Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Static fault localization supported model checking for safety-critical systems

Scientific Students' Association Report

Author:

Mihály Dobos-Kovács

Advisors:

dr. András Vörös

BME, Critical Systems Research Group

dr. Jean-Charles Tournier,
dr. Borja Fernandez Adiego

CERN, BEAMS Department, Industrial Control Systems

2021

Contents

Kivonat

A különböző szoftveres megoldások egyre több feladatot látnak el biztonságkritikus rendszerekben. Példaként lehet említeni a gépjárművek kormányműjét, vagy akár a repülőgépek, atomerőművek irányítórendszerét. Ami ezen rendszerekben közös, az egy esetleges hiba következménye: hatalmas anyagi veszteség, súlyos környezeti kár, vagy akár emberéletek elvesztése.

Ezen biztonságkritikus szoftverkomponensek egyrészt kritikusak a rendszer működése szempontjából, másrészt meglehetősen összetettek. A komponensek helyes működését garantálni kell, ami miatt különféle módszereket lehet bevetni. A tesztelés egy bevett módszer hibák keresésére, éppen ezért minden, biztonságkritikus rendszerek fejlesztését szabályozó szabvány elvárja a használatát. Ugyanakkor a tesztelés önmagában a helyességet nem tudja igazolni. Egy merőben más megközelítés a formális verifikáció, ami a szoftver matematikai modelljét elemezve ad egy bizonyítást a szoftver helyességére vagy egy ellenpéldát egy hiba jelenlétének tanúsítására. Egy ellenpélda — egy hibás teszt nyomához hasonlóan — egy hibához vezető útvonalat ír le a rendszerben, és elemezni lehet a hiba okának feltárása érdekében. Azonban egy komoly probléma, hogy minél összetettebb a vizsgált rendszer, annál összetettebb lesz az ellenpélda is, és nehezebb az értelmezése. Egy ipari rendszer esetén az ellenpélda több száz vagy akár több tízezer utasítást tartalmazhat, amiknek nagy része irreleváns a hiba elhárításához.

A munkám célja egy olyan módszer kidolgozása, ami képes a hiba helyét megállapítani az ellenpéldákban anélkül, hogy azokat futtatni kéne. A módszer egy irodalomban ismert algoritmuson alapszik, ami a leggyengébb előfeltétel alapú érvelést használ ellenpéldák vizsgálatára. Munkám során továbbfejlesztettem az algoritmust, hogy a hiányosságait kijavítsam, valamint, hogy képes legyen a biztonságkritikus rendszerek verifikációs sajátosságait kezelni. Az algoritmus eredményét egyéb heurisztikákkal kombinálva a módszer egy pontszámot rendel a vizsgált szoftver utasításaihoz, ami az adott utasítás hibához való hozzájárulását jelzi. Ezt követően, a pontszámok értelmezésével a fejlesztő képes meghatározni, hogy a vizsgált kód mely kis részében keresse a hiba okát. A módszert C nyelvű szoftvereken, valamint ipari partnerektől származó PLC kódokon értékelem ki, és hasonlítom össze a hatékonyságát az eredeti algoritmuséval.

Abstract

Nowadays, different kinds of software are responsible for numerous features in safety-critical systems. Examples range from the steering mechanism of vehicles to the control system of airplanes or nuclear powerplants, but what binds them together is the consequences of a potential failure: substantial financial loss, catastrophic environmental damage, or the loss of human lives.

These safety-critical software components are essential for the correct functionalities of the system; however, they tend to be quite complex as well. To ensure the correctness of these components, different measures need to be taken. Testing is an efficient way of finding errors, and every standard regulating the development of safety-critical systems requires extensive testing. However, testing alone cannot prove the absence of errors. On the other hand, formal verification takes the mathematical model of the software and yields a mathematical proof of safety; or a counterexample to prove the presence of an error. A counterexample — similarly to a trace of a failing test — describes a path in the system leading to a failure and can be analyzed to find the cause of the issue. However, the more complex the system is, the more complex and difficult to understand the counterexample. In the case of industrial code, the counterexample will contain hundreds or even tens of thousands of lines of instructions, most of which are possibly irrelevant to understanding the cause of the underlying issue.

The goal of this paper is to present a method for localizing faults in counterexamples without executing them. The method is based on a novel algorithm from the literature that uses a weakest precondition-based reasoning to analyze the counterexample for the cause of the failure. In my work, I improved the algorithm to combat its shortcomings and for it to be able to handle the peculiarities of safety-critical software. I combined the algorithm with other heuristics to assign a score to each statement in the program that indicates that instruction's contribution to the error. By interpreting the scores, the developer can identify a small portion of the code to check for the cause of the failure. The method is evaluated on examples of C code as well as on industrial PLC code, and the result will be compared to the original algorithm.

Chapter 1

Introduction

Nowadays, different kinds of software-driven gadgets are becoming part of our everyday lives. Almost everyone carries a smartphone in their pocket, different types of wearable electronics are on the rise, and even simple household appliances have gained smart features. The same phenomenon of heavy reliance on software can be observed in the industry as well, as software-driven solutions tend to be more cost-effective than traditional electromechanical solutions. It follows that the number of software-driven components heavily increased in the so-called safety-critical systems as well. Examples for safety-critical systems range from the steering mechanism of vehicles to the control system of airplanes or nuclear power plants, but what binds them together is the consequences of a potential failure: substantial financial loss, catastrophic environmental damage, or the loss of human lives.

A typical example of an error leading to a catastrophe is the failed launch of Ariane 5 [22], the rocket of the European Space Agency (ESA). After years of development costing about 7 billion dollars, Ariane 5 was supposed to launch on the 4th of June 1996. However, the rocket self-destructed only after 37 seconds after launch. After extensive investigation, the report stated that the missile was destroyed due to a software bug. One of the components stored the rocket's velocity as a 64-bit floating-point number, while another stored it as a 16-bit integer (a legacy from Ariane 4). These components were part of the navigation subsystem. As the conversion between these two formats failed, the rocket lost its ability to navigate, deviated from its designated path, and finally self-destructed to avoid crashing back down on Earth. In the accident, half a billion-dollar worth of cargo was destroyed. However, should this kind of issue be in the reactor's control system of a nuclear powerplant, the result could have been another Chernobyl.

The correct behavior of safety-critical components is of utmost importance. To this end, different measures must be taken to identify errors in the system and to prove its correctness.

Testing is an efficient way of finding errors, and every standard regulating the development of safety-critical systems requires extensive testing. However, testing alone cannot prove the absence of errors, only their presence.

Another completely different approach is formal verification that takes the mathematical model of the software and yields a mathematical proof of safety. Formal verification is a computationally demanding task: it takes all possible states of the software into account, and even the simplest programs can have an immense or even infinite state space. There have been numerous breakthroughs during the past two decades in the field of formal verification and verification methods, becoming part of the development cycle of safety-critical software systems more and more; some standards even require it.

However, there are still challenges when it comes to the application of formal verification. The result of a formal method is either proof that unsafe behavior is unreachable or a counterexample. A counterexample — similarly to a trace of a failing test — describes a path in the system leading to a failure and can be analyzed to find the cause of the issue. Unfortunately, the more complex the system is, the more complex and difficult to understand the counterexample will be. In the case of industrial code, the counterexample will contain hundreds or even tens of thousands of lines of instructions, most of which are possibly irrelevant to understanding the cause of the underlying issue.

The goal of this paper is to present a method for localizing faults in counterexamples without concretely executing them. The method is based on a novel algorithm from the literature that uses a weakest precondition-based reasoning to analyze the counterexample for the cause of the failure. In my work, I improved the algorithm to combat its shortcomings and I extended the approach to handle the peculiarities of safety-critical software. I combined the algorithm with various heuristics to assign a score to each statement in the program that indicates that instruction's contribution to the error. By interpreting the scores, the developer can identify a small part of the code to check for the cause of the failure, or even an IDE support can be provided based on the approach.

The proposed method is evaluated via a custom implementation in the open-source verification framework Theta on examples of C code and industrial PLC code, provided by our industrial partner CERN. The results are compared to the original algorithm.

Chapter 2

Background

This chapter presents the necessary background to understand my work, including the formal and algorithmic background.

2.1 First-order logic

Although mathematical logic has several branches, this paper focuses on *first-order logic (FOL)* [14]. First-order logic has vast expressive power; however, the satisfiability of a first-order formula is generally undecidable algorithmically. Nonetheless, there are specific *theories* [13] (theory of integer arithmetic, theory of arrays, or theory of bit-vectors, for example) that give interpretation to the symbols of a first-order formula, thus loosening the underlying problem and making the satisfiability problem decidable (under certain circumstances).

An *SMT-problem (Satisfiability Modulo Theory)* [8] is a decision problem for logical formulas, in which, when given a first-order formula and the theories used in it, a solver can decide whether there exists a substitution of variables in the formula to concrete values so, after the substitution, the formula evaluates to true; or the formula is unsatisfiable.

An *assignment* is a pair in which the first component is a symbol, and the second is an element of the domain of the symbol, also called the value of the symbol.

The *model* of a first-order formula is a set of assignments, where there are no two assignments for the same symbol, there is an assignment for each symbol, and after substituting each symbol for their value, the formula evaluates to true.

A first-order formula is *satisfiable* if it has at least one model, while a first-order formula is *unsatisfiable* if it has no model satisfying it.

Example 2.1. *Given a first-order formula* $(x < 5 \land x \geq 3 \land y > 7)$ *where* x *and* y *are symbols, and their domain is the set of integers* $(x, y \in \mathbb{Z})$*. An example of an assignment is* $(x = 4)$*. An example model is* $\{(x = 4); (y = 8)\}$*, as substituting these values into the formula, it evaluates to true:* $(4 < 5 \land 4 \geq 3 \land 8 > 7) = \top$*. As there exists a model, the formula is satisfiable. It is worth to be noted that multiple models may exist. For example* $\{(x = 3); (y = 8)\}$ *is also a model of the formula.*

If the formula is $(4 < x \land x < 5)$*, where* $x, y \in \mathbb{Z}$*, then the formula is unsatisfiable, as there is no integer between 4 and 5. However, if* $x, y \in \mathbb{R}$ *then it is satisfiable as* $\{(x = 4.5)\}$ *satisfies it.*

Specialized software, so-called SMT solvers [30], are developed to solve SMT problems. Each SMT solver tends to use a different approach and excels in solving formulas efficiently using a unique set of theories (linear arithmetics, non-linear arithmetics, arrays, or bit-vectors, among others).

SMT solvers accept the SMT-problem in *Conjunctive Normal Form (CNF)* where the conjuncts are also called *assertions.*

Let the $F = F_1 \land F_2 \land ... \land F_n$ SMT-problem be unsatisfiable. The *unsatisfiable core* of F is $UC \subseteq \{F_1; F_2; ...; F_n\}$ subset of the assertions making up F given that the SMT-problem constructed with the elements of UC as assertions is also unsatisfiable. An UC unsatisfiable core of F is also a *minimal unsatisfiable core* of F if every proper subset of UC is satisfiable as an SMT problem. It is worth noting that SMT solvers are capable of calculating the unsatisfiable core of SMT-problems; however the cores are usually not minimal [26].

Example 2.2. *Given a first-order formula* $(x < 5 \land x > 4 \land y > 7)$ *where* $x, y \in \mathbb{Z}$*. The unsatisfiable core* UC *of the formula is* $\{(x < 5); (x > 4)\}$ *as the first-order formula* $(x < 5 \land x > 4)$ *is unsatisfiable.* UC *is also a minimal unsatisfiable core, as both* $(x < 5)$ *and* $(x > 4)$ *is satisfiable on its own.*

It is worth to be noted that $\{(x < 5); (x > 4); (y > 7)\}$ *is also an unsatisfiable core, but not minimal as removing* $(y > 7)$ *leads to an unsatisfiable proper subset.*

2.2 Formal representation of programs

This section presents a formal representation of programs upon which the formal verification and fault localization methods are based.

2.2.1 Control Flow Automata

Computer programs can appear in multiple different formats, for example, in the form of source code. It is easy to read and understand, while on the other hand, the binary created from the source code is not (easily) readable or understandable by a developer, but a computer can execute it without problems. Formal representation is needed to be created from programs to support the formal verification of computer programs.

One of the representations mentioned above is the *Control Flow Automata (CFA)* [10]. The CFA is a (V, L, l_0, E) tuple, where:

- $V = \{v_0, v_1, ...\}$ is the set of *variables* that are present in the program. Each $v_i \in V$ has a D_{v_i} domain.
- $L = \{l_0, l_1, ...\}$ is the set of *control locations.* It can be interpreted as the possible values of the program counter.
- $l_0 \in L$ is the *initial location*, which is active at the start of the program.

- $E \subseteq L \times Ops \times L$ is the set of transitions, where L is the set of control locations, and Ops is the set of operations. A transition is a directed edge between two control locations, one *operation* (or *statement*) labeling each of them. An operation can be:
  - $v_i = expr$: A *deterministic assignment* of a variable, where the value of the right-hand side expression *expr* becomes the value of the left-hand-side variable $v_i \in V$.
  - *havoc* v_i: A *non-deterministic assignment* of a variable, where the value of the variable $v_i \in V$ can be anything valid based in its domain D_{v_i}. Non-deterministic assignments are useful for modeling data coming from the user or other programs.
  - $[cond]$: A *guard*; a transition with a guard can only be executed if the expression inside the guard evaluated to true.

In summary, a CFA can be represented as a directed graph, where the nodes are the program locations, and the labeled edges are the transitions between the locations. The labels stand for the operations during the transition.

The transitions that end in l_i are said to be the *incoming transitions* of l_i, while the transitions that start in l_i are said to be the *outgoing transitions* of l_i. The location with no incoming transition is the *initial location*, while the location with no outgoing transition is a *terminating location*. The location l_i is *branching* if it has at least two outgoing transitions; it is *non-branching* if it is not the initial location, a terminating location, and it is not a branching location.

