Investigating the Border between Constructive and Inductive Proofs of Termination

SVARM at ETAPS 2011, Saarbrücken, Germany

Aliaksei Tsitovich

Formal Verification and Security Lab

University of Lugano, Switzerland

April 02, 2011

The talk is based on joint work with
Daniel Kroening (Oxford, UK), **Natasha Sharygina** (University of Lugano)
and **Christoph M. Wintersteiger** (MSR, Cambridge, UK)

Termination

A program is **terminating** if it does not have infinite executions

Termination

A program is **terminating** if it does not have infinite executions

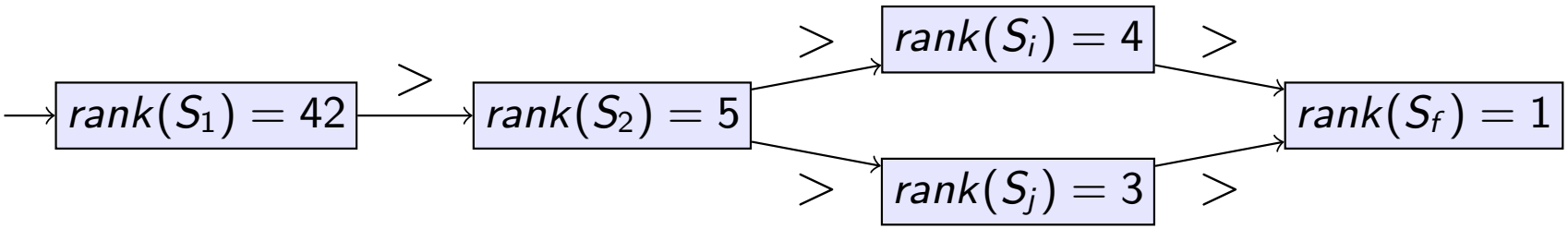
- Turing in 1936 proved there is no general algorithm to decide it.

Termination

A program is **terminating** if it does not have infinite executions

- Turing in 1936 proved there is no general algorithm to decide it.
- But in 1949 he proposed a method that at least tries:

Assign all states of the program with natural numbers, i.e. *rank* them, such that for any pair of consecutive states $s_i, s_{i+1} \in S$ the number is decreasing, i.e., $rank(s_i) > rank(s_{i+1})$

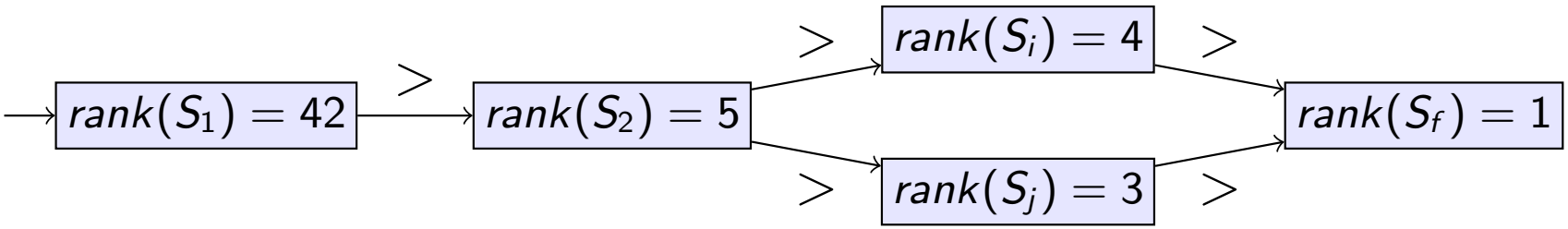


Termination

A program is **terminating** if it does not have infinite executions

- Turing in 1936 proved there is no general algorithm to decide it.
- But in 1949 he proposed a method that at least tries:

> Assign all states of the program with natural numbers, i.e. *rank* them, such that for any pair of consecutive states $s_i, s_{i+1} \in S$ the number is decreasing, i.e., $rank(s_i) > rank(s_{i+1})$



