Cooperative alternation for termination and non-termination

Théo Zimmermann

July 30, 2013

Abstract

I present the work I have realised during my internship at Microsoft Research Cambridge, under the supervision of Byron Cook. The goal was to build a tool to combine already existing termination and non-termination provers with an alternating procedure. We have got some interesting results which confirm the interest of such an alternation. However, the main contributions of my own work are concentrated in the non-termination prover as I have developed new ideas and implemented them.

1 Introduction

The computer science field of verification is aimed at modelling systems, in particular software, in order to prove automatically some properties they must have. These properties are generally separated into two types: safety properties and liveness properties [14]. A safety property is generally stated as "the system must never be in a state where..." so, once we have a model of the system, checking such properties is generally limited to proving certain states are not reachable. To prove that some safety property is not verified, one must only provide a finite trace leading to a defecting state. So every counter-example to a safety property is finite. A lot of research has been done on this topic, providing today some fantastic tools (among them [1]).

Properties of the other type are called liveness properties. They are generally stated as "eventually the system will be in a state where...". In particular, it is helpful to say "eventually, the system will do something" as a system which does nothing will not break any safety property but is not interesting either [8]. A special case of liveness property is termination. Termination of a system is defined as "the system will always eventually stop" which we can write AF$false$ in the formalism of CTL [6]. Proving termination is both useful because proving any liveness property is really close to proving termination [19] and, of course, to track termination bugs (sometimes referred as *infinite loops* although this does not always need to be a loop) which produce an unwanted behaviour which will even not be detected at runtime since it is not easy to distinguish between a non-terminating loop and one which takes a very long time (some tools are

specifically designed to do that [4]). And indeed, finite traces do not suffice to falsify a liveness property.

Much research has been done recently on termination proving, giving rise to many tools ([9], [11], [17], etc). The basic idea is to look for a *ranking function* (or a set of ranking functions) mapping each state of the system into a well-founded order such that for any pair of states, if one can precedes the other one in a given execution, then some function gives it a higher rank [16].

Tools generally split this method into two steps. First one is finding the ranking functions and the second one is proving the decrease of such functions using off-the-shelf safety checkers. We give an idea on how to do that in section 3.

However, as stated by Turing [18], no method for proving termination can be complete. Generally, tools will look for an indefinitely long time for a proof and may never terminate themselves. This is especially inconvenient if the reason for not finding a termination proof is that the system is flawed and does not always terminate. This is why, recent work has been aiming at non-termination proving ([12], [20], etc) and also at integrating both termination and non-termination proving in a single tool [13].

As explained earlier, we cannot exhibit a finite counter-example to termination. Claiming that a system does not terminates requires a proof. Such a proof can be given as a *recurrent set of states*. Recurrent sets for non-termination proving were first introduced by Gupta *et al.* [12]. They are sets of states such that, from any state of such a set, there exist a transition leading back to the set. A second property is that such set is reachable and with this two properties, a recurrent set is a proof of non-termination. Following Harris *et al.* [13], we will call a set, with the first property only, a *partial* recurrent set.

Contrarily to Gupta *et al.* but similarly to Harris *et al.*, we chose to split the search for a proof of non-termination into two steps: finding a partial recurrent set and proving it is reachable. The idea is, exactly as in the case of termination proving, to make use of already good off-the-shelf safety checkers to prove or disprove reachability. This also helps making easier the search for a recurrent set in the first phase. I will present the method and the implementation in section 2.

Alternating between a termination proving attempt and a non-termination proving attempt is a good idea in order not to look for an non-existent proof for too long. This is an even better idea if we can make the two tools (termination and non-termination provers) cooperate. This leads to the main idea, which is developed in this report, first proposed by Byron Cook and Marc Brockshmidt, and which I have been working on while an intern at Microsoft Research Cambridge. T2, the termination prover developed at Microsoft Research Cambridge, proceeds by successive refinements and is able to give information on which part of the system have already been proved terminating, even when it fails to produce a proof of termination. We can therefore remove them from our consideration when we look then at a partial recurrent set in order to prove non-termination. This first cooperative step is a new idea and draws advantage from the specific way in which T2 has been implemented [2]. The second cooperative step is not

a new idea as it was already described by Harris *et al.* If we find a partial recurrent set but not reachable, we can remove it from our consideration before attempting again to prove termination. The concepts and the implementation of the alternation are explained in more details in section 4.

Considering how much had already been done on termination proving, I have focussed more on the non-termination proving part and how to make the two tools cooperate. To synthesize partial recurrent sets, I have been inspired from the original technique described by Gupta *et al.*, based on constraints solving. However, as this technique was limited to deterministic system, I have been adapting it by taking some more inspiration in a unpublished paper by Cook *et al.* [7]. I have implemented my method to make use of Z3, the Microsoft's SMT solver [10].

2 Non-termination proving

Definition 2.1. *A program $P = (\mathcal{L}, \mathcal{T})$ is a directed graph, with program locations $\mathcal{L}$ and transitions $\mathcal{T}$. The canonical start location is $\ell_0 \in \mathcal{L}$. A program state is a pair $(\ell, \boldsymbol{v})$, where $\ell \in \mathcal{L}$ and $\boldsymbol{v}$ is a valuation of the variables $\mathcal{V}$. Transitions are labeled by formulas ρ relating pre-variables $\mathcal{V}$ and post-variables $\mathcal{V}'$. We write $(\boldsymbol{v}, \boldsymbol{v}') \models \rho$ if the valuations $\boldsymbol{v}, \boldsymbol{v}'$ satisfy the transition relation ρ, and $(\boldsymbol{v}, \boldsymbol{v}') \models P$ if such a transition exists in P.*

The basic algorithm for non-termination proving as suggested in the introduction is the following. Note here that for the purpose of presenting this algorithm, we chose to represent a program as a transition system with a set of states $S = \{(\ell, \boldsymbol{v}), \ell \in \mathcal{L} \wedge \boldsymbol{v} \text{ a valuation}\}$, a transition relation $R = \{((\ell, \boldsymbol{v}), (\ell', \boldsymbol{v}')), \ell \xrightarrow{\rho} \ell' \in \mathcal{T} \wedge (\boldsymbol{v}, \boldsymbol{v}') \models \rho\}$ and a set of initial states $I = \{(\ell_0, \boldsymbol{v}), \boldsymbol{v} \text{ any valuation}\}$. In practice, we use formulas to represent such sets.

