RWTH Aachen

Department of Computer Science
Technical Report

Better termination proving through cooperation

Marc Brockschmidt, Byron Cook, Carsten Fuhs

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2013-06

RWTH Aachen · Department of Computer Science · April 2013

Better termination proving through cooperation

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

[1] RWTH Aachen University
[2] Microsoft Research Cambridge
[3] University College London

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 paper 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.

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 following example:

$\mathsf{y} := 1;$
while $\mathsf{x} > 0$ **do**
  $\mathsf{x} := \mathsf{x} - \mathsf{y};$
  $\mathsf{y} := \mathsf{y} + 1;$
done

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.*, $\mathsf{y} > 0$). The two are interrelated: Without $\mathsf{y} > 0$, we cannot prove the validity of the (safety) property "x decreases until 0"; and without "x decreases towards 0", how would we know that we need to prove $\mathsf{y} > 0$?

Several program termination proving tools (*e.g.* [15], [16], [23], [34], [39]) address this problem using a strategy that oscillates between calls to an off-the-shelf safety prover (*e.g.* [1], [4], [11], [26], [31], *etc.*) and calls to a rank function synthesis tool (*e.g.* [2], [7], [8], [35], *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.

In this paper 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, thus indicating the role they play 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. For example, the safety prover can be encouraged not to explore parts of the program that have already been proven terminating. On the rank function synthesis side, we can make use of the full program structure in order to find better termination arguments.

Our approach results in dramatic performance improvements compared to earlier methods and our implementation succeeds on numerous programs on which previous tools fail. In cases where previous tools do succeed, our implementation increases performance by orders of magnitude.

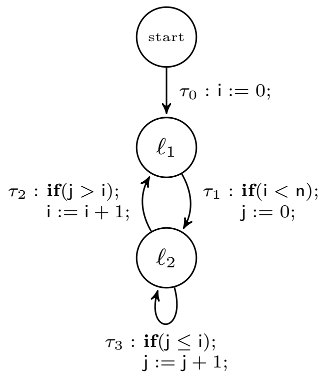
Related work. Numerous tools and techniques exist for termination proving (*e.g.* [5], [7], [8], [10], [15], [17], [20], [21], [29], [34], [39], *etc.*). In many instances our approach is related but essentially incomparable with these previous tools. For example, size-change termination proving [29] sacrifices precision for consistency with a fixed *a priori* finite abstraction and an essentially fixed termination argument. The result is an analysis that will fail to prove termination in more complex cases, but that itself always terminates. This is in contrast to our technique which privileges precision over predictability (*e.g.* we use possibly non-terminating techniques during the search for supporting invariants).

The tools most similar to our own are ARMC [34], TRex [25], CProver [39], HSF [23], Terminator [15], and T2 [16]. As discussed above, the key difference here is in our treatment of shared information. These previous tools share only simple paths with the rank function synthesis procedure, and only the termination argument with the safety-based validity proving procedure. Our cooperation graph, while similar in principle to previous representations (*e.g.* [15]), exposes information in a way that facilitates operations on the graph that would have been difficult or unsound in previous approaches. To see the difference, we look to the experimental results which show a dramatic improvement over previous approaches when our technique is applied.

In order to make use of the information that we have exposed we borrow several existing techniques. For example, we adapt a program simplification strategy from the dependency pair framework [3, 22, 27] to our shared graph as a way of recording lemmas during the proof search. We use a recently developed tech-

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

(a)



(b)

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

nique for efficient rank function synthesis with multiple control-flow locations [2]. Finally, we build upon a recently developed iterative method for finding lexicographic rank functions [16]. Our cooperation graphs facilitate the combination of these complementary techniques, leading to a new tool that outperforms all of the previous approaches.

Limitations. While in theory our approach works in a general setting, in our implementation we are focusing on sequential arithmetic programs (*e.g.* 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.* [30]); in other cases, as is standard in many tools (*e.g.* SLAM [4]), we essentially ignored bitvectors and the heap. Techniques that more accurately and efficiently reason about mixtures of heap and arithmetic are an area of open interest.

2 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. Here we have essentially duplicated the loops in the original program, with non-deterministic transitions from one copy of the program to the other (*i.e.*, τ_4 and τ_5). After duplication, we apply a few known tricks: In the new copy of the program, we follow the approach of Biere *et al.* [6] by adding nodes (*i.e.*, ℓ_1^d and ℓ_2^d) and transitions to take 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

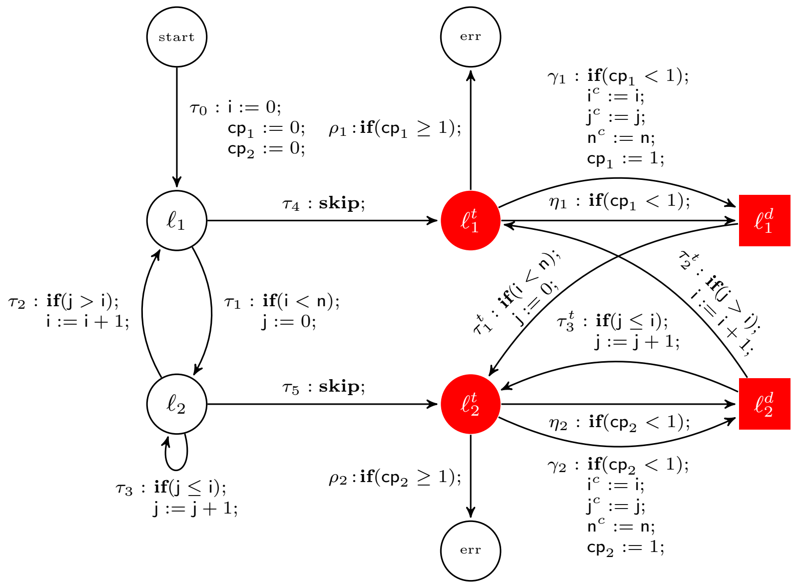


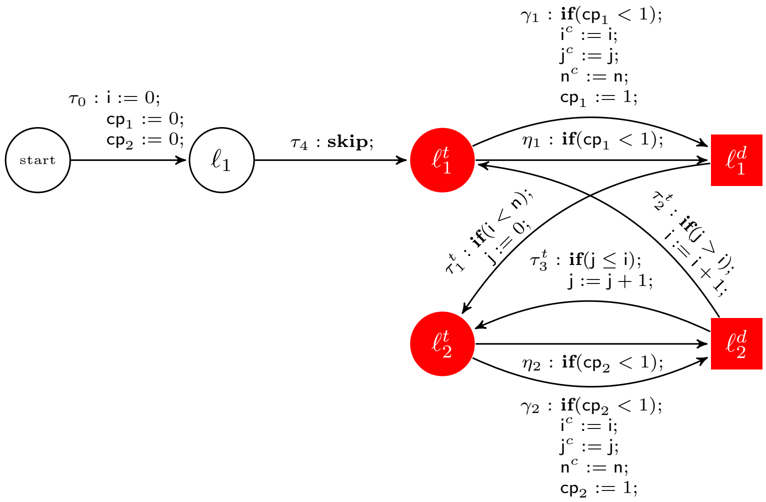
Fig. 2. Cooperation graph derived from Fig. 1

location "err" have been added that—using the approach of Cook *et al.* [15]—can be strengthened later by partial termination arguments. 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—called the termination subgraph—by a procedure built around an efficient rank function synthesis. The search for supporting invariants is performed on the left-hand side—called the safety subgraph—by a safety prover.

