-change termination analyzes a program in terms of its call sequence. We use the following terms to describe call sequences in the program:

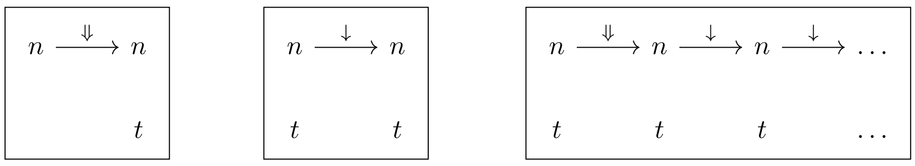
- $f \xrightarrow{c} g$ can occur if a call to g occurs in the body of f
- A call sequence is an infinite or finite sequence $c_1 c_2 c_3 \ldots$ which is well formed if there exists some some sequence of functions $f_0, f_1, f_2, \ldots$ such that $f_0 \xrightarrow{c_1} f_1 \xrightarrow{c_2} f_x \xrightarrow{c_3} \ldots$.
- We can also write a call sequence $cs = c_1 c_2 c_3 \ldots$ as $f_0 \to f_n$ if $f_0 \xrightarrow{c_1} f_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} f_n$.

In this way, a program can be abstracted as a call sequence. With size-change analysis, this information is encoded in a size-change graph.

A size-change graph, $G : f \to g$, is a bipartite graph representing a state transition between (f, v_1) and (g, v_2) as follows:

- Nodes in the graph come from the parameters of f and of g
- Edges in the graph are directed from parameters of f to parameters of g
- An edge can be labeled with $\downarrow$ if the data value strictly decreases in the call, wheras an edge labeled with $\Downarrow$ must have the data value either decrease or stay the same.

A multipath is a graph representing a concatenation of size-change graphs. Below is an example of a set of size-change graphs and a multipath for the factorial example given earlier:



$n \xrightarrow{\Downarrow} n$
t
$n \xrightarrow{\downarrow} n$
t t
$n \xrightarrow{\Downarrow} n \xrightarrow{\downarrow} n \xrightarrow{\downarrow} \ldots$
t t t $\ldots$

(a) $G_1 : \mathit{fact} \to f1$ (b) $G_2 : f1 \to f1$ (c) $G_1 || G_2 || G_2 || \ldots$

Figure 1: Size-change graphs and multipath for `fact`

More formally, let $\mathcal{C}$ be the set of all possible calls in program p. Then, let $\mathcal{G} = \{G_c | c \in \mathcal{C}\}$. Then, given some call sequence cs, $\mathbf{M}^{\mathcal{G}}(cs) = G_{c_1} || G_{c_2} || G_{c_3} || \ldots$, where $||$ represents the concatenation of graphs.

In order to determine if a program is infinitely descending, we must first determine the set of all **infinite** call sequences which represent the program. Given that the first function called by the program is f_i, we call this set $FLOW^{\omega} = \{cs = c_1 c_2 \ldots \mid cs \text{ is well formed and } f_i \xrightarrow{c_1} f_1\}$

The analysis then considers threads in a multipath representing a program p. A thread in a multipath $\mathcal{M}$ is a path starting from any node in the graph - if it contains at least one edge labeled $\downarrow$, it is considered a *descending thread*, and if it contains infinitely many edges labeled $\downarrow$ it is considered an *infinitely decreasing thread.*

Let $DESC^{\omega} = \{cs \in FLOW^{\omega} | \exists \text{ some infinitely decreasing thread in } \mathcal{M}^{\mathcal{G}}(cs)\}$. Then, program p is size-change terminating if and only if $FLOW^{\omega} = DESC^{\omega}$. In this case, any infinite call sequence in p would cause some value to infinitely decrease, which cannot happen due to the well-founded relations defined on program values. Therefore, there must not exist any infinite call sequences, so p terminates for all reachable paths.

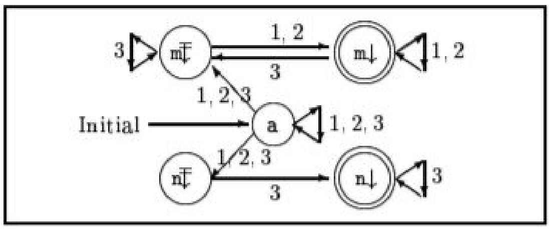
Lee et al. propose two methods for proving equivalence of $FLOW^\omega$ and $DESC^\omega$. Both of these methods are shown to be PSPACE-complete - that is, they are intractable but solvable in polynomial space.

