Termination Analysis via Max-SMT

Marta Grobelna
Supervision: Florian Frohn

SS 2017

Abstract

Software verification is an important discipline of computer science. Incorrect code can lead to large material losses or even to threat to human safety. In the past few decades the complexity of software has increased dramatically. Thus, more efficient verification tools are needed. An important property that verification tools analyze is the termination. A common way to prove termination of a program is to consider the corresponding transition system and to look for ranking functions for each transition. If a transition admits a ranking function then this implies that it is visited only a finite number of times. However, this requirement is in some cases too restricted. Therefore, in this approach one looks for quasi-ranking functions and invariants. This enables the possibility to split a transition and one gets another chance to find a termination argument. In order to find a quasi-ranking function that fulfills as many properties of a ranking function as possible, a Max-SMT solver is used.

1 Introduction

The complexity of software has dramatically increased in the past few decades. Due to this, *software verification* has become more and more complex. Especially for embedded systems the correctness of software is crucial, since failures could lead to large material losses or even to threat to human safety [2, 9]. Therefore, software verification is a very important discipline of computer science. One of the central topics of software verification is *termination analysis* since termination of a program is often necessary in order to guarantee its overall correctness. It deals with the question if a given program terminates or not. The termination problem is similar to the well-known *halting problem* which is, as Alan Turing already proved in 1936 [11], undecidable. A problem is undecidable, if there exits no *decision procedure* for this problem. A decision procedure is an algorithm that always terminates and returns `true` if the problem is satisfiable and `false` if the problem is unsatisfiable. However, it is wrong to think that proving termination is always impossible due to Turing's prove. In recent years many efficient approaches have been developed, which are able to find a *partial* solution for the termination problem. Actually, Turing himself figured out, in 1989 [10], that there are possibilities to solve the problem. The key has been, to rephrase the termination problem, such that it is not only possible to return the answers `terminating` or `non-terminating`, but also to return the answer `unknown`. Obviously, this problem can be solved, as the procedure could always return `unknown`. However, frequent returning of the answer `unknown` is not constructive and therefore the goal is to minimize the number of occurrence of it [4, 7].

For termination analyze an abstract representation of the program is considered. A program which has to be analyzed is represented as a *transition system*. A *state* of a transition system, consists of a location in the program and the current values of the

variables. The idea introduced by Turing was to find, and afterwards to check, *termination arguments*. A termination argument is a function which maps every state to a *well-ordered* structure, e.g., natural numbers. Once such a function is found, it remains to prove that the output of the function decreases in each transition. Such functions are called *ranking functions* [7]. However, finding a single ranking function for whole program is not easy. This is why modern termination analysis tools look for *disjunctive termination arguments*. This approach is based on *Ramsey theorem* [5]. The difference consists in the fact that not a single ranking function is the termination argument but a *set* of ranking functions is the termination argument. This simplifies the search for termination arguments enormously. However, validity check for a set of ranking functions is more complicated. Ramsey's theorem says that it is not enough to proof that a termination argument is valid for single transitions. One has to prove that the termination argument holds for the *transitive closure* of the program's transitions. This is necessary as, proving termination requires finding a ranking function which maps every state to a well-ordered structure. Thus, when using disjunctive termination argument, one has to prove that the disjunctive termination argument is a well-ordered structure. Unfortunately, it does not hold that a union of well-ordered structures is a well-ordered structure. However, it holds that a relation is a well-ordered structure if its *transitive closure* is a well-ordered structure. This is why checking the validity of a set of ranking functions is more complex than for a single ranking function [2, 4, 5, 8].

Many verification tools which analyze termination have too restrict termination requirements. This means that if a transition does not fulfill all properties of well-foundedness, then they usually return `unknown`. This can lead to the problem that a terminating program is not recognized as such. In order to reduce the number of such cases, one does not only look for a ranking function but also for an invariant and a *quasi-ranking function*. A quasi-ranking function does not fulfill all properties of a ranking function. This usually makes is possible to split the corresponding transition and gives a new chance to find a termination argument. However, it is important which properties the quasi-ranking function does not fulfill. For example, if a transition is not decreasing, then there is no chance that the program is terminating. Thus, one introduces *weights* for properties. More important properties get higher weight than less important properties. Therefore, the approach presented in this paper uses *Max-SMT*. Max-SMT is SMT-solver which additionally considers that each formula has a weight. The goal is to find a solution for the set of constraints such that the sum over the weights of satisfied formulas is maximal. This enables the possibility to give priorities to certain properties that a given transition system has to fulfill [4].