The advantage of the duplication (*i.e.* the termination and safety subgraphs) is that we can easily restrict certain operations to either subgraph, but we maintain a connection between them. We use the safety subgraph to describe an over-approximation of all reachable states, while the termination subgraph is an over-approximation of those states for which 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. For example, 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. In our setting, these iterative program simplifications encode the progress of the termination proof search and are directly available to the safety prover when searching for more counterexamples.

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

Termination proof sketch. We now illustrate how termination is proved in our setting. 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 (SCC) containing the found cycle of termination-transitions in the counterexample. We call this enclosing SCC the *SCC context* of a certain cycle. In this case, because the graph is so small, this includes the entire termination subgraph:



By examining a graph that includes extra termination-edges (*e.g.* τ_3^t) we can see 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 and hence, $\mathsf{j} > \mathsf{i}$ looks like a suitable candidate invariant supporting the termination argument $\mathsf{j} - \mathsf{i}$.

Fig. 3 is the state of the cooperation graph after considering 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. The value of this rank function is decreasing 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. Hence, τ_1^t can only be used finitely often and we can

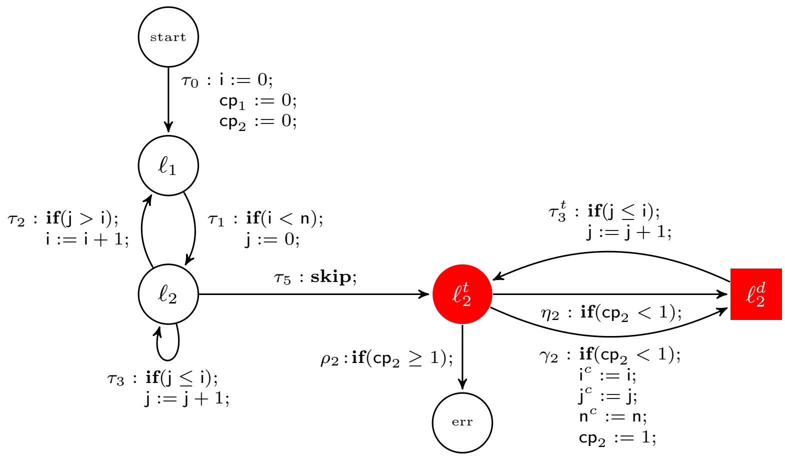


Fig. 3. Cooperation graph after safety and termination analysis on the graph from Fig. 2. Due to termination analysis, the transition τ_2^t has been removed. Afterwards, ℓ_1^t was not part of a non-trivial SCC anymore, so it, its duplicate ℓ_1^d, and the connecting transitions were removed.

remove it from the termination subgraph. Removing this transition is helpful for future iterations of the proof search, as it no longer needs to be considered when searching for further counterexamples. This also allows to remove ℓ_1^t, ℓ_1^d and all transitions connected to the two, as they are not part of a non-trivial strongly connected component anymore and hence cannot occur infinitely often in an execution. Because these nodes and transitions are only used to reason about termination, and not safety, we can soundly remove them. Removing the corresponding node ℓ_1 from the safety subgraph is unsound, as this would make the inner loop unreachable, without requiring a termination proof for it. In our setting we use an incremental implementation of lazy abstraction with interpolation (*à la* IMPACT [31]) to represent the inductive invariants for safety proving. We are not displaying the additional information inferred by this safety proving method in our cooperation graph here.

In the next iteration, starting on Fig. 3, all possible cycles allowed in the termination subgraph use the transition τ_3^t. This transition can easily be proved well-founded with 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, leaving us with a cooperation graph with an empty termination subgraph (*i.e.*, with an empty termination subgraph we are left with what is essentially the original graph from Fig. 1). Thus we have proved termination. (In practice, our algorithm in Fig. 5 handles simple examples such as this one already in a preprocessing step. Here we have given a counterexample-based termination proof for illustration purposes.)

3 Algorithm

In this section, we describe our new termination proving method more formally.

Preliminaries. We represent programs as graphs of program locations connected by transition rules with conditions and assignments to a set of integer variables $\mathcal{V}$. The canonical initial location is called *start*. Program states are tuples $(k, \boldsymbol{x})$, with k the current program location and $\boldsymbol{x}$ a column vector of the values of $\mathcal{V}$ in some fixed order. Transitions are labeled by formulas relating pre- and post-variables, where x' is the post-variable corresponding to the pre-variable x (resp. $\boldsymbol{x}$ are all pre-variables in fixed order, and $\boldsymbol{x'}$ the post-variables). The statement $\mathsf{i} := \mathsf{i} + 1$ is represented as $i' = i + 1$ and $\mathbf{if}(\mathsf{i} < \mathsf{n})$ as $i < n$. For example, the commands on transition τ_1 are equivalent to the formula $i < n \wedge i' = i \wedge j' = 0 \wedge n' = n$. In this paper, we only consider linear program transitions and hence use the constraint system $A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}$ instead of the corresponding formula.

A program execution is a possibly infinite sequence of program states $(k_1, \boldsymbol{x}_1)$, $(k_2, \boldsymbol{x}_2), \ldots$ with $k_1 = start$, $\boldsymbol{x}_1$ freely chosen and for each pair $(k_i, \boldsymbol{x}_i), (k_{i+1}, \boldsymbol{x}_{i+1})$, there is a program transition $(k_i, A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}, k_{i+1})$ such that $A\left(\begin{smallmatrix}\boldsymbol{x}_i\\ \boldsymbol{x}_{i+1}\end{smallmatrix}\right) \geq \boldsymbol{a}$ holds. We call a program terminating if and only if it has no infinite execution.

Finding termination arguments. Past tools used constraint-based approaches for finding rank functions for (sub)programs involving only one program location (*e.g.* ARMC [34] and TERMINATOR [15] use Podelski & Rybalchenko's rank function synthesis method [35], T2 [16] uses the approach due to Bradley *et al.* [7] for lexicographic rank functions). In our setting, we need to find rank functions for the SCC contexts of counterexamples in the termination subgraph, which might involve transitions over several program points. For this purpose we use the lexicographic rank function synthesis due to Alias *et al.* [2] to find linear rank functions for a set of transitions using possibly several program locations.

Given a (finite) set of program transitions $\mathcal{T}$, we prove termination iteratively. When proving that transitions cannot be used infinitely often in an infinite execution, we use an approach from the dependency pair framework [3, 22, 27] to remove them. For this, we choose a sequence of rank functions $f^1, \ldots, f^m$ that measure program states. A *$\mathcal{T}$-orienting rank function* f is a measure of program states in some well-founded ordered domain such that no transition $t \in \mathcal{T}$ allows an increase of this measure, *i.e.*, we require:

$$\bigwedge_{(k, A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}, k') \in \mathcal{T}} \forall \boldsymbol{x}, \boldsymbol{x'}. A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a} \rightarrow f((k, \boldsymbol{x})) \geq f((k', \boldsymbol{x'})) \tag{1}$$

Furthermore, we want that for at least one of the transitions $t = (k, A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}, k')$ in $\mathcal{T}$ the measure is actually decreasing and is bounded from below (0 is a minimal element in our domain):

$$\forall \boldsymbol{x}, \boldsymbol{x'}. A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a} \rightarrow (f((k, \boldsymbol{x})) > f((k', \boldsymbol{x'})) \wedge f((k, \boldsymbol{x})) \geq 0) \tag{2}$$

Similar to the dependency pair framework [3, 22, 27] and to monotonicity constraints [12], we compose lexicographic termination arguments from such $\mathcal{T}$-orienting rank functions. If $\textsc{Decreasing}(\mathcal{T}, f) \subseteq \mathcal{T}$ is the set of transitions for which (2) holds for some f with (1), then for proving termination it suffices to consider executions that use only transitions from $\mathcal{T} \setminus \textsc{Decreasing}(\mathcal{T}, f)$ infinitely often (see also Appendix A).