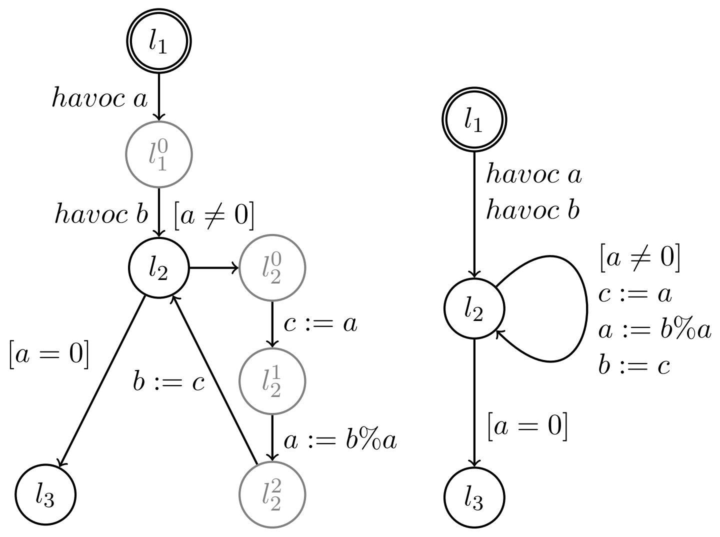
Formally, each transition has exactly one operation associated to it. However, CFAs are often depicted in their *compact form*. In the compact form, a transition can have multiple operations associated to it. A compact transition $(l_i, \{op_1, ..., op_n\}, l_i')$ is equivalent with a set of transitions $\{(l_i, op_1, l_i^1); (l_i^1, op_2, l_i^2); ...; (l_i^{n-1}, op_n, l_i')\}$, where $l_i^1, ..., l_i^{n-1} \in L$ are non-branching locations.

```c
void main() {
    int a, b;

    scanf("%d", &a);
    scanf("%d", &b);

    while(a != 0) {
        int c = a;
        a = b % a;
        b = c;
    }
}
```

(a)



(b) **(c)**

Figure 2.1: The Euclidean algorithm written in C (a), and the corresponding CFA it simple (b) and in compact form (c)

Example 2.3. *On the left side of Figure 2.1, there is an implementation of the Euclidean algorithm written in C. In the middle is a CFA that corresponds to the program on the left. There are two examples of non-deterministic assignment (havoc a and havoc b), three examples of deterministic assignment ($c := a$, $a := b\%a$ and $b := c$), and two examples of a guard ($[a \neq 0]$ and $[a = 0]$). The same CFA can be seen in its compact form on the right-hand side. The compact form can be created by removing the non-branching locations (highlighted with gray in the middle) and concatenating the labels on the corresponding transitions.*

2.2.2 The state-space of a CFA

Each program has its state-space, which is the set of all the possible, reachable states, and transitions between them. A state represents a control location and the values of the variables at a certain point in the operation of the program, while the transitions the operations the program carries out. One *(concrete) state* of the program is a $(l_i, d_1, d_2, ..., d_n)$ tuple, where:

- $l_i \in L$ is the current location,
- $d_1, d_2, ..., d_n$ are the values of the variables, where $d_i \in D_{v_i}$, $n = |V|$ and $d_i = v_i$.

As a CFA can represent a program, we need a method to construct the state-space of the program from the CFA. Given the current state is $(l_i, d_1, d_2, ..., d_n)$, l_i denotes a specific location in the CFA. Let us take a transition $(l_i, op, l'_i) \in E$ leaving this location and modifying the state of the program. Based on *op*, the following state is:

- If *op* is a deterministic assignment $v_k := expr$, then the following state is $(l'_i, d_1, ..., d'_k, ..., d_n)$, where d_k is the value of *expr*, in which all variables are substituted by their $d_1, ..., d_k, ..., d_n$ values. In short, the new value of v_k becomes the expression, while the other variables remain unchanged.
- If *op* is a non-deterministic assignment *havoc* v_k, then the following state is ambiguous. The following state can be $(l'_i, d_1, ..., d'_k, ..., d_n)$, where $d'_k \in D_{v_k}$. In short the value of v_k can be any value that is possible based on its domain, while all other variables remain unchanged, so the number of following states is the size of the domain D_{v_k}.
- If *op* is a guard [*cond*], then the following state is $(l'_i, d_1, ..., d_n)$, if *cond* evaluates to true based on the values $d_1, ..., d_n$. If it evaluates to false, the transition cannot be executed. It follows that the construction of a CFA needs to be careful, so for every state, a transition exists, for which all guards evaluate to true, or else a deadlock occurs.

Example 2.4. *Let the current state be $(l_1, 3, 4)$, where l_1 is the current location, while 3 and 4 are the respective values of variables x and y. Moreover, let the transition be (l_1, op, l_2). Based on op:*

- *If op is deterministic assignment $x := 2$, then the following state is $(l_2, 2, 4)$.*

- *If op is non-deterministic assignment havoc y, then the set of possible following states is:* $\{(l_2, 3, -\infty); ...; (l_2, 3, 0); (l_2, 3, 1); ...; (l_2, 3, \infty)\}$*, if* $D_y = \mathbb{Z}$.
- *If op is guard* $[y = 4]$*, then the following state is* $(l_2, 3, 4)$.
- *If op is guard* $[y \neq 4]$*, then the transition cannot be executed.*

The only thing left is to determine the initial state of the state-space. The CFA has an initial location that can be used, but the value of every variable must also be given. For example, in programs where uninitialized variables contain memory garbage (usually that are written in C, C++), there are multiple initial states, and it is non-deterministic, which one will be chosen. On the other hand, if uninitialized variables are automatically initialized to a specific value, often 0 (for programs written in a managed environment, such as Java, C#, or PLC), then there is only one initial state.

2.2.3 Predicate transformer semantics

Edsger Dijkstra first described predicate transformer semantics [21] to define the formal semantics of imperative programs. They are built on first-order and Hoare logic [18] and define the semantics of the operations in a program.

A first-order formula *satisfies* the concrete state $(l_i, d_1, d_2, ..., d_n)$ of the program if by replacing the symbols in the formula by their respective values $d_1, d_2, ..., d_n$ from the state the formula evaluates to true.

When describing predicate transformer semantics, the semantics is defined along the Hoare triple $\{P\}st\{R\}$, where $P, R \subseteq S$ (sets of states) and $st \subseteq S \times S$ (relation on states). $\{P\}st\{R\}$ means that $\forall s, s' \in S.\ (s \in P \land (s, s') \in op) \implies s' \in R$. P is called *precondition*, R is called *postcondition*, s is called *initial state* and s' is called final state. An intuitive meaning is that if P holds when st is executed, then R will hold as well.

By definition, P and R are sets of states. However, they can also be defined as first-order formulas that denote a set of states that satisfy the formula. This latter definition happens to be more beneficial for reasoning about computer programs.

Example 2.5. *Let us have the following Hoare triple:* $\{x = 0\}\ x := x + 1\ \{x = 1\}$*. In this instance, the first-order formula* $x = 0$ *is the precondition and satisfies all sates of the program where the value of variable* x *is* 0*. Similarly,* $x = 1$ *is the postcondition and satisfies all states of the program where the value of variable* x *is* 1*. The statement* $x := x + 1$ *binds the precondition and postcondition together. It denotes a set of operations that all increase the value of variable* x *by* 1.

Given a precondition $P \subseteq S$ and a statement $st \subseteq S \times S$, then their *strongest postcondition* [25] R is $post(P, st) = R = \{s' | \exists s.\ s \in P \land (s, s') \in st\}$. R is the strongest postcondition as $\forall R \subseteq S.\ \{P\}st\{R\} \implies post(P, st) \subseteq R$. If the postcondition is characterised by a first-order formula, then the strongest postcontition implies any postcondition satisfied by the final state of any execution of st, for any initial state satisfying P. Based on statement st:

- $post(P, x := expr) = \exists x_0.\ x = expr[x \to x_0] \land P[x \to x_0]$, where x_0 is a fresh variable, and $expr[x \to x_0]$ is $expr$ where all occurence of x is replaced by x_0, if st is deterministic assignment $x := expr$,

- $post(P, havoc\ x) = \exists x_0.\ x = P[x \to x_0]$, where x_0 is a fresh variable if st is non-deterministic assignment $havoc\,x$,
- $post(P, [cond]) = P \wedge cond$, if st is guard $[cond]$.

Example 2.6. *Some examples for strongest postconditions for the precondition $x \geq 5 \wedge y \geq 3$. Based on the operation:*

- $post(x \geq 5 \wedge y \geq 3, x := x + y + 10) = \exists x_0.\ x_0 \geq 5 \wedge y \geq 3 \wedge x := x_0 + y + 10$
- $post(x \geq 5 \wedge y \geq 3, havoc\,x) = \exists x_0.\ x_0 \geq 5 \wedge y \geq 3$
- $post(x \geq 5 \wedge y \geq 3, [x \geq 10]) = x \geq 5 \wedge y \geq 3 \wedge x \geq 10 = y \geq 3 \wedge x \geq 10$

A dual concept is the weakest precondition. Given a postcondition $R \subseteq S$ and a statement $st \subseteq S \times S$, then their *weakest (liberal) precondition* [7] P is $pre(st, R) = P = \{s|\forall s'.\ (s, s') \in st \implies s' \in R\}$. P is the weakest precondition as $\forall P \subseteq S.\ \{P\}st\{R\} \implies P \subseteq pre(st, R)$. If the precondition is characterised by a first-order formula, then any precondition satisfying the initial state implies the weakest precondition, for any execution of st and for any final state satisfying P. Based on statement st:

- $pre(x := expr, R) = R[x \to expr]$, where $R[x \to expr]$ is R with all occurence of x is replaced by $expr$, if st is deterministic assignment $x := expr$,
- $pre(havoc\ x, R) = \forall x_0.\ R[x \to x_0]$, where x_0 is a fresh variable if st is non-deterministic assignment $havoc\,x$,
- $pre([cond], R) = cond \implies R$, if st is guard $[cond]$.

Example 2.7. *Some examples for weakest preconditions for the poscondition $x \geq 5 \wedge y \geq 3$. Based on the operation:*

- $pre(x := x + y + 10, x \geq 5 \wedge y \geq 3) = x + y + 10 \geq 5 \wedge y \geq 3$
- $pre(havoc\,x, x \geq 5 \wedge y \geq 3) = \forall x_0.\ x_0 \geq 5 \wedge y \geq 3$
- $pre([x \geq 10], x \geq 5 \wedge y \geq 3) = x \geq 10 \implies (x \geq 5 \wedge y \geq 3)$

The dualism of the strongest postcondition and weakest (liberal) precondition comes from the following equivalence and is commonly used for bidirectional software analysis: $(post(P, st) \implies R) \iff (P \implies pre(st, R))$. This dualism is depicted in Figure 2.2.

2.3 Formal verification

There are numerous algorithms and methods that can check the erroneous behavior in a program. This section presents model checking as a general approach and Counterexample-Guided Abstraction Refinement (or CEGAR for short) as an algorithm to help verify computer software.

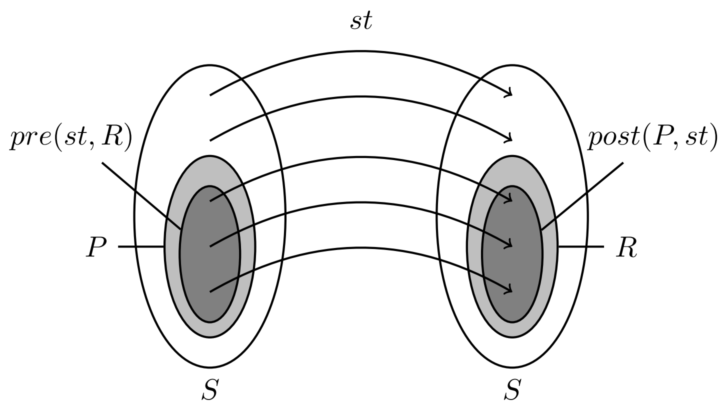


Figure 2.2: An illustration of the strongest postcondition and weakest precondition

2.3.1 Model checking

Given a formal model and a formal requirement (or statement), *model checking* [5] [17] will decide whether the given requirement holds for the given model. The model is safe if mathematical proof exists that the requirement holds for the model. Also, the model is unsafe if mathematical proof exists that the requirement does not hold for the model. It is worth noting that the proof of unsafeness is often an example for which the requirement fails.

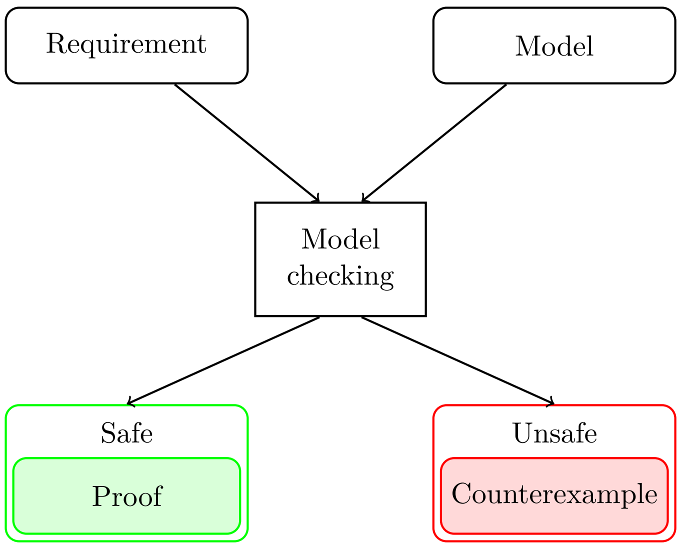


Figure 2.3: The model checking procedure

Model checking is a general approach, and it is not used exclusively for software verification. The notion of model, requirement, and checking needs to be given in terms of a program in order to apply model checking for computer software.