The remainder of this paper is organized as follows. In the next section all needed preliminaries are explained. Afterwards, the Max-SMT approach for proving termination is presented as well as some examples which illustrate the approach. Finally follows the conclusion.

2 Preliminaries

2.1 SAT-solver, SMT-solver and Max-SMT

Every computer program can be interpreted as a logical formula, as each program has a certain syntax and the symbols used to construct this program have certain semantics. Thus, proving correctness of a program can be done by proving that a set of logical formulas describing this program is satisfied. The most basic type of logical formulas are

propositional formulas. Those consist of a conjunction of *clauses* over *literals*. A literal l_i can be either a variable x_i or its negation $\overline{x}_i$. A clause is a disjunction over literals. A propositional formula is *satisfied* if there exists an assignment $\mathfrak{I}$ of the variables occurring in the formula, such that the instantiated formula is equivalent to *true*. If all assignments of the variables satisfy the formula, then the formula is called *tautology*, i.e., it is equivalent to *true*. If there exists no assignment satisfying the formula then it is called *unsatisfiable*, i.e., it is equivalent to *false*. Checking if a set of propositional formulas is satisfiable is the task of a *satisfiability solver* (SAT-solver).

Unfortunately, propositional logic is not expressive enough to model all properties of a program. Therefore, a commonly used logic is *quantifier free first-order logic* with respect to a certain background theory. The corresponding decision problem is called *satisfiability modulo theory (SMT)* problem. Checking quantifier free first-order formulas for satisfiability requires special solvers, as it is important to verify if the formula is satisfiable for a certain theory. Hence, SMT-solvers are usually implemented by combining a SAT-solver with theory solver. A variant of SMT is *Max-SMT*. Given a weighted clause, a Max-SMT solver finds an assignment such that the sum over the weights of satisfied clauses is maximal [1, 2, 7].

2.2 Transition Systems

In order to analyze a program for termination, it is represented by a *transition system*.

Definition 2.1 (Transition System) *A transition system $\mathcal{S}$ is a quadruple ($\mathcal{V}$, $\mathcal{L}$, Θ, $\mathcal{T}$) where*

- *$\mathcal{V}$ is a set of variables occurring in the program,*
- *$\mathcal{L}$ is a set of program locations,*
- *Θ is a mapping from locations to formulas defining the initial values of the variables,*
- *$\mathcal{T}$ is a set of transitions, where each transition is a triple (l, l', ρ) consisting of a pre-location l, a post-location l' and a transition relation ρ which is a formula over the variables $\mathcal{V}$ defining how the values of variables change after the transition. The variable values after taking the transition are denoted by $\mathcal{V}'$ [3, 7].*

A *state* is an assignment of a value to each variable in $\mathcal{V}$ [7]. A *configuration* is a tuple (l, σ) consisting of a location l and a state σ. A potentially infinite sequence of configurations $(l_0, \sigma_0), (l_1, \sigma_1), \ldots$ is called *computation* where the assignment of variables at state σ_0 must satisfy all formulas assigned to the location l_0, i.e., $\sigma_0 \models \Theta(l_0)$. Moreover, for each two configurations (σ_i, l_i) and (σ_{i+1}, l_{i+1}) $(i \in \mathbb{N})$ there must exist a transition from the location l_i to l_{i+1}, i.e., $(l_i, l_{i+1}, \rho) \in \mathcal{T}$ for all consecutive configurations in a computation. Nevertheless, the variables values of the pre-location and the variables values of the post-location must satisfy the transition relation ρ, i.e., $(\sigma_i, \sigma_{i+1}) \models \rho$. Consequently a configuration (l, σ) is called *reachable* if and only if there exists a computation ending at (l, σ) [7].