Consequently, we construct lexicographic termination arguments for a set of transitions $\mathcal{T}$ by iteratively synthesizing such rank functions f. A transition $\delta \in \text{DECREASING}(\mathcal{T}, f)$ can only occur a finite number of times, so we ignore it for the rest of our termination proof and only consider suffixes of infinite executions that do not use δ anymore. In our cooperation graphs, we build upon this observation by removing transitions from the termination subgraph. There, any finite prefix of a computation can be represented using the (unchanged) safety subgraph, while the infinite suffix of a possibly non-terminating computation is represented by the simplified termination subgraph. By repeatedly removing transitions using different rank functions $f^1, \ldots, f^m$, we mirror the progress of building a lexicographic termination argument in the termination subgraph.

Cooperation graphs. The procedure INSTRUMENT, from Fig. 4, is used to construct an initial cooperation graph with transitions $\mathcal{C}$ from a program $\mathcal{P}$ with locations $\mathcal{L}$ and transitions $\mathcal{T}$. We use two mappings SAFETYLOC and TERMINATIONLOC from $\mathcal{L}$ to fresh location names. We first create the safety subgraph of the cooperation graph as a copy of $\mathcal{P}$. For the termination subgraph, we first use SCC_TRANSITIONS to identify all transitions on components that may influence termination, *i.e.*, all non-trivial strongly connected components in the control-flow graph of $\mathcal{P}$, and copy these to the termination subgraph. We then connect the safety and termination subgraphs at cutpoints [19] of the original program, allowing a non-deterministic jump from the safety to the termination location.

We then apply the safety-reduction from Cook *et al.* [16] on cutpoints in the termination subgraph. The point of this reduction is to add an error location that is reachable *iff* the lexicographic termination argument is invalid. For this, we use a mapping CUTPOINTDUPLICATE from cutpoints in the original program to fresh location names. We first "move" all transitions originally starting in the termination copy of the cutpoint p^t to its new duplicate. We then connect p^t to its duplicate by two transitions, one taking a snapshot of the current variable state, one doing nothing. In our example in Fig. 1, ℓ_1 is a cutpoint and we choose $\text{CUTPOINTDUPLICATE}(\ell_1) = \ell_1^d$. The function SNAPSHOT produces the assignments needed to take a snapshot of the variables, *i.e.*, storing copies of variables v in an extra variable v^c and setting an integer flag cp_k that indicates that a snapshot at location k was taken. An example of the result is γ_1 from Fig. 2. Its twin NOSNAPSHOT does not do anything. Note that both resulting transitions can only be used if no snapshot of the program variables was taken at this program point before. Finally, we connect p^t to the error location by a transition that assumes that no decrease was found using the current set of rank functions. This set of rank functions is initially empty, and will be strengthened in the termination proof. Hence, the function NODECREASE only returns the condition stating that a snapshot has been taken (*e.g.*, for ℓ_2^t we have $\mathsf{cp}_2 \geq 1$).

We define projections SAFETY and TERMINATION on the cooperation graph. $\text{SAFETY}(\mathcal{C})$ are the transitions in $\mathcal{C}$ between locations in $range(\text{SAFETYLOC})$, while the projection $\text{TERMINATION}(\mathcal{C})$ are the transitions between locations in $range(\text{TERMINATIONLOC}) \cup range(\text{CUTPOINTDUPLICATE}) \cup \{err\}$. The safety projection $\text{SAFETY}(\mathcal{C})$ is isomorphic to the original program, and the termination

Input: Program with transitions $\mathcal{T}$, start location *start*
Output: Cooperation graph $\mathcal{C}$ with start location $\textsc{SafetyLoc}(start)$

$\mathcal{C} := \emptyset$
for all (ℓ, τ, ℓ') in $\mathcal{T}$ **do**
  $\mathcal{C} := \mathcal{C} \cup \{(\textsc{SafetyLoc}(\ell), \tau, \textsc{SafetyLoc}(\ell'))\}$
end for
for all (ℓ, τ, ℓ') in $\textsc{SCC\_Transitions}(\mathcal{T})$ **do**
  $\mathcal{C} := \mathcal{C} \cup \{(\textsc{TerminationLoc}(\ell), \tau, \textsc{TerminationLoc}(\ell'))\}$
end for
for all p in $\textsc{Cutpoints}(\mathcal{T})$ **do**
  $p^t := \textsc{TerminationLoc}(p)$
  $p^d := \textsc{CutpointDuplicate}(p)$
  $\mathcal{C} := \mathcal{C} \cup \{(\textsc{SafetyLoc}(p), \textbf{skip}, p^t)\}$
  for all (p^t, τ, ℓ') in $\mathcal{C}$ **do**
    $\mathcal{C} := (\mathcal{C} \setminus \{(p^t, \tau, \ell')\}) \cup \{(p^d, \tau, \ell')\}$
  end for
  $\mathcal{C} := \mathcal{C} \cup \{(p^t, \textsc{snapshot}(p), p^d), (p^t, \textsc{nosnapshot}(p), p^d)\}$
  $\mathcal{C} := \mathcal{C} \cup \{(p^t, \textsc{nodecrease}(p), err)\}$
end for
return $\mathcal{C}$

Fig. 4. Procedure Instrument, which initializes a new cooperation graph.

projection corresponds to a termination problem without explicit start state.

In our termination proofs, $\textsc{Safety}(\mathcal{C})$ represents the set of reachable states, and thus remains unchanged. In practice we use an incremental safety prover on this graph to find the necessary inductive invariants on demand. Meanwhile, $\textsc{Termination}(\mathcal{C})$ represents the set of states for which we have not proven termination yet. We change $\textsc{Termination}(\mathcal{C})$ in each iteration of the algorithm by possibly removing transitions and strengthening the conditions of the transitions from cutpoints to the error location. Consequently, questions of reachability and validity of invariants are based on the safety projection.

Refinement algorithm. Our cooperation-based termination procedure is found in Fig. 5. We first use Instrument to create a cooperation graph from our input program. Then, we try to find (partial) lexicographic rank functions to simplify SCCs in the termination part of the graph, where $\textsc{Decreasing}(\mathcal{S}, f)$ identifies the transition rules satisfying (2) from above.

In simple examples such as that of Sect. 2, this preprocessing step can actually already prove termination, by removing all possible paths to the error location before the main loop begins. In more complex cases, we enter the main loop, in which we search for counterexamples to the decrease of the rank functions found so far. Our counterexamples are lassos, with the cycle part in the termination subgraph, starting in some cutpoint p, while the stem can always be represented using only the safety subgraph. In the cycle, we first take a snapshot of the current variable state and then return back to the termination copy of the cutpoint p, finding that the current set of rank functions do not show a decrease.