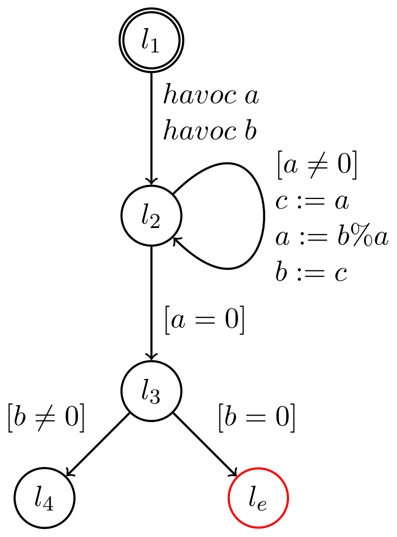
- Let the model be the CFA, as it is a formal representation of the program.
- Let the requirement be that no error location is available. An *error location* is a particular control location in the CFA, which yields an error if the control ever reaches it.

- Let the checking method be an algorithm that can prove whether the control is able ever to reach an error location or not. One possible method is a systematic traversal of the state-space that checks whether a state with an error location for control location or error-state is reachable in it; however, this method is nearly impossible to execute due to the state-space explosion.

The model is said to be *safe* if the requirement holds, and *unsafe* if the requirement does not hold.

```c
1  void main() {
2      int a, b;
3
4      scanf("%d", &a);
5      scanf("%d", &b);
6
7      while(a != 0) {
8          int c = a;
9          a = b % a;
10         b = c;
11     }
12
13     assert(b != 0);
14 }
```

(a)



(b)

Figure 2.4: The Euclidean algorithm written in C (a), and the corresponding CFA it simple (b) and in compact form (c)

Example 2.8. *On the left side of Figure 2.4, there is the Euclidean algorithm written in C. In line 9, there is an assertion. The corresponding CFA can be seen on the right side. It can be observed that the assertion is mapped as two separate branches. The first branch continues the normal flow of the program (l_4), while the other branch marks it as an error location (l_e). The error location is only entered if the condition in the assertion evaluates to false.*

2.3.2 CEGAR

The *Counterexample-Guided Abstraction Refinement (CEGAR)* [16] [27] is an abstraction-based model checking algorithm that has been effectively used to verify computer software. It can use a CFA, among other formalisms, as an underlying model, and it can check for reachability in the state-space, among others, as a requirement.

The size of a program's state-space depends on the number of control locations, the number of variables, and the size of those variables' domain. Out of these, the domain size has the most significant impact on the final size. In the case of two 32-bit integer variables in a program, then at least $2^{32} * 2^{23} = 2^{64} \approx 10^{19}$ states are needed to be represented. If the program had at least eight integer variables with 32-bit integer domains, more states would be needed to store the possible values than the number of atoms in the universe. This phenomenon is called the state-space explosion, and efficient algorithms are needed to handle it.

CEGAR uses abstraction to circumvent state-space explosion. It operates in the *abstract state-space* that consists of abstract states. An *abstract state* is the set of concrete states A (concrete) state is an *error-state* if it has an error location as its control location. It follows that an abstract state is an *abstract error-state* if it contains at least one concrete error-state.

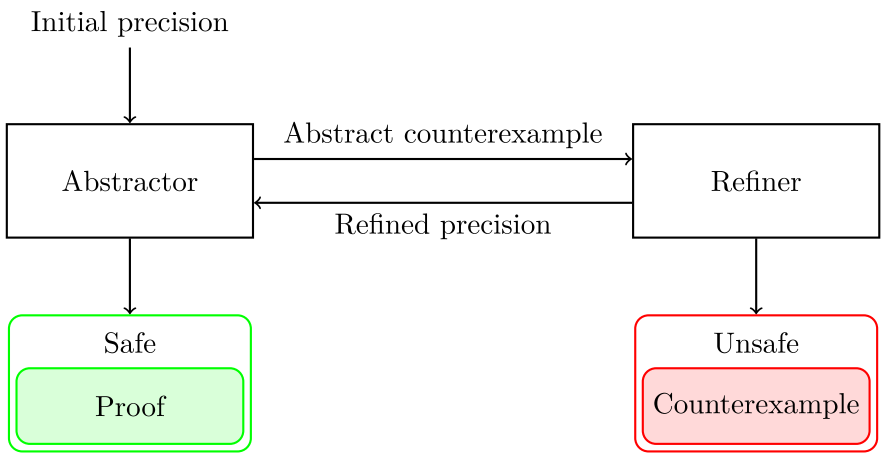


Figure 2.5: The CEGAR-loop

The core of the algorithm is the so-called CEGAR-loop (Figure 2.5) that consists of two distinct parts: the *abstractor* and the *refiner*. In the first part, the abstractor is responsible for building the abstract state-space from the model with a given precision. The abstractor also checks whether an abstract error-state is reachable. As an abstract error-state is an over-approximation of the possible error-states, if no abstract error-state is reachable, then no concrete error-state is reachable; thus, the requirement holds for the model.

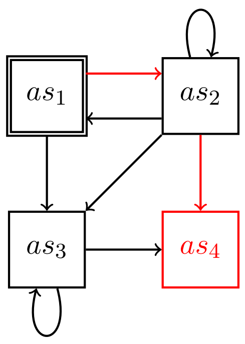
However, if an abstract error-state is reachable, the abstractor produces an *abstract counterexample* (Figure 2.6.a): a path from the initial state to the abstract error-state. Next, the refiner decides whether the counterexample is feasible or spurious.

If a concrete error-state inside of it is reachable, then the abstract counterexample is *feasible* (Figure 2.6.b), so the model fails the requirement. The path from the initial state to the concrete error-state acts as a counterexample.

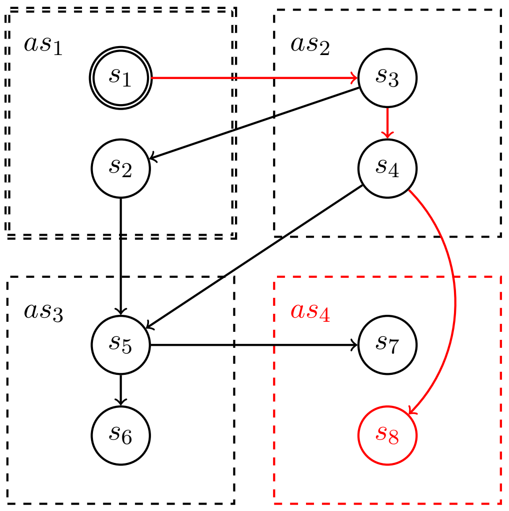
On the other hand, if a concrete error-state is not reachable, then the abstract counterexample is *spurious* (Figure 2.6.c), the reachability of the abstract error-state is the result of the over-approximation. In this case, the precision of abstraction needs to be refined so that the abstract error-state does not contain the unreachable error-state. The precision refinement algorithms are typically built on either unsatisfiable cores or Craig interpolation.

Example 2.9. *Let us assume that the abstractor returned an abstract counterexample seen in Figure 2.6.a. The counterexample starts in the initial abstract state as_1, goes through as_2 and terminates in the abstract error-state as_4. The abstract state-space is an over-approximation of the concrete state-space, so the same abstract state-space can represent multiple concrete state-spaces.*

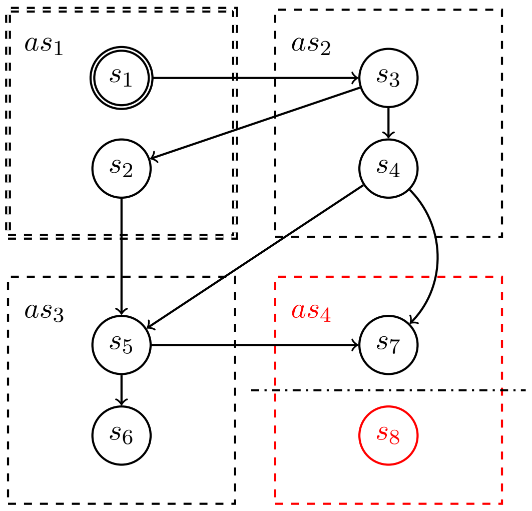
First, assume that the concrete state-space is the one that can be seen in Figure 2.6.b. In this case, the abstract counterexample is feasible, as there is a path (highlighted with red) from the initial state (s_1) to the error state (s_8). As a concrete error-state



(a) An abstract counterexample



(b) A feasible abstract counterexample



(c) A spurious abstract counterexample

Figure 2.6: An abstract counterexample and two possible underlying concrete state-spaces

is reachable, CEGAR will terminate with an unsafe result, with the path (highlighted with red) as a counterexample.

Alternatively, assume that the concrete state-space is the one that can be seen in Figure 2.6.c. In this case, the abstract counterexample is spurious, as there is no path from the initial state (s_1) to the error-state (s_8). The next step is to refine the precision to separate the error-state (s_8) from the abstract error-state (as_4). After the refining step, as_4 will not contain the error-state (s_8), so it will cease to be an abstract error-state; a newly created abstract state, containing s_8, will be the new abstract error-state (containing the states from as_4 below the dash-dotted line).

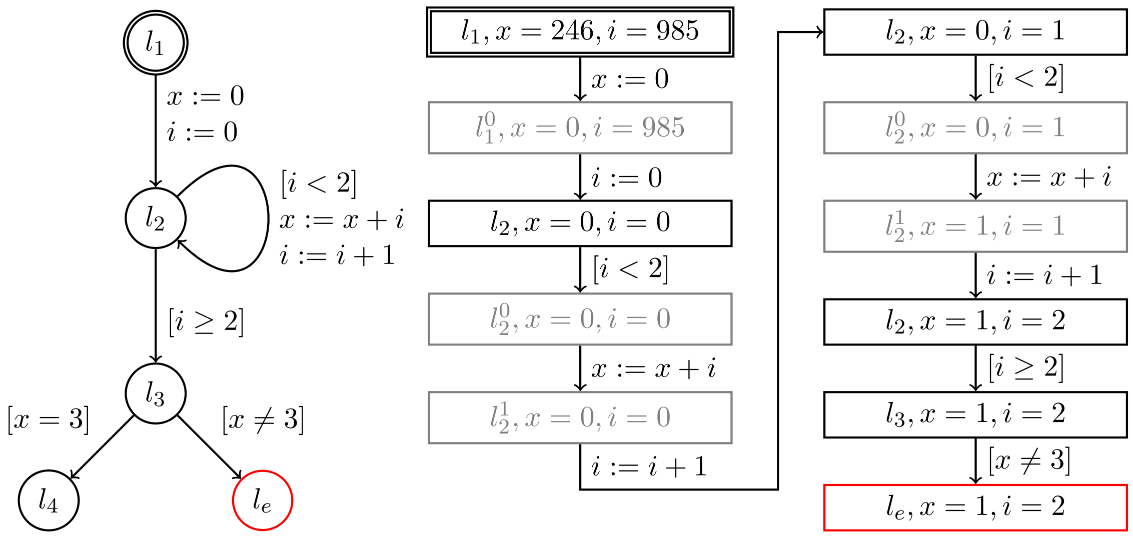
The CEGAR loop keeps repeating itself until it either proves that no abstract error-state is reachable, thus, the requirement holds or gives an example of how a concrete error-state is reachable, thus proving that the requirement does not hold. Each time an abstract error-state is reachable and the refiner proves that the concrete error-state inside is unreachable, the abstraction refines by separating the abstract error-state into at least two other parts. With each refinement, the number of abstract states grows; however, it cannot grow beyond the number of concrete states, which causes the algorithm to terminate at some point.

It is worth noting that multiple types of abstraction can be used with CEGAR: it can use predicate abstraction [28] just as easily as explicit-value abstraction [10] or different kinds of product abstraction [11].

2.3.3 Counterexamples

A *counterexample* [16] is a path from an initial state to a concrete error state. Formally, the counterexample of a CFA is an alternating sequence of states and operations $(s_0, op_0, s_1, op_1, ..., op_{n-1}, s_n$, where s_0 is an initial state and s_n is an error state and op_i is the operation performed to move the execution from s_i to s_{i+1}.

It follows that the values of the program variables can be extracted from the states along with the locations. Using the locations, the path in the CFA can be reconstructed easily.



(a) **(b)**

Figure 2.7: An example CFA (a) with a counterexample (b)

Example 2.10. *A counterexample for a CFA (Figure 2.7.a) can be seen on the right side of the Figure 2.7. The CFA depicts a simple program that sums the first two positive integers in the variable x and asserts that the sum should be $1 + 2 = 3$. However, the program is written in a way that it treats zero as the first positive integer. As a result, the sum of the first two integers will be one, and the assertion will fail.*

Figure 2.7.b depicts a path leading to an error-state, or in other words a counterexample. The counterexample is an alternating list of states and operations, and each state contains (as per the definition in Section 2.2.2) a location and an exact value for each variable.

It is worth noting that there is no information about the initial values of the variables, so there are multiple initial states of the CFA. However, the counterexample requires one concrete initial state, so one will be chosen arbitrarily. Another noteworthy observation is that the loops of the CFA are unrolled in the counterexample: the counterexample contains the steps of the loop the number of times the loop is executed (two in this instance).

2.4 Fault localization

Formal verification is a valuable tool for finding errors in a program but provided we know of an error, finding its cause is an entirely different matter. The goal of fault localization [35] is to identify the locations of the fault automatically to point the developers in the right direction when they try to fix it. For the sake of generality, there are methods that not only detect the location of the faults but offer a fix for them as well; however, these methods are not investigated in this paper.

The traditional tools of fault localization are well known and widely used: logging, assertions, breakpoints, and profiling. Although helpful, these methods are all manual.

Slicing-based methods are often used for fault localization. An important solution is static slicing [34], which analyzes the data and control dependencies inside the program and removes all instructions irrelevant to the failure. Static slicing is effective but tends to leave unnecessary instructions in the slice. Dynamic slicing [2] addresses this issue by incorporating runtime information from a failed test case. The core idea of slicing-based methods is that the slice is many times smaller than the original program, so it is easier to find the cause of the issue in the slice.

