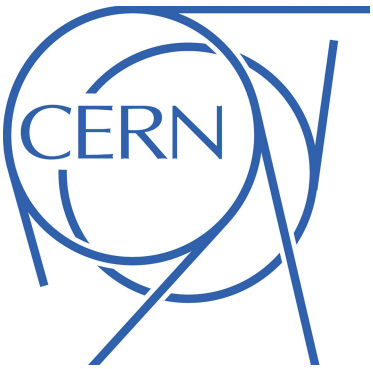
Counterexample analysis of formal verification methods

CERN Summer Student Programme 2021 – Final report

Mihály Dobos-Kovács

Budapest University of Technology and Economics, Hungary

Supervisor: Jean-Charles Tournier



August 27, 2021

1. Introduction

Nowadays, different kinds of software solutions become part of our lives more and more. While 30 years ago home PC's were rare, and Tim Berners-Lee's WorldWideWeb was merely a couple of months old, in the present there is a smart phone in the pocket of every passerby. Moreover, with the rise of 5G and IoT, more and more "smart" gadgets begin to influence our daily lives. In parallel to this process, the industry began to heavily rely on computers as well, as nearly every corner of a production line is driven by computers in a modern factory.

A special category of software systems is the safety-critical systems. Usually a fault in a safety-critical system can lead to immersive financial loss, catastrophic environmental effect, or it can even cost human lives. Typical examples of such safety-critical systems are found in airplanes, nuclear power-plants, or even in CERN's LHC.

To ensure the safety of these systems, they are rigorously tested in an isolated, safe environment according to the strict standards regulating the development and operation of said systems. However, accidents can still happen, and a textbook example is the failed first test-fight of European Space Agency's Ariane 5 rocket in 1996. Although every component was operating correctly, a failed conversion of a 64-bit floating point value to a 16-bit signed integer value caused the navigation system of the rocket to lose track the rocket's position, and the rocket self destructed costing more than $300m.

A completely different approach of finding faults in systems is formal verification, which takes the mathematical model of the system, and mathematically proves some property on it – or provides a counterexample that demonstrates why the property does not hold. It is possible to find extremely rare bugs using formal verification, that were missed in testing, because they only happen in every ten-thousand years, but these methods have their own challenges.

One challenge, is that the counterexample given by a verification method is usually rather long and complex, and it takes a huge amount of time to analyze it, and find the cause of the issue. The goal of my project was to find and develop methods that are capable of automatically analyze the counterexamples, and point the developers in the right directions. I implemented the algorithms using CERN's PLCverif software, that is capable of the formal verification of PLC code.

2. Counterexample analysis

Technically speaking, formal verification is a formal method that takes a formal model and a formal requirement, and gives a mathematical proof whether the requirement holds on the model. If it does not hold, it produces a counterexample that reproduces the conditions of the failing requirement.

When formal verification is applied on PLC code, the formal model will be the Control Flow Automaton (CFA) [1], which describes a computation graph of the PLC code consisting of locations and edges with the statements on them. Moreover, some of the locations are marked as error locations, and the requirement is that none of the error locations is reachable. These error locations

are generated automatically from assertions in the code. Without much detail, in the context of a CFA, the counterexample is the list of inputs to give to the PLC code, which will lead to (one of) the assertion(s) to fail.

```
1:  FUNCTION “find_max”
2:   VAR_INPUT
3:     x1: INT;
4:     x2: INT;
5:     x3: INT;
6:   END_VAR
7:
8:   VAR_OUTPUT
9:     max: INT;
10:  END_VAR
11:
12:  max := x1;
13:  IF max <= x2 THEN
14:    max = x2;
15:  END_IF;
16:  IF max >= x3 THEN
17:    max = x3;
18:  END_IF;
19:
20:  //#ASSERT max >= x2;
21: END_FUNCTION
```

