Cooperation For Better Termination Proving

Marc Brockschmidt[1], Byron Cook[2], and Carsten Fuhs[3]

1 **RWTH Aachen University, Germany**
`brockschmidt@cs.rwth-aachen.de`
2 **Microsoft Research and University College London, United Kingdom**
`bycook@microsoft.com`
3 **University College London, United Kingdom**
`c.fuhs@cs.ucl.ac.uk`

Abstract

One of the difficulties of proving program termination is managing the subtle interplay between the finding of a termination argument and the finding of the argument's supporting invariant. In this extended abstract we propose a new mechanism that facilitates better cooperation between these two types of reasoning. In an experimental evaluation we find that our new method leads to dramatic performance improvements.

Keywords and phrases Termination analysis, safety proving, rank functions

1 Introduction

When proving program termination we are simultaneously solving two problems: the search for a termination argument, and the search for a supporting invariant. Consider the example to the right. To prove termination of this program we are looking to find both a termination argument (*i.e.*, "x decreases until 0") and a supporting invariant (*i.e.*, $y > 0$). The two are interrelated: Without $y > 0$, we cannot prove the validity of the argument "x decreases until 0"; and without "x decreases towards 0", how would we know that we need to prove $y > 0$?

```
y := 1;
while x > 0 do
    x := x − y;
    y := y + 1;
done
```

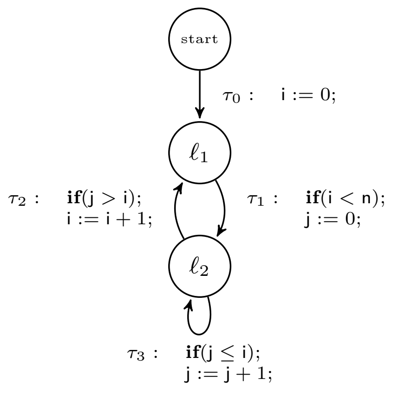
Several program termination proving tools (*e.g.* [9], [10], [11], [15], [17]) address this problem using a strategy that oscillates between calls to an off-the-shelf safety prover (*e.g.* [1], [3], [8], [12], [14], *etc.*) and calls to a rank function synthesis tool (*e.g.* [2], [5], [6], [16], *etc.*). In this setting a candidate termination argument is iteratively constructed. The safety prover proves or disproves the validity of the current argument via the search for invariants. Refinement of the current termination argument is performed using the output of a rank function synthesis tool when applied to counterexamples found by the safety prover.

A difficulty with this approach is that currently, the underlying tools do not share enough information about the overall state of the termination proof. For example, the rank function synthesis tool is only applied to the single path through the program described by the counterexample found by the safety prover, while the context of this single path is not considered at all. Meanwhile, the safety prover is unaware of things such as which paths in the program have already been deemed terminating and how those paths might contribute to other potentially infinite executions. The result is lost performance, as the underlying tools often make choices inappropriate to the common goal of fast termination proving.

Here we introduce a technique that facilitates cooperation between the underlying tools in a termination prover, thus allowing for decisions more appropriate to the common good of proving program termination. The idea is to use a single representation of the state of the termination proof search—called a *cooperation graph*—that both tools operate over. Nodes in the graph are marked as either termination-nodes or safety-nodes to indicate their role

```
i := 0;
while i < n do
    j := 0;
    while j ≤ i do
        j := j + 1;
    done
    i := i + 1;
done
```

(a)


start
τ_0 : i := 0;
ℓ_1
τ_2 : if(j > i); i := i + 1;
τ_1 : if(i < n); j := 0;
ℓ_2
τ_3 : if(j ≤ i); j := j + 1;

(b)

Figure 1 Textual and control-flow graph representation of skeleton bubble sort routine

in the state of the proof. With this additional information exposed, we can now represent the progress of the termination proof search by modifying the termination subgraph. This has practical advantages: the safety prover can be encouraged not to explore parts of the program that have already been proven terminating, and the rank function synthesis can make use of the full program structure in order to find better termination arguments.

Our approach results in significant performance improvements over earlier methods and our implementation succeeds on numerous programs where previous tools fail. In cases where previous tools do succeed, our implementation boosts performance by orders of magnitude.

Limitations. While in theory our approach works in a general setting, our implementation focuses on sequential arithmetic programs (so these programs do not use the heap or bitvectors). In some cases we have soundly abstracted C programs with heap to arithmetic programs (*e.g.* using a technique due to Magill *et al.* [13]); in other cases, as is standard in many tools (*e.g.* SLAM [3]), we essentially ignored bitvectors and the heap.

The full version of the present short paper has been published in [7].