Statistics and spectrum-based methods [1] usually require more than a single failed execution. They usually require multiple execution traces (i.e., a whole test suite) and analyze which part of the code was executed during a passing and a failing trace. Processing this information, these methods can pinpoint the most likely statements to cause the actual failure. Advanced statistics-based methods use machine learning [4] or data mining [15] as well.

Program state-based techniques also rely on successful executions. Usually, the states of the program during a successful execution are compared to the states of the program during a failed execution. An effective and popular algorithm called delta debugging [36] tries to modify the states of the failed execution by using information from the successful execution step-by-step (hence the name delta debugging) and re-executes the modified failing trace to find the cause of the issue.

Model-based methods usually take a model of the program and either use a formal specification, an oracle implementation, or successful executions to find the cause of the fault. These methods tend to use model checking algorithms [24], but some solutions use symbolic execution [29] as well.

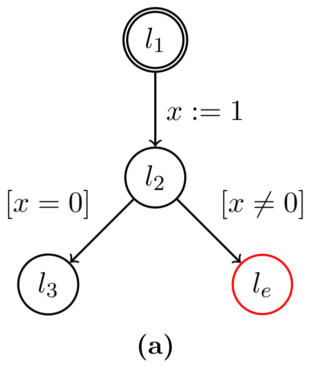
The common part of the previous approaches is that they all require additional data from the user besides an error to determine the cause. Unfortunately, there are cases when a test suite is not available or creating an oracle is infeasible. Formal verification usually finds hidden, obscure errors missed by the rigorous testing, so the statistical information around the failure might be lacking. Moreover, there are cases of mission-critical systems where the cause of the error is a hardware design fault [6]. In these cases, the hardware faults need to be corrected in software, and usually, there is no test suite dedicated for that.

Some algorithms only require a failing trace to work and are generally applied in cases where additional information is not available or expensive to produce. Out of these algorithms, some examples target a particular domain, such as function block diagrams [31] and there are algorithms that can work on a more generic CFA formalism [33].

2.4.1 Whodunit

As mentioned earlier, there are numerous methods available that localize faults in programs. However, they usually require additional information, like multiple successful or failing traces, or use expensive model checking or constraint solving. In contrast, Wang et al. (2006) [33] devised a single path-based method for reasoning about software failures using weakest preconditions. The algorithm will be called *Whodunit* in this paper, owing to the title of the article it was published in.

The input of the algorithm is a counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$, where the last state is an error state. The counterexample describes the failing trace in the program. Moreover, we can conclude that the last operation, op_{n-1} is a guard $[\neg c]$ which comes from the failed assertion.



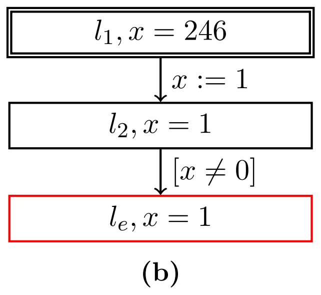


Figure 2.8: An example CFA (a) with a counterexample (b)

Example 2.11. *An example of this statement can be seen in Figure 2.8. Since error locations are created from assertions, the input transition of the error location will always be labeled by a guard, which is the negated assertion.*

Weakest preconditions are generally used in model checking methods to refine the precision based on an infeasible abstract counterexample. However, Whodunit uses weakest preconditions for a different reason. The input counterexample is feasible, so weakest preconditions are used to find a minimal set of conditions needed by the program to stay on the same path without violating the assertion.

The core idea of the algorithm is the *infection chain.* The infection chain is a list of predicates starting from the failed assertion c. The next step is calculated from the previous step by applying a modified weakest precondition predicate transformer $WP(st, R)$:

- $WP(x := expr, R) = R[x \to expr]$
- $WP(havoc\ x, R) = R$
- $WP([cond], R) = R \wedge cond$
- $WP(\{op_0, op_1, ..., op_n\}, R) = WP(op_0, WP(op_1, ..., WP(op_n, R))...))$

There are two differences compared to the weakest precondition:

- The first difference is the result of the non-deterministic assignment *havoc* x. In the concrete counterexample, each variable is assigned an exact value, including the non-deterministically assigned variable x in this case, so it is not needed to apply the original transformer. Non-deterministic assignments model inputs and will be required later on, but they are not part of the infection chain.

- The second difference is the result of the guard $[cond]$. The concrete counterexample is a concrete, executable path in the program, so it is known that every guard evaluates to true. Taking this information into account, $WP([cond], R) = pre([cond], R) \wedge cond = (cond \implies R) \wedge cond = cond \wedge R$.

Example 2.12. *Some examples for the modified weakest precondition:*

- $WP(x := x + y + 10, x \geq 5 \wedge y \geq 3) = x + y + 10 \geq 5 \wedge y \geq 3$
- $WP(havoc\, x, x \geq 5 \wedge y \geq 3) = x \geq 5 \wedge y \geq 3$
- $WP([x \geq 10], x \geq 5 \wedge y \geq 3) = x \geq 10 \wedge (x \geq 5 \wedge y \geq 3)$
- $WP(\{x := x+y+10, [x \geq 10]\}, x \geq 5 \wedge y \geq 3) = WP(x := x+y+10, WP([x \geq 10], x \geq 5 \wedge y \geq 3)) = WP(x := x+y+10, x \geq 10 \wedge (x \geq 5 \wedge y \geq 3)) = x+y+10 \geq 10 \wedge (x + y + 10 \geq 5 \wedge y \geq 3)$

In the end, the infection chain of counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$ where the last operation, op_{n-1} is a guard $[\neg c]$ is

$$WP(\{op_0, op_1, ..., op_{n-1}\}, c) = c' \wedge (c'_0 \wedge c'_1 \wedge ... \wedge c'_l) ,$$

where c' is transformed from the given assertion c through variable substitutions in case of deterministic assignments, and each c'_i is transformed from a guard $[cond]$ through variable substitutions in case of deterministic assignments. It is worth noting that the resulting formula is a list of conjuncts, and each guard in the counterexample adds a new conjunct, while each assignment transforms one or more conjuncts.

A formula f' is *transformed* from formula f if it can be created from f by variable substitutions. An operation op_i is a *transforming statement* of f if op_i is deterministic assignment $x := expr$ and during the creation of f', x is substituted for $expr$.

Example 2.13. *Let us assume that we have $f_1 = (x \geq 5)$ and $f_2 = (y \geq 3)$ and an operation op as $x := x+y+10$. In this case $f'_1 = (x+y+15)$ and op is a transforming statement of f_1. op is not a transforming statement of f_2, as the variables written by op are not in f_2.*

Given a counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$ and a formula f the *set of transforming statements* of the formula is

$$\begin{aligned} TS(\{op_0, op_1, ..., op_{n-1}\}, f) = \{&op_i \mid op_i \text{ is a transforming statement of } f \\ &\text{when calculating } WP(\{op_0, op_1, ..., op_{n-1}\}, f)\} \ . \end{aligned}$$

The *input valuation* is a set of assignments that assigns an exact value to each input variable of the program. The input valuation I can be extracted from the counterexample:

the input variables are identified by non-deterministic assignments, while their values can be extracted from the state.

> **Example 2.14.** *Let us assume, that our program has two input variables x and y. This means, that there are (at least) two non-deterministic assignments in the program: one for x and one for y. In this case the input valuation I can be $I = (x = 4) \wedge (y = 3)$ if the counterexample assigns 4 to variable x and 3 to variable y.*

The main theorem behind Whodunit is that the SMT problem constructed using the infection chain and the input valuation of the assertion is unsatisfiable. This theorem is called the *proof of infeasibility.* More formally, given a counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$ where the last operation, op_{n-1} is a guard $[\neg c]$, the proof of infeasibility is:

$$I \wedge WP(\{op_0, op_1, ..., op_{n-1}\}, c) = \emptyset.$$

Generally, the proof of infeasibility consists of a set of conjuncts. The input valuation is a set of valuations for each input variable, while the infection chain was shown to consist of a set of conjuncts: the transformed assertion and the transformed guards. Given, that

$$(I_0 \wedge I_1 \wedge ... \wedge I_m) \wedge c' \wedge (c'_0 \wedge c'_1 \wedge ... \wedge c'_l) = \emptyset,$$

there exists a minimal set of conjuncts of I and a minimal set of conjuncts of $WP(\{...\}, c)$ denoted respectively by I_{sub} and WP_{sub}, such that $I_{sub} \wedge WP_{sub} = \emptyset$. We call $I_{sub} \wedge WP_{sub}$ the *minimal proof of infeasibility.* An algorithm for calculating the minimal proof of infeasibility can be seen in Algorithm 2.1.

Algorithm 2.1: Minimal proof of infeasibility

```
   input  : $I$, $WP(\{...\}, c)$
   output: $I_{sub}$, $WP_{sub}$
1  $I_{sub} \leftarrow I$, $WP_{sub} \leftarrow WP(\{...\}, c)$
2  for each $c'_i$ conjunct $c_i \in WP_{sub}$ do
3      drop $c'_i$ from $WP_{sub}$
4      if $I_{sub} \wedge WP_{sub} = \emptyset$ then
5          drop $c'_i$ permanently
6      else
7          add $c'_i$ back to $WP_{sub}$
8      end
9  end
10 for each $I'_i$ conjunct $I_i \in I_{sub}$ do
11     drop $I'_i$ from $I_{sub}$
12     if $I_{sub} \wedge WP_{sub} = \emptyset$ then
13         drop $I'_i$ permanently
14     else
15         add $I'_i$ back to $I_{sub}$
16     end
17 end
```

Whodunit combines the previous steps to identify the cause of the assertion failure. Given a counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$, where op_{n-1} is $[\neg c]$, Whodunit first calculates the infection chain $WP(\{op_0, op_1, ..., op_{n-1}\}, c)$ step-by-step. During this process, the transforming statements of each conjunct in the infection chain are recorded as well. In each step, while calculating the infection chain, the intermediate result is checked. There are two outcomes:

1. The infection chain becomes empty before reaching the start of the counterexample: $WP(\{op_i, op_{i+1}, ..., op_{n-1}\}, c) = \emptyset$. In this case, WP_{sub} needs to be calculated as soon as the infection chain becomes unsatisfiable and all conjuncts in WP_{sub} are considered as the cause of the failure.

2. The infection chain is calculated successfully: $WP(\{op_0, op_1, ..., op_{n-1}\}, c) \neq \emptyset$. In this case I_{sub} and WP_{sub} is calculated and their conjuncts are the cause of the assertion failure.

In the end, the transforming statements corresponding to the remaining conjuncts in WP_{sub} are denoted as the cause of the failure: $\{s \mid s \in TS(f)$ if $f' \in WP_{sub}\}$. The algorithm can be seen in Algorithm 2.2.

Algorithm 2.2: Whodunit

```
input  : A counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$, where $op_{n-1}$ is $[\neg c]$
output: Subset of $TS(\{op_0, op_1, ..., op_{n-1}\}, c)$ causing the assertion failure $c$
1  $WP \leftarrow c$
2  for $i \leftarrow n-1$ to $0$ do
3      $WP \leftarrow WP(op_i, WP)$
4      if $WP = \emptyset$ then
5          calculate $WP_{sub}$
6          return $\{s \mid s \in TS(f)$ if $f' \in WP_{sub}\}$
7      end
8  end
9  calculate $I_{sub}$ and $WP_{sub}$
10 return $\{s \mid s \in TS(f)$ if $f' \in WP_{sub}\}$
```

Example 2.15. *An example of the Whodunit algorithm can be seen in Figure 2.9. The C code describes an algorithm calculating the difference between two numbers received as parameters. This difference must be positive, as seen in the assertion in Line 14. However, the direction of the comparison in Line 4 is wrong, so the assertion fails. An example for the failure is $x1 = 3$ and $x2 = 4$ which will lead to $diff = -1$. The counterexample goes through lines 2, 3, 4, 5, and 14.*

Whodunit starts in Line 14 by the assertion $diff \geq 0$. The next step is Line 5. Since it is a deterministic assignment, the conjuncts in WP are transformed, and the transformation statements are recorded: Line 5 transforms the assertion. The next step is Line 4. As it is a guard, a new conjunct is added to WP. However, the algorithm stops here, as WP is unsatisfiable at this moment.

At the end of the algorithm, the transforming statements related to the conjuncts in WP are blamed for the assertion failure. In this case, Line 5 is the culprit. It can

```c
 1  int compute_diff(int x1, int x2) {
 2      int diff;
 3      if(x1 != x2) {
 4          if(x1 < x2) {
 5              diff = x1 - x2;
 6          }
 7          else {
 8              diff = x2 - x1;
 9          }
10      }
11      else {
12          diff = 0;
13      }
14      assert(diff >= 0);
15  }
```

(a) An example function in C

Line	WP	$TS(diff \geq 0)$	$WP = \emptyset$
14	$(diff \geq 0)$	no	no
5	$(x1 - x2 \geq 0)$	yes	no
4	$(x1 < x2) \wedge (x1 - x2 \geq 0)$	no	yes

(b) The execution of Whodunit

Figure 2.9: An example for Whodunit in action

> *be seen that the issue is indeed the assignment in Line 5, but one might argue that the guard in Line 4 is at least equally if not more at fault here.*

Chapter 3

Fault localization in formal counterexamples

This chapter presents a method for analyzing the counterexamples produced by the formal CEGAR algorithm. It also presents the algorithms used in the method, as well as the methodology and reasoning behind the solution. Finally, this chapter introduces a prototype implementation that serves as a proof of concept.

3.1 Overview of approach

Safety is an essential aspect of the development of safety-critical systems. To ensure safe behavior, safety-critical systems are submitted to rigorous testing procedures required by every standard regulating the development of said systems. However, testing has its limits, as it can only prove the presence of errors, not the absence of them. Moreover, although there are methodologies to design test suites, testing is not guaranteed to find the errors in the system.