Input: Program with start state *start*, transitions $\mathcal{T}$
Output: "**Terminating**" or "**Unknown**"

$\mathcal{C} := \textsc{Instrument}(\mathcal{T})$
for all $\mathcal{S}$ in $\text{SCC}\textsc{s}(\textsc{Termination}(\mathcal{C}))$ **do**
  while $\exists$ $\mathcal{S}$-orienting rank function f **do**
    $\mathcal{C} := \mathcal{C} \setminus \textsc{Decreasing}(\mathcal{S}, f)$
    $\mathcal{S} := \mathcal{S} \setminus \textsc{Decreasing}(\mathcal{S}, f)$
  end while
end for
while $\exists$ counterexample $(\mathit{stem}, \mathit{cycle})$ from $\textsc{SafetyLoc}(\mathit{start})$ to err in $\mathcal{C}$ **do**
  $\mathcal{S} := \text{SCC\_}\textsc{Context}(\textsc{Termination}(\mathcal{C}), \mathit{cycle})$
  if $\exists$ $\mathcal{S}$-orienting rank function f **then**
    $\mathcal{C} := \mathcal{C} \setminus \textsc{Decreasing}(\mathcal{S}, f)$
    $\mathcal{C} := \textsc{Strengthen}(\mathcal{C}, \mathit{cycle}, f)$
  else if $\exists$ any rank function f for *cycle* **then**
    $\mathcal{C} := \textsc{Strengthen}(\mathcal{C}, \mathit{cycle}, f)$
  else
    return "Unknown"
  end if
end while
return "Terminating"

Fig. 5. Procedure REFINEMENT, which oscillates between a safety prover and a rank function synthesis tool using a cooperation graph.

The counterexample is then used to synthesize a new rank function f. We first determine the SCC context of the cycle in the termination subgraph of the cooperation graph. We then try to find a rank function that is non-increasing for the SCC context of the cycle and decreases for our counterexample. If we find such a rank function, we remove any transitions that we have proven decreasing in all cases from the termination subgraph. We additionally use STRENGTHEN to restrict the transition from the cutpoint p in the counterexample to the error location further, *i.e.*, we only allow going to the error location if the newly found rank function does not decrease.

The procedure STRENGTHEN refines the partial termination argument in the safety-representation, as is done in previous tools (*e.g.* [15], [16], [23], [34], [25], [39]). We use lexicographic termination arguments as in Cook *et al.* [16]. Such an argument has the form $\langle f_1, \ldots, f_n \rangle$, where the f_i are the rank functions at some cutpoint p. If we can find a $\mathcal{S}$-orienting rank function, we can always prepend it to an existing lexicographic termination argument (computed "bottom-up"). If no such rank function could be found, we fall back to the method from Cook *et al.* [16], synthesizing a rank function such that a lexicographic termination argument for the counterexamples found so far can be constructed.

We then construct constraints that only allow a transition if an iteration did not make the variable state decrease w.r.t. this argument. Formally, we use the snapshots of old variable values to construct the pre-state $s = (p, \boldsymbol{x})$ at the beginning of the loop iteration and create the post-state $s' = (p, \boldsymbol{x}')$ of the loop iteration from the current variables. Then, Strengthen encodes the following condition:[4]

$$\neg\big(\bigvee_{1\leq i\leq n} f_i(s) > f_i(s') \wedge f_i(s) \geq 0 \wedge (\bigwedge_{1\leq j<i} f_j(s) \geq f_j(s'))\big)$$

Strengthen uses the cycle passed as an argument to determine at which cutpoint to strengthen the termination argument. If we could find no rank function for the cycle, we give up.[5] Finally, if no counterexamples exist anymore, we report termination. For a correctness proof of our termination proving procedure in Fig. 5, please see Appendix A.

As in earlier work, this use of Strengthen allows the termination prover to speculate termination arguments based on single counterexamples. The safety prover then has to find invariants proving the speculated argument to be correct, or provide more counterexamples. However, in our cooperation graph setting, the safety prover is helped in this by the removal of transitions proven to be terminating. This both speeds up the state space exploration and avoids to refute many spurious counterexamples. Moreover, by splitting executions into finite prefixes (in the safety subgraph) and possibly infinite suffixes (in the termination subgraph), the safety prover can infer "eventual invariants", *i.e.*, formulas that always hold after a finite time.

4 Evaluation

To evaluate the usefulness of our idea we have compared our implementation against the following tools/configurations:

- Terminator [15], which implements an oscillation between rank function synthesis and safety using termination arguments expressed as transition invariants [36].
- T2 [16], which implements a Terminator-like oscillation between rank function synthesis and safety using termination arguments expressed as lexicographic rank functions.
- Cooperating-T2: Our implementation of the procedure from Fig. 5, which is based on T2.[6]
- ARMC [34], which also implements a Terminator-like procedure. Note that, as of the writing of this paper, Rybalchenko's C-to-clauses converter was not complete, and thus we could not compare against HSF. Based on experience with the two tools we expect that ARMC and HSF will have comparable results when proving termination [38].
- AProVE [21], a termination prover based on the dependency pair framework [3, 22, 27] and including an implementation of the rank function synthesis *à la* Alias *et al.* [2]. AProVE does not generate invariants on demand and hence always uses the supporting invariant **true**.

[4] Disjunctions in transition conditions can be expressed using several transitions.

[5] Actually, we then attempt to prove non-termination, but the details of that procedure are orthogonal to the point of this paper.

[6] For details on accessing a source-based release of this tool, please see Appendix C.

– APROVE+INTERPROC, which uses the abstract interpretation tool INTERPROC [28] to generate as many invariants as possible using the Octagon abstract domain [33] before running APROVE.
– SIZE-CHANGE/MCNP, an implementation of termination proofs via monotonicity constraints [12], an efficient generalization of the size-change principle [29]. The abstraction from integer programs to monotonicity constraints is implemented in APROVE.
– KITTEL, another termination prover based on termination proving techniques from rewriting systems [18].

During our evaluation we ran tools on a set of 449 termination proving benchmarks drawn from a variety of applications that were also used in prior tool evaluations (*e.g.* Windows device drivers, the APACHE web server, the POSTGRESQL server, integer approximations of numerical programs from a book on numerical recipes [37], integer approximations of benchmarks from LLBMC [32] and other tool evaluations). Of these, 260 are known to be terminating and 181 are known to be non-terminating. For a handful examples, no result is known. These include the Collatz conjecture, and the remaining are very large and hence have not been analyzed manually. Our benchmarks and results can be found at

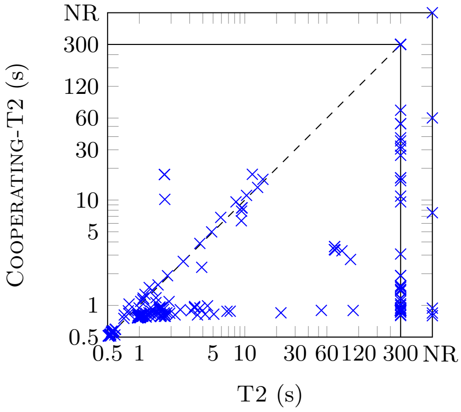
`http://verify.rwth-aachen.de/brockschmidt/Cooperating-T2/`

Experiments for TERMINATOR, T2, and COOPERATING-T2 were performed on a quadcore 2.26GHz E5520 system with 4GB of RAM and running Windows 7. All other experiments were performed on a quadcore 3.07GHz Core i7 system with 4GB of RAM and running Debian Linux 6. We ran all tools with a timeout of 300 seconds. When a tool returned early without a definite result or crashed, we display this in the plots using the special "NR" (no result) value.

