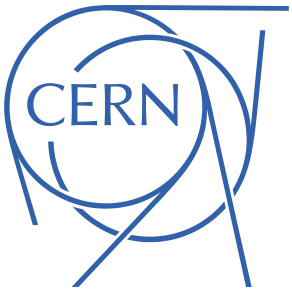
Evaluating compositional verification options for PLCverif

Milán Mondok

Advisor: Borja Fernandez Adiego

September 2021



Abstract

Model checking is a computationally complex task and state space explosion often hinders successful verification. Compositional (also called modular) verification techniques aim to tackle this by decomposing model checking problems into smaller sub-problems that are potentially easier to compute, and reasoning about the original problem through them. The main focus of my summer student project was evaluating the state of the art in compositional verification and examining how they could be applied in PLCverif, a model checking framework for PLC programs developed at CERN.

1 Introduction

This report consists of two parts. In the first part (see Section 2), I present a new experimental CEGAR variant that works with black-box model checkers, which I implemented in the PLCverif framework. In the second part (see Section 3), I briefly introduce the state of the art in compositional verification.

2 Counterexample-guided abstraction refinement with component-abstraction

Counterexample-guided abstraction (CEGAR) refinement is a widely used model checking technique, which aims to tackle state space explosion. The model checking in this case is performed on an abstract model, which is an over-approximation of the original model. This abstract model has a smaller state space, but it contains behaviour that the concrete model doesn't. The level of abstraction is determined by the current precision. If the abstract model is found correct, then the CEGAR loop terminates with a 'correct' result. However, if an abstract counterexample is found, then further analysis is required to determine if it is present in the concrete model as well. If we find that it is present, then we conclude a 'violated' result. However, if the abstract counterexample is found spurious, meaning it isn't present in the concrete model, then the abstraction has to be refined accordingly. For an overview of this algorithm, see Fig. 1.

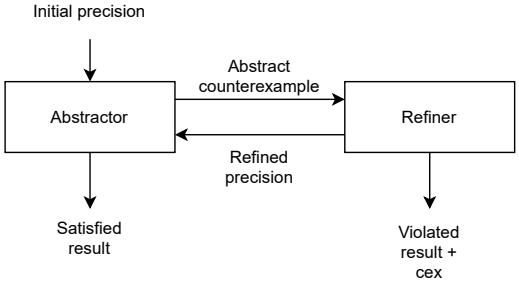


Figure 1: An overview of the CEGAR algorithm

CEGAR can be used with various abstract domains, for example predicate abstraction, explicit-value analysis or zone abstraction. In this report I present a new abstract domain: component abstraction. Component abstraction is special because it doesn't have to be implemented inside a model checker, it is enough to use a model checker as a black-box when implementing it. This special requirement stems from the architecture of PLCverif. PLCverif compiles the PLC sources into an intermediate CFA representation, carries out reductions on it, and then transforms it to the input languages of various model checkers, such as CBMC, NuSMV and Theta. The novel component-abstraction CEGAR algorithm fits into this workflow, equipping model checkers with CEGAR capabilities without the need to modify them (note that Theta supports CEGAR out of the box, so no significant improvement is excepted there).

The CFA metamodel The PLC sources in PLCverif are translated to CFA networks. CFA networks consist of multiple automata and have a main automaton that is invoked when the network is executed. Control-flow automata are labeled directed graphs, where each node corresponds to a location (representing a value of the program-counter), and each directed edge corresponds to a transition. Transitions can either be call transitions or assignment transitions. An assignment transition can have multiple conditions and assignments. A call transition invokes another automaton and can have input and output assignments associated to it. When an automaton is invoked, first the input assignments are carried out, then the control is passed to the initial location of the called automaton. After reaching the

end location of the called automaton, output assignments are carried out and execution continues from the target state of the call transition. Calls cannot be recursive.

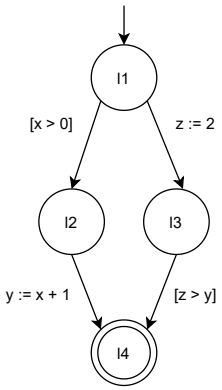


Figure 2: A control flow automaton

Fig. 2 shows an example of a control flow automaton. Locations are denoted with circles and transitions with directed edges. Conditions appear between squared brackets, for example $[x > 0]$, while assignments are denoted with $:=$, for example $y := x + 1$. The initial location ($l1$) has a single incoming empty transition, and the end location has a double outline.

