On Compositional Safety Verification with Max-SMT

Author: Fabian Böller
Supervision: David Korzeniewski

SS 2017

Abstract

This paper is based on the article 'Compositional Safety Verification with Max-SMT' ([1]) written by Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodrìguez-Carbonell and Albert Rubio. They present an automated compositional program verification technique to verify safety properties for arbitrary integer programs. In this paper we will inspect the inner workings of the algorithm with an example program which is minimal but complex enough to cover major parts of the algorithm.

1 Introduction

1.1 Compositional safety verification

Safety verification is the task to ensure that an assertion at a certain program location is always true regardless of the input. Formally a program is safe for an assertion if every evaluation path of the program leads to the fulfillment of the assertion. While this definition describes a global analysis of the whole program, it is also possible to analyze components of a program and combine the results. The advantage of this compositional method is its scalability, but in many cases there is a loss in precision due to fewer information in the separated analysis.

Since variables in a program can potentially be global and therefore a change of a variable at a location can have a huge impact on other parts of the program, it is not possible to analyze parts of the program completely independent. Therefore the presented algorithm analyzes parts semi-independently, where the analysis of the parts gets combined and eventually reanalyzed.

1.2 Structure

This paper aims at explaining the various distinctions of cases of the algorithm for compositional safety verification with a single example. In the third section of the first chapter this example is presented. The second chapter covers formal definitions of program graphs and invariants and illustrates them with the presented example.

The third chapter introduces the algorithm. At first an overview over the basic analysis of a whole program is given and explained with the example, after that narrowing and the analysis of components are explained. For the analysis of components Max-SMT-Solving, an extension of SMT-Solving, is introduced.

In the fourth chapter a summary of the method is given.

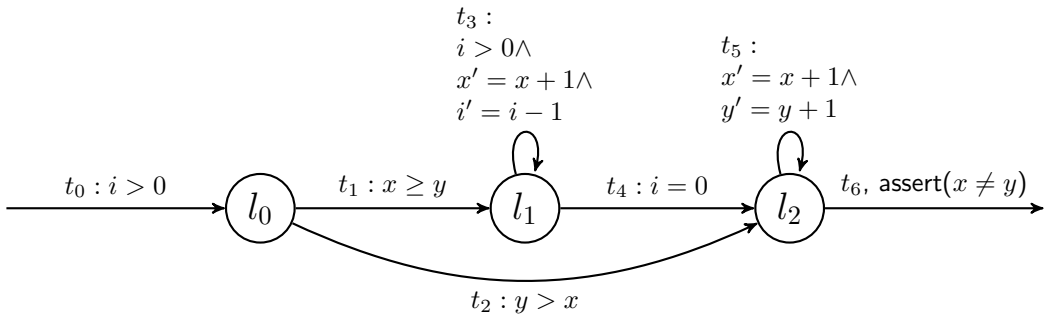

Figure 1: Example program

1.3 Example program

To show the inner workings of the algorithm, we use a single example (1) and explain based on that the different techniques of the algorithm.

It has three input parameters x, y and $i > 0$. The task of the safety verification is to assure, that after every possible evaluation $x \neq y$ holds.

We can easily show this property of the program intentionally: Starting at l_0 we can move on to l_2 if $y > x$. With every iteration with t_5 also $y + 1 > x + 1$ holds and therefore $x \neq y$ holds in the end of the program. Starting at l_0 we can also move to l_1 if $x \geq y$. Since $i > 0$ we must use t_3 at least one time before moving on to l_2. Since t_3 increments the value of x, $x > y$ holds after one usage of t_3. Again t_5 can not change the relation between x and y and therefore $x \neq y$ also holds for this case.

2 Preliminaries

After the informal introduction of the example program in the previous chapter, this chapter introduces formal definitions. In the first part a formal definition of program graphs is presented. In the second part the different types of invariants, which are necessary for the analysis, are introduced.

2.1 Program

We define $\mathcal{L} = \{\ell_0, \ldots, \ell_n\}$ as the set of program locations, where ℓ_0 denotes the unique start location of a program. The example program uses the location set $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$ and ℓ_0 is its start location.