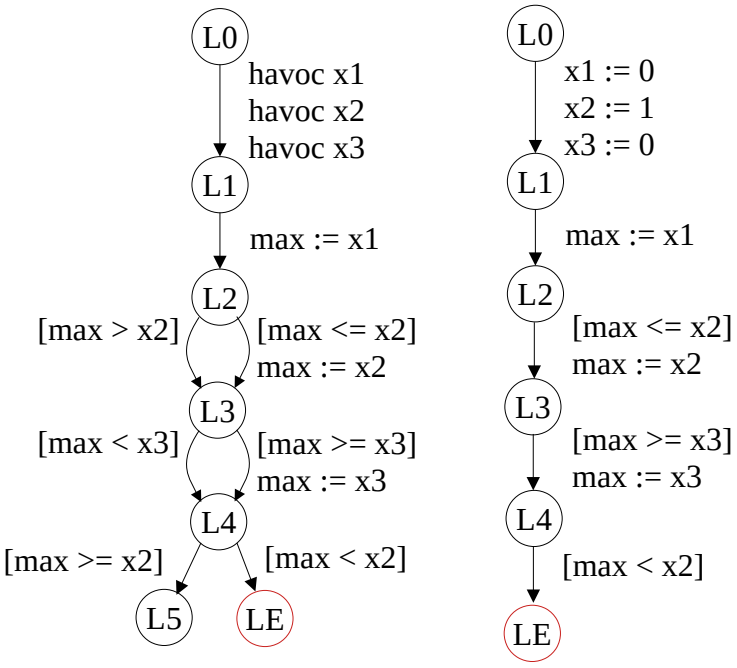


Figure 1: An example of a PLC code and its CFA model with a counterexample

In the figure above on the left side a PLC function can be seen that receives three inputs and returns the greatest of them. However, there is an error in line 16 in the comparison operator, which leads to the assertion in line 20 to fail sometimes. A counterexample for which the assertion fails is 0-1-0. In the middle of the figure above is the CFA model of the PLC code. It can be seen that input variables are mapped to havocs, the conditional statements to guards, and the assignments to assignments. It can also be observed how the assertion in line 20 is mapped to the error location (LE). On the right side is the counterexample, with the input variables having a concrete value, and only containing the statements that are executed for those input values.

Given the counterexample and the CFA, my task was to find an algorithm that is capable to point the developer to line 16, where the error resides. As part of the project I took an extensive look at the literature to find algorithms capable of achieving something similar to this.

My first search yielded results that are capable of analyzing the counterexamples of one particular domain. I found algorithms that are capable of this for function block diagrams [2][3]. These were able to identify the location of the issue inside the function block diagram given a counterexample. However, I did not find such methods designed specifically for PLC programs (or the UNICOS framework).

Next, I focused on methods that are independent of the domain. I found several kinds of algorithms that promised to locate the fault in the code. There were testing based methods [4][5]. These methods required the presence of failing and passing tests, and used statistical approaches to

determine the place of the error. Next, there were model checking based methods [6][7], which try to search additional, passing traces by applying additional constraints to the code. Then, there is delta debugging [8], which uses automated testing to identify the difference between a failing and a passing test case and localize the fault based on that. What all these methods have in common, that they require passing traces as well, so they are a dynamic approach and require the verified PLC code to be executed. This latter can be problematic, of only a small part of the code was verified.

To overcome these issues I looked for methods that are completely static, and do not require passing traces to work. I managed to find two algorithms, which actually both used the same core idea, weakest precondition reasoning. The first algorithm was whodunit [9] that utilized weakest precondition reasoning, minimized the calculated state space, and marked the transforming statements inside the minimized state space as the offending statements. Another approach used cascade fault localization [10], that also used weakest preconditions and then used unsatisfiable cores to find the offending statements.