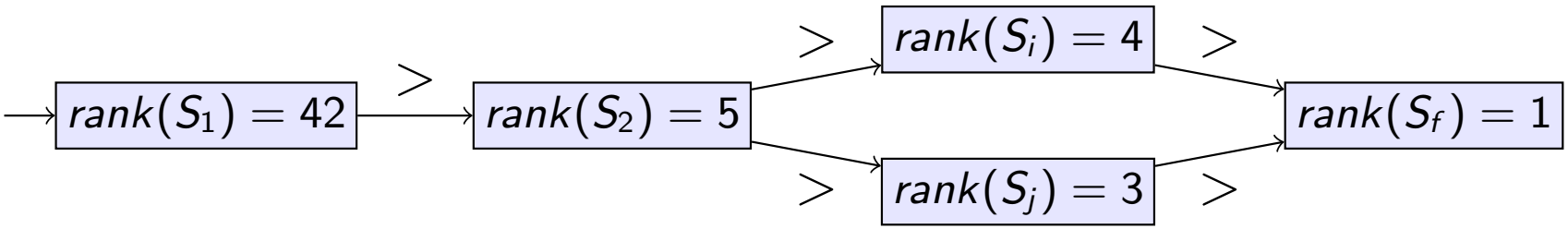
Termination can be concluded from **well-foundedness** of program's transition relation bounded to only reachable states

Termination

A program is **terminating** if it does not have infinite executions

- Turing in 1936 proved there is no general algorithm to decide it.
- But in 1949 he proposed a method that at least tries:

> Assign all states of the program with natural numbers, i.e. *rank* them, such that for any pair of consecutive states $s_i, s_{i+1} \in S$ the number is decreasing, i.e., $rank(s_i) > rank(s_{i+1})$



Termination can be concluded from **well-foundedness** of program's transition relation bounded to only reachable states

Discovery of a magic *rank* function remained a major problem for years

Transition Invariants

Definition (Podelski, Rybalchenko 2004)

A **transition invariant** T for program $P = \langle S, I, R \rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.

In contrast, **state invariant** — a superset of a set of reachable states, i.e. $R^*(I) \subseteq V$.

Transition vs State Invariant

State invariant preserves a common property over states

while

Transition invariant reflects a common property over transitions between the states

Example

Example

```
#define SIZE 256
int c[SIZE];
int main()
{
  unsigned i=SIZE-1;
  while(i>0)
  {
    c[i] = i;
    i=i-1;
  }
}
```

Transition invariant:

$i' < i$

Termination

Theorem (Termination by Podelski, Rybalchenko'04)

A program P is terminating iff there exists a (disjunctively) well-founded transition invariant for P.

Inductive Transition Invariant

Theorem (T., Sharygina, Wintersteiger, Kroening at TACAS'11)

A binary relation T is a transition invariant for the program $\langle S, I, R \rangle$ if it is **compositional** *(transitive) and $R \subseteq T$.*

$$\exists s_i, s_j, s_k \in S \;.\; \neg \left(T(s_i, s_j) \wedge T(s_j, s_k) \Rightarrow T(s_i, s_k) \right)$$

Inductive Transition Invariant

Theorem (T., Sharygina, Wintersteiger, Kroening at TACAS'11)

A binary relation T is a transition invariant for the program $\langle S, I, R \rangle$ if it is **compositional** *(transitive) and $R \subseteq T$.*

$$\exists s_i, s_j, s_k \in S \;.\; \neg \left(T(s_i, s_j) \wedge T(s_j, s_k) \Rightarrow T(s_i, s_k) \right)$$

A program is terminating if there exists a well-founded **compositional** transition invariant for it.

Algorithm #1

Terminator (by Cook, Podelski and Rybalchenko in 2006) uses **Binary Reachability Analysis**, model checker-based non-terminating counterexample extraction and employs **per-path** *rank* **generator** to construct a d.wf. transition invariant.

- give a full program (loop) to MC and get a yet non-ranked path from it
- find a ranking function for the path; add another disjunct to the d.wf TI
- update a (reachability) assertion in the program to reflect the change in TI and ask MC for another non-ranked path

Algorithm #2

Compositional Termination Analysis (Kroening, Sharygina, T., Wintersteiger at CAV'10) improved over TERMINATOR by path construction via loop unwinding and employing compositionality condition as a sufficient criterion.

- give MC a "program" of 1 loop iteration and get a non-ranked path
- find a ranking function for it; add another disjunct to the d.wf TI
- update an assertion to reflect the change in TI and ask MC for another non-ranked path
- if no more unranked path of 1 loop iteration exists then check TI for compositionality. If it is not — continue with a "program" of 2 loop iterations.