Abstraction In the novel abstract domain, the precision is a set of automata, which are going to appear in the abstract model unmodified (we call them "explicit automata"). The abstract model is constructed by replacing the automata which are not explicit with an over-approximation. An over-approximation of an automaton is constructed by collecting all variables that either the automaton itself or automata that the automaton calls assign to, and assigning nondeterministic values to them. The abstract automaton will only contain two locations and a single transition between them that assigns nondeterministic values to all potentially affected variables. Fig. 3 illustrates what the automaton from Fig. 2 will look like after abstraction.

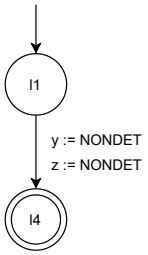


Figure 3: The automaton from Fig. 2 after abstraction

Refinement If the result of the model checking on the abstract model is 'safe', then we can conclude that the concrete model is safe as well (because the abstraction was an

over-approximation). However, if the result is 'unsafe', then we have to concretize the counterexample and potentially refine the abstraction. To do this we have to identify parts of the counterexample that correspond to an abstracted automaton. As these automata were replaced with two locations and a single transition, we simply have to look for two consecutive states in the counterexample, where the first state has the initial location, and the second state has the end location of an abstracted automaton, in our example, $l1$ and $l4$, respectively. To check if this step is possible in the concrete model, we have to carry out a separate model checking step, where we assign the values in $l1$ as initial values and we check whether the values in state $l4$ are reachable in that location in the concrete model. If all such abstracted steps are concretizable, then we conclude that the model is 'unsafe'. If one of them isn't, then we refine the abstraction by marking the corresponding automaton as explicit.

Conclusions and future work I created an initial implementation of the approach described above in PLCverif, and it works correctly on simple models. I tested it on a few larger models as well, but got some incorrect results, which I couldn't debug yet, due to the limited amount of time I had. I also found that the abstraction-method, i.e. assigning nondeterministic values to all assigned variables of the automaton is too aggressive, we usually lose too much information with it and have to refine. This could be improved by introducing optional pre- and postconditions that could be given as comments in the PLC code. They then appear as assumptions in the abstract model and lead to a more precise abstraction (similarly to the approach in 3.3). These conditions of course would also have to be verified, and the result of that could give further insight to the users about the programs that they develop.

3 The state of the art

In this section I briefly present the state-of-the-art approaches in compositional verification, and their relevance to PLCverif.

3.1 Assume-guarantee reasoning

Assume-guarantee reasoning [1] focuses on compositional verification of concurrent programs: replacing a single analysis over the global state space with localized smaller analyses of the individual components. The biggest challenge of the approach is synthesizing appropriate assumptions, as they have to be strong enough prove a desired property, while being weak enough to allow for an efficient analysis.

Papers on assume-guarantee reasoning are usually automata-theoretic. The exact automata-theoretic details are not relevant to PLCverif, so I will try to avoid going into details about them. The systems are defined as regular languages and modeled using finite automata (DFA). Assumptions and properties are also modelled using DFAs, but they have a single error state. A component M violates the property P if the error state is reachable in $M \| P$. The operator $\|$ denotes the composition of two components in such a way, that all their common actions are synchronized, and all others are interleaved.

Assume guarantee approaches typically employ the following assume-guarantee rule:

$$\frac{\{\, A \,\}\ M_1\ \{\, P \,\} \qquad \{\, true \,\}\ M_2\ \{\, A \,\}}{\{\, \text{true} \,\}\ M_1 \| M_2\ \{\, \text{P} \,\}}$$

This is a Hoare rule, which is used for writing formal proofs of systems. The main building blocks of these rules are the so-called Hoare triples, for example $\{ true \}\ M_2\ \{ A \}$ and $\{ A \}\ M_1\ \{ P \}$. If both triples above the line are met, then the triple below the line holds.