Figure 1 illustrates a simple program and its transition system. For the quadruple defining this transition system holds $\mathcal{V} = \{x, y, z\}$, $\mathcal{L} = \{l_0, l_1\}$, $\Theta = \{(l_0, true), (l_1, false)\}$ and $\mathcal{T} = \{(l_0,\ l_1,\ \rho_{\tau_0} := y \geq 1,\ x' = x - 1,\ y' = y,\ z' = z),\ (l_1,\ l_1,\ \rho_{\tau_1} := y < z,\ x' = x + 1,\ y' = y,\ z' = z - 1),\ (l_1,\ l_0,\ \rho_{\tau_2} := y \geq z,\ x' = x,\ y' = x + y,\ z' = z)\}$.

A transition system, and so the corresponding program, is *terminating* if all possible computations are finite. Hence, in order to prove that a program is terminating, the transition system must not have transitions which can be executed infinitely often in one computation [7].

```
int x, y, z
while y ≥ 1 do
    x−−
    while y < z do
        x++, z−−
    end while
    y = x + y
end while
```

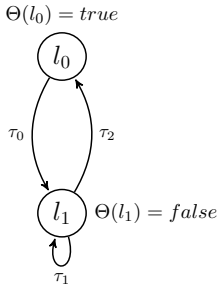

Figure 1: Example of a program and its transition systems.

2.3 Invariants

A well-known technique to check if a certain property holds after executing a sequence of program instructions, is to introduce *assertions* at the corresponding locations. More formally one can say that an assertion is a first-order formula which belongs to a certain location. An assertion I at the location l is called *invariant* if for any reachable configuration (l, σ) it holds that σ satisfies I. Considering the program 1 one can find for example the invariant $y \geq 1$ for the location l_1.

A map that assigns an invariant to each location is called *invariant map*. An *inductive invariant map* μ assigns to each location an invariant such that $\mu(l) \wedge \rho \models \mu(l')'$ holds. The expression $\mu(l')'$ stands for the invariant after substituting the variables $\mathcal{V}$ in $\mu(l')$ with the variables values after taking the transition $\mathcal{V}'$. More formally inductive invariant map can be defined as follows [6, 7].

Definition 2.2 (Inductive Invariant Map) *An inductive invariant map is a map fulfilling the two properties*

- ***Initiation**: For every location $l \in \mathcal{L} : \Theta(l) \models \mu(l)$*
- ***Consecution**: For every transition $\tau = (l, l', \rho) \in \mathcal{T} : \mu(l) \wedge \rho \models \mu(l')'$.*

Note that not every invariant map is inductive. Figure 2 illustrates a transition system. The map $\mu(l_0) = true,\ \mu(l_1) = y > 0$ is not an inductive invariant map, as it does not fulfill consecution. It requires that $\mu(l_0) \wedge \rho \models \mu(l_1)'$, i.e., $true\ \wedge x > 0 \models y > 0$. This is obviously not satisfied here.

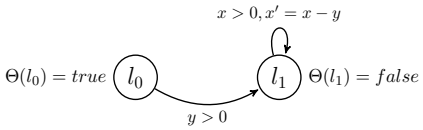

Figure 2: Example of a non-inductive invariant map.

Invariant maps play an important role in termination analysis, due to the fact that they can help to prove that a transition is disabled (a transition which can never be executed). A transition (l, l', ρ) is disabled if and only if there exists an invariant $\mu(l)$ such that $\mu(l) \wedge \rho$ is unsatisfiable. Moreover, inductive invariant maps can help to find *ranking function* [7].

2.4 Ranking Functions

A non-terminating program has at least one infinite computation. Since programs have only a finite number of locations the infinite computation must visit a certain subset of

location an infinite number of times. Such subset of locations belongs to the same *strongly connected component*. A strongly connected component (SCC) of a directed graph (here: transition system), is a part of a graph such that for each node there exists a path to all the remaining nodes [3, 7, 1].