Algorithm #3

Loop summarization using relational domains (T., Sharygina, Wintersteiger, Kroening at TACAS'11) focus only on one loop iteration and instead iterates over possible candidates for compositional d.wf. TI

- take a loop and generate a set of relations between pre- and post-states of the loop iteration
- check each candidate for being a super-set of transition relation of one iteration, i.e. $T \supseteq R$
- if it is, check T for compositionality and wf.-ness (can be avoided by an appropriate selection of candidates)

Types of termination proofs

Terminator — using Binary Reachability Analysis (BRA)
VS
Compositional Termination Analysis (CTA)
VS
LoopFrog

Terminator is a "proof by construction"

Compositional Termination Analysis uses "proof by construction" to obtain the base case and check it to be enough using inductive step (compositionality criterion)

LoopFrog "guesses" candidates, check each for being a base case of inductive proof and tries if the inductive step holds for it too

Possible ways to continue

Termination with Compositionality

A program P is terminating **iff** there exists a well-founded **compositional** transition invariant for P.

- The forward direction of the proof was already used in the previous work
- The reverse direction of the proof follows from the definition of TI

Questions to think about:

- Can we define a criterion for loops that allow (easy) inductive termination proof?

Questions to think about:

- Can we define a criterion for loops that allow (easy) inductive termination proof?
- Does there exist a method to enumerate all possible base cases of possible inductive termination proof?

Questions to think about:

- Can we define a criterion for loops that allow (easy) inductive termination proof?
- Does there exist a method to enumerate all possible base cases of possible inductive termination proof?
- Can we find the method that generalizes the discovery of inductive termination proof without ranking function discovery?

Abstract domains for well-founded transition invariants discovery

#	Constraint	Meaning
1	$i' < i$ $i' > i$	A numeric variable i is strictly decreasing (increasing).
2	$x' < x$ $x' > x$	Any loop variable x is strictly decreasing (increasing).
3	$sum(x', y') < sum(x, y)$ $sum(x', y') > sum(x, y)$	Sum of all numeric loop variables is strictly decreasing (increasing).
4	$max(x', y') < max(x, y)$ $max(x', y') > max(x, y)$ $min(x', y') < min(x, y)$ $min(x', y') > min(x, y)$	Maximum or minimum of all numeric loop variables is strictly decreasing (increasing).
5	$(x' < x \wedge y' = y) \vee$ $(y' < y \wedge x' = x)$ $(x' > x \wedge y' = y) \vee$ $(y' > y \wedge x' = x)$	A combination of strict increase or decrease for one of loop variables while the remaining ones are not updated.

Algorithm #4?

Find a "complete" way to enumerate invariant candidates in LoopFrog

1. Enumerate all possible permutations of variables in the loop to use as possible lexicographical ordering, i.e. move gradually from domain #1 "$i' > i$ to domain #5 "$(i, j, k, ...)' > (i, j, k, ...)$".

   Huge number of variants — $n! + n!/(n-1)! + ... + n!/1!$ only for permutations without considering the decrease/increase

Algorithm #4?

Find a "complete" way to enumerate invariant candidates in LoopFrog

1. Enumerate all possible permutations of variables in the loop to use as possible lexicographical ordering, i.e. move gradually from domain #1 "$i' > i$ to domain #5 "$(i, j, k, ...)' > (i, j, k, ...)$".

   Huge number of variants — $n! + n!/(n-1)! + ... + n!/1!$ only for permutations without considering the decrease/increase, but:

   - we can explore them gradually in a tree-like structure
   - we can apply several loop iterations to remove a majority of them as unfeasible

Algorithm #4?

Find a "complete" way to enumerate invariant candidates in LoopFrog

1. Enumerate all possible permutations of variables in the loop to use as possible lexicographical ordering, i.e. move gradually from domain #1 "$i' > i$ to domain #5 "$(i, j, k, ...)' > (i, j, k, ...)$".

   Huge number of variants — $n! + n!/(n-1)! + ... + n!/1!$ only for permutations without considering the decrease/increase, but:

   - we can explore them gradually in a tree-like structure
   - we can apply several loop iterations to remove a majority of them as unfeasible

2. The remaining candidates are checked for being a (compositional) TI
3. The wf.-ness check requires ensuring a minimal/maximal value for the considered variables to be a loop invariant.

Algorithm #4?

Find a "complete" way to enumerate invariant candidates in LoopFrog

1. Enumerate all possible permutations of variables in the loop to use as possible lexicographical ordering, i.e. move gradually from domain #1 "$i' > i$ to domain #5 "$(i, j, k, ...)' > (i, j, k, ...)$".