We define $\mathcal{V} = \{v_1, \ldots, v_n\}$ as the set of variables which occur in a program. We define $\mathcal{V}' = \{v_1', \ldots, v_n'\}$ as annotated variables to distinguish between previous and later values. In the example program occur the variables x, y and i. Their annotated versions are x', y' and i'.

An evaluation $\upsilon : \mathcal{V} \to \mathbb{Z}$ assigns each program variable an integer value. A state $s = (\ell, \upsilon)$ represents the values of variables at a certain location $\ell \in \mathcal{L}$. A valid starting state in the example program could be for instance $(\ell_0, \upsilon_{[x \to 1, y \to 1, i \to 1]})$.

We denote with $\mathcal{F}(\mathcal{V})$ the set of formulas consisting of conjunctions of linear inequalities over the variables $\mathcal{V}$. We define a program as a directed graph consisting of the program locations $\mathcal{L}$ and a set of transitions $\mathcal{T} = \{(\ell, \tau, \ell') \mid \ell, \ell' \in \mathcal{L}, \tau \in \mathcal{F}(\mathcal{V} \cup \mathcal{V}')\}$ between those locations. The example program consists of seven transitions. If it contains for a variable v no assignment $v' = \ldots$, we assume the trivial assignment $v' = v$. Therefore the formal representation of t_3 is for instance $(\ell_1, i > 0 \wedge x' = x + 1 \wedge i' = i - 1 \wedge y' = y, \ell_1)$.

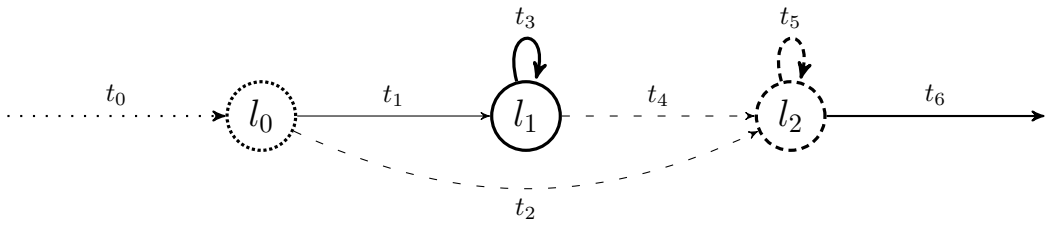

Figure 2: The example program with the SCCs and entry transitions marked

An evaluation step with a transition $t = (\ell, \tau, \ell') \in \mathcal{T}$ leads a program state (ℓ, υ) to another program state (ℓ', υ') if and only if $\upsilon \models \tau$ and $\upsilon' \models \tau$. We then write $(\ell, \upsilon) \to_t (\ell', \upsilon')$. For instance $(\ell_1, \upsilon_{[i\to 0, x\to 1, y\to 2]}) \to_{t_4} (\ell_2, \upsilon_{[i\to 0, x\to 1, y\to 2]})$ is a valid evaluation step in the program, while $(\ell_1, \upsilon_{[i\to 1, x\to 1, y\to 2]}) \to_{t_4} (\ell_2, \upsilon_{[i\to 1, x\to 1, y\to 2]})$ is not a valid evaluation step, since $i = 0$ does not hold for both evaluations.

We define a program component $\mathcal{C} \subseteq \mathcal{T}$ as a strongly connected component (SCC) of a program. We denote with $\mathcal{E}_\mathcal{C}$ its entry transition set which consists of all transitions $t = (\ell, \tau, \ell')$ such that ℓ' appears in $\mathcal{C}$ but $t \notin \mathcal{C}$.

The example program contains three SCCs. Every location represents in this case its own SCC. On the one hand there is the trivial SCC $C_0 = \emptyset$ for the location ℓ_0, on the other hand there are the nontrivial SCCs $C_1 = \{t_3\}$ and $C_2 = \{t_5\}$. The entry transitions of those SCCs are $\mathcal{E}_{\mathcal{C}_0} = \{t_0\}$, $\mathcal{E}_{\mathcal{C}_1} = \{t_1\}$ and $\mathcal{E}_{\mathcal{C}_2} = \{t_2, t_4\}$. Figure 2 shows the three SCCs and their entry transitions.