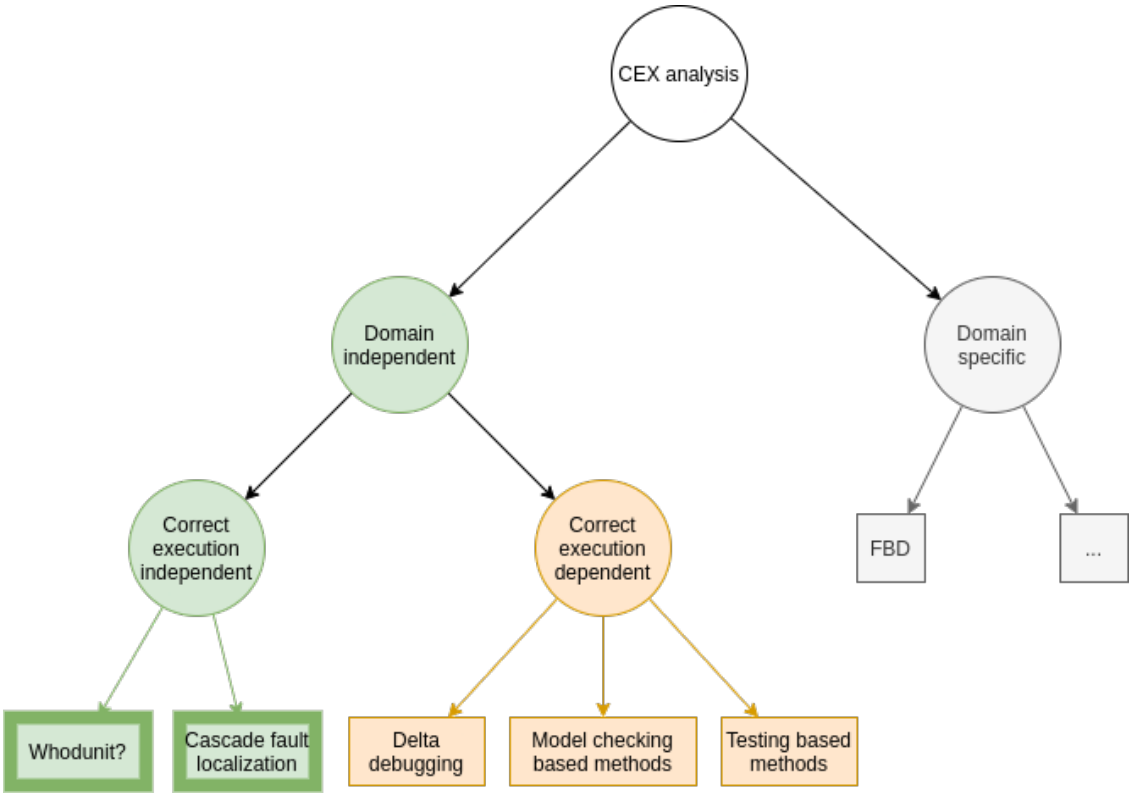


Figure 2: Counterexample analyzing algorithms

3. Whodunit

During the summer, I focused on whodunit, as it seemed easier to integrate and to better fit our purpose, but unfortunately it could not be applied without any modification.

The core of the algorithm takes the assertion at the end of the counterexample, and calculates its weakest precondition backwards, as long as the state space is satisfiable, and makes note of the the transforming statements, that are statements that transform the assertion. Should the state space

become unsatisfiable, the transforming statements in the minimal unsatisfiable set of the state space are marked as the cause of the assertion failure. In this case, we can speak of an internal error. On the other hand, if the state space is satisfiable until the beginning, then the cause of the error can be found in the values of the input variables.

Although the premise of the algorithm is promising, I encountered several problems. The first, and biggest problem comes in the way that PLCverif models assertions. As a PLC code can contain multiple assertions, PLCverif introduces a variable named `__assertion_error` with an initial integer value of 0. Should the first assertion fail, its value will be set to 1, should the second fail, to 2, etc… At the end, the only assertion in the resulting CFA is an assertion stating that this variable must be 0.