Due to the definition of SCCs, it is reasonable for proving termination to show that there exists no SCC in the transition system that can be visited an infinite number of times. In order to show this, one has to show for each transition belonging to the SCC that either the transition is *disabled* or one can find a ranking function for it. A transition is disabled if it can never be executed [7]. A ranking function is defined as follows

Definition 2.3 (Ranking Function) *Let $\tau = (l, l', p)$ be a transition where l and l' belong to the same SCC C. A function $R : v \to \mathbb{Z}$ with $v \in \mathcal{V}$ is called a ranking function if it fulfills the following three properties*

- ***Boundedness**: $p \models R \geq 0$,*
- ***Strict Decrease**: $p \models R > R'$ where R' is the value after executing the transition,*
- ***Non-increase**: For every $\hat{\tau} = (\hat{l}, \hat{l'}, \hat{p}) \in \mathcal{T}$ such that $\hat{l}, \hat{l'} \in C$: $\hat{p} \models R \geq R'$.*

Thus, a ranking function maps to each transition belonging to a SCC a rank such that there is no transition which increase the rank (non-increase), there is at least one transition which decrease the rank (strict decrease) and the rank is never less than zero (boundedness). Therefore, a transition which admits a ranking function is executed a finite number of times and can be removed from the SCC. This may lead to a decomposition of the SCC into several smaller SCCs. For the new SCC again one has to find either ranking functions for the transitions or one can prove that a transition is disabled. Thus, one can execute this procedure recursively to prove termination of a program [3, 7].

Again considering the program 1, for example z is a ranking function for the transition τ_1. As mentioned before, at l_0 one can find the invariant $y \geq 1$. This invariant can be used to strengthen the transition relation for the transition τ_1 as follows $\rho_{\tau_1} \wedge y \geq 1$. As ρ_{τ_1} contains the constraint $y < z$, z fulfills boundedness ($y < z \wedge y \geq 1$ implies $z > 1$). Moreover, ρ_{τ_1} contains the constraint $z' = z - 1$ thus, z also strictly decreasing. In order to show that z is also non-increasing one has to look at the other transitions in the SCC. As both transition relations ρ_{τ_0} and ρ_{τ_2} contain the constraint $z' = z$, z also is non-increasing [7].

2.5 Constraint-based Program Verification

The basic task of termination analysis is to find invariant properties and ranking function and then to prove that those are valid. In order to find invariant properties and ranking function one needs to find templates for them which are first-order constraints. If there exists a satisfying assignment for the templates then a linear invariant or a ranking function results [6, 7]. As invariants are associated with a certain location, the template for linear invariant has the form

$$I_l(\mathcal{V}) := \exists i_{l,0}, \ldots, i_{l,n} \forall v_1, \ldots, v_n.\ i_{l,0} + \sum_{i=1}^{n} i_{l,v_i} \cdot v_i \leq 0 \tag{1}$$

where v_i are the program variables and $i_{l,j}$ ($j \in \{0, \ldots, n\}$) are the unknown parameters. Similarly to this, the template for ranking function has the form

$$R(\mathcal{V}) := \exists r_0, \ldots, r_n \forall v_1, \ldots, v_n.\ r_0 + \sum_{i=1}^{n} r_{v_i} \cdot v_i \tag{2}$$

where v_i are the program variables and r_{v_i} are the unknown parameters. However, this templates are not quantifier free first-order formulas. In order to remove the universal quantifiers, one can apply *Farkas' Lemma*. The result is a set of existentially quantified constraints. The resulting problem can be solved by a SMT-solver [6, 7].

3 Termination Analysis via Max-SMT

3.1 General Approach

Proving termination requires the prove that all SCCs are visited only a finite number of times. This in turn means that for every single SCC one has to find termination arguments (ranking function or disability argument) for each transition in the SCC. If a transition belonging to a certain SCC admits a ranking function (or is disabled), then it can be removed from the SCC. Usually this causes a decomposition of the SCC into several smaller SCCs. This procedure is executed recursively on the transition system.

As termination arguments are needed for each transition of an SCC, a constraint system must be constructed. If this constraint system is satisfiable, then it will entail a linear inequality. The entailed linear inequality is either a ranking function or a disability argument [7].