   Huge number of variants — $n! + n!/(n-1)! + ... + n!/1!$ only for permutations without considering the decrease/increase, but:

   - we can explore them gradually in a tree-like structure
   - we can apply several loop iterations to remove a majority of them as unfeasible

2. The remaining candidates are checked for being a (compositional) TI
3. The wf.-ness check requires ensuring a minimal/maximal value for the considered variables to be a loop invariant.

- Is it possible to prove the completeness of this method?
- Does it "repeat" the linear ranking functions?
- How can we minimize a number of variables to consider?

Algorithm #5?

Combine **Liveness-to-Safety** rewriting by Biere et.al with interpolation to prune terminating paths from the MC check

1. Save the loop pre-state
2. Apply one loop unwinding and check if the saved stated can be equal to the current post-state
3. If not, find an interpolant to over-approximate a loop iteration and conjunct it with another copy of loop iteration (i.e., explore longer path)
4. If yes, generalize a CE to precondition and report as non-termination
5. Do it until the interpolant becomes compositional (?)

Algorithm #5?

Combine **Liveness-to-Safety** rewriting by Biere et.al with interpolation to prune terminating paths from the MC check

1. Save the loop pre-state
2. Apply one loop unwinding and check if the saved stated can be equal to the current post-state
3. If not, find an interpolant to over-approximate a loop iteration and conjunct it with another copy of loop iteration (i.e., explore longer path)
4. If yes, generalize a CE to precondition and report as non-termination
5. Do it until the interpolant becomes compositional (?)

- Does it make sense for an interpolant to be compositional?
- Can we tune interpolation algorithm such that interpolant is compositional?
- Is it a dual of TERMINATOR?

Any answers?

More details available at:

- `http://www.verify.inf.usi.ch/loopfrog/termination`
- `http://www.cprover.org/termination/`

Loop summarization with transition invariants

SummarizeLoop-TI(L)
input: Single-loop program L with a set of variables X
output: Loop summary
begin
    $T := \top$
    foreach Candidate $C(X, X')$ **in**
    PickInvariantCandidates(*Loop*) **do**
        if IsInvariant(L, C) $\wedge$ IsCompositional(C) **then**
            $T := T \wedge C$
    return "$X_{PRE} := X; X = *;$ `assume`$(T(X_{PRE}, X));$"
end

IsCompositional **(transitivity) check**

$$\exists s_i, s_j, s_k \in S \;.\; \neg\,(C(s_i, s_j) \wedge C(s_j, s_k) \Rightarrow C(s_i, s_k))$$

Only existential quantification

$\Downarrow$

Computationally "cheap" SAT/SMT check is possible

Loop summarization with transition invariants

IsInvariant(L, C)
input: Single-loop program L with a set of variables X, cand. invariant C
output: TRUE if C is invariant for L; FALSE otherwise
begin
    return Unsat($L(X, X') \wedge \neg C(X, X')$)
end

Well, but what about termination?

Definition (Well-foundedness)

A relation R is **well-founded (wf.)** over S if for any non-empty subset of S there exists a minimal element (with respect to R), i.e. $\forall X \subseteq S \,.\, X \neq \emptyset \implies \exists m \in X \,.\, \forall s \in X \;\; (s, m) \notin R$.

Restricted class of transition invariants

Observation

If T is a **strict order relation** for a **finite** set $K \subseteq S$ and is a **transition invariant** for the program $\langle S, I, R \rangle$, then T is **well-founded**.

Restricted class of transition invariants

Observation

If T is a **strict order relation** for a **finite** set $K \subseteq S$ and is a **transition invariant** for the program $\langle S, I, R \rangle$, then T is **well-founded**.

$\Downarrow$

Corollary

A program terminates if it has a transition invariant T that is also a finite strict order relation.

LoopFrog greets you!

Static analyzer for ANSI-C programs

`www.verify.inf.usi.ch/loopfrog`

Evaluation

LoopFrog
vs
Terminator — (using Binary Reachability Analysis (BRA))
vs
Compositional Termination Analysis (CTA)

TERMINATOR (by Cook, Podelski and Rybalchenko in 2006) uses **Binary reachability analysis**, **model checker**-based counterexample extraction, and **per-path** *rank* **generator** to construct a d.wf. transition invariant.

Compositional Termination Analysis (by Kroening, Sharygina, Tsitovich and Wintersteiger CAV'10) improved over BRA by path construction via loop unwinding; and employing compositionality condition as a sufficient criterion.