We define an assertion as a pair (t, φ) where $t \in T$ and $\varphi \in \mathcal{F}(\mathcal{V})$. We say that a program is safe for an assertion if and only if $\forall(\ell_0, \upsilon_0) : (\ell_0, \upsilon_0) \to_P^* \circ \to_t (\ell, \upsilon) \Rightarrow \upsilon \models \varphi$. We say that a program is conditional safe for an assertion if from a state where a precondition holds every path to an assertion also leads to the fulfillment of the assertion. The example program contains one assertion $(t_6, x \neq y)$. It is safe for this assertion, if from each run from l_0, where the values of x and y are arbitrary integers and the value of i is a positive integer, at t_6 the condition $x \neq y$ always holds. The SCC at l_2 is conditional safe for an assertion, if there exists a precondition (for instance $x \neq y$), such that the assertion holds for every run from l_0 which does satisfy the precondition directly before entering the SCC. The goal is to show that the program is safe for the assertion $(t_6, x \neq y)$. The algorithm uses the conditional safety property for this purpose.

2.2 Invariants

An invariant is defined as an assignment $\mathcal{I} : \mathcal{L} \to \mathcal{F}(\mathcal{V})$ if and only if for all reachable states (ℓ, υ) it holds that $\upsilon \models \mathcal{I}(\ell)$. For instance a function with $\mathcal{I}(\ell) = i \geq 0$ for each location $\ell \in \mathcal{L}$ is an invariant of the example program.

An invariant is inductive if and only if additionally it holds that $\top \models \mathcal{I}(\ell_0)$ and for all $(\ell, \tau, \ell') \in \mathcal{P}$ it holds that $\mathcal{I}(\ell) \wedge \tau \models \mathcal{I}(\ell')'$. We call the first initiation condition and the latter consecution condition. $\mathcal{I}$ with $\mathcal{I}(\ell_0) = i > 0$, $\mathcal{I}(\ell_1) = x > y \vee i > 0$ and $\mathcal{I}(\ell_2) = x \neq y$ is a valid inductive invariant of the example program.

An invariant is an conditional inductive invariant if and only if for all $(\ell, \upsilon) \to_\mathcal{P} (\ell', \upsilon')$ it holds that $\upsilon \models \mathcal{Q}(\ell) \Rightarrow \upsilon' \models \mathcal{Q}(\ell')$. In contrast to inductive invariants those conditional inductive invariants does not hold from the program start on, but from a specific state on. We abbreviate the term conditional inductive invariant with CII from this point on.

3 The Algorithm

In the previous chapter we covered the formal basics of a program graph and invariants and introduced an example graph, which we will inspect in this chapter.

The algorithm uses a bottom-up approach. It analyzes each component (SCC) of a program and combines the results. In a first step we will abstract from the analysis of the components, treat them as given and introduce the combination of the analysis to give an overview first. The analyzer of the components is called CondSafe, while the overall combination is called CheckSafe.

3.1 CheckSafe

Figure 3 shows a sequence diagram with the execution of CheckSafe on the example program. In this section we will use this example execution to explain the idea of the overall algorithm.

The given program consists of the three SCCs C_0, C_1 and C_2. The SCC C_2 has an exit transition t_6 which is annotated with an assertion $x \neq y$. The overall algorithm first tries to find a CII which holds directly before C_2 and which satisfies the assertion condition. This is displayed as CondSafe($x \neq y$) in the sequence diagram. The only transition in C_2 is t_5. If t_5 occurs in an evaluation, it has no effect on the equality of x and y, since it increments both x and y with the same constant. Therefore we can conclude that if we can show that prior to C_2 for every evaluation $x \neq y$ holds, it will also hold after C_2.

Unfortunately CondSafe is only able to find either the CII $x > y$ or $y > x$. This does not affect correctness since both CIIs imply $x \neq y$, but it affects completeness. Therefore the authors of the original paper introduced a technique they called narrowing. The idea behind narrowing is to first try to prove safety with the CII CondSafe returns. If this fails CheckSafe is then able to backtrack, filter evaluations and force CondSafe to return a different CII.