3.2 Constructing Constraint System for Termination Analysis

Since solving the constraint system should lead to a ranking function or an invariant, the corresponding properties must somehow be encoded in this system. Therefore, the corresponding constraint system will be constructed out of these constraints:

- **Initiation**: For $l \in \mathcal{L} : \mathbb{I}_l := \Theta(l) \vdash I_l$,
- **Consecution**: For $\tau = (l, l', \rho) \in \mathcal{T} : \mathbb{C}_\tau := I_l \wedge \rho \vdash I'_{l'}$,
- **Disability**: For $\tau = (l, l', \rho) \in \mathcal{T} : \mathbb{D}_\tau := I_l \wedge \rho \vdash 1 \leq 0$,
- **Boundedness**: For $\tau = (l, l', \rho) \in \mathcal{T} : \mathbb{B}_\tau := I_l \wedge \rho \vdash R \geq 0$,
- **Strict Decrease**: For $\tau = (l, l', \rho) \in \mathcal{T} : \mathbb{S}_\tau := I_l \wedge \rho \vdash R > R'$,
- **Non-increase**: For $\tau = (l, l', \rho) \in \mathcal{T} : \mathbb{N}_\tau := I_l \wedge \rho \vdash R \geq R'$

The first two constraints encode the properties of inductive invariant map. Disability encodes the fact, that if a transition is disabled, then it cannot be executed and so the invalid inequality $1 \leq 0$ can be derived. The last three constraints encode the properties of ranking functions. The two symbols R and I correspond to the previously defined templates for ranking function and linear invariant (see equations 1 and 2). For example in Fig. 1 holds

$$\begin{aligned} I_{l_0}(\mathcal{V}) &= \exists i_{l_0,0},\ i_{l_0,x},\ i_{l_0,y},\ i_{l_0,z}\ \forall x,\ y,\ z.\ i_{l_0,0} + i_{l_0,x} \cdot x + i_{l_0,y} \cdot y + i_{l_0,z} \cdot z \leq 0 \\ I_{l_1}(\mathcal{V}) &= \exists i_{l_1,0},\ i_{l_1,x},\ i_{l_1,y},\ i_{l_1,z}\ \forall x,\ y,\ z.\ i_{l_1,0} + i_{l_1,x} \cdot x + i_{l_1,y} \cdot y + i_{l_1,z} \cdot z \leq 0 \\ R(\mathcal{V}) &= \exists r_0,\ r_x,\ r_y,\ r_z, \forall x,\ y,\ z.\ r_0 + r_x \cdot x + r_y \cdot y + r_z \cdot z \end{aligned} \tag{3}$$

The constraints for the example have the following form

$$\begin{aligned} &\mathbb{I}_{l_0} = \Theta(l_0) \vdash I_{l_0},\ \mathbb{I}_{l_0} = \Theta(l_0) \vdash I_{l_0} \\ &\mathbb{D}_{\tau_0} = I_{l_0} \wedge p_{\tau_0} \vdash 1 \leq 0,\ \mathbb{D}_{\tau_1} = I_{l_1} \wedge p_{\tau_1} \vdash 1 \leq 0,\ \mathbb{D}_{\tau_2} = I_{l_1} \wedge p_{\tau_2} \vdash 1 \leq 0 \\ &\mathbb{C}_{\tau_0} = I_{l_0} \wedge p_{\tau_0} \vdash I'_{l_1},\ \mathbb{C}_{\tau_1} = I_{l_1} \wedge p_{\tau_1} \vdash I'_{l_1},\ \mathbb{C}_{\tau_2} = I_{l_1} \wedge p_{\tau_2} \vdash I'_{l_0} \end{aligned}$$

$$\mathbb{B}_{\tau_0} = I_{l_0} \wedge p_{\tau_0} \vdash R \geq 0,\ \mathbb{B}_{\tau_1} = I_{l_1} \wedge p_{\tau_1} \vdash R \geq 0,\ \mathbb{B}_{\tau_2} = I_{l_1} \wedge p_{\tau_2} \vdash R \geq 0$$
$$\mathbb{S}_{\tau_0} = I_{l_0} \wedge p_{\tau_0} \vdash R > R',\ \mathbb{S}_{\tau_1} = I_{l_1} \wedge p_{\tau_1} \vdash R > R',\ \mathbb{S}_{\tau_2} = I_{l_1} \wedge p_{\tau_2} \vdash R > R'$$
$$\mathbb{N}_{\tau_0} = I_{l_0} \wedge p_{\tau_0} \vdash R \geq R',\ \mathbb{N}_{\tau_1} = I_{l_1} \wedge p_{\tau_1} \vdash R \geq R',\ \mathbb{N}_{\tau_2} = I_{l_1} \wedge p_{\tau_2} \vdash R \geq R'$$