The first method proposed is composing a Büchi automaton - a type of ω-automaton which is a finite representation of infinite sequences. Given two sets of infinite sequences, S_1 and S_2, along with two (potentially distinct) automata which accept each sequence in their corresponding set, is decidable to determine if $S_1 = S_2$. It is relatively trivial to construct an automata which accepts all sequences in $FLOW^\omega$. Therefore, if an automata to accept sequences in $DESC^\omega$ is created, it is decidable to check if $FLOW^\omega = DESC^\omega$.

The specific automata created is a Büchi automaton with state transitions corresponding to some size change graph and states corresponding to edges of the form $x \Downarrow$ or $x \downarrow$. The automaton is created in such a way that it accepts a call sequence cs if the multipath $\mathcal{M}^{\mathcal{G}}(cs)$ has an infinitely descending thread. If this is the case, it enters a state of the form $x \downarrow$ infinitely many times, so these states are designated as accepting.

```
a(m, n) = if m=0 then n+1 else
          if n=0 then 1:a(m−1, 1)
               else 2:a(m−1,
                        3:a(m, n−1))
```

(a) Example terminating program



(b) Büchi Automaton

Figure 2: Example program and automaton from [11]

However, explicitly creating an automaton in this way can be costly and cumbersome. An alternate method involves analyzing a subset of the transitive closure of the set of all size-change graphs. The composition of G_1 and G_2 contains the edge $x \overset{\downarrow}{\rightarrow} z$ if there is some edge $x \rightarrow y$ in G_1 and $y \rightarrow z$ in G_2 where either $x \overset{\downarrow}{\rightarrow} y$ or $y \overset{\downarrow}{\rightarrow} z$. It will contain the edge $x \overset{\Downarrow}{\rightarrow} y$ if neither edge is strictly decreasing.

This analysis considers only the idempotent elements of the transitive closure. Lee et al. show that Program `p` is not size change terminating if there is an idempotent element with no edge of the form $x \overset{\downarrow}{\rightarrow} x$. Conversely, `p` is size-change terminating if every idempotent G has an edge $x \overset{\downarrow}{\rightarrow} x$ [8]. Suppose there is some idempotent G', that is $G_{cs} = G_{cs}; G_{cs}$ where G_{cs} is a graph composition representing some call sequence `cs` and $G_1; G_2$ is the composition of G_1 and G_2. Assuming p is size-change terminating, the call sequence $cs, cs, \ldots$ contains some infinitely descending thread. Because there are finitely many variables, some variable x must be visited finitely many times, so there must be some thread of the form $x \overset{\downarrow}{\rightarrow} x$. Because G_{cs} is idempotent, $G_{cs}; G_{cs}; \ldots = G_{cs}$, so G_{cs} must also have the edge $x \overset{\downarrow}{\rightarrow} x$. The reverse direction is shown in [11, 8] but is not required for an understanding of the justification.

2.2 Implications

Both methods proposed for providing size-change termination are decidable, which makes size-change analysis seem tempting as a method for approximating the undecidable problem of size-change termination. However, creating a size-change graph which actually represents the program is itself undecidable. Methods for getting a size-change graph which adequately approximates the program have been developed. One method, by Codish et al. [4], modifies the size-change graph abstraction slightly such that the search for a "good enough" abstraction can be encoded as a SAT problem.

However, while one issue is finding an appropriate approximation to the program's size-change graph, the performance bottleneck is the actual analysis. A polynomial time approximation for this step has been developed [2]. Because analyzing a specific size-change graph in the second approach described can be done in polynomial time, the real issue is the exponential number of graphs generated in the closure. However, by using heuristics based on the structure of the graph, a finite number of graphs can be analyzed. In practice, this has been successfully applied to most cases, with some user annotation required in certain cases.

A unique element of size-change analysis is the direct generation of a program abstraction in the form of a size-change graph. Therefore, many other approaches create related abstractions which allow the analysis to be performed in different programming contexts. Of note, recently an approach to detect termination at runtime has been developed [13], shown to be equivalent to static size-change analysis in many cases.

3 Transition Invariants

3.1 Methodology

Transition invariants extend the ideas of loop invariants to reason about the course of program execution across its entire execution, rather than the state of the program at any given state. In this sense, transition invariants may be the most natural way to reason about program termination - and it is by no means a surprise that this was the basis behind Turing's original approach to proving program termination [12].

The idea behind Turing's method is to find a termination argument f which, given some well-founded relation $<\in A \times A$, maps a state σ to A. If the program transitions from state σ to σ', then the termination argument should satisfy $f(s') < f(s)$ - that is, the value of the termination argument should decrease upon each iteration of the program.

If an appropriate termination argument is found, then this is sufficient to prove program termination. If the state of the program can be mapped to a continually decreasing value, then there cannot be infinite execution of the program. This is because $f(s)$ must eventually reach some minimal value.