Algorithm 1 Non-termination proving technique

Input: Program $P = (S, R, I)$
  $Q \leftarrow S$
  loop
    match RECURRENCE($R \cap (Q \times Q)$) **with**
    | Nil $\rightarrow$ **return** "Failed to prove non-termination"
    | Set(C) $\rightarrow$
      match SAFETY(R, I, C) **with**
      | Reachable($Path$) $\rightarrow$
        return "Proved non-termination with $(Path, C)$"
      | Unreachable(Inv) $\rightarrow$ $Q \leftarrow Inv \cap Q$
  end loop

As explained before, the idea of proving non-termination with recurrent sets comes from Gupta *et al.* [12] but we chose to check reachability separately because it is easier to find a *partial* recurrent set and then let a good safety checker do the reachability analysis for us. If we find a set

which is not reachable, we ask the safety checker for an invariant Inv such that any reachable state verifies the invariant but it rules out the spurious recurrent set. Then we continue, taking this new invariant into account.

2.1 An example

The following example[1] has been a leading one to implement the non-termination proving and to check if results were conform to expectations. The process described is the actual one, with my current implementation:

Input: $\mathsf{x} \leq 0 \wedge -1 \leq \mathsf{k} \leq 1$
while $\mathsf{x} \neq 0$ **do**
    $\mathsf{x} \leftarrow \mathsf{x} + \mathsf{k}$
end while

Algorithm 1 first attempts to find a partial recurrent set, leading to $C = [\![\mathsf{x} \geq 1 \wedge \mathsf{k} \geq 0]\!]$[2]. The reachability check shows that it is not reachable and provides the invariant $Inv = [\![\mathsf{k} \leq 1 \wedge \mathsf{x} \leq 0]\!]$. Taking this invariant into account, the second attempt to find a partial recurrent set will lead to $C = [\![\mathsf{x} \leq -1 \wedge \mathsf{k} \leq 0]\!]$ which will then be proved reachable.

2.2 Synthesizing partial recurrent sets

I implemented and tried algorithm 1. First, I was using a technique to find partial recurrent sets for which we had already an implementation [3] but as it found too few partial recurrent sets for our purposes, I dropped it and developed instead a new technique adapted from the one described by Gupta *et al.* [12]. It would still be interesting however to compare both techniques and see if there are some cases where the former is better.

Gupta *et al.* restrict recurrent sets to states which are all on the same given location. This location corresponds to the start point of a loop and thus they must enumerate exhaustively every loop in the program (which is a restriction as some infinite non-terminating runs can be aperiodic). On the contrary, we will look at strongly connected subgraphs[3] (SCS) in the graph representation of the program and will search recurrent sets within those.

Looking at SCS rather than at loops allows us to handle nested loops without any added complication. Moreover, the technique scales easily provided we enumerate SCS by size: the program will detect very well little non-terminating parts of a larger program and have a lot more information (invariants) when reaching the larger ones.

So we adapt the notion of recurrent sets to several program locations $\mathcal{L}$. Here, one set of recurrent states $\mathcal{R}_\ell$ is assigned to each program location ℓ and we require that for each state $(\ell, \boldsymbol{v}) \in \mathcal{R}_\ell$, there is some state $(\ell', \boldsymbol{v}') \in \mathcal{R}_{\ell'}$ and a transition ρ connecting the two.

Proposition 2.2 (Recurrent states)**.** *Let $P = (\mathcal{L}, \mathcal{T})$ be a program. The transition relation of P is not well-founded if and only if there are sets $\mathcal{R}_\ell$ with:*

[1] Note that all our programs contain only integers
[2] Here we are using $[\![-]\!]$ to represent the semantic translation of formulas to sets
[3] We are not necessarily restricting to strongly connected components

1. $\exists \ell \in \mathcal{L}, \mathcal{R}_\ell \neq \emptyset$
2. $\forall \ell \in \mathcal{L}, \forall \boldsymbol{v} \in \mathcal{R}_\ell, \exists \ell' \in \mathcal{L}, \exists \boldsymbol{v}' \in \mathcal{R}_{\ell'}, ((\ell, \boldsymbol{v}), (\ell', \boldsymbol{v}')) \models P$

We call $\bigcup_{\ell \in \mathcal{L}} \mathcal{R}_\ell$ a partial recurrent set.

We describe an infinite set of states by mapping each location in the program with the set of solutions of a system of linear inequalities. We reuse the idea of setting, *a priori*, a template $T_\ell \boldsymbol{v} \leq \boldsymbol{t}_\ell$, but we do so for each location ℓ. Here, T_ℓ is a matrix with p_ℓ rows and n columns. n is the number of variables in the program but p_ℓ can be arbitrarily chosen. We will take for instance $\forall l \in \mathcal{L}, p_l = p$, starting with a low value and if we do not find a partial recurrent set, we can try adding more constraints by increasing the value of p. Note that we will not be able to describe every kind of recurrent sets but with more linear constraints we are able to describe more sets. Finding the templates becomes also harder then.