	Term	Non-Term
COOPERATING-T2	91.4%	96%
APROVE	73.5%	n.a.
KITTEL	73.1%	n.a.
T2	70.5%	99%
APROVE+INTERPROC	69.0%	n.a.
TERMINATOR	66.0%	100%
SIZE-CHANGE/MCNP	58.2%	n.a.
ARMC	51.5%	n.a.

Fig. 8. Evaluation overview

The results of our test runs are displayed in Figs. 6–8. Fig. 6 contains two plots which chart the difference between our new procedure and T2's previous procedure, in log scale. Plot **(a)** represents the results when applied to programs that terminate. Plot **(b)** contains the results from non-terminating benchmarks. Here both configurations of T2 use an approach similar to the approach used in TNT [24]. Our method from Fig. 5 has a fixed overhead, making non-terminating proofs for examples where the first counterexample already suffices to find a non-termination argument slower. Additionally, our method exposes a performance/non-termination bug in Z3 in a few cases, leading to some additional timeouts. In non-terminating examples with many other terminating loops, our program simplifications speed up the search for a non-termination proof. Fig. 7 compares our procedure to the other termination proving tools (to accommodate that not all tools support non-termination proofs, we only consider those examples that are not known to be non-terminating here). Fig. 8 gives the percentages of benchmarks proved



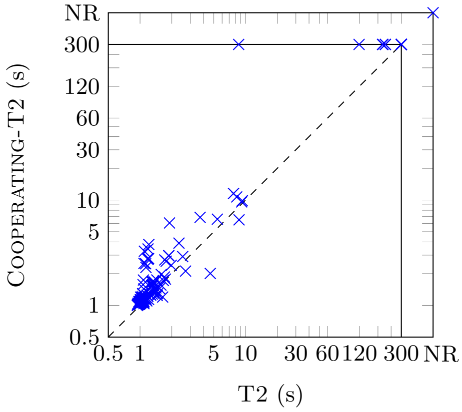


Fig. 6. Evaluation results of Cooperating-T2 vs. Standard T2, in log scale. Plot **(a)** represents the results of the two tools on terminating benchmarks, **(b)** represents non-terminating benchmarks. Timeout=300s. NR="No Result". The NR cases are due to failure of the underlying safety prover to find an inductive invariant.

(non-)terminating by the respective tools. The improvement for terminating benchmarks is dramatic: Cooperating-T2 times out or fails far less often than competing tools. On non-terminating benchmarks, the difference is small.

Discussion. Overall, the performance gains of our approach over previous techniques are dramatic. Our method does not just speed up termination proving, it makes a dramatic improvement in cases where previous tools time out or fail. Our experimental results also show how important supporting invariants are, *e.g.* in Cooperating-T2 vs. AProVE we see that many results cannot be obtained with the invariant **true**, even though AProVE also uses modern rank function synthesis algorithms (*e.g.* [2]). Furthermore, the result of AProVE+Interproc indicates that an eager search for invariants in a preprocessing step is not a suitable solution to this problem, as this leads to more timeouts. In-depth analysis shows that these are not only due to timeouts in the preprocessing tool Interproc, but that the wealth of generated invariants also slows down the later termination proof. As expected [16], the performance of ARMC and Terminator is worse than that of T2. Thus, since our approach improves dramatically over T2, it also represents an improvement over ARMC and Terminator.

5 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. In this paper 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

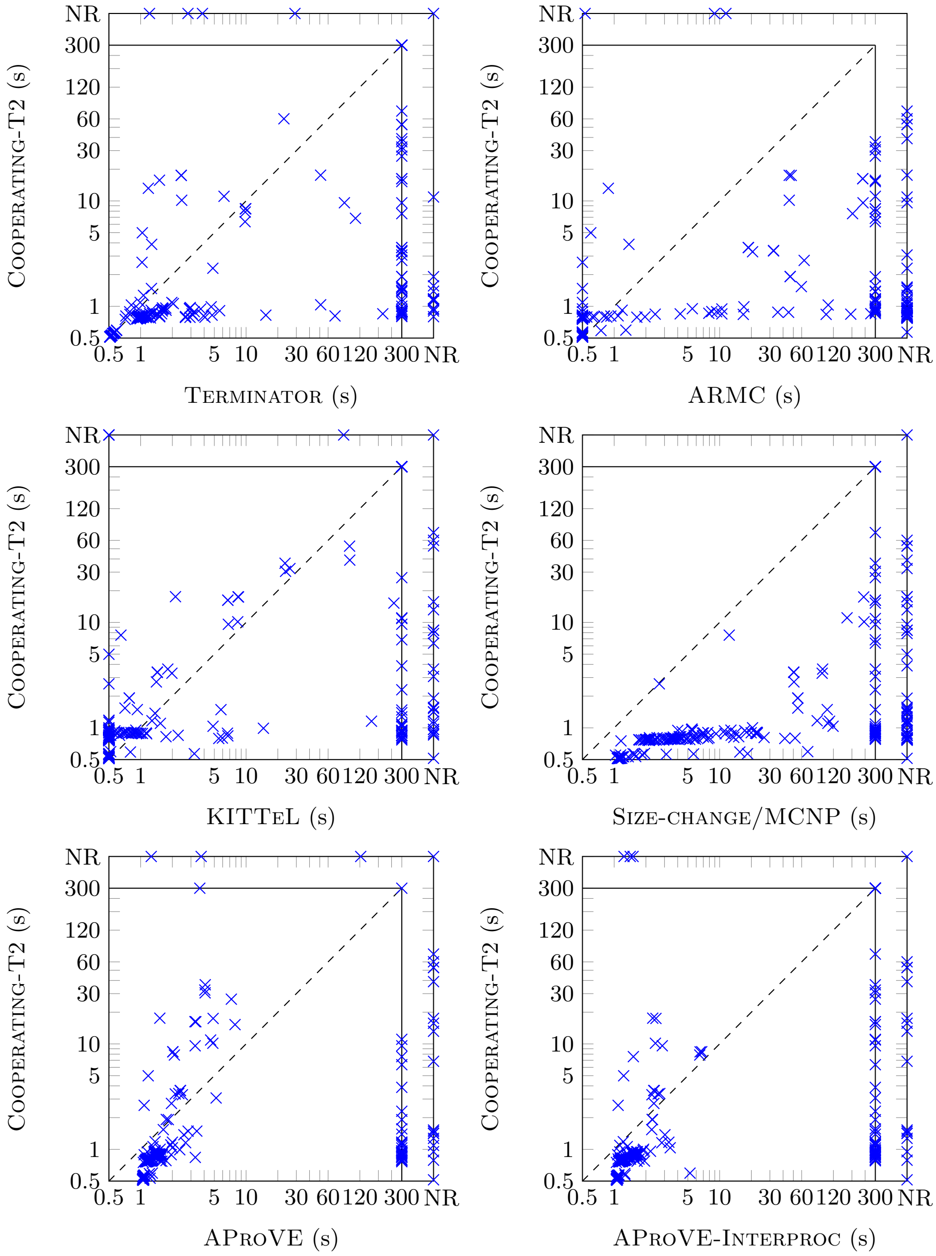


Fig. 7. Evaluation results of Cooperating-T2 vs. other termination proving tools (see Fig. 6 for comparison to T2). Scatter plots are in log scale. Timeout=300s. NR="No Result", indicating failure of the tool.

share their intermediate findings. As we have demonstrated experimentally, our approach leads to dramatic performance gains.