Now, the constraints must somehow be combined. Since an inductive invariant should be computed, the constraint encoding initiation should be a conjunction over all locations $\mathcal{L}$ in the regarding SCC. The other property needed for inductive invariant map is the consecution. However, it is important to pay attention to the fact that this property refers to transitions. Therefore, one has to consider that a transition can also be disabled. Thus, each transition in the SCC either fulfills the consecution property or it is disabled, since this should hold for all transitions ($\mathcal{T}$ is the set of all transitions) in the SCC, a conjunction over the disjunction of the two properties is needed. Hence, the sub-formula looks like this until now

$$\bigwedge_{l \in \mathcal{L}} \mathbb{I}_l \wedge \bigwedge_{\tau \in \mathcal{T}} (\mathbb{C}_\tau \vee \mathbb{D}_\tau)$$

As already mentioned, once a transition admits a ranking function it can be discarded from the transition system. This implies that the set of transitions $\mathcal{T}$ decreases to a new set of transitions $\mathcal{T}'$. Now, the properties for ranking functions are considered. Since these properties also refer to transitions, one also has to pay attention to the possibility that a transition can be disabled. Moreover it is required, that either there is at least one disabled transition or no transition increase the measure. This lead to the following sub-formula

$$\bigwedge_{\tau \in \mathcal{T}'} \mathbb{N}_\tau \vee \bigvee \tau \in \mathcal{T}' \mathbb{D}_\tau$$

A transition which can admit a ranking function must fulfill two more requirements. It must be bound and strictly decreasing. Due to the disability possibility this sub-formula has the following form

$$\bigvee_{\tau \in \mathcal{T}'} ((\mathbb{B}_\tau \wedge \mathbb{S}_\tau) \vee \mathbb{D}_\tau)$$

All in all, the constraints system has the form

$$\bigwedge_{l \in \mathcal{L}} \mathbb{I}_l \wedge \bigwedge_{\tau \in \mathcal{T}} (\mathbb{C}_\tau \vee \mathbb{D}_\tau) \wedge \left(\bigwedge_{\tau \in \mathcal{T}'} \mathbb{N}_\tau \vee \bigvee_{\tau \in \mathcal{T}'} \mathbb{D}_\tau \right) \wedge \bigvee_{\tau \in \mathcal{T}'} \left((\mathbb{B}_\tau \wedge \mathbb{S}_\tau) \vee \mathbb{D}_\tau \right) \tag{4}$$

The drawback of this solution is that it allows only one invariant for each ranking function. This is not always suitable in praxis. Another drawback is that here a single ranking function or disability argument is returned as proof for termination. Therefore, the next subsection deals with Max-SMT which can be used to make the termination analysis more accurate [7].

3.3 Max-SMT for Proving Termination

If the presented constraint system is satisfiable then at least one transition can be discarded. If the constraint system is unsatisfiable then no termination argument could be found. Thus, the termination analysis would result with the answer `unknown`. This means that the analyzed program is not necessarily non-terminating. Such result only implies that some properties that a ranking function must fulfill are not satisfied. A function

which does not fulfill all properties of a ranking function is called *quasi-ranking function*. This means that this function is a potential ranking function. Depending on which properties the function does not fulfill, one can transform the corresponding transition system and again try to find termination arguments for the resulting equivalent transition system. In order to see which ranking function properties are not fulfilled, one can add three additional constraints to the original constraint. Each of the additional constraints is responsible for a different property. It is important to underline that the goal is to find a function which fulfills the maximum number of properties. Therefore, if a function does not fulfill a property it will be penalized. Thus, one assigns to each of the additional constraints a weight that indicate how badly it is that the corresponding property is not fulfilled.