An example of the function f for a simple iterative program is given in Figure 3. Here, `input` refers to a function which nondeterministically returns either 0 or 1 [6].

```
x := input();
y := input();
while x > 0 and y > 0 do
    if input() = 1 then
        x := x - 1;
        y := y + 1;
    else
        y := y - 1
    fi
done
```

$f = 2x + y$ will prove termination of the program

Figure 3: Program termination with a monolithic termination argument

More formally, a given program can be represented via the tuple $< W, I, R >$ where

- W is the set of program states
- I is the set of initial states
- R is a transition relation, which is a binary relation in $W \times W$, which shows how states can transition in the program.

A transition invariant T is a superset of the transitive closure of R, R^+. This superset is specifically restricted to states that are actually accessible in the program: $T \supseteq R^+ \cap (Acc \times Acc)$, where Acc is the set of accessible states in the program [14]. If the transition invariant is well-founded (as before), the program will terminate.

Therefore, this approach must find a relation which is

1. Well-founded
2. Describes a transition between any possible path of accessible states.

Finding a single termination argument which serves as a transition invariant has been the subject of research and successfully applied in a number of cases [16], however there are multiple challenges involved with this approach. Depending on the specific output of the termination argument, it may or may not be possible to actually find a termination invariant with this method. Even if it is, the termination argument may have to be so long that it is in practice impossible to find one that is actually a superset of R^+ [6].

In order to solve these issues, Podelski and Rybalchenko proposed instead attempting to find disjunctively well-founded termination relations. A disjunctively well-founded relation is the union of well-founded relations: $T = T_1 \cup T_2 \cup \ldots \cup T_n$, where $T_1, \ldots, T_n$ are all well-founded relations.

In practice, this looks like the disjunction of termination arguments found in the previous example. For example, a disjunctively-well founded termination relation for the previous example would be $f = (\text{x decreases by at least } 1 \wedge x > 0) \vee (\text{y decreases by at least } 1 \wedge x > 0)$. However, if we only consider the behavior of this relation across one iteration, we can "prove" some programs terminate which actually do not (See figure 4). This does not give us the desired property of soundness, making the method useless unless other modifications are made.

```
x := input();
y := input();
while x > 0 and y > 0 do
    if input() = 1 then
        x := x - 1;
        y := y + 1;
    else
        y := y - 1
        x := x + 1
    fi
done
```

Although either x or y will decrease from one iteration to another, the program does not terminate. Consider the case where input() alternately outputs 0 and 1.

Figure 4: Nonterminating program

The issue is that the union of well-founded relations may not itself be well-founded. For a monolithic termination argument (with no disjunction) to form a transition invariant, it was sufficient to show $R \subseteq T$. However, for a disjunctively well-founded relation, we must show $R+ \subseteq T$. In other words, we must show that $f(s') < f(s)$ not just for states s, s' which occur in consecutive iterations, but for any pair of states s, s' for which s' is reachable after one or more iterations.

In practice, it is much easier to find a disjunctive termination argument f than a monolithic termination argument, using techniques which will be described in the next section. However, because we have to prove that $R+ \subseteq T$ instead of just $R \subseteq T$, it is much more different to show that a termination argument is valid. As will be discussed, many of the attempts to improve the performance of methods using disjunctively well-founded arguments relate to this step of proving validity.

3.2 Implications

Once a termination argument is found, it can be encoded as a loop assertion and then analyzed using existing techniques for symbolic execution, abstract interpretation, and safety checking [6]. Consider for example, the method in Figure 5 for encoding the disjunctive invariant given in the previous section.

In order to generate a termination argument in the first place, one method is to start with an empty set of well-founded relations and iteratively generate candidates using rank function synthesis, adding them to the argument until it is determined to be valid or a real (non-spurious)

```
copied := 0
x := input();
y := input();
while x > 0 and y > 0 do
    if copied := 1 then
        assert ((oldx >= x + 1 and oldx > 0) or
                (oldy >= y + 1 and oldy > 0)
    elsif input() = 1 then
        copied = 1
            oldx := x;
            oldy := y
    fi
    ...
done
```

Figure 5: Encoding of termination argument checking

counterexample is found. This, combined with abstraction refinement, form the basis for the TERMINATOR algorithm [5], which has successfully been applied to systems code of over 30K lines, and has been able to identify numerous true bugs.