Algorithm 2 RECURRENCE: procedure to find partial recurrent sets

Input: Relation R

```
1:  all_scs ← list of strongly connected subgraphs of R
2:  for p = 1 to max do
3:    for all scs ∈ all_scs do
4:      for all ℓ ∈ locations of scs do
5:        new template T_ℓ with p unknown linear constraints
6:      end for
7:      match solve constraint system with
8:        | Unsat → continue
9:        | Model(m) →
10:         assign values provided by m to templates T_ℓ
11:         return "Partial recurrent set found with templates T_ℓ"
12:       | Timeout → remove scs from all_scs and continue
13:   end for
14: end for
15: return "No partial recurrent set was found"
```

At line 7 of algorithm 2 we call a constraint solver. Indeed, we solve the problem by a reduction to arithmetic, non-linear, constraint systems. We present now different constraint systems we could theoretically use, with respect to the kind of program we are working on. All of them are restricted to linear updates but we may also allow non-deterministic updates (*i.e.* user inputs or random numbers) and non-deterministic branching. For instance, the original idea by Gupta *et al.* is restricted to programs using only linear arithmetic without non-determinism at all.

We will use approximations in most cases in order to be able to apply Farkas' lemma and get rid of universal quantifiers before calling a SMT solver.

Proposition 2.3 (Farkas' lemma)**.** *Let A be a $p \times n$ matrix, $\boldsymbol{b} \in \mathbb{R}^p$, $\boldsymbol{c} \in \mathbb{R}^n$, $\delta \in \mathbb{R}$, the following statements are equivalent:*

1. $\forall \boldsymbol{x} \in \mathbb{R}^n, A\boldsymbol{x} \leq \boldsymbol{b} \rightarrow \boldsymbol{c}^\intercal \boldsymbol{x} \leq \delta$

2. $\exists \boldsymbol{\lambda} \in \mathbb{R}^p, \boldsymbol{\lambda} \geq 0 \wedge \boldsymbol{\lambda}^\intercal A = \boldsymbol{c}^\intercal \wedge \boldsymbol{\lambda}^\intercal \boldsymbol{b} \leq \delta$

2.2.1 Constraint systems for deterministic updates only

In case there are only deterministic updates, we can do the following. For each transition $(\ell_i, \rho_i, \ell'_i) \in \mathcal{T}$, we separate ρ_i into two parts, a guard constraint $G_i \boldsymbol{v} \leq \boldsymbol{g}_i$ (G_i has q_i rows and n columns) and an update statement $\boldsymbol{v}' = U_i \boldsymbol{v} + \boldsymbol{u}$.

Contrarily to Gupta *et al.*, we want to handle, at least, programs with non-deterministic branching, meaning that there might be several possible transitions to trigger when we are in one given state; we require that *at least one* leads back to the partial recurrent set. So we would like to solve the most general constraint:

$$\forall \boldsymbol{v} \bigwedge_{\ell \in \mathcal{L}} \left(T_\ell \boldsymbol{v} \leq \boldsymbol{t}_\ell \rightarrow \bigvee_{(\ell, \rho_i, \ell'_i) \in \mathcal{T}} G_i \boldsymbol{v} \leq \boldsymbol{g}_i \wedge T_{\ell'_i} U_i \boldsymbol{v} \leq \boldsymbol{t}_{\ell'_i} - T_{\ell'_i} \boldsymbol{u}_i \right)$$

which is then a necessary and sufficient condition for the conditions from Prop. 2.2. However, this introduces a disjunction that makes it impossible to apply Farkas' Lemma and will make it harder for a constraint solver to handle.

A first idea of approximation (making again Farkas' lemma usable) was rather to solve:

$$\forall \boldsymbol{v} \bigwedge_{(\ell_i, \rho_i, \ell'_i) \in \mathcal{T}} \left(T_{\ell_i} \boldsymbol{v} \leq \boldsymbol{t}_{\ell_i} \rightarrow G_i \boldsymbol{v} \leq \boldsymbol{g}_i \wedge T_{\ell'_i} U_i \boldsymbol{v} \leq \boldsymbol{t}_{\ell'_i} - T_{\ell'_i} \boldsymbol{u}_i \right)$$

Here, we require that *every* transition be enabled for a state in the partial recurrent set and that they all lead back to it. So, we are actually looking for specific *closed recurrent sets* (as defined in [5]). When using this idea, we must also ensure that every location for which the set of recurrent states is not empty has at least one transition departing from it.

I introduced a new idea of approximation which is the one I implemented:

$$\bigwedge_{\ell \in \mathcal{L}} \bigvee_{(\ell, \rho_i, \ell'_i) \in \mathcal{T}} \forall \boldsymbol{v} \left(T_\ell \boldsymbol{v} \leq \boldsymbol{t}_\ell \rightarrow G_i \boldsymbol{v} \leq \boldsymbol{g}_i \wedge T_{\ell'_i} U_i \boldsymbol{v} \leq \boldsymbol{t}_{\ell'_i} - T_{\ell'_i} \boldsymbol{u}_i \right)$$

Here, we require that from each location one transition is enabled for all states in the partial recurrent set and leads back to it. This is less restricted than the previous approximation in the case where there are several transitions departing from one location (non-deterministic branching) but it still allows us to apply Farkas' lemma.