Formal verification uses an entirely different approach to testing. It automatically scans the whole state-space of the system, it can find every error, even the most obscure ones, and it is able to prove that the system is safe. Nowadays, more and more standards regulating safety-critical systems require the application of formal methods.

However, from the point of view of a simple developer, there are limitations on the application of formal methods. Formal methods usually require knowledge in the area for different reasons:

1. The system under verification and the requirement need to be formalized, as formal methods work on mathematical models.
2. Formal methods tend to have many parameters that modify the behavior of the algorithm, and it is usually required to choose them based on the properties of the system under verification for the best possible performance.
3. The result of formal methods needs to be analyzed and possibly mapped back to the source code of the system.

The first issue is usually handled by applying different solutions together. Formal verification tools tend to have language frontends that can parse industrial code written in standard languages like C, Java or PLC, and create the formal model without explicit knowledge in the area. A similar approach is the usage of assertions as requirements, as assertions are part of every language (or standard library).

The second issue presents more of a challenge. Choosing the parameters of a formal method requires extensive knowledge of that algorithm, and choosing the wrong parameters might make the verification impossible. In recent years, with the intense increase in the computation capacity, a new method became available to solve this issue. Advanced tools tend to use feature detection and a dynamic portfolio [3] to analyze the system and decide the best parameters for verification.

However, the last issue still persists. Analyzing the results of formal methods requires knowledge of that method and usually takes an immersive amount of effort. Model checking methods check all states of a system, and in an unsafe case, a counterexample is reported. The counterexample represents a path, a list of states in the state-space of the system. However, the bigger the system is, the bigger the counterexample might be (in terms of number of variables).

In the case of industrial systems, it is especially common that the system consists of hundreds of thousands of lines of codes and has thousands of variables. An issue deep in the state-space of such a system would yield an enormous counterexample that would be extremely difficult to analyze. However, analyzing the counterexample would have its own issues:

1. The system is developed by multiple developers, who only know a part of the system. Many developers have to interact even to understand the counterexample.
2. If the system is developed inside a framework, the developers need to understand the exact inner workings of the framework to understand the counterexample: even if a third party developed the framework. As an example, CERN tends to develop PLC code inside the UNICOS framework [23].

The aforementioned issues are similar to the difficulties that development teams face during system tests or when they analyze the logs of system crashes. To try and ease the burden, fault localization methods come to the rescue that try to find the cause of issues in the system. Fault localization methods are automatic, although they usually require some information about the software: the very least a known fault.

The main goal of this paper is to present a method for assisting the development of safety-critical systems to adopt the application of formal verification. Safety-critical systems are usually developed according to a more rigid waterfall model, where the testing step is done after the implementation and by an independent team. In this scenario, additional information like a complete test suite with adequate coverage for fault localization to work is likely unavailable during the implementation phase to assist the developers in finding core issues. Thereby, this paper presents a method combining a model checking method with a fault localization approach that analyzes the counterexamples produced by the formal algorithm and does not rely on any additional information besides the counterexample.

The overview of the approach can be seen in Figure 3.1. The first step requires language frontends that can parse industrial C, PLC, or other types of code and create a CFA from them. The language frontends are also tasked with extracting an assertion from the source as well. Using the CFA and the assertion, the CEGAR algorithm is executed as a model checking method. If the algorithm concludes its input is safe, the whole process terminates

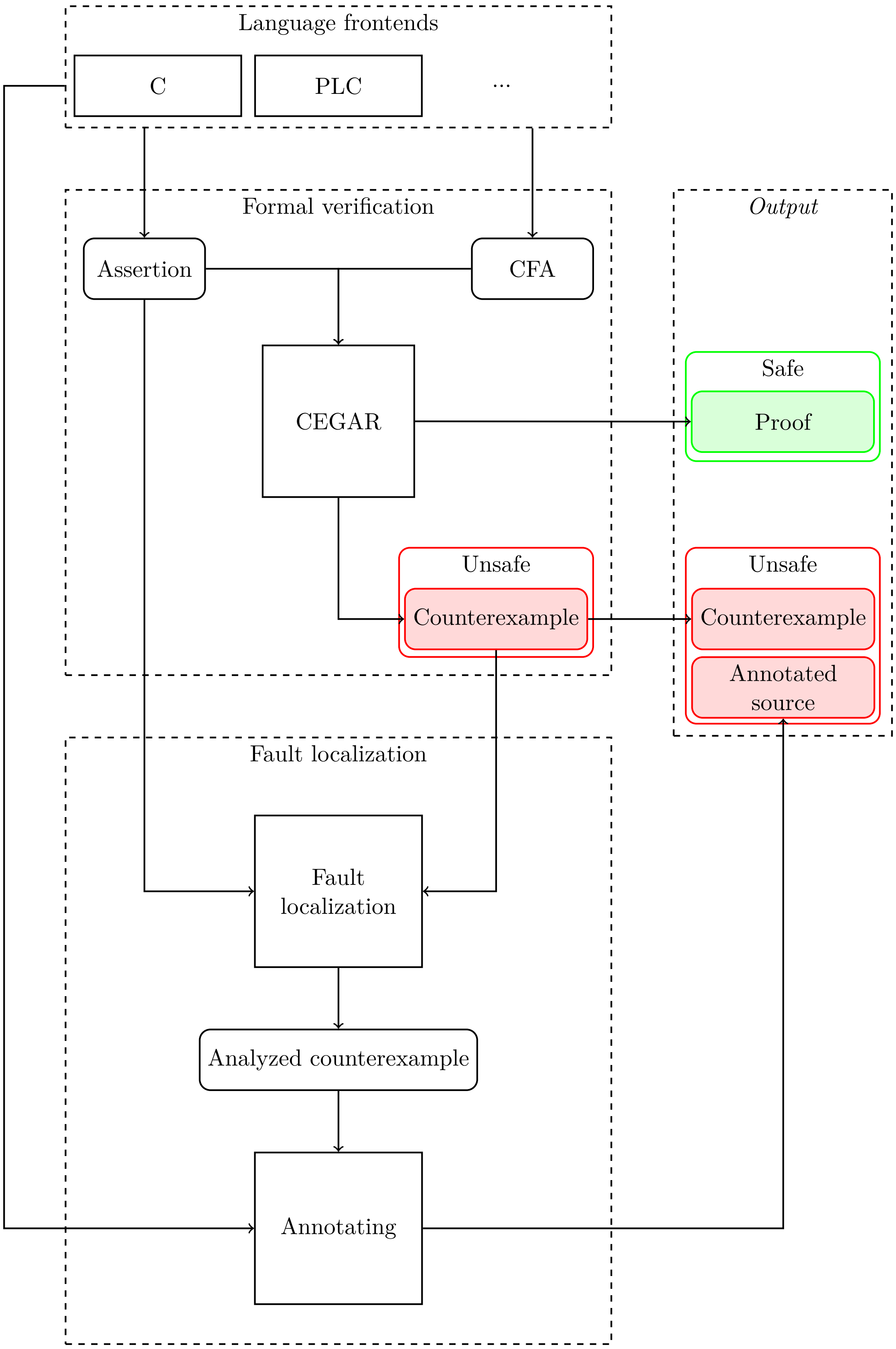


Figure 3.1: The overview of the approach

with a safe result (and a proof for that). However, should the CEGAR algorithm terminate with an unsafe result, the counterexample will be analyzed further.

The fault localization method requires a formal counterexample and the failed assertion as input and analyzes the counterexample. After that, using information from the language frontends and from the analyzed counterexample, the source code will be annotated with information regarding the cause of the failure. This annotated source code accompanies the counterexample as part of the unsafe result of the method.

3.2 Producing counterexamples

The approach described in the previous section requires a formal method for the counterexample. Generally, many model checking methods produce a counterexample, which all essentially share the same format. However, the task at hand usually indicates the formal method to use.

Bounded model checking [12] is one of the earliest examples of model checking. Bounded model checking checks the state-space until a certain depth (or bound) is reached. If it encounters an error state until then, it can provide a counterexample. However, it is only able to prove correctness if the state-space is smaller than the bound. With optimal configuration parameters, bounded model checking is an effective tool for finding issues but generally has a bad performance when it comes to proving correctness.

In this method, I opted for an abstraction-based model checking algorithm, as it is generally able to prove correctness, as well as find errors effectively, although abstraction-based methods are computationally more expensive than bounded model checking. Out of the abstraction-based methods, I chose CEGAR as a tool of convenience. CEGAR is well known, has been profoundly studied in the past two decades, and is heavily configurable.

For the sake of generality, it needs to be mentioned that the fault localization method is independent of the model checking method used. Moreover, the counterexample may even come from a failed test case: the trace of the test need to be formalized for that.

However, a limitation is that each algorithm might use slightly different formalization as input, and the fault localization method highly depends on the predicate transformer semantics. CEGAR enables us to work with reasonably simple statements, as it only requires three: deterministic assignments of simple variables, non-deterministic assignments of simple variables, and guards: the ones defined by the CFA formalism. Should the counterexample be a failed C test case, additional statements arise: if statements, while statements, assignments to arrays, structs, pointers, among others. It is possible to support more kinds of statements by declaring their formal semantics through the predicate transformers, but for the sake of simplicity, the rest of the paper works with strict CFA semantics.

3.3 Fault localization

This section presents the used fault localization method. Should the formal verification fail, the fault localization will analyze the produced counterexample. Fault localization builds on an improved Whodunit algorithm and introduces a novel method for assigning scores to the lines of the source code: the higher the score, the more likely it is that that line contributes to the failure of the assertion.

3.3.1 Limitations of Whodunit

All in all, Whodunit [33] has a decent performance when it comes to fault localization. However, the algorithm has multiple shortcomings that limit its usefulness:

1. Whodunit only denotes assignments as causes of a failure (see Algorithm 2.2). However, there are obviously situations where the issue is in the condition of a guard.

   Moreover, there are situations where no assignment is needed for a failure: I/O intensive applications tend to have an infinite cycle in their core. In each cycle, input is read by a non-deterministic assignment, and the data is processed. There might be assertions that fail due to a missing check of the input. In this case, the counterexample only contains guards and non-deterministic assignments (relevant to the assertion failure), and Whodunit ignores them both. An example can be seen in Listing 3.1.

   ```
   void main() {
       int a;
       int b = 2;
       while(true) {
           scanf("%d", &a);
           if(a) {
               b = b * b;
               assert(!a && b);
           }
       }
   }
   ```

   Listing 3.1: An assertion failure caused by non-deterministic assignments

2. Whodunit is likely to terminate early based on the exact formalization of the assertion. Language frontends tend to allow multiple assertions in the source code. In this case, a new variable (usually called `__assertion_failure`) is introduced in the CFA with 0 as a starting value. Should the first assertion fail in the source code, the value of this variable will be set to 1. Should the second assertion fail, the value will be set to 2, and so on. In the end, a single global assertion is defined, asserting the value of this variable to be precisely 0 (a.k.a. no assertion failed).

   If Whodunit is applied to such input, it will terminate the algorithm when `__assertion_failure` is assigned. At this point, there are two clauses in the WP: `__assertion_failure` $= 0$ coming from the assertion, and `__assertion_failure` $= 1$ coming from the assertion, which is a contradiction: the algorithm will terminate without really finding the underlying cause of the failure. An example can be seen in Figure 3.2: Whodunit would determine that the cause of the assertion failure is the assignment in Line 13.

3. Finally, the algorithm for calculating the minimal proof of infeasibility (see Algorithm 2.1) is generally ineffective. The algorithm in itself contains a cycle iterating through all conjuncts in WP and I, and for each conjunct, it checks unsatisfiability. Generally, these checks are encoded as SMT problems, but solving SMT problems is resource-intensive and time-consuming. Decreasing the number of SMT problems to solve could improve the performance considerably.

```c
void main() {
    int n;
    int s = 1;


    scanf("%d", &n);

    for(int i = 0; i < n; i += 2) {
        s += 2;
    }

    assert(s % 2 == 0);




}
```

(a)

```c
void main() {
    int n;
    int s = 1;
    int __assertion_failure = 0;

    scanf("%d", &n);

    for(int i = 0; i < n; i += 2) {
        s += 2;
    }

    if(s % 2 != 0) {
        __assertion_failure = 1;
    }

    assert(__assertion_failure == 0);
}
```

(b)

Figure 3.2: An example program (a) and an equivalent version depicting the formalization used by typical language frontends (b)

Example 3.1. *Let us assume, that* $WP = (x > 5) \wedge (x < 4) \wedge (y = 4)$*. To minimize* WP*, three SMT-problems will be checked:* $(x > 5) \wedge (x < 4)$*,* $(x > 5) \wedge (y = 4)$ *and* $(x < 4) \wedge (y = 4)$*. Out of these three options only the first is unsatisfiable, so* $WP_{sub} = (x > 5) \wedge (x < 4)$*.*

3.3.2 Improving Whodunit

To improve upon the issues described in the previous section, the original algorithm described in Algorithm 2.2 was modified at multiple places. First of all, the issue of only blaming deterministic assignments is addressed. Next, the algorithm will be modified to handle the peculiarities of language frontends. Finally, a new, more effective algorithm is introduced for finding the minimal proof of infeasibility.

3.3.2.1 Including guards in the set of transforming statements

Generally speaking, the guards that share variables with the assertion (in any of its transformed state) are the guards interesting regarding the failure of the assertion. To capture this, let an operation op_i be a *guard statement* of f if op_i is a guard $[cond]$ and the intersection of the set of variables in $cond$ and the set of variables in any f' created by variable substitutions from f is not empty.