In the triple $\{ A \}\ M_1\ \{ P \}$, A is an assumption and P is a property. The rule expresses that if the precondition A is met, then the component M_1 ensures the postcondition P. Remember that A and P are also DFAs. The triple $\{ A \}\ M_1\ \{ P \}$ can be checked by checking the reachability of error states in $A \| M \| coP$, where coP is the complement of P and an error state is a state that is accepting in coP. All of this put together in our case means that if M_2 always ensures the postcondition A, and given A, M_1 ensures P, then the composition of the two systems ensures P given $true$ (i.e. in all cases).

What this means for compositional verification is that in order to show that the composite system fulfills the property P, it is enough to show that M_1 given $true$ fulfills A and M_2 given A fulfills P, which are potentially easier to decide. The hard part however is finding an appropriate assumption A, it has to be strong enough so that $\{ A \}\ M_1\ \{ P \}$ holds, and also permissive enough so that $\{ true \}\ M_2\ \{ A \}$ holds as well.

The construction of the assumption A happens using the L* algorithm, which is an algorithm that is capable of learning an unknown regular language, in our case A. In order to be able to learn the language, L* needs a teacher who is able to correctly answer 2 questions:

- *Membership queries*: decide if a given string is part of the language. In the context of model checking, a string means a single execution, i.e. a sequence of state labels.
- *Equivalence queries*: decide if the language of the candidate DFA given by L* is identical to the language that we want to learn. In our case this will mean deciding if the candidate assumption A is a good assumption. (By checking if $\{ A \}\ M_1\ \{ P \}$ and $\{ true \}\ M_2\ \{ A \}$ hold.)

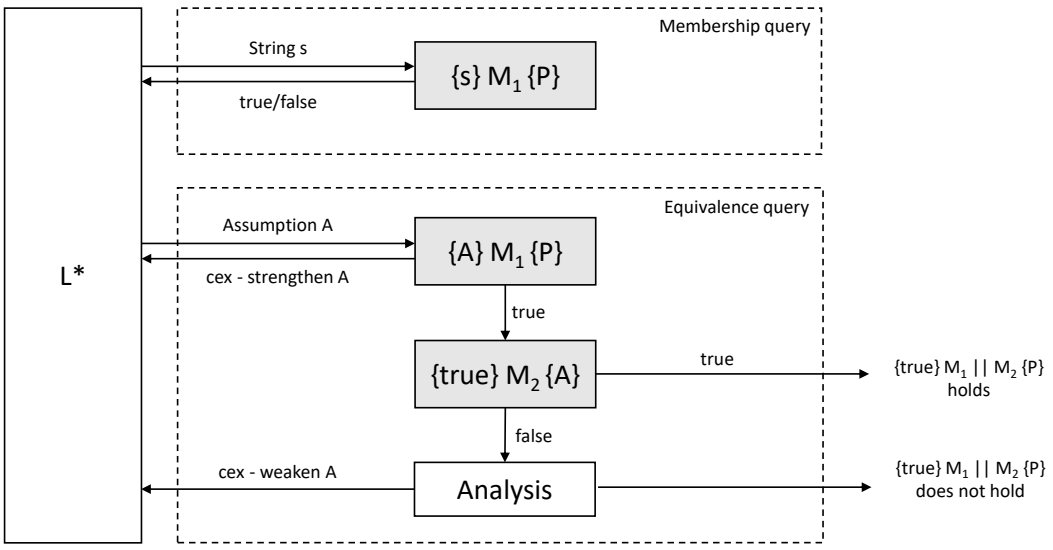


Figure 4: Assume-guarantee model checking

The teacher in this case is going be us carrying out various model checking tasks. Figure

4 shows the process of assume-guarantee reasoning using the L* algorithm and the Hoare rule from above. The L* algorithm repeatedly queries the teacher with membership queries. For each of these membership queries we have to answer if the string s satisfies the triple $\{ s \}\ M_1\ \{ P \}$. (This is because a string s can only be a member of the language of the assumption A, which fulfills $\{ A \}\ M_1\ \{ P \}$, if $\{ s \}\ M_1\ \{ P \}$). Remember that s is a trace in our case, so checking if $\{ s \}\ M_1\ \{ P \}$ can be done by simulating s on $M_1 \| P$ and seeing if an error state is reached. If an error state is reached, then the answer is $false$, else the answer is $true$.