The issue with this approach, that in the counterexample the assertion is a variable being 0, and a couple of statements before that, there is an assignment setting this variable to another value. Whodunit will highlight this assignment as the cause of the error, and technically speaking it is right: there is an assignment setting the variable to a positive value, and a failed assertion stating it should be 0, so the issue is with the assignment.

To circumvent this issue I modified the algorithm to use an iterative approach. After the algorithm stops, I remove the locations and edges from the counterexample that it has already processed. Next, I find a new assertion. I achieved this by iterating the counterexample backwards, and finding the first guard. This guard can be converted to an assertion by negating it, and the original algorithm can be restarted from this point.

My next issue was, that this algorithm treated guards as a condition on a single boolean variable, which has been assigned beforehand. As this is not the case with PLCverif, I modified the algorithm to include guards' conditions in the set of transforming statements next to the assignments.

My final modification was made to make the result of the algorithm more detailed and specific. I created a (parameterizable) scoring mechanism that assigns a score to each statement in the CFA based on how likely it is to be the cause of the assertion failure. I awarded scores for the statement being a transforming statement, and for it being in the minimal set of conjuncts of the weakest state space. Next, I deducted points based on how much iterations was needed for that statement to be marked.

There is an example for a counterexample in the figure below. The first assertion is that the variable `__assertion_error` is 0. By calculating the weakest precondition, the algorithm terminates at the next step, as there is an assignment on the variable to be one. In the next iteration, a new assertion is chosen, and it is `max >= x2`, as this is the first assertion from that point. By calculating the weakest preconditions, the set of conjuncts in the state space will be `x3 >= x2`, `x2 <= x3`, `x1 <= x2`, while `x1 = 0`, `x2 = 1`, `x3 = 0`. The minimal set of conjuncts in this unsatisfiable state space is `x2 = 1`, `x3 = 0` and `x3 < x2`. The assertion was only transformed between L3 and L4 by the statement `max := x3`.

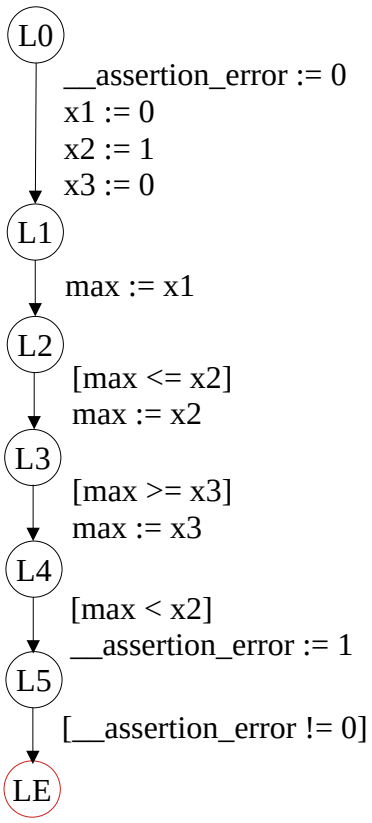


Figure 3: A counterexample to analyze

All in all, two statements will receive positive scores for being the cause of the failure. The first will be the statement `__assertion_error := 1`, which should be ignored, as explained above. The second will be `max := x3`, which is a transforming statement. It can be seen, that the issue in the code is indeed with that assignment, as it is the assignment that can assign a value less than the maximum.

4. Side-projects

Besides counterexample analysis, I worked on multiple side-projects whose goal was to improve PLCverif. Previously, I have worked with and developed Theta [11], a formal verification framework, one of the formal verification backends for PLCverif. My goal was to improve PLCverif's Theta support, as it was using a multiple year old build at the start of my internship.

As part of my work, I first focused on integer representation. In software, integers are represented on a finite amount of bits, so they have a finite range (and they overflow at the border). In contrast to that, formal verification methods tend to use an abstraction for the sake of performance, and treat these integers as mathematical integers, so they have an infinite range. There are two practical consequences to this: first, these integers do not overflow, second, one cannot apply bitwise operations on them.