We assume the analyzer finds the CII $x > y$ first. The algorithm will now try to recursively find further CIIs such that $x > y$ holds after each component occurring directly before C_2. In the sequence diagram those are the calls CheckSafe($t_2, x > y$) and CheckSafe($t_4, x > y$).

The entry transitions of C_2 are t_2 and t_4. The SCC that has t_2 as exit transition is $C_0 = \emptyset$, the SCC that has t_4 as exit transition is $C_1 = \{t_3\}$. We need to find a CII for both SCCs, otherwise narrowing is necessary. Since $y > x$ has to be satisfied for t_2 to occur in an evaluation, both $y > x$ and $x > y$ need to be satisfied by the same evaluation step. Those conditions contradict each other, therefore there is no evaluation satisfying the postcondition and no precondition can be found to imply the postcondition. This has the effect of CondSafe($x > y$) to be None in the sequence diagram.

The SCC before t_4 is C_1. We need to find a CII that ensures that after C_1 it holds that $x > y$. If we run CondSafe($x > y$) for this SCC it could return $i > 0 \wedge x \geq y$ as precondition for t_1, because it ensures that t_3 is run at least once and therefore implies that $x > y$ holds at t_4.

The literals of both preconditions now get propagated independently to C_0. This is displayed by CheckSafe($t_1, i > 0$) and CheckSafe($t_1, x \geq y$) in the sequence diagram. $x \geq y$ is directly implied by the condition of t_1 and therefore this part of the CII is safe. For the safety prove of $i > 0$ we need to inspect C_0. Since C_0 does not contain any transition, CondSafe($i > 0$) for the SCC will detect that the postcondition has already to be met at the entry transitions, therefore the preconditions equal the postconditions. Because