After a certain number of membership queries, the L* algorithm constructs a conjecture, which is a DFA that the algorithm assumes to be equivalent to the language it has to learn, resulting in an equivalence query. In our case the conjecture will be a candidate for the assumption A. To verify this conjecture we first check if it is strong enough, i.e. if $\{ A \}\ M_1\ \{ P \}$ holds. If not, then the A is too weak and a counterexample demonstrating this is returned to L*. If however $\{ A \}\ M_1\ \{ P \}$ is true, then $\{ true \}\ M_2\ \{ A \}$ is checked. If this is true as well, then the model checking is completed, the composite system satisfies the property, i.e. $\{ \text{true} \}\ M_1 \| M_2\ \{ \text{P} \}$ holds. If not then the counterexample is examined in an analysis step to decide if the source of this is that $\{ \text{true} \}\ M_1 \| M_2\ \{ \text{P} \}$ does not hold, or that A is too strong and it has to be weakened.

Relevance for PLCverif The greatest advantage of assume-guarantee reasoning is that it automatically learns the assumptions, meaning that it doesn't require extra input from the users, which makes it easy to use. We cannot apply it in PLCverif however, because it was developed for concurrent systems, where the different components are parallel processes, that either communicate through shared memory or messages, which is fundamentally different from the type of composition that is present in PLC programs, i.e. functions. It is also discouraging that in an evaluation of assume-guarantee reasoning in [2] they found that it often leads to a larger state space than the original model had.

3.2 Function summarization

Function summarization [3] is an incremental verification technique. The core idea of the approach is storing meaningful data about functions during successful verification runs to reuse them during subsequent runs. This approach is implemented in the FunFrog [4] and HiFrog [3] model checkers developed at the University of Lugano. Both FunFrog and HiFrog are bounded model checkers for C programs, that use the CProver infrastructure, similarly to CBMC and ESBMC.

Function summarization is applicable when the model checking task consists of subsequently executable sub-tasks, between which the model remains unchanged. For example checking different assertions on the same C source code. After a successful verification of a sub-task, function summaries are computed and stored in a database for all involved functions. Function summaries are over-approximations of their respective functions and are obtained using interpolation, (Craig interpolation in the case of FunFrog and more advanced techniques in the case of HiFrog).

Subsequent runs can exploit function summaries by replacing functions with their summaries, and thus simplifying the model, resulting in faster execution. Given that the summaries are over-approximations however, they can contain spurious behaviour that isn't present in the concrete functions. To discard spurious errors, a counterexample-guided re-

finement loop is implemented, which replaces summaries with the concrete functions whenever it is necessary.

```
1  #include "math.h"
2  int nondet_int();
3
4  int num()
5  {
6      int s = 0;
7      for(int i=0; i<10; i++)
8      {
9          s = s + abs(nondet_int());
10     }
11     return s;
12 }
13
14 int main()
15 {
16     int a, b;
17     a = num();
18     assert(a >= 0);
19
20     b = num();
21     assert(a + b >= 0);
22 }
```

Figure 5: A simple C program illustration for function summarization.

For an illustration, let's consider the C program in Fig. 5. The second line contains the declaration of the *nondet_int* function, this is a function that the model checker recognizes and handles as a function that returns nondeterministic integers. The function *num* initializes a variable to 0, then adds the absolute values of nondeterministic integers to it, before returning it. In function *main*, we have two assertions, the first one asserts that the return value of *num* will be greater than or 0. The second one asserts that the sum of two return values of the *num* function will be greater than or 0. It is easy to see that in this simple example both assertions will hold.