2 Illustrating Example

We illustrate our approach using the example in Fig. 1, which displays a bubble-sort like program (the manipulation of the data has been abstracted away). In our setting we use a graph—called a *cooperation graph*—to facilitate sharing of information between a safety prover and a rank function synthesis procedure. See Fig. 2 for the cooperation graph at the start of the proof search. We start with the control-flow graph of the original program from Fig. 1, which we keep for reasoning about safety (*i.e.*, (un)reachability from the initial program states). Intuitively, this part of the graph is for the finite prefix of a run from the initial location to a loop with a (hypothetical) infinite suffix of the run. For this infinite suffix, we have duplicated the loops of the original program (in the form of the *strongly-connected components (SCCs)* of the graph with locations ℓ_1^t and ℓ_2^t). We connect the two parts of the graph with non-deterministic transitions from one copy of the program to the other (*i.e.*, τ_4 and τ_5). Technically, the cooperation graph contains a superset of the transitions in the initial program, yet if we can prove that there is no infinite run from the initial location where ℓ_1^t or ℓ_2^t occur infinitely often, this implies termination of the original program as well.

After duplication, we also apply a few known tricks: In the new copy of the program, we follow the approach of Biere *et al.* [4] by adding nodes (*i.e.*, ℓ_1^d and ℓ_2^d) and transitions to take

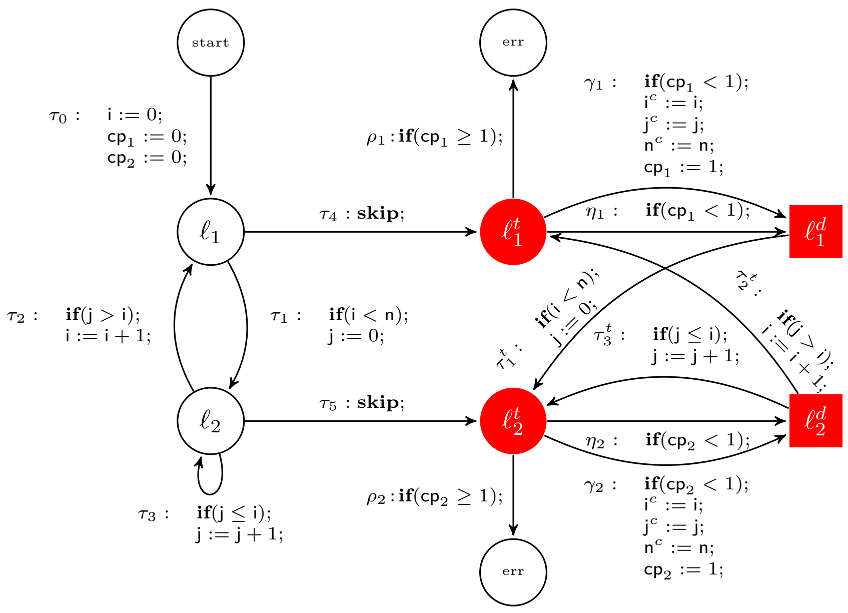

start
err
τ_0 : i := 0; cp$_1$:= 0; cp$_2$:= 0;
ℓ_1
τ_4 : **skip**;
ρ_1 : **if**(cp$_1$ ≥ 1);
γ_1 : **if**(cp$_1$ < 1); i^c := i; j^c := j; n^c := n; cp$_1$:= 1;
ℓ_1^t
η_1 : **if**(cp$_1$ < 1);
ℓ_1^d
τ_2 : **if**(j > i); i := i + 1;
τ_1 : **if**(i < n); j := 0;
τ_1^t : **if**(i < n); j := 0;
τ_3^t : **if**(j ≤ i); j := j + 1;
τ_2^t : **if**(j > i); i := i + 1;
ℓ_2
τ_5 : **skip**;
ℓ_2^t
ℓ_2^d
η_2 : **if**(cp$_2$ < 1);
τ_3 : **if**(j ≤ i); j := j + 1;
ρ_2 : **if**(cp$_2$ ≥ 1);
γ_2 : **if**(cp$_2$ < 1); i^c := i; j^c := j; n^c := n; cp$_2$:= 1;
err