Similarly to the definition of the set of transforming statements, given a counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$ and a formula f, the *set of guard statements* of the formula is

$$\begin{aligned} GS(\{op_0, op_1, ..., op_{n-1}\}, f) = &\{op_i \mid op_i \text{ is a guard statement of } f \\ &\text{when calculating } WP(\{op_0, op_1, ..., op_{n-1}\}, f)\} \ . \end{aligned}$$

Taking this new set of statement into account, the algorithm should be modified to return both the transformation and guard statements of the formulas in WP_{sub}, or more formally, return $\{s \mid s \in TS(f) \cup GS(f) \text{ if } f' \in WP_{sub}\}$.

Example 3.2. *Take, for example, the C program in Listing 3.1. The assertion is states that* $(a = 0) \land (b \neq 0)$. *This can fail very easily in a single cycle: let a be 1. The rest of the algorithm follows according to Table 3.1.*

Calculating the minimal proof of infeasibility will leave us with $WP_{sub} = (a = 0) \land (a \neq 0)$. *As it can be seen, there is no transformation statement associated with any of the conjuncts in the minimal proof of infeasibility, so the original algorithm would have terminated without giving a single result. In contrast to that, the improved algorithm will "highlight" the condition of the if statement in Line 6, as it is a guard statement of conjunct* $(a \neq 0) \in WP_{sub}$.

Line	WP	$TS(...)$	$GS(...)$	$WP = \emptyset$
8	$(a = 0) \land (b \neq 0)$	no	no	no
7	$(a = 0) \land (b * b \neq 0)$	yes	no	no
6	$(a = 0) \land (b * b \neq 0) \land (a \neq 0)$	no	yes	yes

Table 3.1: An execution of the modified Whodunit algorithm

3.3.2.2 Applying the algorithm iteratively for the whole counterexample

To address the issue of early termination, the algorithm was modified to take an iterative approach. In a nutshell, one iteration of the algorithm takes an assertion and determines the cause. Then it calculates a new assertion to use in the next iteration until the start of the counterexample is reached.

The goal of every fault localization method is to identify statements responsible for an assertion failure. However, a more generalized version of this goal can be to explain the causes of key decisions in the program that ultimately lead to the assertion failure. This corresponds with the idea that the issue might not be a faulty instruction but instead the path that the program ultimately takes for the given inputs.

The algorithm starts by accepting a $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$ counterexample and a c assertion: it also holds, that the last operation op_{n-1} is the guard of the negated assertion $[\neg c]$. The first iteration of the algorithm executes the original Whodunit algorithm on this input. At the end of the iteration, there are two possibilities:

1. The algorithm terminated as it has reached the start of the counterexample.
2. The algorithm terminated as WP became unsatisfiable due to an internal contradiction.

In the first case, the iterative algorithm should terminate as well by reporting the cause. However, in the second case, the question arises: why did that internal contradiction occur? The input of the algorithm is a single (erroneous) path of the program, so the root cause of every issue, including the presence of an internal contradiction, is the path it took. To find the cause of the internal contradiction, the question is why the program would take that path?

The paths in a program are diverging at guards. To answer the previous question, the nearest guard to the place where the previous iteration concluded should be found. The key observation is that by negating this guard, we have a counterexample (the remaining part of the original from the nearest guard), an assertion (the negated guard), and the last operation of the counterexample is the negated assertion. Given these inputs, the Whodunit algorithm can identify the causes of why the counterexample took that path.

After the second iteration has terminated, the same reasoning can be made for a third iteration, then a fourth, and so on. To summarize, the algorithm of Iterative Whodunit can be seen in Algorithm 3.1.

Algorithm 3.1: Iterative Whodunit

```
input  : A counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$, where $op_{n-1}$ is $[\neg c]$
output: Statements causing the assertion failure $c$
1  $ST \leftarrow \emptyset, CEX \leftarrow (s_0, op_0, s_1, ..., op_{n-1}, s_n), C \leftarrow c$
2  while true do
3      $ST \leftarrow ST \cup$ Whodunit($CEX$, $C$)
4      $CEX \leftarrow CEX \setminus$ operations and states processed by Whodunit
5      if end of CEX reached then
6          return $ST$
7      end
8      $l \leftarrow \arg\max_i op_i \in CEX$ is guard $[cond]$
9      $CEX \leftarrow (s_0, op_0, s_1, ..., op_l, s_{l+1}), C \leftarrow \neg cond : op_l$ is $[cond]$
10     if end of CEX reached then
11         return $ST$
12     end
13 end
```

Example 3.3. *Let us take the program in Figure 3.2.b as an example. The assertion fails for $n = 0$, and the counterexample goes through lines 2, 3, 4, 6, 8, 12, 13, and 16. The rest of the Iterative Whodunit algorithm follows.*

In the first iteration, the counterexample is lines 2, 3, 4, 6, 8, 12, 13 and 16, while the assertion is `__assertion_failure` $= 0$. *The first iteration of Whodunit will conclude that the assignment in Line 13 is the cause of the failure. Whodunit processed lines 13 and 16, so they are removed from the counterexample.*

The following assertion will be the nearest guard, which is the guard in Line 12. As a result, the counterexample for the next iteration will be lines 2, 3, 4, 6, 8, and 12, while the assertion will be $s\%2 = 0$ (the negation of the guard). The second iteration will conclude that Line 3 is the cause of the assertion failure (as n is 0, the cycle was not executed). Whodunit processed lines 3, 4, 6, 8, and 12, removing them from the counterexample.

The following assertion will be the nearest guard. However, there are no more guards in the counterexample; the start of the counterexample has been reached, the algorithm terminates.

The output of the algorithm is the union of statements the iterations blamed for the failures: Line 3 and Line 13. After analyzing the program, we can conclude that *the real cause of the issue is indeed the assignment in Line 3 (it should assign an even value for the assertion to hold), which the original algorithm would not have discovered.*

3.3.2.3 Improving the algorithm for finding the minimal proof of infeasibility

The final issue of Whodunit is the ineffective minimal proof of infeasibility algorithm. The original algorithm described in Algorithm 2.1 iterates through all of the conjuncts and checks whether the proof of infeasibility is still unsatisfiable without them: if so, it removes the conjunct; if not, it leaves it in place.

Unfortunately, this algorithm results in multiple calls to an SMT-solver: an SMT-problem will be constructed for each conjunct. Moreover, due to the structure of the SMT problems, interactive solvers do not have an advantage either.

Whodunit uses the minimal proof of infeasibility instead of the proof of infeasibility to make the result more precise: the operation only removes conjuncts, so fewer statements will be reported at the end of the algorithm. However, instead of using the minimal proof of infeasibility, the "almost minimal proof of infeasibility" would also suffice. If it is smaller in the number of conjuncts than the actual proof of infeasibility and only slightly larger than the actual minimal proof of infeasibility, the result would only report a couple more lines to consider. Moreover, the bigger the counterexample is, the less it actually influences the final result of the algorithm.

A way to implement the "almost minimal proof of infeasibility" is to use the unsatisfiable core of the proof of infeasibility. SMT-solvers implementing the unsatisfiable core work towards a minimal unsatisfiable core, even if they do not always find it. On the other hand, an immense advantage of using unsatisfiable cores is that it requires only one call to the SMT-solver: the unsatisfiability of the proof of infeasibility should be checked, and the unsatisfiable core calculated.

Example 3.4. *Let us assume, that* $WP = (x > 5) \wedge (x < 4) \wedge (y = 4)$*. To minimize* WP*, one SMT-problems will be checked:* $(x > 5) \wedge (x < 4) \wedge (y = 4)$*. As it is unsatisfiable, the unsatisfiable core can be queried:* $UC = \{(x > 5), (x < 4)\}$*. Using the unsatisfiable core,* $WP_{sub} = (x > 5) \wedge (x < 4)$*.*

3.3.3 The complete algorithm

The complete algorithm summarizing the changes made to the original Whodunit algorithm can be seen in Algorithm 3.2. The core of the algorithm is the Whodunit algorithm defined between lines 1 and 13. This version calculates guard statements and returns them as part of its answer compared to the original. Moreover, it uses unsatisfiable cores to calculate the "almost minimal proof of infeasibility."

In lines 14-26 is the Iterative Whounit algorithm also introduced in Algorithm 3.1. The Iterative Whodunit executes the improved Whodunit algorithm in multiple iterations: it finds new suitable assertions to continue the reasoning between iterations.

3.3.4 Scoring mechanism

The accuracy of the algorithm can be increased even more by introducing a scoring mechanism into the algorithm. This mechanism will assign a score to each statement of the

Algorithm 3.2: The improved Whodunit algorithm

```
input  : A counterexample $(s_0, op_0, s_1, ..., op_{n-1}, s_n)$, where $op_{n-1}$ is $[\neg c]$
output: Statements causing the assertion failure $c$
1  Function Whodunit(CEX, C) is
2      $WP \leftarrow C$
3      $(s_0, op_0, s_1, ..., op_{n+1}, s_n) \leftarrow CEX$
4      for $i \leftarrow n-1$ to 0 do
5          $WP \leftarrow WP(op_i, WP)$
6          if $WP = \emptyset$ then
7              $WP_{sub} \leftarrow$ UC($WP$)
8              return $\{s \mid s \in TS(f) \cup GS(f)\ \textit{if}\ f' \in WP_{sub}\}$
9          end
10     end
11     $I_{sub}, WP_{sub} \leftarrow$ UC($I \wedge WP$)
12     return $\{s \mid s \in TS(f) \cup GS(f)\ \textit{if}\ f' \in WP_{sub}\}$
13 end
14 $ST \leftarrow \emptyset, CEX \leftarrow (s_0, op_0, s_1, ..., op_{n-1}, s_n), C \leftarrow c$
15 while true do
16     $ST \leftarrow ST \cup$ Whodunit(CEX, C)
17     $CEX \leftarrow CEX \setminus$ operations and states processed by Whodunit
18     if end of CEX reached then
19         return $ST$
20     end
21     $l \leftarrow \arg\max_i op_i \in CEX$ is guard $[cond]$
22     $CEX \leftarrow (s_0, op_0, s_1, ..., op_l, s_{l+1}), C \leftarrow \neg cond : op_l$ is $[cond]$
23     if end of CEX reached then
24         return $ST$
25     end
26 end
```

source code: the higher the score is, the more likely it is that the statement is the cause of the assertion failure.

A common wisdom of software debugging is that the source of the issue is close to its observable effect. Translating it to the language of fault localization, the previous statement means that the cause of the assertion failure is usually close to the assertion itself. The original Whodunit algorithm considered this phenomenon. However, this led to the early termination of the algorithm and caused it to miss the real cause multiple times.

By applying Whodunit iteratively, the result can be lost in the result of multiple iterations. Usually, the statements in fault are the ones returned by the first couple of iterations; later iterations instead serve as additional might-be-useful information.

To incorporate this information into the improved Iterative Whodunit algorithm, each statement will be assigned a score based on the number of iterations required for that statement to be reported. To assign a score based on this information, the mathematical functions i^{-1} or $e^{-}i$ can be used, where i is the number of iterations.

One last issue that needs to be solved is the issue of statements that are in a cycle. Such statements might be returned by multiple iterations as well. In this case, I opted to sum the scores of the different iterations, as a statement returned by multiple iterations might be worth investigating. The iteration-wise decreasing nature of the score ensures that a statement returned by multiple iterations does not suppress other lines worth investigating.

3.4 Implementation

A prototype implementation was created to serve as a proof-of-concept for the method introduced in this chapter (Figure 3.1). The implementation builds upon Theta [32], an open-source, generic, modular, and configurable model checking framework developed by the Critical Systems Research Group of Budapest University of Technology and Economics. The implementation can consume preprocessed C files as input and execute fault localization on them. Moreover, the implementation is integrated with PLCverif [19], a tool supporting the formal verification of PLC programs developed by CERN.

3.4.1 Theta

Theta [32] is an open-source, generic, modular and configurable model checking framework. Theta can receive problems in various engineering formats, including C or statecharts. It converts the input to one of its supported formal representations using a dedicated frontend. Theta supports the formal representations *Control Flow Automata (CFA)* [10], *Symbolic Transition Systems (STS)* [27], or *Timed Automata (XTA)* [9].

The core of Theta is the abstraction-refinement-based CEGAR algorithm. For the sake of extensibility, the CEGAR algorithm is independent of the formalisms. However, it still needs information from the formalisms. This information is provided to the CEGAR algorithm by interpreters (see Figure 3.3), which depend on a specific formalism.

Theta provides a generic SMT solver interface that the analysis algorithms can depend on. This interface supports incremental solving, unsatisfiable cores, and interpolants as well. Typically, the interpreter calls the SMT solver interface when abstract successor states need to be calculated or by the refiner when the feasibility of an abstract path needs to be checked. Theta currently provides only one implementation of the SMT solver interface, which uses an older version of Microsoft's Z3 [20] solver.

3.4.2 Fault localization framework

The formal verification and fault localization algorithms and the approach described in Figure 3.1were implemented in Theta.

The CEGAR algorithm of Theta was given a CFA, with the requirement being that a designated error location is unreachable. The assertion was extracted from the guard leading to the error location. Given a CFA, Theta's CEGAR algorithm can be configured via multiple parameters that are independent of the output of an unsafe program: the counterexample.