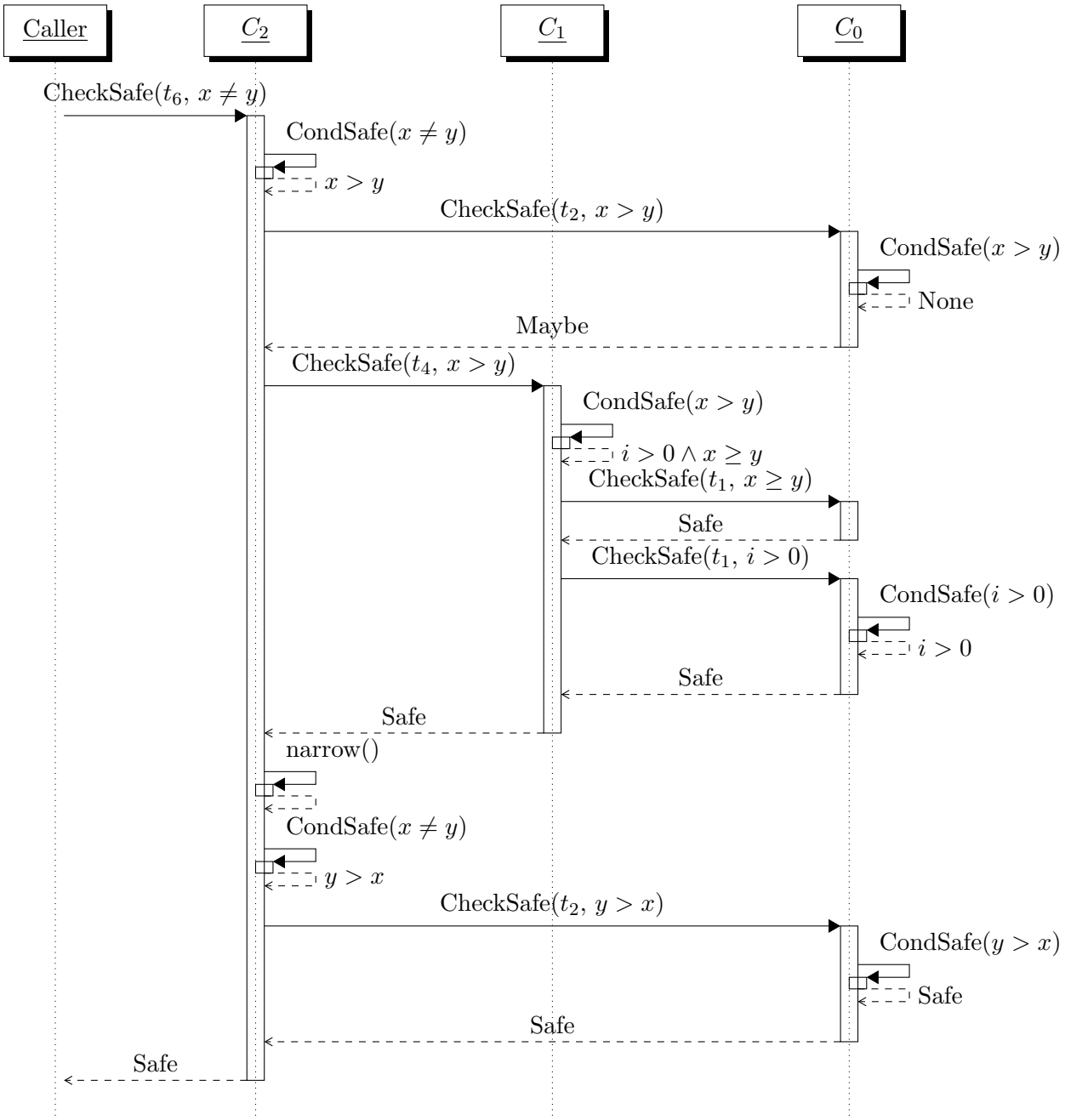

Figure 3: Sequence diagram for the example program

the only entry transition t_0 of C_0 implies $i > 0$, CondSafe proves the SCC as safe and propagates this result back to C_2.

At this point CheckSafe knows that $x > y$ holds after every occurrence of t_4 for every evaluation, but it does not hold for t_2. Therefore we need to narrow the program and find the CII $y > x$ instead. Since we already found t_4 to be safe, we call CheckSafe($t_2, y > x$) for C_2. For the exit transition t_2 of C_0 the condition of the transition now implies the postcondition $y > x$. Therefore this path is also proved safe.

Algorithm 1 CheckSafe

```
1: Input: The program, an SCC, the entry transitions of the SCC, an exit transition with
   an assertion
2: if the exit transition already implies the assertion then
3:     return Safe
4: else if the exit transition is an initial transition then
5:     return Maybe
6: Call CondSafe for the SCC with the given assertion
7: if no CII could be found then
8:     return Maybe
9: for all entry SCCs do
10:    for all literals of the according condition of the CII do
11:        Call CheckSafe for the entry SCC and with the literal as assertion
12: if all calls returned Safe then
13:    return Safe
14: return the result of a call to CheckSafe with a narrowed version
```

3.2 Narrowing

Until this point we abstracted from the inner workings of narrowing. The idea of narrowing is to manipulate the original program, such that the algorithm CondSafe is able to find a different CII to show safety for not yet proved entries. In the example program it would be sufficient to show that the post condition $x > y$ holds after each evaluation step before the location l_2. But this can not be shown for one entry transition, therefore we try to use narrowing.

Algorithm 2 Narrowing

```
1: for all entry transitions do
2:     for all literals of the CII do
3:         if literal could not be proved safe for this transition then
4:             Add a conjunct with the negated literal to the transition
5: for all transitions of the SCC do
6:     Add a conjunct with the negated CII at the start location to the transition
7:     Add a conjunct with the negated CII at the end location to the transition
```

We exclude those evaluations we could not prove safe by adding the negated literals of the CII to those entry transitions that could not be proved safe for the literal. In the example program we add to t_2 the additional constraint $\neg(x > y)$ which is equivalent to $x \leq y$. This way we guarantee that no evaluation with $x > y$ at the entry transitions passes to C_2 and therefore they got excluded from the program.