The TERMINATOR algorithm provides an effective method for finding termination arguments - however, the majority of its running time (>99%) is spent verifying the correctness of candidate termination arguments due to the path explosion of safety-checking methods. Therefore, much of the subsequent work in the field has been dedicated to speeding up this verification. One obvious approach is to identify a subset of candidate termination arguments to check. One such method, which relies on so-called compositional transition invariants, allows for a finite number of iterations in the checking process with the loss of a slight amount of precision [10]. Other methods attempt to abstract away portions of the program. Loop summarization attempts to replace nested loops with an appropriate transition argument, and builds the overall invariant based on this [15]. Other methods attempt to abstract away irrelevant information in the ranking functions generated so the checking is as efficient as possible, such as Chawdhary et al. who abstract a program as a trace, or a sequence of states [3].

4 Comparison and Conclusion

One major difference between these two approaches is that size-change termination is a stronger property than just program termination. To prove program termination, it is sufficient to show that all reachable paths terminate. However, size-change termination requires all possible paths (not just those reachable from the start of the program) terminate [9].

Just because of this difference, however, it does not mean that the two approaches are completely orthogonal. In the paper on loop summarization by Tsitovich et al., it is noted that in order to perform the analysis, variables must be "havoced" in order to find an argument, much like the "memoryless" property of size-change termination. In fact, many abstractions that are used to find transition invariants closely mirror specific classes of size-change graphs [9].

Ultimately, the reason to use one method over the other depends mostly on the domain. While there is no actual restriction on what types of program each method can analyze, size-change analysis is most natural for functional programs and transition invariants make the most sense for imperative and logic programs. Size-change analysis also avoids the use of external safety checkers or theorem solvers, avoiding the problems of path explosion and the reliance on external progress that methods relating to transition invariants come across.

In either case, the majority of problems faced by both methods relate to modern language features, such as runtime structures, concurrency, and untyped programs. Recent work has attempted to address this - in particular, work in termination analysis for program bytecode has seen significant development. One such method generates a constraint logic program for which many techniques, using the methods described here, are available to prove termination [1].

References

[1] Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., and Zanardini, D. Termination analysis of java bytecode. In *Formal Methods for Open Object-Based Distributed Systems* (Berlin, Heidelberg, 2008), G. Barthe and F. S. de Boer, Eds., Springer Berlin Heidelberg, pp. 2–18.

[2] Ben-amram, A. M. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst 29*, 2007.

[3] Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., and Yang, H. Ranking abstractions. In *Programming Languages and Systems* (Berlin, Heidelberg, 2008), S. Drossopoulou, Ed., Springer Berlin Heidelberg, pp. 148–162.

[4] Codish, M., Fuhs, C., Giesl, J., and Schneider-Kamp, P. Lazy abstraction for size-change termination. vol. 6397, pp. 217–232.

[5] Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. *SIGPLAN Not. 41*, 6 (June 2006), 415–426.

[6] Cook, B., Podelski, A., and Rybalchenko, A. Proving program termination. *Commun. ACM 54*, 5 (May 2011), 88–98.

[7] Giesl, J. Termination analysis for functional programs using term orderings. In *Static Analysis* (Berlin, Heidelberg, 1995), A. Mycroft, Ed., Springer Berlin Heidelberg, pp. 154–171.

[8] Heizmann, M., Jones, N. D., and Podelski, A. Size-change termination and transition invariants. In *Proceedings of the 17th International Conference on Static Analysis* (Berlin, Heidelberg, 2010), SAS'10, Springer-Verlag, pp. 22–50.

[9] Heizmann, M., Jones, N. D., and Podelski, A. Size-change termination and transition invariants. In *Static Analysis* (Berlin, Heidelberg, 2010), R. Cousot and M. Martel, Eds., Springer Berlin Heidelberg, pp. 22–50.

[10] Kroening, D., Sharygina, N., Tsitovich, A., and Wintersteiger, C. M. Termination analysis with compositional transition invariants. In *Computer Aided Verification* (Berlin, Heidelberg, 2010), T. Touili, B. Cook, and P. Jackson, Eds., Springer Berlin Heidelberg, pp. 89–103.

[11] Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. *SIGPLAN Not. 36*, 3 (Jan. 2001), 81–92.

[12] Morris, F., and Jones, C. An early program proof by alan turing. *Annals of the History of Computing 6*, 2 (1984), 139–143.

[13] Nguyen, P. C., Gilray, T., Tobin-Hochstadt, S., and Horn, D. V. Size-change termination as a contract. *CoRR abs/1808.02101* (2018).

[14] Podelski, A., and Rybalchenko, A. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 2004), LICS '04, IEEE Computer Society, pp. 32–41.

[15] Tsitovich, A., Sharygina, N., Wintersteiger, C. M., and Kroening, D. Loop summarization and termination analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2011), Springer, pp. 81–95.

[16] Walther, C. On proving the termination of algorithms by machine. *Artificial Intelligence 71*, 1 (1994), 101–157.