The counterexample Theta produces is an alternating list of states and the operations between them, starting from the initial location and ending in the error location. The fault localization method described in Section 3.3 was implemented in Java for easy integration in Theta. The algorithm directly consumes the counterexample produced by Theta and

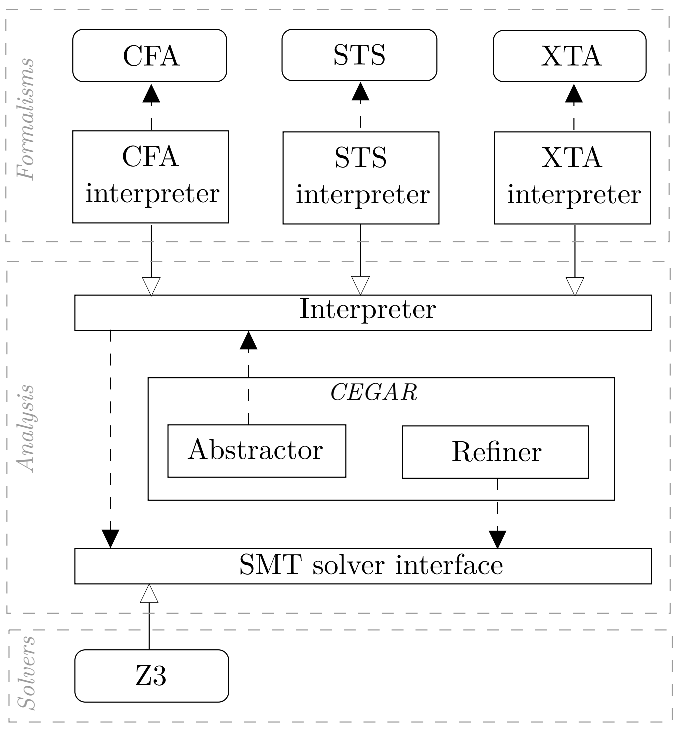


Figure 3.3: The simplified architecture of Theta

outputs a map in which statements from the counterexample are assigned a score (Figure 3.4).

3.4.3 Language frontends

The implementation described in the previous chapter is independent of the language frontend as of yet. It follows that a language frontend needs to provide the following two features:

1. It needs to be able to parse a project in the source language and generate a CFA from it.
2. It needs to be able to process the annotated counterexample and map the scores back to the source code.

3.4.3.1 C frontend

Theta can parse preprocessed C codes and create a CFA from them out of the box via one of its tools. The input of the tool is a preprocessed C file. The tool creates a CFA based on different configurations: the generated CFA might depend on the integer representation mode, among others. Besides the CFA, the tool provides a traceability model that maps each location in the CFA to lines in the source code and each transition to a statement.

The fault localization implementation can extract source code information from the traceability model and use it to map the scores in the annotated counterexample back to the source code level.

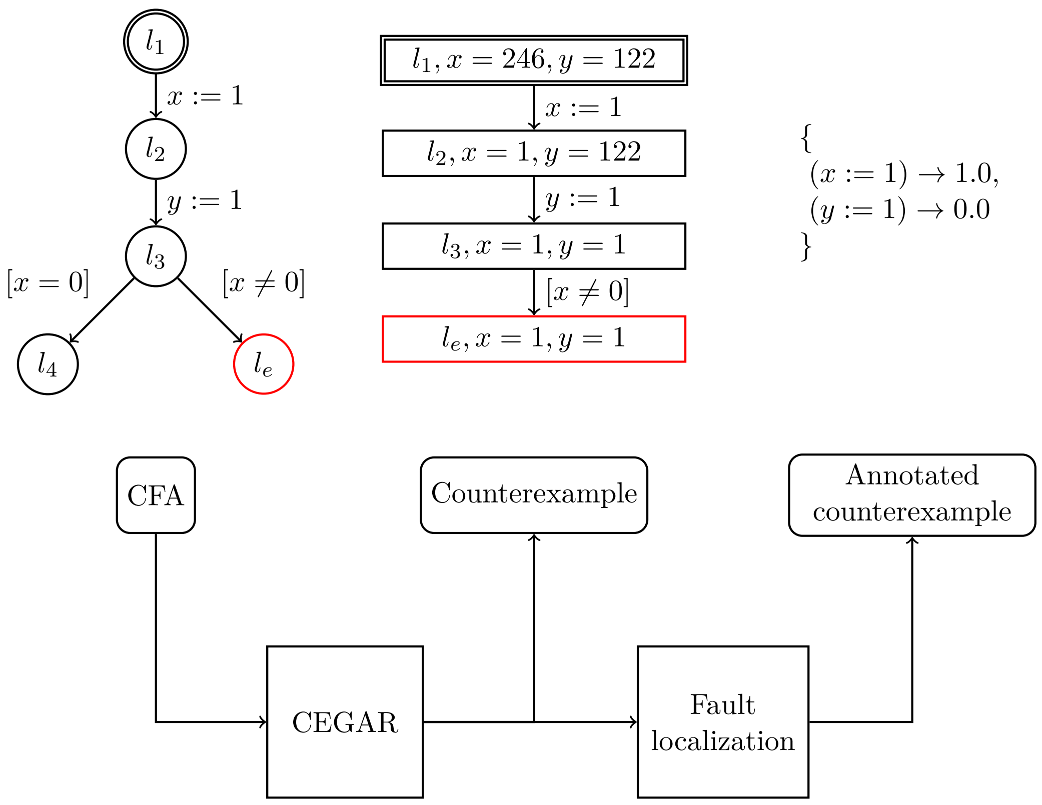


Figure 3.4: The data between the steps of the implementation

3.4.3.2 PLC frontend

PLCverif [19] is a tool supporting the formal verification of PLC programs developed by CERN. Architecture-wise, PLCverif processes PLC projects and generates an internal formal representation of them. Then, the formal representation is converted to the formal representation of one of the supported verification backends: CBMC, NuSMV, or Theta. After the verification succeeds, the result, including the counterexample, is parsed and mapped back to the PLC level.

To do so, PLCverif maintains a traceability model containing a mapping from the internal representation of PLCverif to the formal representation of the backend. Using the traceability model, it is possible to extract source code information and map the annotated counterexample back to the source code level.

3.5 The scope and limitations of the approach

This section analyzes the theoretical scope and limitations of the approach and the additional limitations of the implementation.

3.5.1 Theoretical scope and limitations

To apply the approach, the input program needs to be formalized into a CFA. Variables and control flow are generally easy to formalize; however, there are pitfalls. The common issue tends to be the formalization of pointers in native languages like C and handling compound data structures like structs, unions, or arrays.

CFAs define only simple variables. Thereby, structures are usually decomposed, unions are mapped to a variable with a type capable of handling bitwise operations, and arrays are supported as a custom type. However, the support for pointers is generally a complicated issue to solve, as it requires a complex memory model heavily increasing the required computational power.

The CEGAR algorithm can be applied to multiple formalisms besides CFA. It supports symbolic transition systems or timed automatons as well that produce a similar counterexample. So the question is whether the fault localization method supports counterexamples from different formalisms?

The core of the fault localization algorithm Whodunit is the weakest precondition predicate transformer that also defines the formal semantics of the operations of a CFA. Other formalisms use other kinds of operations, and the formal semantics of these operations can also be defined by the weakest precondition. Generally, if the weakest precondition defines the operations of a formalism, Whodunit can analyze the counterexample produced by CEGAR for that formalism.

3.5.2 Limitations of the implementation

The implementation depends on multiple software components, each contributing its own limitations. Generally, the language frontends only support a part of their language. The C frontend:

- Pointers are not supported.
- Function invocation is supported only by inlining: recursion is not supported.
- Unions are not supported.
- Only standard C files are supported. Pre-ANSI C syntax is not supported.
- Preprocessor directives are not supported.
- Support for floating points is algorithmically limited.

PLCverif has a similar list of limitations. The most notable one is that it only supports the grammar for the Siemens flavor of PLC code (with the grammar for the Schneider grammar being under development).o

Chapter 4

Evaluation

This chapter presents the evaluation of the method presented in this paper. The algorithms are evaluated using a custom implementation with code from a standard fault localization benchmark and industrial code provided to us by CERN.

4.1 Case study

This section presents a case study describing the whole approach. The subject of the case study is the program in Listing 4.1 that reads a number in each iteration of an infinite cycle and performs some operation based on its value. The assertion to check can be seen in Line 8.

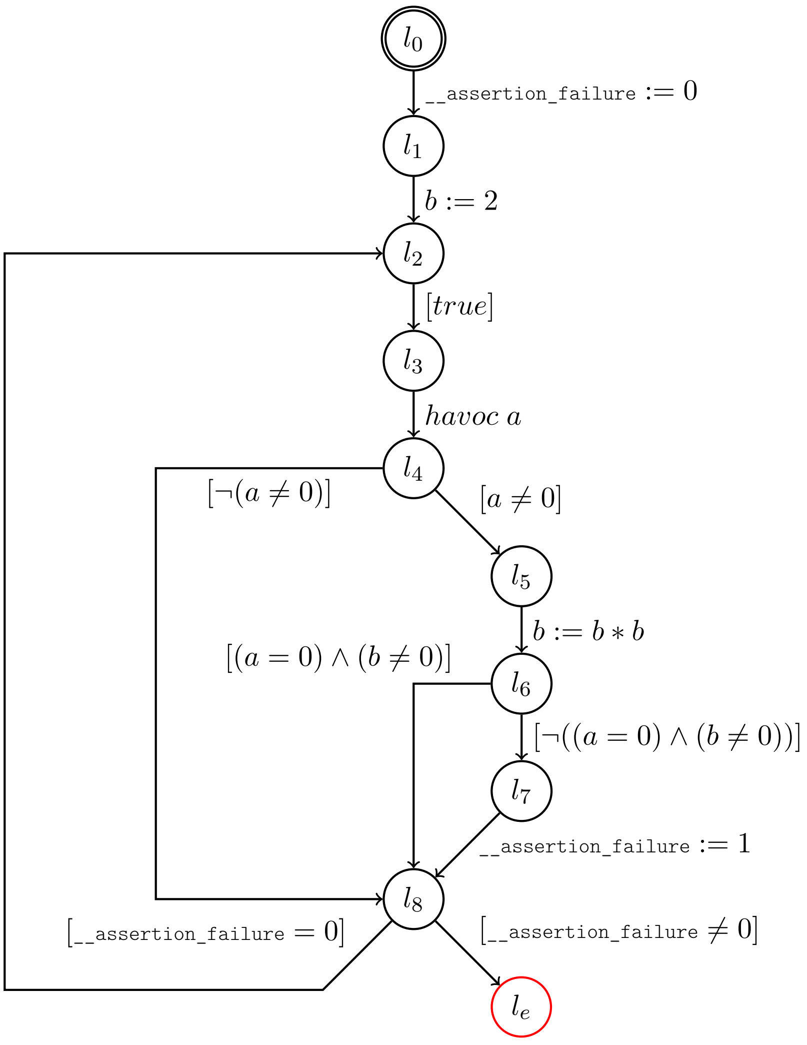
```
1  void main() {
2      int a;
3      int b = 2;
4      while(true) {
5          scanf("%d", &a);
6          if(a) {
7              b = b * b;
8              assert(!a && b);
9          }
10     }
11 }
```

Listing 4.1: An example C program for the purpose of the case study

4.1.1 Providing the CFA

The first step of the approach described in Figure 3.1 is to parse the C code and create a CFA. The CFA of the code in Listing 4.1 can be seen in Figure 4.1.

As it can be seen, the assertion has been modified. The assertion in the CFA asserts that a variable named `__assertion_failure` is zero. This variable is set to zero at the beginning of the program and set to one in the transition between l_7 and l_8. This part of the code corresponds to the original assertion: it is implemented as a small branch in l_6, l_7 and l_8 by setting `__assertion_failure` to one if the original assertion does not hold.



l_0
__assertion_failure $:= 0$
l_1
$b := 2$
l_2
$[true]$
l_3
$havoc\ a$
l_4
$[\neg(a \neq 0)]$
$[a \neq 0]$
l_5
$b := b * b$
$[(a = 0) \wedge (b \neq 0)]$
l_6
$[\neg((a = 0) \wedge (b \neq 0))]$
l_7
__assertion_failure $:= 1$
l_8
[__assertion_failure $= 0$]
[__assertion_failure $\neq 0$]
l_e

Figure 4.1: The CFA generated from the code in Listing 4.1

4.1.2 Executing the CEGAR algorithm

Given a CFA and an error location, the CEGAR algorithm can be executed. CEGAR will determine that the program is unsafe, as the error location is reachable. A suitable counterexample can be seen in Table 4.1.

By checking the counterexample step-by-step, it can be seen that the value of the variable `__assertion_failure` is indeed 1 at the end, so the assertion is violated. It can also be seen, that for values $a = 1$ and $b = 4$ the original assertion (between s_6 and s_7 does indeed not hold.

4.1.3 Fault localization

The modified Whodunit algorithm will consume the counterexample in Table 4.1 to determine the cause of the assertion failure. The result of the algorithm can be seen in Table 4.2.

The fault localization algorithm will require three iterations. The first iteration will determine that assertion `__assertion_failure` $= 0$ assertion fails, as the variable was assigned the value 1 a step before. In the second iteration, the assertion is the negated guard in the

State	Operation	Location	a	b	`__assertion_failure`
s_0		l_0	122	224	86
	`__assertion_failure` $:= 0$				
s_1		l_1	122	224	0
	$b := 2$				
s_2		l_2	122	2	0
	$[true]$				
s_3		l_3	122	2	0
	$havoc\ a$				
s_4		l_4	1	2	0
	$[a \neq 0]$				
s_5		l_5	1	2	0
	$b := b * b$				
s_6		l_6	1	4	0
	$[\neg((a = 0) \land (b \neq 0)]$				
s_7		l_7	1	4	0
	`__assertion_failure` $:= 1$				
s_8		l_8	1	4	1
	[`__assertion_failure` $\neq 0$]				
s_9		l_e	1	4	1

Table 4.1: A counterexample for the CFA in Figure 4.1

Iteration	Counterexample	Assertion	Cause
1	$(s_0, ..., s_9)$	[`__assertion_failure` $= 0$]	`__assertion_failure` $:= 1$
2	$(s_0, ..., s_7)$	$[(a = 0) \land (b \neq 0)]$	$[a \neq 0]$
3	$(s_0, ..., s_3)$	$[false]$	

Table 4.2: Fault localization for the counterexample in Table 4.1

previous operation stating that $(a = 0) \land (b \neq 0)$. The algorithm concludes that the cause of this failure is the condition in the guard created from the condition of the if statement in Listing 4.1: $[a \neq 0]$. The third iteration terminates immediately as the assertion would be $false$, which can never come to pass. As there are no more guards in the counterexample, the algorithm terminates.