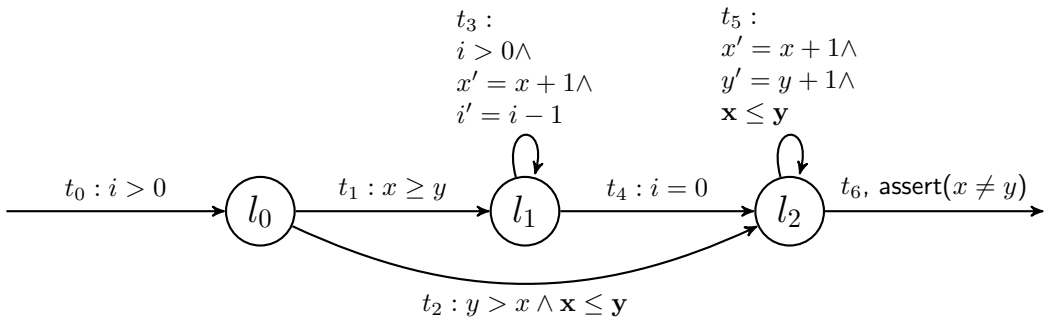

Figure 4: Narrowed example program

Additionally we can modify the transitions inside an SCC to exclude further evaluations, because $x > y$ is either true at the start location or at the end location of the transition.

After narrowing we obtain the program graph presented in figure 4. The added conditions are bold.

The addition of narrowing has the effect of adding an additional layer of backtracking to the main algorithm. Without narrowing CheckSafe already uses a tree-like exploration from the exit state on to the initial states. Narrowing adds even more possible calls to CheckSafe at each step of the algorithm.

3.3 CondSafe

Another procedure the algorithm CheckSafe uses is the algorithm CondSafe. It analyzes SCCs of the program and tries to find a CII such that a post condition is implied.

For performance reasons it only takes a post condition which consists of a single clause. This is not a real restriction, because each conjunction of conditions can be split into single transitions and be analyzed independently.

The aim is to find an invariant that consists of as less conjunctions of clauses as possible. Therefore the algorithm uses an iterative approach. In a first step it tries to find an invariant consisting of a single clause. If this is not possible, it expands its search spectrum and searches for clauses consisting of two clauses. The algorithm terminates if either an invariant with a specific count of clauses is found or a certain constant is reached.

Algorithm 3 CondSafe

```
1: Input: An SCC, the entry transitions of the SCC, an exit transition with an assertion
2: k ← 1
3: repeat
4:     Construct a formula 𝔽_k for the SCC and the assertion
5:     Call the Max-SMT-Solver with 𝔽_k
6:     if it returned a solution then
7:         return an invariant assigning each location the according condition from 𝔽_k
8:     k ← k + 1
9: until k > MAX_CONJUNCTS
10: return None
```

To find a CII with a certain size of clauses the algorithm uses a Max-SMT-Solver, which will be explained in the next section.

3.3.1 Max-SMT

Let $\mathcal{P}$ be a fixed set of propositional variables. We call p and $\neg p$ literals for all $p \in \mathcal{P}$. We define a clause as disjunction of literals $l_1 \vee \cdots \vee l_n$ and a propositional formula as conjunction of clauses $C_1 \wedge \cdots \wedge C_m$.

With a SAT-Solver it is possible to determine if a propositional formula is satisfiable. An SMT-Solver is able to check the satisfiability of a formula with literals from a given background theory. For both solvers there exist extensions called Max-SAT and Max-SMT. Those extensions provide the possibility to declare constraints as soft with a weight. A soft constraints does not have to be satisfied, but the solver will try to maximize the sum of the weights of the soft constraints.

Formally a Max-SMT problem is a formula $H_1 \wedge \cdots \wedge H_n \wedge [S_1, \omega_1] \wedge \cdots \wedge [S_m, \omega_m]$, where H_i denote the hard clauses, which also occur in an SMT problem, and $[S_i, \omega_i]$ denote the soft clauses and their weight.

To find CIIs for program parts we create hard constraints for the consecution and safety conditions and soft constraints for the initiation conditions. While the consecution conditions ensure that the invariant is inductive and the safety conditions ensure that the assertion is implied, the initiation conditions do not need to be fulfilled and thus the resulting invariant might depend on additional preconditions.