Let's see how function summarization would work on the simple example in Fig. 5. We will have two runs, one for each assertion, starting with the first. At the start, the summary database is empty, so during the first run, there are no substitutions. The first run concludes that the first assertion holds, this means that we can extract function summaries. Let's assume that the interpolation returned the summary $return\_value >= 0$ for the function *num*, where *return_value* refers to the return value of the function, which we stored in the summary database. As we proceed to the next run where we check the second assertion, we replace the function *num* with the previously obtained summary, hiding all the implementation details. Running the analysis on this simplified model returns that the second assertion holds as well. In this case, the abstract model was detailed enough to prove the second assertion, because we got lucky with the interpolant. This isn't always the case however, as the summary could have easily been weaker, for example $return\_value >= -2$. This is a correct over-approximation of the function as well, and we have no control over what the solver will return. If this were the case, then a second iteration would have been required, meaning function summarization wouldn't have been helpful.

Relevance for PLCverif As PLCverif is already capable of generating C sources for the CBMC model checker, which also uses the CProver infrastructure, I tried running HiFrog and FunFrog on these sources. I wasn't satisfied with these runs, the results seemed inaccurate, for which the most probable explanation is that the currently generated C source code for CFIs contains complex data structures and language elements like "pointers that point to structs that contain pointers that also point to structs". Another hardship in applying function summarization stems from the requirement that the model must remain unchanged between the verification of different assertions. A model with multiple assertions is entirely possible in PLCverif, however in these cases only the PLC-level model is constant. The CFA is usually different for each assertion even though it is generated from the same PLC source, because of the different reductions that are applied to remove anything that is irrelevant to the currently checked assertion. This means that some reductions would have to be turned off.

3.3 Function contracts

The modular verification approach which uses so called "function contracts" is perhaps the simplest of the approaches presented here. Among other tools, it is used in Boogie[5], a modular model checker for the Boogie IVL (intermediate verification language), and Solc-verify[6], a model checker for Solidity smart contracts.

This approach can be used with languages that contain procedures and calls, like the C programming language, Boogie IVL, or the Structured Text language for PLCs. The procedures have to be annotated with pre- and postconditions, this is what we call a "function contract". Preconditions are conditions that must be met before the procedure can be executed. Postconditions are guarantees that a procedure ensures given that its preconditions are met. The verification problem in this case is deciding whether all procedures ensure their postconditions given their preconditions. During modular verification, each procedure is verified independently, where calls to other procedures are replaced with the contract of the called procedure (which is essentially an assumption of the preconditions and an assertion of the postconditions). In order to run a successful verification, appropriate pre- and postconditions are required, which cannot be obtained trivially, and whose formulation often requires domain-specific knowledge. This means that extra input from the developers is required, which can make this approach harder to apply in practice.

For a simple example, let's consider the Boogie IVL program in Figure 6a. The program has two integer variables, x and y, and two procedures, *add* and *main*. Procedure *add* is annotated with two preconditions, $x >= 0$ and $y >= 0$, and a postcondition, $x >= y$. Informally, this reads as "the procedure *add* can only be executed if $x >= 0$ and $y >= 0$, and in return it ensures that after its execution x will be greater than or equal to y". The procedure *main* assigns values to the variables, then calls *add* and makes an assertions. We can see in Figure 6b what this program will be transformed to if we want to modularly verify procedure *main*. The call to the procedure *add* is replaced with the assumption of its preconditions, nondeterministic assignments to all its assigned variables, and the assertion of its postconditions. The modular model checking in this case would consist of two verification steps: first we would check if procedure *add* complies with its contract, and then check if procedure *main* is correct if we replace *add* with its contract.

Relevance for PLCverif Implementing this modular verification approach in a model checking tool is relatively easy, however applying it in practice is hard, because writing good

```
1  var x : int;
2  var y : int;
3
4  procedure add()
5      requires x >= 0;
6      requires y >= 0;
7      ensures  x >= y;
8  {
9      x := x + y;
10 }
11
12 procedure main()
13 {
14     x := 2;
15     y := 3;
16     call add();
17     assert(x == 5);
18 }
```

(a) Boogie program example illustrating function contracts.

```
1  var x : int;
2  var y : int;
3
4  procedure main ()
5  {
6      x := 2;
7      y := 3;
8      assume(x >= 0); // requires
9      assume(y >= 0); // requires
10     havoc x;        // assignment
11     assert(x >= y); // ensures
12     assert(x == 5);
13 }
```

(b) Transformation of the program for modular verification.