2.2.2 Constraint systems for deterministic and non-deterministic updates

In the case there are some non-deterministic updates, we can adapt the last method, with inspiration found in [7]. We still decompose transitions

into a guard and an update but potentially non-deterministic updates are represented by $U_i\boldsymbol{v} + U_i'\boldsymbol{v}' \leq \boldsymbol{u}_i$. U_i and U_i' are matrices with r_i rows and n columns and they have the following property:

$$\forall \boldsymbol{v} \exists \boldsymbol{v}' (G_i\boldsymbol{v} \leq \boldsymbol{g}_i \rightarrow U_i\boldsymbol{v} + U_i'\boldsymbol{v}' \leq \boldsymbol{u}_i) \tag{1}$$

This property is important because it allows us to decide whether a transition can be triggered only by evaluating constraints on initial conditions (so without the need to find if there exists a conforming update). Thus we will be able to write constraint system (2) with universal quantifiers only and then apply Farkas' lemma.

Here we encode the facts that at least one transition must always be enabled and that this transition must always lead back to the partial recurrent set:

$$\bigwedge_{\ell \in \mathcal{L}} \bigvee_{(\ell, \rho_i, \ell_i') \in \mathcal{T}} \Bigg[\forall \boldsymbol{v} \left(T_\ell \boldsymbol{v} \leq \boldsymbol{t}_\ell \rightarrow G_i \boldsymbol{v} \leq \boldsymbol{g}_i\right) \wedge \forall \boldsymbol{v} \forall \boldsymbol{v}' \left(T_\ell \boldsymbol{v} \leq \boldsymbol{t}_\ell \wedge U_i \boldsymbol{v} + U_i' \boldsymbol{v}' \leq \boldsymbol{u}_i \rightarrow T_{\ell_i'} \boldsymbol{v}' \leq \boldsymbol{t}_{\ell_i'} \right) \Bigg] \tag{2}$$

And we can again apply Farkas' lemma:

$$\bigwedge_{\ell \in \mathcal{L}} \bigvee_{(\ell, \rho_i, \ell_i') \in \mathcal{T}} \Bigg[\exists \Lambda_\ell \geq 0, \Lambda_\ell' \geq 0, \Lambda_\ell'' \geq 0, \begin{pmatrix} \Lambda_\ell & 0 \\ \Lambda_\ell' & \Lambda_\ell'' \end{pmatrix} \begin{pmatrix} T_\ell & 0 \\ U_i & U_i' \end{pmatrix} = \begin{pmatrix} G_i & 0 \\ 0 & T_{\ell_i'} \end{pmatrix} \wedge \begin{pmatrix} \Lambda_\ell & 0 \\ \Lambda_\ell' & \Lambda_\ell'' \end{pmatrix} \begin{pmatrix} \boldsymbol{t}_\ell \\ \boldsymbol{u}_i \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{g}_i \\ \boldsymbol{t}_{\ell_i'} \end{pmatrix} \Bigg]$$

We use Z3 to solve this constraint system which is non-linear as Λ_ℓ and T_ℓ are both unknown. This is one reason for the complexity of this problem and, in particular, explaining that we cannot add too much constraints before making Z3 time out.

2.3 Finding guards and updates with non-determinism

Getting a representation verifying property (1) is not necessarily easy. Given a sequence of conditions and assignments, we must draw guards and updates. Guards must only involve pre-variables what is not straightforward, given that there might be some postconditions as in:

$$\mathsf{x} := \mathsf{x} - 1;$$
$$\mathit{assume}(\mathsf{x} \geq 0);$$

In this case, it is relatively easy to see what the guard should be:

$$\mathsf{x} \geq 1$$

There might be however more complexed cases:

$$\mathsf{x} := nondet();$$
$$assume(\mathsf{x} \geq \mathsf{y});$$
$$assume(\mathsf{x} \leq \mathsf{z});$$

Here, to verify property (1), the guard must be:

$$\mathsf{y} \leq \mathsf{z}$$

When variables involved in the conditions are computed by deterministic updates only, then we can substitute and draw from there guards on pre-variables only. But if we are in a case similar to the latter example, we need another method. We have chosen to apply in all cases Fourier-Motzkin quantifier elimination to get rid of all intermediary and post-variables.

Proposition 2.4. *Given a set of linear formulas involving variables of the form x^i where $i \in \mathbb{N} \cup \{\text{post}\}$ and x is in the set of variables of the program, we apply Fourier-Motzkin algorithm and set:*

- *guards are the linear formulas in which all variables with $i \neq 0$ have been eliminated;*
- *updates are the linear formulas in which all variables with $i \neq 0 \wedge i \neq$ post have been eliminated.*

Then, by defining G, $\boldsymbol{g}$, U, U' and $\boldsymbol{u}$ accordingly to the fact that variables with $i = 0$ represent pre-variables (value of the variable before taking the transition) and variables with $i =$ post represent post-variables (value of the updated variable), property (1) is verified.

Proof. The guard $G\boldsymbol{v} \leq \boldsymbol{g}$ can be drawn from the update $U\boldsymbol{v} + U'\boldsymbol{v}' \leq \boldsymbol{u}$ by eliminating every post-variable with the Fourier-Motzkin algorithm. By definition of variables elimination, $\forall \boldsymbol{v}(G\boldsymbol{v} \leq \boldsymbol{g} \rightarrow \exists \boldsymbol{v}' U\boldsymbol{v} + U'\boldsymbol{v}' \leq \boldsymbol{u})$. Thus, we have property (1). □

2.4 Tackling some harder examples

Not finding any recurrent set is one of the main problem of our method and this is often due to taking too long to solve the constraint system. Ideas to improve this include allowing more time before timing out, allowing different numbers of constraints in the template with respect to the location, catching timeouts and adding some clever analysis, reducing the number of variables involved by making an analysis of active variables or working with subsets of strongly connected components[4] to reduce the size of the constraint system to solve.

There are also some cases where approximation (2) is not good because it has no solution, even if there exists some recurrent set.