In the definition of the condition we use for each location in the SCC the templates $I_{\ell,k}$ and $I_{\ell,j,k}$, where $I_{\ell,k}$ is a conjunction of k templates $I_{\ell,j,k}$.

$$I_{\ell,k}(\mathcal{V}) \equiv \bigwedge_{1 \leq j \leq k} I_{\ell,j,k}(\mathcal{V}) \tag{1}$$

$$I_{\ell,j,k}(\mathcal{V}) \equiv i_{\ell,j} + \sum_{v \in \mathcal{V}} i_{\ell,j,v} * v \leq 0 \tag{2}$$

For instance the template for the SCC $\mathcal{C}_2 = \{t_5\}$ and $k = 1$ is

$$I_{\ell_2,1}(\{x, y, i\}) \equiv i_{\ell_2,1} + i_{\ell_2,1,x} * x + i_{\ell_2,1,y} * y + i_{\ell_2,1,i} * i \leq 0$$

We write $I'_{\ell,k}$ instead of $I_{\ell,k}$ if we use $v' \in \mathcal{V}'$ instead of $v \in \mathcal{V}$.

With those templates we define the initiation condition for a transition $t = (\ell, \tau, \ell') \in \mathcal{E}_\mathcal{C}$ and numbers $j, k \in \mathbb{N}$ with $1 \leq j \leq k$ as $\mathbb{I}_{t,j,k}$. For each transition $t = (\ell, \tau, \ell') \in \mathcal{C}$ and number $k \in \mathbb{N}$ we define the consecution condition as $\mathbb{C}_{t,k}$. The safety condition we define for an exit transition $t_{\text{exit}} = (\tilde{\ell}_{\text{exit}}, \tau_{\text{exit}}, \ell_{\text{exit}})$ as $\mathbb{S}_k$.

$$\mathbb{I}_{t,j,k} \equiv \tau \Rightarrow I'_{\ell',j,k} \tag{3}$$

$$\mathbb{C}_{t,k} \equiv I_{\ell,k} \wedge \tau \Rightarrow I'_{\ell',k} \tag{4}$$

$$\mathbb{S}_k \equiv I_{\tilde{\ell}_{\text{exit}},k} \wedge \tau_{\text{exit}} \Rightarrow \varphi' \tag{5}$$

For instance for the SCC $\mathcal{C}_2$ and $k = 1$ the two initiation conditions are $\mathbb{I}_{t_2,1,1}$ and $\mathbb{I}_{t_4,1,1}$. The single consecution condition is $\mathbb{C}_{t_5,1}$. The single safety condition is $\mathbb{S}_1$.

$$\mathbb{I}_{t_2,1,1} \equiv y > x \wedge i' = i \wedge x' = x \wedge y' = y \Rightarrow I'_{\ell_2,1,1} \tag{6}$$

$$\mathbb{I}_{t_4,1,1} \equiv i = 0 \wedge i' = i \wedge x' = x \wedge y' = y \Rightarrow I'_{\ell_2,1,1} \tag{7}$$

$$\mathbb{C}_{t_5,1} \equiv I_{\ell_2,1} \wedge x' = x + 1 \wedge y' = y + 1 \wedge i' = i \Rightarrow I'_{\ell_2,1} \tag{8}$$

$$\mathbb{S}_1 \equiv I_{\ell_2,1} \wedge i' = i \wedge x' = x \wedge y' = y \Rightarrow x' \neq y' \tag{9}$$

Those templates for the three different conditions we can put together in a Max-SMT problem:

$$\mathbb{F}_k \equiv \bigwedge_{t\in\mathcal{C}} \mathbb{C}_{t,k} \wedge \mathbb{S}_k \wedge \bigwedge_{t\in\mathcal{E}_\mathcal{C}, 1\leq j\leq k} (\mathbb{I}_{t,j,k} \vee \neg p_{\mathbb{I}_{t,j,k}}) \wedge \bigwedge_{t\in\mathcal{E}_\mathcal{C}, 1\leq j\leq k} [p_{\mathbb{I}_{t,j,k}}, \omega_\mathbb{I}]$$