Figure 6: A simple Boogie IVL program and its transformation for modular verification.

contracts requires human input and domain knowledge. It could however easily complement other methods, such as the CEGAR approach in Section 2, where the component abstraction could easily be augmented with optional pre- and postconditions, which could make the abstraction more precise without adding a significant performance overhead.

3.4 Horn clauses for multiple instances

A very interesting approach presented in [7] is also about compositional verification of PLC programs, but instead of splitting the problems into smaller parts it focuses on cases where there are many instances of the same function block in a PLC program.

In this paper they encode the program as constrained Horn clauses (CHC) instead of SMT formulas. Constrained Horn clauses can be familiar from logical programming languages like Prolog and Datalog. A CHC is a formula $(p_1 \wedge p_2 \wedge ... \wedge p_k \wedge \varphi) \Rightarrow h$, where p_i are predicates over the variables, ϕ is a logical formula called the constraint, and h called the head is a predicate over the variables as well. This is usually written as $h \leftarrow p_1, p_2, ..., p_k, \varphi$, which can be read as "to show h, show p_1, show p_k, . . ., and show φ" (in Prolog $:-$ is used instead of $\leftarrow$). The satisfiability of CHC formulas (or rather lists of CHC formulas) can be decided using CHC solvers, which are similar to SMT solvers.

For demonstration purposes, let's consider the CFAs in Fig. 7. The *main* CFA starts in its location 0, then y gets assigned 0, after that x gets assigned $y + 2$, and then the other CFA f is called. In the other CFA x gets assigned $x + 1$, then $[x > 0]$ is assumed.

Initial state A CFA can be characterized using CHCs with the following formulas. Let l be a variable that encodes the location of the CFA. Let $S(l, v_1, v_2, ..., v_n)$ be a predicate over the variables that characterizes the (transitively) reachable valuations of the CFA S in

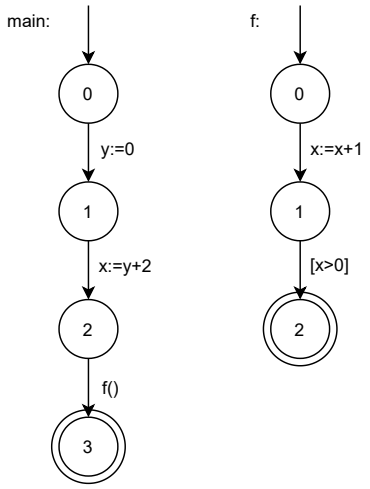


Figure 7: Example CFAs

the location l. We constrain the initial location to 0 with the following clause:

$$S(l, v_1, v_2, ..., v_n) \leftarrow l = 0 \tag{1}$$

For the CFA *main* in our example:

$$main(l, x, y) \leftarrow l = 0 \tag{2}$$

Transitions Transitions of the CFA can be characterized using the following CHC formulas, where primed variables denote the values of the variables in the resulting state, and the action on a transition between the locations l and l' is characterized with the formula $T(l', v_1', v_2', ..., v_n', l, v_1, v_2, ..., v_n)$ (which I will explain below):

$$S(l', v_1', v_2', ..., v_n') \leftarrow S(l, v_1, v_2, ..., v_n), T(v_1', v_2', ..., v_n', v_1, v_2, ..., v_n) \tag{3}$$

Actions: The actions on the transitions are characterized with the following predicates:

- Assignments $v := expr$ with the formula $v' = expr \wedge \bigwedge_{x \in V, x \neq v} x' = x$, where V is the set of variables.
- Assumes $[\varphi]$ with the formula $\varphi \wedge \bigwedge_{v \in V} v' = v$

Putting all of this together, the first two transitions of the *main* CFA are formulated this way:

$$main(l', x', y') \leftarrow main(l, x, y), l' = 1, l = 0, y' = 0, x' = x \tag{4}$$

$$main(l', x', y') \leftarrow main(l, x, y), l' = 2, l = 1, x' = y + 2, y' = y \tag{5}$$

This (equation 5) can be read as "location 2 of the CFA *main* is reachable with a valuation (x', y'), if location 1 is reachable with the valuation (x, y), and the valuations satisfy the formula $x' = y + 2 \wedge y' = y$".