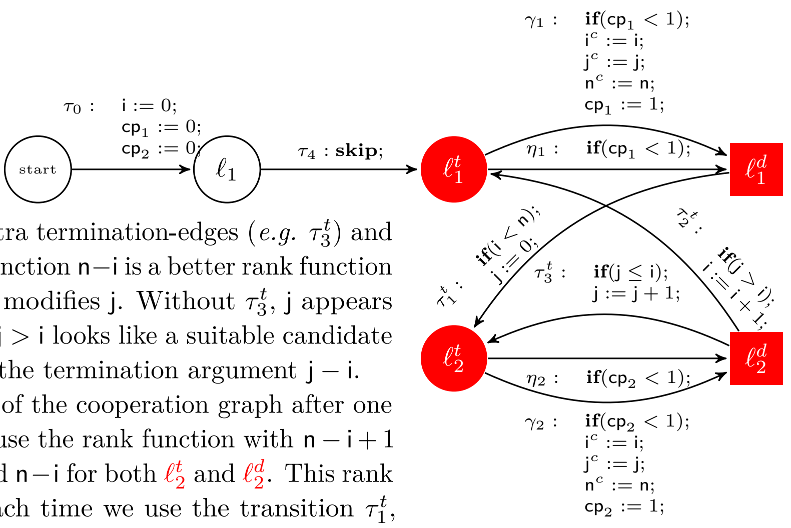
Figure 2 Cooperation graph derived from Fig. 1

a snapshot of variable values (*i.e.*, γ_1 and γ_2). The current values of variables $\mathsf{i}, \mathsf{j}, \mathsf{n}$ are stored in copies $\mathsf{i}^c, \mathsf{j}^c, \mathsf{n}^c$ and the flag cp_k is set to indicate that a snapshot was taken at location ℓ_k. Furthermore, new transitions to an error location "err" have been added that can be strengthened later by partial termination arguments *à la* Cook *et al.* [9]. Proving this error location unreachable then implies a termination proof for the input program. In the resulting graph, reasoning about termination is performed on the right-hand side (the *termination subgraph*) by a procedure built around an efficient rank function synthesis. We search for supporting invariants on the left-hand side (the *safety subgraph*) via a safety prover.

Via this duplication to the termination and safety subgraphs, we can easily restrict certain operations to either subgraph, yet still maintain a connection between them. The safety subgraph describes an over-approximation of all reachable states, while the termination subgraph is an over-approximation of those states whose termination has not been proven yet. This allows us to perform operations in the one half that may not make sense (or may be unsound) in the other: when we prove that transitions in the termination subgraph can only be used finitely often, we can simply remove them, as they cannot contribute to infinite executions. This is only sound because the safety subgraph remains unchanged in this simplification, which keeps the set of reachable states unchanged and hence allows reasoning about safety/invariants. These iterative program simplifications encode the progress of the termination proof search and are directly available to the safety prover.

The graph structure guides the safety prover to unproven parts of the program yielding relevant counterexamples and allowing the rank function synthesis to produce better termination arguments. If these do not allow a program simplification, they still guide the invariant generation by the safety prover for nodes in the safety subgraph. The invariants in turn support reasoning about the validity of termination arguments in the termination subgraph.

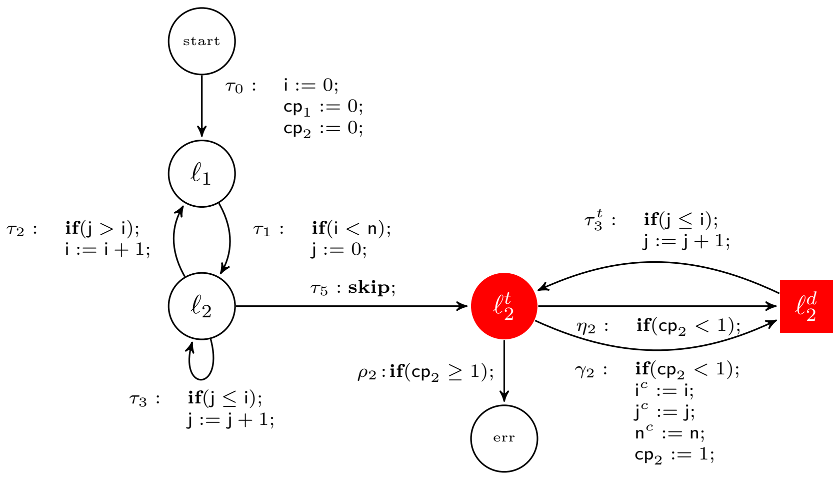
Termination proof sketch. In our example, we begin searching for a path from the "start" location to the error location "err". We might, for example, choose the path $\langle \tau_0, \tau_4, \gamma_1, \tau_1^t, \eta_2, \tau_3^t, \eta_2, \tau_2^t, \rho_1 \rangle$ where τ_0 is drawn from the safety subgraph and the other transitions come from the termination subgraph. Here, $\langle \gamma_1, \tau_1^t, \eta_2, \tau_3^t, \eta_2, \tau_2^t \rangle$ form a cycle in the execution, returning back to location ℓ_1^t. In our approach we do not simply use this command sequence directly to search for a new termination argument (as is done in previous tools). Instead, we additionally consider all transitions from the termination subgraph that enter and exit nodes in the strongly connected component containing the found cycle of termination-transitions in the counterexample. In this case, because the graph is so small, this includes the entire termination subgraph:



γ_1 : if(cp$_1$ < 1); i^c := i; j^c := j; n^c := n; cp$_1$:= 1;
τ_0 : i := 0; cp$_1$:= 0; cp$_2$:= 0;
start
ℓ_1
τ_4 : skip;
ℓ_1^t
η_1 : if(cp$_1$ < 1);
ℓ_1^d
τ_1^t : if(i < n); j := 0;
τ_2^t : if(j > i); i := i + 1;
τ_3^t : if(j ≤ i); j := j + 1;
ℓ_2^t
η_2 : if(cp$_2$ < 1);
ℓ_2^d
γ_2 : if(cp$_2$ < 1); i^c := i; j^c := j; n^c := n; cp$_2$:= 1;

This graph has extra termination-edges (*e.g.* τ_3^t) and shows that the rank function $\mathsf{n}-\mathsf{i}$ is a better rank function than $\mathsf{j}-\mathsf{i}$ because τ_3^t modifies j. Without τ_3^t, j appears as a constant so that $\mathsf{j} > \mathsf{i}$ looks like a suitable candidate invariant supporting the termination argument $\mathsf{j}-\mathsf{i}$.

Fig. 3 is the state of the cooperation graph after one counterexample. We use the rank function with $\mathsf{n}-\mathsf{i}+1$ for both ℓ_1^t and ℓ_1^d, and $\mathsf{n}-\mathsf{i}$ for both ℓ_2^t and ℓ_2^d. This rank function decreases each time we use the transition τ_1^t, and the condition $\mathsf{i} < \mathsf{n}$ implies that the rank function is bounded from below. So τ_1^t can only be used finitely often and we can remove it from the termination subgraph. This also allows to remove ℓ_1^t, ℓ_1^d and all transitions connected to the two, as they are not on a non-trivial SCC anymore and thus cannot occur infinitely often in an execution. Removing the corresponding node ℓ_1 from the safety subgraph is unsound, as this would make the inner loop unreachable, without any termination proof for it.



start
τ_0 : i := 0; cp$_1$:= 0; cp$_2$:= 0;
ℓ_1
τ_2 : if(j > i); i := i + 1;
τ_1 : if(i < n); j := 0;
ℓ_2
τ_3 : if(j ≤ i); j := j + 1;
τ_5 : skip;
ℓ_2^t
τ_3^t : if(j ≤ i); j := j + 1;
ℓ_2^d
η_2 : if(cp$_2$ < 1);
ρ_2 : if(cp$_2$ ≥ 1);
err
γ_2 : if(cp$_2$ < 1); i^c := i; j^c := j; n^c := n; cp$_2$:= 1;

Figure 3 Cooperation graph after safety and termination analysis on the graph from Fig. 2. Due to termination analysis, the transition τ_2^t has been removed. Afterward, ℓ_1^t was not part of a non-trivial SCC anymore, so it, its duplicate ℓ_1^d, and the connecting transitions were removed.

In the next iteration, starting on Fig. 3, all possible cycles in the termination subgraph use the transition τ_3^t. We prove this transition well founded via the rank function $\mathsf{i}-\mathsf{j}$ for the locations ℓ_2^t and ℓ_2^d, allowing us to remove τ_3^t and then, ℓ_2^t, ℓ_2^d and all connected transitions. This yields a cooperation graph with an empty termination subgraph (so we are left with what is essentially the original graph from Fig. 1). Thus we have proved termination.

3 Conclusion

One of the difficulties for reliable and scalable program termination provers is orchestrating the interplay between the reasoning about progress and the search for supporting invariants. We have developed a new method that facilitates cooperation between these two types of reasoning. Our representation gives the underlying tools the whole picture of the current proof state, allowing both types of reasoning to contribute towards the greater goal and also to share their intermediate findings. Our experiments (which we cannot present here for space reasons; details on experiments and benchmarks are available at `http://verify.rwth-aachen.de/brockschmidt/Cooperating-T2/` and in [7]) indicate dramatic performance gains.

The full version of this short paper has been published at [7], and our implementation in T2 is available for download at `http://research.microsoft.com/en-us/projects/t2/`.

References

1 Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: an interpolation-based algorithm for inter-procedural verification. In *Proc. VMCAI '12*.

2 Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. SAS '10*.

3 Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. CAV '01*.

4 Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *Proc. FMICS '02*.

5 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. CAV '05*.

6 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Proc. ICALP '05*.

7 Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*.

8 Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS '05*.

9 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*.

10 Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*.

11 Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. PLDI '12*.

12 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proc. SPIN '03*.

13 Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*.

14 Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*.

15 Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *Proc. PADL '07*.

16 Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. VMCAI '04*.

17 Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Proc. TACAS '11*.