The new constraints are denoted as $p_\circ$ where $\circ = \{\mathbb{B}, \mathbb{S}, \mathbb{N}\}$ which indicates which of the properties boundedness ($\mathbb{B}$), strict decrease ($\mathbb{S}$) and non-increase ($\mathbb{N}$) is violated. The corresponding weight is denoted as $w_\circ$. The new constraint system has the following form

$$\begin{aligned} &\bigwedge_{l\in\mathcal{L}} \mathbb{I}_l \wedge \bigwedge_{\tau\in\mathcal{T}} (\mathbb{C}_\tau \vee \mathbb{D}_\tau) \wedge \left(\bigwedge_{\tau\in\mathcal{T}'} \mathbb{N}_\tau \vee \bigvee_{\tau\in\mathcal{T}'} \mathbb{D}_\tau \vee p_{\mathbb{N}} \right) \\ \wedge &\bigvee_{\tau\in\mathcal{T}'} \left(((\mathbb{B}_\tau \vee p_{\mathbb{B}}) \wedge (\mathbb{S}_\tau \vee p_{\mathbb{S}})) \vee \mathbb{D}_\tau \right) \wedge [\neg p_{\mathbb{B},w_{\mathbb{B}}}] \wedge [\neg p_{\mathbb{S},w_{\mathbb{S}}}] \wedge [\neg p_{\mathbb{N},w_{\mathbb{N}}}] \end{aligned} \tag{5}$$

In order to find the optimal solution a Max-SMT solver is used. The solver either finds a ranking function or the weights indicate which invariants or quasi-ranking functions are best suitable. The most important property that a quasi-ranking function must fulfill is the non-increase. If this property is not fulfilled, one cannot use the corresponding function as ranking function. However, the resulting invariant map could still help finding an another potential ranking function. Therefore, this property should get the largest weight. In contrast to this ranking functions have the weight 0 [7].

In order to illustrate how this approach works, termination analysis for the program 1 is presented. The formula yielded by the Max-SMT solver is denoted as R. As already explained, at the beginning the Max-SMT solver can find the invariant $y \geq 1$ and the ranking function z for τ_1. Thus, R satisfies the formula above. This means, that one can remove the transition τ_1 and τ_2 can strengthen with the invariant $y \geq 1$ (for τ_0 it is redundant). Thus, for τ_2' holds $\rho_{t_2'} := y \geq 1,\ y \geq z,\ x' = x,\ y' = x + y,\ z' = z$. The resulting transition system is shown in Fig. 3 (b). Not, the approach must be applied on the new transition system. Here, the Max-SMT solver cannot find a ranking function, but a quasi-ranking function x for τ_0. This quasi-ranking function is strict decreasing (ρ_{τ_0} contains the constraint $x' = x - 1$) and it is non-increasing (τ_2' does not increase the value of x). However, x is not bounded. In this case one can split τ_0 into two new transitions $\tau_{0.1}$ and $\tau_{0.2}$. Due to the fact that x is non-increasing, its value can be positive and later negative. Therefore, transition $\tau_{0.1}$ can be strengthen with the invariant $x \geq 0$. As this invariant bounds x for this transition, $\tau_{0.1}$ can be removed from the transition system. For the other transition one can add the invariant $x < 0$. Thus, for $\tau_{0.2}$ holds $\rho_{\tau_{0.2}} := x < 0,\ y \geq 1,\ x' = x - 1,\ y' = y,\ z' = z$. The resulting transition system is shown in Fig. 3 (c). Again the approach must be applied on the new transition system. Adding the invariant $x < 0$ to the transition relation $\tau_{0.2}$ implies that this invariant holds in l_1. Now, Max-SMT solver considers τ_2'. The function y is bounded as $\rho_{\tau_2'}$ contains the constraint $y \geq 1$. Now, consider the constraint $y' = x + y$. Due to the invariant $x < 0$, y is also strict decreasing. Nevertheless, $\tau_{0.2}$ contains the constraint $x' = x - 1$. This means that x is never increased and so is y never increased. Thus, y is a ranking function for τ_2'. Therefore, τ_2' can be removed from the transition system. The resulting transition