Future work. We have focused on a method to improve performance of termination analysis for arithmetic programs. Our technique could be adapted for additional contexts. For example, a finite-state model checker could potentially make use of similar information when proving safety properties resulting from the liveness-to-safety reduction from Biere *et al.* [6]. The techniques developed here can possibly be adapted to proving termination of heap-based programs, perhaps by using shape analysis techniques in the safety subgraph to learn arithmetic invariants for the termination subgraph. Finally, we expect that the approach developed here adapts naturally to the problem of CTL and LTL model checking (*e.g.* via [13] and [14]), but we have not looked into this in detail yet.

This technical report is an extended version of the conference paper [9].

Acknowledgments. We thank Christian von Essen, Jürgen Giesl, Heidy Khlaaf, Peter O'Hearn, Carsten Otto, and Abigail See for valuable discussions and the anonymous reviewers for helpful comments.

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 Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2), 2000.
4. Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. CAV '01*.
5. Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. In *Proc. POPL '07*.
6. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *Proc. FMICS '02*.
7. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In *Proc. CAV '05*.
8. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Proc. ICALP '05*.
9. Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*. To appear.
10. Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.
11. Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS '05*.
12. Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. SAT-based termination analysis using monotonicity constraints over the integers. *Theory and Practice of Logic Programming*, 11(4-5), 2011.
13. Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In *Proc. POPL '11*.

14. Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. In *Proc. CAV '11*.
15. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*.
16. Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*.
17. Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1-2), 1987.
18. Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. In *Proc. RTA '11*.
19. Robert W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, Proc. of Symposia in Applied Mathematics. American Mathematical Society, 1967.
20. Alfons Geser. *Relative Termination*. PhD thesis, Universität Passau, Germany, 1990.
21. Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*.
22. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3), 2006.
23. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. PLDI '12*.
24. Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*.
25. William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In *Proc. SAS '10*.
26. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proc. SPIN '03*.
27. Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2), 2005.
28. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. CAV '09*.
29. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*.
30. Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*.
31. Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*.
32. Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE '12*.
33. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.
34. Andreas Podelski and Andrey Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *Proc. PADL '07*.
35. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. VMCAI '04*.
36. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Proc. LICS '04*.
37. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. 1989.
38. Andrey Rybalchenko. Private communication, 2013.
39. Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *Proc. TACAS '11*.

Appendix A. Proof of Correctness

To prove the correctness of our approach, we formalize the intuitive view of cooperation graphs as a tool to split (infinite) program executions into a finite prefix (in the safety subgraph) and an infinite suffix (in the termination subgraph). For this, we first consider the structure of infinite program executions. As we only have finitely many program transitions, an infinite execution has to eventually re-use only a limited set of recurrent transitions, *i.e.*, transitions used infinitely often in the execution. For the following, we fix a set of transitions $\mathcal{T}$ (with the canonical start location *start*) to represent our program.

Definition 1. *Let* $\rho = (k_1, \boldsymbol{x}_1), (k_2, \boldsymbol{x}_2), \ldots$ *be an infinite execution of* $\mathcal{T}$*. Let* t_i *be the transition between states* $(k_i, \boldsymbol{x}_i)$ *and* $(k_{i+1}, \boldsymbol{x}_{i+1})$*. Then, there is an index* n *and a set* $\mathcal{R} \subseteq \mathcal{T}$ *of* recurrent transitions *such that for every* $m > n$*, there is a* $m' > m$ *with* $t_{m'} = t_m \in \mathcal{R}$*,* i.e.*, every element of* $\mathcal{R}$ *appears infinitely often in* ρ*. W.l.o.g. we choose a minimal* n *such that* k_n *is a cutpoint.*

We then call $\rho_s = (k_1, \boldsymbol{x}_1), \ldots, (k_n, \boldsymbol{x}_n)$ *the* stem *of* ρ *and* $\rho_t = (k_n, \boldsymbol{x}_n), (k_{n+1}, \boldsymbol{x}_{n+1}), \ldots$ *the* loop *of* ρ*.*[7] *We define the set of recurrent transitions of the program* $\mathcal{T}$ *as* $\mathcal{T}_\omega = \{t \in \mathcal{R} \mid \exists$ *an execution* ρ *with recurrent transitions* $\mathcal{R}\}$*.*

We now show that this natural split of infinite executions is mirrored in the construction of our cooperation graphs:

Lemma 1 (Cooperation graphs sound). *Let* $\mathcal{C} = \textsc{Instrument}(\mathcal{T})$*,* ρ *be an infinite execution of* $\mathcal{T}$*, and* n*,* ρ_s*, and* ρ_t *as in Def. 1. Then, there are corresponding state sequences* ρ'_s *and* ρ'_t *such that* $\rho'_s\,\rho'_t$ *is an infinite execution in* $\mathcal{C}$*, with* ρ'_s *in* $\textsc{Safety}(\mathcal{C})$ *and* ρ'_t *in* $\textsc{Termination}(\mathcal{C})$*.*

Proof. As $\textsc{Safety}(\mathcal{C})$ is just a copy of the original program, we can use a prefix

$$\rho'_s = (\textsc{SafetyLoc}(k_1), \boldsymbol{x}_1), \ldots, (\textsc{SafetyLoc}(k_n), \boldsymbol{x}_n)$$

Here, line 3 of the algorithm $\textsc{Instrument}$ ensures that the corresponding transitions exist.

By choice of n, k_n is a cutpoint. Hence, our cooperation graph contains a transition $(\textsc{SafetyLoc}(k_n), \mathbf{skip}, \textsc{TerminationLoc}(k_n))$. As all remaining transitions in the execution after $(k_n, \boldsymbol{x}_n)$ get used an infinite number of times, they have to be part of an SCC, and hence, by line 6 of the algorithm, copies of them exist in $\textsc{Termination}(\mathcal{C})$. Consequently, we can then construct ρ'_t starting with $(\textsc{TerminationLoc}(k_n), \boldsymbol{x}_n)$ by simulating the infinite suffix $(k_n, \boldsymbol{x}_n), \ldots$ of the original execution ρ. For this simulation, we can just use the copied transition. Only at cutpoints, where lines 8–16 of $\textsc{Instrument}$ introduce additional transitions to take value snapshots, additional steps need to be introduced in ρ'_t to accommodate the use of the (unrestricted) $\textsc{nosnapshot}$ transition. $\square$

[7] Note that ρ_t does not need to periodic.

In cooperation graphs, the termination subgraph is an over-approximation of possible recurrent transitions. Our main algorithm Refinement removes transitions from the termination subgraph when it can prove that they are not recurrent, refining the over-approximation. First, we recall the constraints on rank functions used to remove transitions.