Of course, there are ways around this issue, the first are bitvectors. They also store the numbers on a finite amount of bits, they overflow, and they support bitwise operations. However, they are

significantly slower to use in formal verification methods, is it is a trade-off between precision and speed that the user has to consider.

There is a middle ground and that is using mathematical integers with modulo operators. This method replaces all arithmetic operation with the modulo version of said operation. This way, these variables are able to overflow, but still do not support bitwise operations, but have less of a performance impact than bitvectors.

My next side-project was supporting dynamic array indexing in PLCverif. Up until this point, PLCverif required the array indexes to be constant numbers, so it can enumerate the arrays. However, there were examples, especially in the UNICOS framework, where dynamic array indexing was needed, and the usage of constant indices were not possible.

Fortunately, Theta supports dynamic arrays, so I was able to create an array representation setting in PLCverif's Theta backend, in which the user is able to chose whether they want to enumerate arrays, or use dynamic ones. If possible, it is worth to enumerate arrays as this way the verification is faster.

As part of my work, I integrated these additional integer representations[1] into the Theta backend, and the user can chose out of three options based on their needs. Moreover, I integrated the array representation[2] option into the Theta backend as well, so the user can choose between enumeration or dynamic indexing.

5. Conclusion

I applied to this internship in the hope of a professional challenge, and a chance to find out what it is like to work in one of the world's leading scientific institutions. During the summer, I worked on difficult, but interesting problems, that do not have one clear solution, and the scientific community has been trying to solve it for a long time, namely counterexample analysis. Apart from this, I worked on more practical tasks as well, like integrating Theta's newest version into PLCverif, as well as extending PLCverif's support with integer representation options and dynamic array indexing.

My one disappointment is that I was not able to attend in person due to COVID, as I believe it would have added much to this experience. I have always wanted to experience what it is like to live and work abroad, even if for a short amount of time. Moreover, I have already visited Geneva once, and would of liked to have the opportunity to explore it further.

All in all, I am content with the last two months, as I believe I have developed professionally. In the future, I would like to have a chance to experience what it is truly like to work at CERN.

1 https://gitlab.com/plcverif-oss/cern.plcverif/-/merge_requests/4
2 https://gitlab.com/plcverif-oss/cern.plcverif/-/merge_requests/6

Bibliography

[1]: Beyer D., Henzinger T.A., Théoduloz G., Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis, 2007
[2]: Polina Ovsiannikova and Igor Buzhinsky and Antti Pakonen and Valeriy Vyatkin, Visual counterexample explanation for model checking with Oeritte, 2020
[3]: Pakonen, Antti and Buzhinsky, Igor and Vyatkin, Valeriy, Counterexample Visualization and Explanation for Function Block Diagrams, 2018
[4]: Jones, J.A. and Harrold, M.J. and Stasko, J, Visualization of test information to assist fault localization, 2002
[5]: Renieres, M. and Reiss, S.P., Fault localization with nearest neighbor queries,
[6]: Ball, Thomas and Naik, Mayur and Rajamani, Sriram K., From Symptom to Cause: Localizing Errors in Counterexample Traces, 2003
[7]: What Went Wrong: Explaining Counterexamples, What Went Wrong: Explaining Counterexamples, 2002
[8]: Cleve, H. and Zeller, A., Locating causes of program failures, 2005
[9]: Wang C., Yang Z., Ivančić F., Gupta A., Whodunit? Causal Analysis for Counterexamples, 2006
[10]: Yi, Qiuping and Yang, Zijiang and Liu, Jian and Zhao, Chen and Wang, Chao, Explaining Software Failures by Cascade Fault Localization, 2015
[11]: Tóth, Tamás and Hajdu, Ákos and Vörös, András and Micskei, Zoltán and Majzik, István, Theta: A framework for abstraction refinement-based model checking, 2017