In the end, the algorithm returned two statements responsible: `__assertion_failure` $:= 1$ and $[a \neq 0]$. However, they were returned by two different iterations, so the scoring mechanism will assign a different score value to them. Assuming the scoring function is i^{-1} where i is the number of iterations, `__assertion_failure` $:= 1$ will be assigned the score 1, while $[a \neq 0]$ will be assigned the score 0.5 as it is seen in Table 4.3.

Statement	Score
`__assertion_failure` $:= 1$	1.0
$[a \neq 0]$	0.5

Table 4.3: The result of the fault localization

4.1.4 Interpreting the results

To interpret the result, the statements in Table 4.3 need to be mapped back to the source code in Listing 4.1. Using the traceability model created by the model, we can map statement `__assertion_failure` := 1 back to Line 8 (the assertion itself) and statement $[a \neq 0]$ to Line 6 (the if statement).

Based on the result, the most likely cause of the assertion failure is the assertion itself. Ignoring this result, which is present because of how programs are formalized, the next candidate is the if statement. We can see that the condition of the if statement and the assertion are indeed in contradiction, so further changes are required by the developer.

4.2 Evaluating on C programs

The algorithm was evaluated on the Siemens TCAS (Traffic Collision Avoidance System) from the Software-artifact Infrastructure Repository (SIR) that models an aircraft conflict detection system and is commonly used to evaluate fault localization methods. The benchmark is based on a C code of 173 lines that implements a collision-avoiding algorithm. The benchmark also contains 40 variations of the C code, each injected with a fault that causes an assertion failure.

The algorithms were evaluated on whether they were able to find the issue in hand, the number of statements returned, and the required time. To compare the result with the original Whodunit algorithm, the original was run with two iterations to circumvent the early termination issue mentioned in Section 3.3.1. Some results can be seen in Table 4.4 with the variants being randomly selected to be represented here. The entire table can be found in Appendix A.1.

Variant	Original Whodunit			Modified Whodunit		
	Size	Time	Found	Size	Time	Found
v1	6	0.76	yes	14 (8)	0.37	yes
v9	6	0.73	yes	12 (6)	0.38	yes
v10	15	0.72	no (1)	23 (16)	0.39	yes
v14	8	0.73	yes	16 (8)	0.39	yes
v15	11	0.78	yes	20 (13)	0.40	yes
v20	9	0.76	no (2)	17 (11)	0.37	yes
v34	15	0.78	no (1)	25 (17)	0.39	yes
v35	3	0.74	no (1)	16 (4)	0.37	yes
v40	8	0.79	yes	20 (8)	0.38	yes

Table 4.4: The result of the fault localization

The size of the result was measured in lines of code, while the time was in seconds. The *Found* column contains whether the injected error was in the instructions returned by the algorithm. If it was not found, the number in the parentheses represents the distance to the nearest line returned in lines of code. The *Size* column of the Modified Whodunit contains the size of the returned instruction set. In parentheses is the size of the returned instruction set until the iteration that the offending instruction was returned with.

As it can be seen, the original algorithm is not always able to find the faulty lines. On the other hand, the modified algorithm manages to do that. The original algorithm returns fewer lines as the cause of the fault. Although the modified algorithm returns twice as many lines on average, many of these superfluous instructions are part of the result due to the iterative approach. Only considering the first few iterations, the number of returned lines is only slightly larger.

The most significant difference is the time required by the algorithms. On average, the modified algorithm requires half the time required by the original.

4.3 Applying the approach to industrial PLC code

The proposed algorithm and the implementation were also tested using PLC code. CERN provided us with three PLC projects. All three projects contained multiple assertions, but for the sake of computational feasibility, only one assertion was checked at a time. Five assertions were checked altogether: three from one project and one-one from the other two. The criteria for selecting the assertions were that they have to be proven correct by PLCverif.

During the tests, faults were inserted into the source code of the project manually. After that, the formal verification was expected to return an unsafe result, and the fault localization method was used to point out the issue. To compare the result with the original Whodunit algorithm, the original was run with two iterations to circumvent the early termination issue mentioned in Section 3.3.1. Moreover, two kinds of faults were differentiated: one that would be formalized as a faulty deterministic assignment and one that would be formalized as a faulty guard.

Project A and Project B were part of the UNICOS base library, a standard library used by all PLC projects inside CERN. Ensuring the proper functioning of these components is of utmost importance, as a failure in them could affect hundreds of programs. Due to computational feasibility, one assertion was selected from both projects. Project C, on the other hand, is a PLC program implementing a safety procedure. Three assertions were chosen from this project for the test.

Pro.	Size	Ass.	Fault	Original Whodunit			Modified Whodunit		
				Size	Time	Found	Size	Time	Found
A	141	A	assignment	8	0.76	yes	17 (10)	0.37	yes
			guard	0	0.76	no (∞)	17 (10)	0.38	yes
B	768	A	assignment	15	2.41	yes	82 (20)	1.25	yes
			guard	16	2.28	no (5)	86 (33)	1.31	yes
C	2312	A	assignment	25	7.63	yes	217 (31)	3.11	yes
			guard	29	8.03	no (32)	208 (34)	3.67	yes
		B	assignment	25	7.27	no (167)	236 (45)	3.12	yes
			guard	28	7.52	no (34)	212 (22)	3.84	yes
		C	assignment	32	7.42	yes	204 (41)	3.24	yes
			guard	28	7.26	no (47)	187 (35)	3.42	yes

Table 4.5: The result of the fault localization

The results of the testing can be seen in Table 4.5. The size of the project and the result were measured in lines of code, while the time was in seconds. The *Found* column contains whether the injected error was in the instructions returned by the algorithm. If it was not found, the number in the parentheses represents the distance to the nearest line returned in lines of code. The *Size* column of the Modified Whodunit contains the size of the returned instruction set. In parentheses is the size of the returned instruction set until the iteration that the offending instruction was returned with.

As it can be seen, the original algorithm was not able to find the faults injected into guards. However, generally, it was able to find lines close to the fault at hand. In one case (Project A, Assertion A), the original algorithm returned an empty result. Early termination has also hindered the original (Project C, Assertion B, fault injected to assignment).

The Modified Whodunit algorithm managed to always return the offending line. However, the size of the returned instruction set was significantly larger. Considering that the results should be interpreted in the order of decreasing scores, it can be seen that the first couple of iterations always returned the line in question, and it was only slightly higher than the size of the result of the Original Whodunit.

However, the most significant difference was in the time required by the algorithms to execute. Generally, the modified algorithm terminated in almost half the time required by the original, even considering that the modified performed much more iterations.

4.4 Summary

The proposed method and implementation were evaluated using both C programs and industrial PLC codes. All in all, we can conclude that the modified algorithm has better performance, tend to find the causes better than the original, but achieves this at the cost of generally returning more instructions. Another lesson is that in most cases, it is not necessary to run iterations until the start of the counterexample: it is enough to stop after the first few iterations.

However, the results warrant further investigation. It would be beneficial to compare the Modified Whodunit algorithm with other fault localization algorithms and with other standalone tools as well. Moreover, it would be advantageous to check the effect of a maximum number of iterations option to the Modified Whodunit algorithm.

Chapter 5

Conclusion

This paper presented an approach to apply a custom fault localization method for formal counterexamples. The approach uses an abstraction-refinement-based model checking algorithm to produce counterexamples and a weakest precondition-based fault localization algorithm.

The fault localization method is based on an algorithm from the literature. The shortcomings of the algorithm were identified and I extended the algorithm to support better fault localization and to be used in various industrial settings.

A prototype implementation was developed to check the viability of the approach. The implementation is capable of processing C programs and PLC code as well. The approach was tested with a C benchmark designed for fault localization methods and using industrial PLC implementation as well. The result showed that the new algorithm improved in performance, found the cause of the failure more often in return for less precise results.

As a summary, a novel approach was introduced that was successfully applied to fault localization problems on formal counterexamples, and the method was proven to work on industrial code as well.

5.1 Future work

In the future, I plan to work on the limitations of the implementation as well as evaluate the approach more thoroughly:

- The algorithm should be compared to other state-of-the-art methods from literature.
- It should be examined whether there is a need to maximize the iteration count.
- The limited floating-point support should be addressed, as floating points are often used in critical embedded systems.
- The fault localization algorithm should be extracted into a standalone tool so that other formal verification backends of PLCverif may take advantage of it.

Acknowledgements

I would like to thank Jean-Charles Tournier and Borja Fernandez Adiego for supervising my work during my internship at CERN. I would also like to thank Ignacio David López Miguel for helping me a lot with PLCverif during the summer.

Supported by the ÚNKP-21-2 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

This research was funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003).

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[2] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[3] Elvio Amparore, Bernard Berthomieu, Gianfranco Ciardo, Silvano Dal Zilio, Francesco Gallà, Lom Messan Hillah, Francis Hulin-Hubard, Peter Gjøl Jensen, Loïg Jezequel, Fabrice Kordon, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Andrew Miner, Emmanuel Paviot-Adet, Jiří Srba, Yann Thierry-Mieg, Tom van Dijk, and Karsten Wolf. Presentation of the 9th edition of the model checking contest. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 50–68, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17502-3.

[4] Luciano C Ascari, Lucilia Y Araki, Aurora RT Pozo, and Silvia R Vergilio. Exploring machine learning techniques for fault localization. In *2009 10th Latin American Test Workshop*, pages 1–6. IEEE, 2009.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

[6] Levente Bajczi, András Vörös, and Vince Molnár. Will my program break on this faulty processor? formal analysis of hardware fault activations in concurrent embedded software. *ACM Trans. Embed. Comput. Syst.*, 18(5s), October 2019. ISSN 1539-9087. DOI: `10.1145/3358238`. URL `https://doi.org/10.1145/3358238`.

[7] Mike Barnett and K Rustan M Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, 2005.

[8] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[9] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *International hybrid systems workshop*, pages 232–243. Springer, 1995.

[10] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013.

[11] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38. IEEE, 2008.

[12] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 2003.

[13] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[14] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.

[15] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Multiple fault localization with data mining. In *SEKE*, pages 238–243, 2011.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[17] Edmund M Clarke, Thomas A Henzinger, and Helmut Veith. Introduction to model checking. In *Handbook of Model Checking*, pages 1–26. Springer, 2018.

[18] John P. Cleave. *A Study of Logics*. Oxford University Press, 1991.

[19] Dániel Darvas, Enrique Blanco Vinuela, and Vince Molnár. Z3: An efficient smt solver. In *17th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2019.

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

[21] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[22] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[23] Philippe Gayet, Renaud Barillere, et al. Unicos a framework to build industry like control systems: Principles & methodology. *10th ICALEPCS*, 2005.

[24] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electronic Notes in Theoretical Computer Science*, 174(4): 95–111, 2007.

[25] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, pages 1–7, 2009.

[26] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. Minimal unsatisfiable core extraction for smt. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64. IEEE, 2016.

[27] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable cegar framework with interpolation-based refinements. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 158–174. Springer, 2016.

[28] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In *Handbook of Model Checking*, pages 447–491. Springer, 2018.

[29] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 91–100. IEEE, 2011.

[30] David Monniaux. A survey of satisfiability modulo theory. In *International Workshop on Computer Algebra in Scientific Computing*, pages 401–425. Springer, 2016.

[31] Polina Ovsiannikova, Igor Buzhinskyt, Antti Pakonen, and Valeriy Vyatkin. Visual counterexample explanation for model checking with oeritte. In *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 01–10. IEEE, 2020.

[32] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: `10.23919/FMCAD.2017.8102257`.

[33] Chao Wang, Zijiang Yang, Franjo Ivančić, and Aarti Gupta. Whodunit? causal analysis for counterexamples. In *International Symposium on Automated Technology for Verification and Analysis*, pages 82–95. Springer, 2006.

[34] Mark David Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* University of Michigan, 1979.

[35] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8): 707–740, 2016.

[36] Andreas Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

Appendix

A.1 TCAS benchmark data

Variant	Original Whodunit			Modified Whodunit		
	Size	Time	Found	Size	Time	Found
v1	6	0.76	yes	14 (8)	0.37	yes
v2	11	0.79	yes	20 (12)	0.38	yes
v3	15	0.78	no (1)	28 (17)	0.37	yes
v4	13	0.79	yes	21 (13)	0.34	yes
v6	13	0.75	no (1)	22 (15)	0.41	yes
v7	12	0.78	no (2)	24 (14)	0.35	yes
v8	7	0.71	yes	16 (8)	0.37	yes
v9	6	0.73	yes	12 (6)	0.38	yes
v10	15	0.72	no (1)	23 (16)	0.39	yes
v11	14	0.82	yes	23 (15)	0.35	yes
v13	7	0.72	no (4)	16 (8)	0.35	yes
v14	8	0.73	yes	16 (8)	0.39	yes
v15	11	0.78	yes	20 (13)	0.40	yes
v20	9	0.76	no (2)	17 (11)	0.37	yes
v21	8	0.83	no (1)	18 (9)	0.38	yes
v22	10	0.85	yes	21 (10)	0.40	yes
v23	11	0.73	yes	22 (12)	0.34	yes
v26	7	0.79	yes	15 (9)	0.38	yes
v27	6	0.73	no (2)	16 (7)	0.36	yes
v29	10	0.77	yes	19 (10)	0.37	yes
v30	11	0.75	no (1)	23 (14)	0.37	yes
v31	13	0.71	yes	24 (13)	0.32	yes
v32	8	0.82	yes	18 (9)	0.41	yes
v33	11	0.76	no (2)	23 (13)	0.39	yes
v34	15	0.78	no (1)	25 (17)	0.39	yes
v35	3	0.74	no (1)	16 (4)	0.37	yes
v36	12	0.76	yes	21 (13)	0.35	yes
v38	11	0.73	no (2)	20 (12)	0.36	yes
v40	8	0.79	yes	20 (8)	0.38	yes

Table A.1: TCAS benchmark data