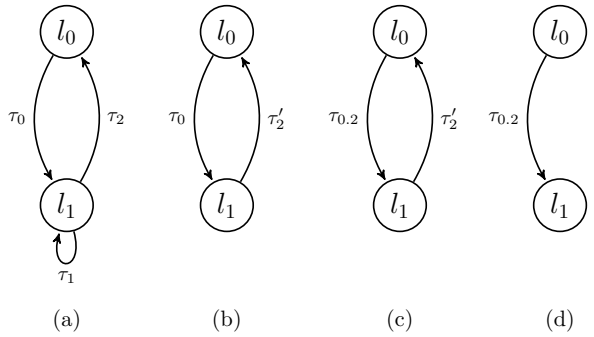

Figure 3: Program and its transition systems.

system is shown in Fig. 3 (d). The new transition system is an acyclic graph. An acyclic graph does not contain infinite computations and $\tau_{0.2}$ is disabled. Thus, the program is terminating [7].

This example showed how the Max-SMT approach can be used to prove termination of a program. It showed that when Max-SMT yields a ranking function or disability argument, the corresponding transition can be removed from the transition system. Otherwise, if Max-SMT found only a quasi-ranking function and an invariant then the transition system is transformed in a different way. The example showed that in the case where the resulting formula R is non-increasing and strictly decreasing but not bounded, one can split the corresponding transition into two transitions and add to the one transition the constraint $R \geq 0$ and to the other $R < 0$. However, the example did not show all possibilities this approach offers. Consider the transition relation $\rho_\tau := x > 0, x' = x$. Max-SMT can find the quasi-ranking function x. It is bounded and not-increasing (there is no other transition increasing it). In this case τ can be split into two new transitions τ_1 and τ_2. As, x is non-increasing it holds that $x \geq x'$ and therefore one can add to τ_1 the invariant $x > x'$ and to τ_2 the invariant $x = x'$. The first transition can be removed because x fulfills now all three properties of a ranking function. For the second transition Max-SMT must look for more invariants [7].

Another interesting case is when the resulting formula only satisfies the non-increase property. This holds for example for this transition relation $\rho_\tau = x' = x$ (there are no other transitions increasing x). Here, one can also split τ into two new transitions τ_1 and τ_2. Since x is not bounded, it can be either positive ($x \geq 0$) or it can be negative ($x < 0$). Thus, to one of the two new transitions one can add the invariant $x \geq 0$ and to the other $R < 0$. As in both cases it is not sufficient to add only one invariant, Max-SMT must look for other invariants [7].

One has to pay attention to the fact that Max-SMT should not be allowed to add redundant invariants and ranking functions. A redundant invariant (ranking function) is an invariant (ranking function) that can be implied from the previously computed invariants (ranking functions). This could cause the problem that the Max-SMT approach would never terminate, as it would always introduce redundant termination arguments [7].

4 Conclusion

In this approach Max-SMT was used to prove termination of imperative programs. The goal was to either find a ranking function or a disability argument or a quasi-ranking function and an invariant. If Max-SMT found a quasi-ranking function and an invariant then this usually enabled splitting the transition such that in the next iteration Max-SMT

could again look for termination arguments again [7].

This approach has been implemented and compared with T2 [5]. The results for both approaches were similar. However, since T2 can solve some benchmarks sets much better than many other approaches, using Max-SMT and quasi-ranking functions seems to be a sensible way for solving termination problems [7].

References

[1] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer Publishing Company, Incorporated, 3rd edition, 2012.

[2] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[3] M. Colón and H. Sipma. Practical methods for proving program termination. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 442–454, London, UK, UK, 2002. Springer-Verlag.

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

[5] B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 47–61, Berlin, Heidelberg, 2013. Springer-Verlag.

[6] A. K. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *International Journal on Software Tools for Technology Transfer*, 15(4):291–303, Aug 2013.

[7] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using max-smt. In *2013 Formal Methods in Computer-Aided Design*, pages 218–225, Oct 2013.

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

[9] M. Shedeed, G. Bahig, M. W. Elkharashi, and M. Chen. Functional design and verification of automotive embedded software: An integrated system verification flow. In *2013 Saudi International Electronics, Communications and Photonics Conference*, pages 1–5, April 2013.

[10] A. Turing. The early british computer conferences. chapter Checking a Large Routine, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[11] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.