[4]If we are not already looking at every strongly connected subgraph

2.4.1 Non-determinism

Here is a first example[5]:

START: 1;		
FROM: 1;	$\mathsf{x} := nondet();$	**TO**: 2;
FROM: 2;	$assume(\mathsf{x} \geq 1);$	**TO**: 1;
FROM: 2;	$assume(\mathsf{x} \leq -1);$	**TO**: 1;

In this case, we might think about two different kinds of recurrent sets: $[\![\mathsf{x} > 0]\!]$ and $[\![\mathsf{x} \neq 0]\!]$. The second one is not expressible with a linear template. So we want our tool to be able to find the first one.

The problem here is that $[\![\mathsf{x} > 0]\!]$ cannot be found by solving the constraint system with approximation (2). Indeed, the first transition does not necessarily always lead back to the recurrent set.

First solution: abstracting. By adding smart assumptions in transitions with non-deterministic updates, we can sometimes tackle this kind of issues. For instance, here, it would require only to add $assume(\mathsf{x} \geq 1)$ to the first transition, after the non-deterministic update of x. Such an abstraction can be found during the constraint solving.

Second solution: getting rid completely of non-deterministic variables. This is much more radical but also the logical next step. Since a variable which was last updated to a non-deterministic value does not count at all to know if the template of the arrival location will be verified, we can replace it by a fresh variable. This is equivalent to keeping an existential quantifier for every updated variable that is totally free, or to abstracting with the conditions provided by the template.

We will assume $\boldsymbol{v}' = \begin{pmatrix} \boldsymbol{v}'_1 \\ \boldsymbol{v}'_2 \end{pmatrix}$ with every variable in $\boldsymbol{v}'_1$ which actually appears in $U_i\boldsymbol{v} + U'_i\boldsymbol{v}' \leq \boldsymbol{u}_i$ and every variable in $\boldsymbol{v}'_2$ which does not (consequently, the update can be re-written $U_i\boldsymbol{v} + U'_i \begin{pmatrix} \boldsymbol{v}'_1 \\ 0 \end{pmatrix} \leq \boldsymbol{u}_i$). The template $T_{\ell'_i}\boldsymbol{v}' \leq \boldsymbol{t}_{\ell'_i}$ is re-written $T_{\ell'_i} \begin{pmatrix} \boldsymbol{v}'_1 \\ 0 \end{pmatrix} \leq \boldsymbol{t}_{\ell'_i} - T_{\ell'_i} \begin{pmatrix} 0 \\ \boldsymbol{v}'_2 \end{pmatrix}$ before we apply Farkas' lemma for all $\boldsymbol{v}$ and all $\boldsymbol{v}'_1$. $\boldsymbol{v}'_2$ is set to be a vector of fresh variables.

I implemented this second solution and many new program tests are proved non-terminating. However, we observe also that the constraint solving is more complicated as some tests lead more quickly to a timeout[6]. This is probably due to having more multiplications of unknown variables.

2.4.2 No approximation

We might have wanted to try to solve the most general constraint too, as it has not the problem of being an approximation:

[5] In this example and in later ones, the original program contained a condition $assume(\mathsf{x} \neq 0)$ which was automatically split into two linear conditions.

[6] It would be good to check that no test that was solved before times out now.

$$\forall \boldsymbol{v} \bigwedge_{\ell \in \mathcal{L}} \left(T_\ell \boldsymbol{v} \leq \boldsymbol{t}_\ell \rightarrow \bigvee_{(\ell, \rho_i, \ell'_i) \in \mathcal{T}} \exists \boldsymbol{v}' (G_i \boldsymbol{v} \leq \boldsymbol{g}_i \wedge U_i \boldsymbol{v} + U'_i \boldsymbol{v}' \leq \boldsymbol{u}_i \wedge T_{\ell'_i} \boldsymbol{v}' \leq \boldsymbol{t}_{\ell'_i}) \right)$$

Note that this constraint is non-linear, contains universal quantifiers and that we cannot apply Farkas' lemma. Consequently, it is much harder to handle by T2 and it has not proved to be very useful. On a set of unit tests I had (some of them being very simple), very few did not lead Z3 to time out when solving the most general constraint. They were some (but not all) programs with only one variable. Half of these tests could already be handled very well with the approximations[7]. The other half could be handled with the improvement described in paragraph 2.4.1[8]. The later are all similar to the following:

START: 1;
FROM: 1; $assume(\mathsf{x} \leq -1)$; $\mathsf{x} := nondet()$; **TO**: 1;
FROM: 1; $assume(\mathsf{x} \geq -1)$; $\mathsf{x} := nondet()$; **TO**: 1;

It would be interesting to determine exactly which kind of example can be handled and which cannot because we might then try to combine the best approximation (2) and the most general constraint, with respect to the location, in the same constraint system.

2.4.3 Disjunction

The following example really requires disjunctive templates:

START: 1;
FROM: 1; $assume(\mathsf{x} \leq -1)$; $\mathsf{x} := 1$; **TO**: 1;
FROM: 1; $assume(\mathsf{x} \geq 1)$; $\mathsf{x} := -1$; **TO**: 1;

Solution: If we allow recurrent sets to be represented by a disjunction of templates $\bigvee_{d=1}^{q_\ell} T_\ell^d \boldsymbol{v} \leq t_\ell^d$, the constraint system (2) becomes:

$$\bigwedge_{\ell \in \mathcal{L}} \bigwedge_{d=1}^{q_\ell} \bigvee_{(\ell, \rho_i, \ell'_i) \in \mathcal{T}} \left[\forall \boldsymbol{v} \left(T_\ell^d \boldsymbol{v} \leq \boldsymbol{t}_\ell^d \rightarrow G_i \boldsymbol{v} \leq \boldsymbol{g}_i \right) \wedge \right.$$
$$\left. \forall \boldsymbol{v} \forall \boldsymbol{v}' \left(T_\ell^d \boldsymbol{v} \leq \boldsymbol{t}_\ell^d \wedge U_i \boldsymbol{v} + U'_i \boldsymbol{v}' \leq \boldsymbol{u}_i \rightarrow \bigvee_{d'=1}^{q_\ell} T_{\ell'_i}^{d'} \boldsymbol{v}' \leq \boldsymbol{t}_{\ell'_i}^{d'} \right) \right]$$