Evaluation

LoopFrog
VS
Terminator — (using Binary Reachability Analysis (BRA))
VS
Compositional Termination Analysis (CTA)

LoopFrog 1	$sum(x', y') < sum(x, y)$ or $sum(x', y') > sum(x, y)$
LoopFrog 2	$i' < i$ or $i' > i$
Terminator	Cook et al. PLDI '06 and TACAS'10
CTA	Kroening, Sharygina, Tsitovich, Wintersteiger CAV'10

All algorithms are implemented in Cprover/SatAbs framework

Evaluation

LoopFrog
VS
Terminator — (using Binary Reachability Analysis (BRA))
VS
Compositional Termination Analysis (CTA)

We experimented with a large number of ANSI-C programs including:

- The SNU real-time benchmark suite that contains small C programs used for worst-case execution time analysis;
- The Powerstone benchmark suite as an example set of C programs for embedded systems;
- The Verisec 0.2 benchmark suite by Ku et al.;
- **Windows device drivers** (from Windows Device Driver Kit 6.0).

Benchmark group	Method	T	NT	TO	Time
Power-stone 135 loops in 14 benchmarks	LoopFrog 1	26	109	0	197.67
	LoopFrog 2	66	69	0	2972.63
	CTA	60	70	5	6519.45+
	Terminator	66	58	11	5111.02+
SNU-real-time 109 loops in 17 benchmarks	LoopFrog 1	25	84	0	595.06
	LoopFrog 2	75	34	0	816.21
	CTA	64	35	10	7084.68+
	Terminator	62	35	12	4182.92+
Verisec 0.2 244 loops in 160 benchmarks	LoopFrog 1	33	211	0	11.38
	LoopFrog 2	44	200	0	22.49
	CTA	34	208	2	1207.62+
	Terminator	40	204	0	4040.53

Columns 3 to 6 state number of loops proven to terminate **T**, possibly non-terminate **NT**, time-out **TO**. Time is computed only for loops noted in T and NT; '+' is used to denote the cases were at least one time-outed loop was not considered.

Windows device drivers

Benchmark group	Method	T	NT	TO	Time
SDV FLAT DISPATCH HARNESS 557 loops in 30 benchmarks	LoopFrog 1	135	389	33	1752.1
	LoopFrog 2	215	201	141	10584.4
	CTA	166	160	231	25399.5
SDV FLAT DISPATCH STARTIO HARNESS 557 loops in 30 benchmarks	LoopFrog 1	135	389	33	1396.0
	LoopFrog 2	215	201	141	9265.8
	CTA	166	160	231	28033.3
SDV FLAT HARNESS 635 loops in 45 benchmarks	LoopFrog 1	170	416	49	1323.0
	LoopFrog 2	239	205	191	6816.4
	CTA	201	186	248	31003.2
SDV FLAT SIMPLE HARNESS 573 loops in 31 benchmarks	LoopFrog 1	135	398	40	1510.0
	LoopFrog 2	200	191	182	6814.0
	CTA	166	169	238	30292.7
SDV HARNESS PNP DEFERRED IO REQUESTS 177 loops in 31 benchmarks	LoopFrog 1	22	98	57	47.9
	LoopFrog 2	66	54	57	617.4
	CTA	80	94	3	44645.0
SDV HARNESS PNP IO REQUESTS 173 loops in 31 benchmarks	LoopFrog 1	25	94	54	46.6
	LoopFrog 2	68	51	54	568.7
	CTA	85	86	2	15673.9
SDV PNP HARNESS SMALL 618 loops in 44 benchmarks	LoopFrog 1	172	417	29	8209.5
	LoopFrog 2	261	231	126	12373.2
	CTA	200	177	241	26613.7
SDV PNP HARNESS 635 loops in 45 benchmarks	LoopFrog 1	173	426	36	7402.2
	LoopFrog 2	261	230	144	13500.2
	CTA	201	186	248	41566.6
SDV PNP HARNESS UNLOAD 506 loops in 41 benchmarks	LoopFrog 1	128	355	23	8082.5
	LoopFrog 2	189	188	129	13584.6
	CTA	137	130	239	20967.8
SDV WDF FLAT SIMPLE HARNESS 172 loops in 18 benchmarks	LoopFrog 1	27	125	20	30.3
	LoopFrog 2	61	91	20	202.0
	CTA	73	95	4	70663.0