The propositional variables are necessary, because the Max-SMT-Solver is only able to maximize variables and no whole conditions. With the used formula we achieve that if $p_{\mathbb{I}_{t,j,k}}$ is decided to be true, the initiation condition $\mathbb{I}_{t,j,k}$ also needs to be satisfied. If the Max-SMT-Solver could maximize expressions, it would be possible to write

$$\mathbb{F}_k \equiv \bigwedge_{t\in\mathcal{C}} \mathbb{C}_{t,k} \wedge \mathbb{S}_k \wedge \bigwedge_{t\in\mathcal{E}_\mathcal{C}, 1\leq j\leq k} [\mathbb{I}_{t,j,k}, \omega_\mathbb{I}]$$

For the example the actual formula is equal to:

$$\mathbb{F}_1 \equiv \mathbb{C}_{t_5,1} \wedge \mathbb{S}_1 \wedge ((\mathbb{I}_{t_2,1,1} \vee \neg p_{\mathbb{I}_{t_2,1,1}}) \wedge (\mathbb{I}_{t_4,1,1} \vee \neg p_{\mathbb{I}_{t_4,1,1}})) \wedge ([p_{\mathbb{I}_{t_2,1,1}}, \omega_\mathbb{I}] \wedge [p_{\mathbb{I}_{t_4,1,1}}, \omega_\mathbb{I}])$$

If we assume the Max-SMT-Solver returned $y > x$ as maximally satisfying condition for $\mathbb{F}_1$, we can verify if this is a model. The condition $y > x$ relates to the template instantiated with $I_{\ell,1}(\{x, y, i\}) \equiv 1 + (-1) * y + 1 * x + 0 * i \leq 0$. Inserted into the initiation, safety and consecution conditions we get:

$$\mathbb{I}_{t_2,1,1} \equiv i = 0 \wedge i' = i \wedge x' = x \wedge y' = y \Rightarrow y' > x' \tag{10}$$
$$\mathbb{I}_{t_4,1,1} \equiv y > x \wedge i' = i \wedge x' = x \wedge y' = y \Rightarrow y' > x' \tag{11}$$
$$\mathbb{C}_{t_5,1} \equiv y > x \wedge x' = x + 1 \wedge y' = y + 1 \wedge i' = i \Rightarrow y' > x' \tag{12}$$
$$\mathbb{S}_1 \equiv y > x \wedge i' = i \wedge x' = x \wedge y' = y \Rightarrow x' \neq y' \tag{13}$$

The consecution and safety conditions we can simplify such that every model is still a model:

$$\mathbb{C}_{t_5,1} \equiv y > x \Rightarrow y + 1 > x + 1 \tag{14}$$
$$\mathbb{S}_1 \equiv y > x \Rightarrow x \neq y \tag{15}$$

We can directly see that both the consecution and the safety condition are valid. Since the initiation conditions does not need to be satisfied, $y > x$ can potentially be a result of the Max-SMT-Solver, if we assume that it is a maximal solution.

4 Conclusion

In this paper we explained the method of compositional safety verification with an example program. This example program covered many distinctions of cases and therefore enabled us to take a deeper look at the inner workings of the algorithm. We inspected several things left out in the original paper:

1. The exploration of multiple entry SCCs by CheckSafe
2. The effect of narrowing if a path to the SCC is proved safe and another could not be proved safe
3. The behavior of CheckSafe if a CII with multiple conjunctions is found

On the other hand there are cases which could be investigated in subsequent studies. We focused on SCCs with only one transition. This affects the search for CIIs and narrowing. Inspecting multiple transitions in an SCC has for the search for CIIs just the effect of giving further consecution constraints to the Max-SMT-Solver. For narrowing it affects both the added constraints in the SCC as well as the added constraints to the entry transitions, if they lead to different entry locations. Further studies could therefore inspect SCCs with multiple transitions to give a fine-grained explanation regarding narrowing.

References

[1] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodriguez-Carbonell, and A. Rubio. Compositional safety verification with max-smt. In *Proceedings of FMCAD'15*. IEEE – Institute of Electrical and Electronics Engineers, September 2015.