The second disjunction making it again impossible to apply Farkas' lemma, we then approximate with:

[7] flipflop.t2, rewrite.t2 and w1.t2
[8] n-32.t2, p-32.t2 and p-33.t2

$$\bigwedge_{\ell\in\mathcal{L}}\bigwedge_{d=1}^{q_\ell}\bigvee_{(\ell,\rho_i,\ell'_i)\in\mathcal{T}}\Bigg[\forall\boldsymbol{v}\left(T_\ell^d\boldsymbol{v}\le\boldsymbol{t}_\ell^d\to G_i\boldsymbol{v}\le\boldsymbol{g}_i\right)\wedge \bigvee_{d'=1}^{q_\ell}\forall\boldsymbol{v}\forall\boldsymbol{v}'\left(T_\ell^d\boldsymbol{v}\le\boldsymbol{t}_\ell^d\wedge U_i\boldsymbol{v}+U'_i\boldsymbol{v}'\le\boldsymbol{u}_i\to T_{\ell'_i}^{d'}\boldsymbol{v}'\le\boldsymbol{t}_{\ell'_i}^{d'}\right)\Bigg]$$

For the given example, there is no simple linear template for a recurrent set. However, if we specify we are looking for a disjunctive template with two clauses, then the obvious one is found (whatever constraint system we use, be with or without approximation). However, for now, our program cannot properly handle this disjunctive template because of the reachability checker which was not prepared to that.

Note that disjunctive templates are useful only in the case where every execution has to pass several times the same location while variables are taking different values that cannot be represented with the same linear constraint (as in our example). It is not necessary and it does not help in the case of totally non-deterministic updates as in paragraph 2.4.1. The problem then comes from the approximation: once we are in one particular clause, we are compelled to stay in the particular clauses which are linked with it and we do not actually make use of the wealth of the disjunctive template.

Note also that there is another solution to such problems we tackle with disjunctive recurrent sets: it is to duplicate problematic locations. Indeed the following example is very well handled by our basic version:

START: 1;
FROM: 1; $assume(\mathsf{x}\le -1)$; $\mathsf{x}:=1$; **TO**: 1;
FROM: 1; $assume(\mathsf{x}\ge 1)$; $\mathsf{x}:=-1$; **TO**: 1;
FROM: 1; $assume(\mathsf{x}\le -1)$; $\mathsf{x}:=1$; **TO**: 2;
FROM: 1; $assume(\mathsf{x}\ge 1)$; $\mathsf{x}:=-1$; **TO**: 2;
FROM: 2; $assume(\mathsf{x}\le -1)$; $\mathsf{x}:=1$; **TO**: 2;
FROM: 2; $assume(\mathsf{x}\ge 1)$; $\mathsf{x}:=-1$; **TO**: 2;
FROM: 2; $assume(\mathsf{x}\le -1)$; $\mathsf{x}:=1$; **TO**: 1;
FROM: 2; $assume(\mathsf{x}\ge 1)$; $\mathsf{x}:=-1$; **TO**: 1;

2.4.4 Partition

In the following example, very close to the one of paragraph 2.4.1, we propose a different solution:

START: 1;
FROM: 1; $\mathsf{x}:=nondet()$; **TO**: 2;
FROM: 2; $assume(\mathsf{x}\ge 0)$; **TO**: 1;
FROM: 2; $assume(\mathsf{x}\le 0)$; **TO**: 1;

Solution: The problem here is that we can take both transitions with respect to the preconditions but none every time. To tackle this, we can allow different transitions to be taken but decide which of them to select according to a linear partition. In the following formula, Farkas' lemma can still be applied on the first part:

$$\bigwedge_{\ell\in\mathcal{L}}\left[\bigwedge_{(\ell,\rho_i,\ell'_i)\in\mathcal{T}}\left(\forall\boldsymbol{v}\left(T_\ell\boldsymbol{v}\le\boldsymbol{t}_\ell\wedge Q_{\ell'_i}\boldsymbol{v}\le\boldsymbol{q}_{\ell'_i}\to G_i\boldsymbol{v}\le\boldsymbol{g}_i\right)\wedge\right.\right.$$
$$\forall\boldsymbol{v}\forall\boldsymbol{v}'\left(T_\ell\boldsymbol{v}\le\boldsymbol{t}_\ell\wedge Q_{\ell'_i}\boldsymbol{v}\le\boldsymbol{q}_{\ell'_i}\wedge U_i\boldsymbol{v}+U'_i\boldsymbol{v}'\le\boldsymbol{u}_i\to T_{\ell'_i}\boldsymbol{v}'\le\boldsymbol{t}_{\ell'_i}\right)\Big)\wedge$$
$$\left.\forall\boldsymbol{v}\bigvee_{(\ell,\rho_i,\ell'_i)}Q_{\ell'_i}\boldsymbol{v}\le\boldsymbol{q}_{\ell'_i}\right]$$

The last part is added there to specify that $\{[\![Q_{\ell'_i}\boldsymbol{v}\le\boldsymbol{q}_{\ell'_i}]\!],(\ell,\rho_i,\ell'_i)\in\mathcal{T}\}$ defines indeed a partition (with possibly some of its elements to be empty). It contains the only lasting (universal) quantifier, after we have applied Farkas' lemma. However, we might express it otherwise, without needing for a quantifier, in particular by restricting the type of partition we can use. For instance, we might solve the problem above by using a partition with only one linear inequality: if $\mathsf{x}\le 0$ or if $\mathsf{x}\ge 0$. I did not implement this last idea because it is both very specific and not so useful given the other improvements.