Calls This is where the main contribution of this paper lies. In other CHC based approaches the calls of the CFA would simply be inlined resulting in a single CFA. What this paper proposes instead is a solution in which whenever a CFA is called from another CFA, and the valuation of the calling CFA is the same as in a previous call, it will be recognized and the previous results will be reused.

To achieve this, the predicates $S(l, v_1, ..., v_n)$ are extended to $S(l, v_1, ..., v_n, v_1*, ..., v_n*)$ to capture valuations that are transitively reachable when the CFA S is entered with the valuation $v_1*, ..., v_n*$. The valuation $v_1*, ..., v_n*$ is essentially a "snapshot of the memory" of the calling automaton at the start of the call.

This way, the transition formula from equation 3 changes to the following:

$$\begin{aligned} S(l', v_1', ..., v_n', v_1*, ..., v_n*) \leftarrow S(l, v_1, ..., v_n, v_1*, ..., v_n*), \\ T(v_1', v_2', ..., v_n', v_1, v_2, ..., v_n) \end{aligned} \tag{6}$$

The values $v_1*, ..., v_n*$ are carried over in a transition: they simply encode the valuation that the CFA was entered with (i.e. the values of the variables in the source state of the call transition).

For example the transitions of the called CFA f are characterized the following way:

$$f(l', x', y', x*, y*) \leftarrow f(l, x, y, x*, y*), l' = 1, l = 0, x' = x + 1, y' = y \tag{7}$$

$$f(l', x', y', x*, y*) \leftarrow f(l, x, y, x*, y*), l' = 2, l = 1, x > 0, x' = x, y' = y \tag{8}$$

The call from CFA *main* to CFA f is encoded the following way (this expresses that location 3 of the *main* CFA is reachable with a given valuation if the location 2 of the CFA f is reachable with the same valuation and valuation that f was entered with is also reachable in location 2 of the caller):

$$main(l'', x', y') \leftarrow main(l, x, y), f(l', x', y', x, y), l'' = 3, l' = 2, l = 2 \tag{9}$$

The called CFA is instantiated the following way (this essentially expresses that if the location 2 of the CFA *main* is reachable with the valuation (x, y), then the location 0 of the called CFA f is also reachable with the same valuation):

$$f(l', x, y, x, y) \leftarrow main(l, x, y), l' = 0, l = 2 \tag{10}$$

Reachability After constructing these formulas, to decide the reachability of a given valuation $(v_1, ..., v_n)$ in a given location l, in the CFA S, one simply has to query a CHC solver with the formula $S(l, v_1, ..., v_n)$. For example $main(l = 2, x = 2, y = 0)$ would be satisfiable, but $main(l = 1, x = 0, y = 1)$ would not.

Relevance for PLCverif This approach is implemented in the ARCADE.PLC platform, which is a model checking framework for PLC programs that supports the Siemens S7 and the IEC 61131 ST and IL languages. The platform is similar to PLCVERIF in that it is also eclipse-based and that it also transforms PLC programs to CFAs. The main difference is that while PLCVERIF transforms the CFAs to the inputs of other standalone model checkers, ARCADE.PLC carries out the model checking itself, using the Z3 solver. Based on what I found, the platform is not open source, the GUI version of the tool isn't publicly available anywhere, but a link to the CLI version can be found in the paper.

References

[1] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu. *Compositional Reasoning*, pages 345–383. Springer International Publishing, Cham, 2018.

[2] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2), May 2008.

[3] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. Hifrog: Smt-based function summarization for software verification. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 207–213, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[4] O. Sery, Grigory Fedyukovich, and Natasha Sharygina. Funfrog: Bounded model checking with interpolation-based function summarization. In *Tenth International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Thiruvananthapuram, India, 2012. To appear at the conference. The pdf-version is not final.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. *Verified Software. Theories, Tools, and Experiments*, page 161–179, 2020.

[7] Dimitri Bohlender and Stefan Kowalewski. Leveraging Horn clause solving for compositional verification of PLC software. *Discrete Event Dyn. Syst.*, 30(1):1–24, 2020.