Definition 2. *Let $\mathcal{S}$ be a set of transitions. We call f an $\mathcal{S}$-orienting rank function if and only if there is some non-empty subset $\mathcal{S}_> \subseteq \mathcal{S}$ such that the following two conditions hold:*

$$\bigwedge_{(k,A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x}'\end{smallmatrix}\right)\geq \boldsymbol{a},k')\in\mathcal{S}} \forall \boldsymbol{x},\boldsymbol{x}'.A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x}'\end{smallmatrix}\right) \geq \boldsymbol{a} \to f((k,\boldsymbol{x})) \geq f((k',\boldsymbol{x}'))$$

$$\bigwedge_{(k,A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x}'\end{smallmatrix}\right)\geq \boldsymbol{a},k')\in\mathcal{S}_>} \forall \boldsymbol{x},\boldsymbol{x}'.A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x}'\end{smallmatrix}\right) \geq \boldsymbol{a} \to (f((k,\boldsymbol{x})) > f((k',\boldsymbol{x}')) \wedge f((k,\boldsymbol{x})) \geq 0)$$

We set Decreasing$(\mathcal{S}, f) = \mathcal{S}_>$*.*

Based on this, we can prove that our transition removal in the termination subgraph is sound, as in similar approaches in the dependency pair framework [22, 27] such as the reduction pair processor. We show that if t appears in the set of infinitely recurrent transitions of some transition set $\mathcal{S}$, there is no $\mathcal{S}$-orienting rank function f with $t \in$ Decreasing$(\mathcal{S}, f)$ that would allow removal of t.

Lemma 2 (Transition removal sound). *Let $\mathcal{S}$ be a set of transitions and $t \in \mathcal{S}_\omega$. Then, there is no $\mathcal{S}$-orienting rank function f with $t \in$ Decreasing$(\mathcal{S}, f)$.*

Proof. We prove by contradiction. If $t \in \mathcal{S}_\omega$, there is an infinite $\mathcal{S}$-execution $\rho = (k_1, \boldsymbol{x}_1), (k_2, \boldsymbol{x}_2), \ldots$ such that the transition t is used infinitely often, *i.e.*, we have an infinite series of indices $\iota_1, \iota_2, \ldots$ such that t is used between $(k_{\iota_i}, \boldsymbol{x}_{\iota_i})$ and $(k_{\iota_i+1}, \boldsymbol{x}_{\iota_i+1})$. Then, when applying f to the states in ρ and considering the constraints from Def. 2, we get the following series of inequalities:

$$f((k_{\iota_1}, \boldsymbol{x}_{\iota_1})) > f((k_{\iota_1+1}, \boldsymbol{x}_{\iota_1+1})) \geq \ldots \geq f((k_{\iota_2}, \boldsymbol{x}_{\iota_2})) > f((k_{\iota_2+1}, \boldsymbol{x}_{\iota_2+1})) \geq \ldots$$

Transitivity of the order $>$ then implies:

$$f((k_{\iota_1}, \boldsymbol{x}_{\iota_1})) > f((k_{\iota_2}, \boldsymbol{x}_{\iota_2})) > \ldots$$

Thus, the value of $f((k_j, \boldsymbol{x}_j))$ is infinitely decreasing. But Def. 2 also enforces that the value of f is non-negative at each use of t. Hence, no infinite sequence in which $t \in$ Decreasing$(\mathcal{S}, f)$ is used again and again exists. □

Hence, if we find a $\mathcal{S}$-orienting rank function for an SCC in the termination subgraph, we can safely remove all decreasing transitions, as they are not infinitely recurring and hence cannot cause non-termination. We can now finally prove correctness of our algorithm Refinement.

Theorem 1. *If* Refinement($\mathcal{T}$) *returns "**Terminating**", $\mathcal{T}$ has no infinite executions.*

Proof. We combine the results from Lemmas 1 and 2. Assume that $\mathcal{T}$ has an infinite execution ρ, but Refinement returns "**Terminating**". Let $\mathcal{C}$ = Instrument($\mathcal{T}$).

By Lem. 1, there exists an execution $\rho'_s\,\rho'_t$ in Instrument($\mathcal{T}$) corresponding to ρ. As in Def. 1, ρ has an infinite suffix using only a restricted set of infinitely recurrent transitions $\mathcal{R} \subseteq \mathcal{T}$. By Lemma 2, the counterparts of $\mathcal{R}$ in Termination($\mathcal{C}$) can never be decreasing under a Termination($\mathcal{C}$)-orienting rank function. Thus, the simplification steps in lines 4 and 11 of Refinement do not remove these transitions. Hence, as "**Terminating**" is only returned when no counterexample is found anymore, the condition on the path to the error location generated by Strengthen has to be always false.

This implies that there is a sequence of rank functions $f_1, \ldots, f_n$ on some cutpoint p such that

$$\neg\left(\bigvee_{1\leq i\leq n} f_i(s) > f_i(s') \wedge f_i(s) \geq 0 \wedge \left(\bigwedge_{1\leq j<i} f_j(s) \geq f_j(s')\right)\right)$$

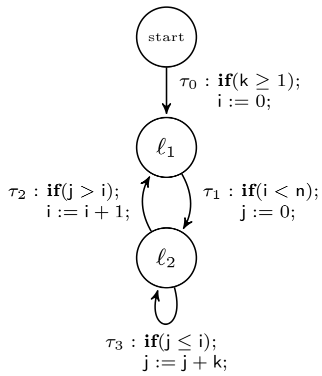
is always false for program states s, s' connected by one loop iteration over p. Hence, we have that the following holds for all possible such states:

$$\bigvee_{1\leq i\leq n} f_i(s) > f_i(s') \wedge f_i(s) \geq 0 \wedge \left(\bigwedge_{1\leq j<i} f_j(s) \geq f_j(s')\right)$$

By an argument similar to Lem. 2, this implies infinite descent in the well-founded lexicographic order of $>$ on n-tuples over the well-founded domain for our rank functions, and hence is a contradiction to our initial assumption that ρ is infinite. $\square$

```
if k ≥ 1 then
    i := 0;
    while i < n do
        j := 0;
        while j ≤ i do
            j := j + k;
        done
        i := i + 1;
    done
fi
```

(a)



(b)

Fig. 9. Textual and control-flow graph representation of a variation of Fig. 1

Appendix B. Extended Example

To see both phases of our algorithm REFINEMENT in action, consider the program in Fig. 9, which is a variation of our leading example from Fig. 1. Using INSTRUMENT, we obtain the cooperation graph $\mathcal{C}$ in Fig. 9. We have one non-trivial SCC $\mathcal{S}$ in the termination subgraph TERMINATION($\mathcal{C}$), made up by the nodes $\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$ and their connecting transitions. Our synthesis yields the $\mathcal{S}$-orienting rank function f^1, made up by the following functions for the single locations:

$$f^1_{\ell_1^t}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i} + 1 \qquad f^1_{\ell_1^d}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i} + 1$$
$$f^1_{\ell_2^t}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i} \qquad f^1_{\ell_2^d}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i}$$