2.5 Checking reachability and getting invariants

In order to to check the reachability of partial recurrent sets, we call an interpolation-based safety checker (see [15] for the main ideas). To do so, we connect every location in the recurrent set with a new error location and we guard these new transitions with the templates defining the recurrent set. It is then equivalent to test the reachability of the recurrent set or of the error location.

During the safety check, an execution tree is built. In case the partial recurrent set was not reachable, we draw the invariant from the execution tree. Each node of the execution is mapped to a set of constraints representing the current state. For a given location (in the program), the new invariant is the disjunction of all these sets of constraints (understood as conjunctions) for nodes in the tree which correspond to the same location.

We then strengthen any transition starting from a given location with a constraint that this invariant must hold.

3 Termination proving

I remind here the main algorithm used in the current version of T2, first described by Brockshmidt *et al.* [2]. T2 works by successive, lasso-shaped-counter-example-based refinements. As briefly described in the introduction, it relies on both a ranking function synthesis tool and a

safety checker which tests the ranking functions and provides potential counter-examples.

The alternation between the two tools is the place for a reinforced cooperation through the use of a cooperation graph. Indeed, the program is represented as a graph in which every state and transition has been duplicated to allow safety checks and termination proving to work on different parts of the graph soundly and yet to be able to share their information. We have also used this graph soundly for non-termination proving as every run through it can be mapped to an actual program run (in particular in the case of an infinite path which will be mapped to an infinite execution provided such path is reachable).

T2 also includes already some non-termination proving techniques to check if a provided counter-example might be genuine.

Algorithm 3 The termination prover T2

Input: Cooperation graph C

1: **for all** strongly connected component S **do**
2: **while** $\exists$ S-orienting rank function f **do**
3: remove strictly decreasing parts
4: **end while**
5: **end for**
6: **while** $\exists$ counter-example $(stem, cycle)$ **do**
7: $S \leftarrow$ strongly connected component containing $cycle$
8: **if** $\exists$ S-orienting rank function f **then**
9: remove strictly decreasing parts
10: strengthen $cycle$ by adding that f should not decrease during an infinite loop
11: **else if** $\exists f$ rank function for $cycle$ **then**
12: strengthen $cycle$ by adding that f should not decrease during an infinite loop
13: **else**
14: try to prove non-termination or **return** "Unknown"
15: **end if**
16: **end while**
17: **return** "Terminating"

I have used T2 as the termination prover in my tool combining termination and non-termination proving. However, since T2 was already very efficient and I had to attest specifically of the usefulness of my tool, I had to consider how to alleviate T2 power.

Our goal, for a final version of the tool, will not be to alleviate T2 but rather to alternate without waiting for the whole algorithm to unroll. Possible restrictions of algorithm 3 in order to alternate more quickly include reducing the loop from line 2 to 4 to only one iteration, reducing the loop from 6 to 16 to only one iteration or removing it completely (as the first loop already proves termination in most cases) and removing the non-termination proving techniques. All the choices are described in

section 4.

4 Alternation

As described in the introduction, the alternation procedure is rather simple as we merely try successively to prove termination and non-termination by calling the sub-procedures described in sections 3 and 2. We are working on the cooperation graph which is necessary for the termination prover and which will be directly transformed by it to remove terminating parts (first cooperative step). The non-termination prover removes newly found unreachable parts as described in subsection 2.5.

As soon as we try this procedure, we get a lot more programs which are proved either terminating or non-terminating (see section 5), thus highlighting the usefulness of such an alternation.

However, it is harder to find the good rate of alternations and speed up the proof finding.

We first tried to alternate after each refinement by T2 (see algorithm 4) but it took then very long for some termination proving.

In the current version, we do the following. During the preliminary steps of the termination prover (lines 1 to 5 of algorithm 3), we also look for partial recurrent sets for every strongly connected component. New invariants are added each time an unreachable partial recurrent set is found. We may alternate at the end of these preliminary steps provided at least one partial recurrent set has been found (in this case, if the non-termination proving step leads to nothing more we will not do again the preliminary steps when coming back to termination proving). Then, during the main loop, each time we generate a potential counter-example, we first try to prove it is genuine by looking for a recurrent set, before searching for a new ranking function and proving the counter-example is spurious.

During this step when we look for a recurrent set on the locations of the cycle of the counter-example, we try two techniques: the one we described earlier and a simpler one which was already implemented in T2 (finding an invariant for the loop and checking if it defines a recurrent set). This last method works surprisingly well in a large number of cases.

Alternating more frequently is intended to accelerate the finding of a proof of either termination or non-termination. However, we can observe in practice many cases for which it delays termination proving by repeated and unnecessary attempts to prove non-termination. We should find a way to detect if new progress in the termination proving justify or not to attempt again to prove non-termination.

5 Results

Some improvements of the procedure might still be useful, as well as a precise setting of parameters (parameters include lower and larger numbers of constraints we use, delays before timing out during the constraint

Algorithm 4 Alternation at each step

Input: Cooperation graph C

```
loop
  call T2 with only one counter-example finding
  snapshot the state of T2
  if T2 returned "Terminating" then
    return "Terminating"
  else if T2 returned "Counter-example (stem, cycle)" then
    { T2 has given up because it could not find a ranking function for this
    counter-example }
    call non-termination proving sub-procedure with instruction to look at
    cycle first
    if sub-procedure returned "Non-terminating" then
      return "Non-terminating"
    else if sub-procedure returned "New invariant inv" then
      strengthen C with inv
    else
      { sub-procedure has not find anything of interest }
      return "(Non-)termination proving failed"
    end if
  else
    { T2 cannot say yet but has not given up }
    call non-termination proving sub-procedure with some restrictions (a
    shorter timeout, simpler constraint systems...)
    if sub-procedure returned "Non-terminating" then
      return "Non-terminating"
    else if sub-procedure returned "New invariant inv" then
      strengthen C with inv
    else
      { sub-procedure has not find anything of interest }
      resume T2 at snapshot
    end if
  end if
end loop
```

solving, which can differ with respect to the step in the algorithm when we do the solving). Yet, the first results are encouraging[9]

We can see that the average time to prove either termination or non-termination is not much better with our new method. However, more examples are handled, which is even more important. The most interesting cases are the ones for which we prove termination but were unable to do so with T2 alone and the ones for which we prove non-termination but were unable to do so with our new non-termination technique alone[10].

6 Related work

As explained in section 2, we have adapted Gupta et al's method [12], to several locations and to handle non-determinism. It was also the aim of [7] (whose content was later in most part reused in [5]) which defines a new notion of *closed recurrent sets*. I was inspired by them but I introduced a new approximation which allows us to look for more general recurrent sets.

Chen *et al.* [5] propose an implementation for closed recurrent sets which does not rely on constraint-solving but only on reachability checks. I had considered using it before starting my own non-termination proving tool but I gave up because it does not distinguish at all between finding partial recurrent sets and proving them reachable. On the contrary, the interest of such a method is to combine them nicely. Yet, for our purposes, the distinction was very important as finding unreachable recurrent sets is at the core of one of our cooperative steps.

Our work can seem very similar to Haris *et al.*'s [13]. Here are however some important differences. The core difference comes from the way they prove termination.

Their termination proving step is based upon an under-approximation of the transition relation of each loop. They look for termination arguments on the under-approximation and then check them on the whole loop. When they can find a counter-example, they add it to the under-approximation. Thus, they can avoid to look at many terminating parts because they will never be added to the under-approximation. But each step brings a more complex approximation on which termination proving will be more and more difficult. On the contrary, T2 is driven by counter-examples. Finding partial termination arguments still relies on looking closely at the parts which might be non-terminating. However, every new termination argument helps removing a lot of terminating parts and thus each step makes the termination proving easier and closer.

Our non-termination prover also takes advantage of it because there are less loops to look at. Thus, we can consider all of them rather than concentrating on loops which comes from counter-examples we could not handle. This is a second difference in our approches. It explains why our tool is as great as proving termination and non-termination.

[9] Please see `http://tzim.fr/testT2` for the complete results.

[10] The non-termination technique alone was not tested on a full handset of tests. Such examples include curious4.t2, ex31.t2 and fun6.t2

Finally, as explained earlier, our approach to prove non-termination, even if close from previous approaches, is novel. The fact that we can handle several program locations allows us to work on strongly connected subgraphs rather than loops. Thus we do not have to enumerate every loop (as in [12]), nor to specify one loop which the non-termination tool should work on (as in [13]). Finally, we do not need a specific strategy to tackle nested loops.

7 Conclusion

Our work highlights two novel things. Firstly, we can generalize a constraint-based approach to non-termination proving in a clever way such that:

- we do not rely anymore on enumerating or specifying loops to work on;
- non-determinism can be nicely handled in the same kind of tool.

Secondly, the new approach to prove termination that is the core of T2 fits well, without any change, as a part of a tool alternating between termination and non-termination proving.

We can also confirm the usefulness of such a technique as described by Harris *et al.* and first implemented in the tool called TRex [13] and the advantage of separating the search for a partial recurrent set and the check of its reachability.

Finally, by generalizing the method first described by Gupta *et al.*, we showed once more how much a big step it was in the non-termination proving field and how such a method is not limited at all to the framework which the authors had originally thought it for.

Greetings: I would like to thank Microsoft for the fantastic work environment they provide, Byron Cook for hosting and tutoring me and giving me some great boosts, Marc Brockshmidt for being there when I needed help and all the other interns for their stimulating camaraderie.

References

[1] Thomas Ball and Sriram K Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[2] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV*, volume 13, 2013.

[3] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and nullpointerexceptions for java bytecode. In *Formal Verification of Object-Oriented Software*, pages 123–141. Springer, 2012.

[4] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 161–169. IEEE Computer Society, 2009.

[5] Hongyi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. Working paper, 2013.

[6] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[7] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Disproving termination with overapproximation. Unpublished, 2013.

[8] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y Vardi. Proving that programs eventually do something good. In *ACM SIGPLAN Notices*, volume 42, pages 265–276. ACM, 2007.

[9] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *ACM SIGPLAN Notices*, volume 41, pages 415–426. ACM, 2006.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *Automated Reasoning*, pages 281–286. Springer, 2006.

[12] Ashutosh Gupta, Thomas A Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Acm Sigplan Notices*, volume 43, pages 147–158. ACM, 2008.

[13] William R Harris, Akash Lal, Aditya V Nori, and Sriram K Rajamani. Alternation for termination. In *Static Analysis*, pages 304–319. Springer, 2011.

[14] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[15] Kenneth L McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136. Springer, 2006.

[16] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 32–41. IEEE, 2004.

[17] Andreas Podelski and Andrey Rybalchenko. Armc: the logical choice for software model checking with abstraction refinement. In *Practical Aspects of Declarative Languages*, pages 245–259. Springer, 2007.

[18] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[19] Moshe Y Vardi. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1):79–98, 1991.

[20] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Tests and Proofs*, pages 154–170. Springer, 2008.