Using these, we have the following inequalities for the transitions in the SCC (where relevant parts of the transition formula are displayed as preconditions):

$$\begin{array}{lrcl}
\gamma_1 : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} + 1 \geq \mathsf{n}' - \mathsf{i}' + 1 \\
\eta_1 : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} + 1 \geq \mathsf{n}' - \mathsf{i}' + 1 \\
\tau_1^t : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} + 1 > \mathsf{n}' - \mathsf{i}' \\
\tau_1^t : & \mathsf{n} > \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} + 1 \geq 0 \\
\gamma_2 : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} \geq \mathsf{n}' - \mathsf{i}' \\
\eta_2 : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} \geq \mathsf{n}' - \mathsf{i}' \\
\tau_2^t : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} + 1 & \Rightarrow & \mathsf{n} - \mathsf{i} \geq \mathsf{n}' - \mathsf{i}' + 1 \\
\tau_3^t : & \mathsf{n}' = \mathsf{n} \wedge \mathsf{i}' = \mathsf{i} & \Rightarrow & \mathsf{n} - \mathsf{i} \geq \mathsf{n}' - \mathsf{i}'
\end{array}$$

Hence, we have DECREASING($\mathcal{S}, f$) $= \{\tau_1^t\}$. We remove τ_1^t from $\mathcal{C}$ and $\mathcal{S}$. Now, a second (trivial) rank function can be found removing all transitions that

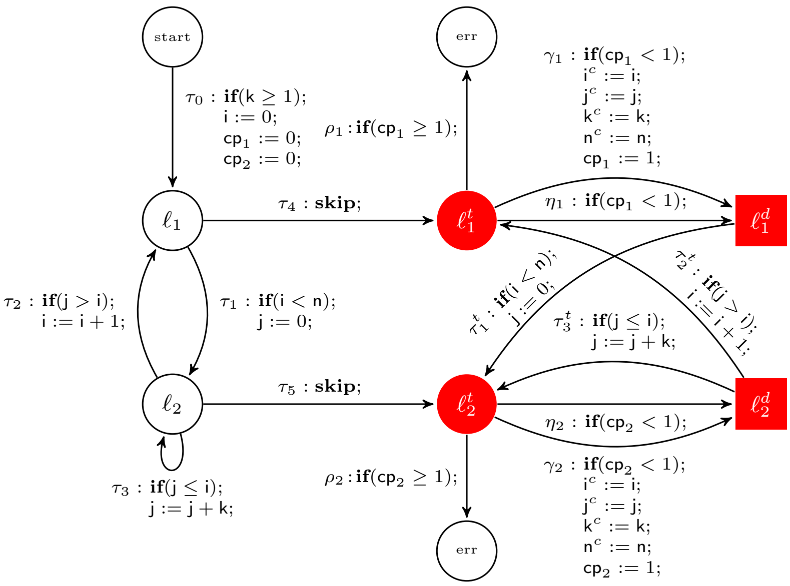


Fig. 10. Cooperation graph derived from Fig. 9

are not part of an SCC anymore:

$$f^2_{\ell_1^t}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = 2 \qquad f^2_{\ell_1^d}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = 1$$
$$f^2_{\ell_2^t}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = 3 \qquad f^2_{\ell_2^d}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = 3$$

This allows to remove the transitions γ_1, η_1, and τ_2^t. The remaining cooperation graph is shown in Fig. 11.

No further preprocessing is possible here, as the invariant $\mathsf{k} \geq 1$ is not known for the inner loop.[8] We switch to the second phase of the algorithm, searching for a counterexample to the current termination argument. The first returned counterexample might be $\langle \tau_0, \tau_1, \tau_5, \gamma_2, \tau_3^t, \rho_2 \rangle$. We extract the loop $\langle \gamma_2, \tau_3^t \rangle$. As no invariants were inferred in the meantime, there still exist no $\mathcal{S}$-orienting rank functions. We fall back to the approach from Cook *et al.* [16] and extract a formula from the loop, as usual removing variables introduced by our instrumentation:

$$\mathsf{i}' = \mathsf{i} \wedge \mathsf{j}' = \mathsf{j} + \mathsf{k} \wedge \mathsf{k}' = \mathsf{k} \wedge \mathsf{n}' = \mathsf{n}$$

Furthermore, we extract a formula from the stem, yielding $\mathsf{k} \geq 1 \wedge \mathsf{i} < \mathsf{n} \wedge \mathsf{j} = 0$. We then generate a lexicographic rank function for the one counterexample found

[8] In fact, our implementation experiences show that a simple, limited invariant propagation is useful, as it is far cheaper than using several (more costly) counterexample iterations to propagate trivial invariants only when needed. Hence, our implementation proves this example terminating in its preprocessing step.

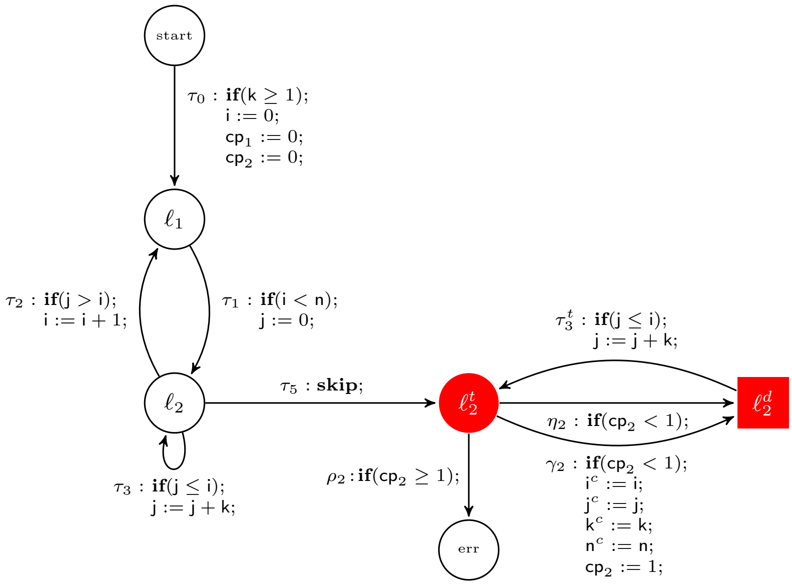


Fig. 11. Cooperation graph after preprocessing Fig. 10

so far, using its stem as precondition, obtaining $f(\mathsf{i}, \mathsf{j}, \mathsf{k}, \mathsf{n}) = \mathsf{i} - \mathsf{j}$. This is then used to strengthen the transition ρ_2, yielding the following ρ_2':

$$\begin{aligned}\rho_2' : &\ \mathbf{if}(\mathsf{cp}_2 \geq 1);\\ &\ \mathbf{if}(\neg(\mathsf{i}^c - \mathsf{j}^c > \mathsf{i} - \mathsf{j} \land \mathsf{i}^c - \mathsf{j}^c \geq 0));\end{aligned}$$

Using the strengthened ρ_2' instead of ρ_2, we restart the safety prover on our example. To prove safety, it infers the invariant $\mathsf{k} \geq 1$ at location ℓ_2^d, and does not return any further counterexamples.

Appendix C. Implementation

The contributions of this paper have been implemented in a new version of the tool T2. A source-based release is available at the following URL:

`http://research.microsoft.com/en-us/projects/t2/`

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2010-01 *	Fachgruppe Informatik: Jahresbericht 2010
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05	Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10	Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11	Martin Zimmermann: Parametric LTL Games
2010-12	Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13	Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
2010-14	Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15	Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-16	Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles
2010-17	Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18	Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
2010-19	Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games
2011-01 * Fachgruppe Informatik: Jahresbericht 2011
2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06 Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
2011-18 Kamal Barakat: Introducing Timers to pi-Calculus
2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04	Marcus Gelderie: Strategy Machines and their Complexity
2012-05	Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06	Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-07	André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08	Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09	Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains
2012-10	Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12	Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15	Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations
2012-16	Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets
2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.