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

Formal verification driven test generation in automotive software development

Author:

Mihály Dobos-Kovács

Advisors:

Dr. András Vörös
BME Fault-Tolerant Systems Research Group

Dr. András Balogh
thyssenkrupp Components Technology Hungary Kft.

2019

Table of contents

Table of contents ... **1**

Abstract ... **3**

Összefoglaló ... **4**

1. Introduction ... **5**

2. Background ... **7**

2.1. First-order logic ... *7*

2.2. Formal representation of programs ... *8*

2.2.1. Control Flow Automata ... 8

2.2.2. The state-space of a CFA ... 9

2.2.3. The abstract state-space of a CFA ... 11

2.3. CEGAR ... *13*

2.3.1. Model checking ... 13

2.3.2. CEGAR algorithm ... 15

2.3.3. Building abstraction with predicates ... 16

2.3.4. Refining with predicate abstraction ... 18

2.4. Testing ... *19*

2.4.1. Basics of testing ... 19

2.4.2. Black box testing ... 20

2.4.3. White box testing ... 21

2.5. AUTOSAR ... *23*

2.5.1. Application Software Components ... 24

2.5.2. Runtime Environment ... 25

2.5.3. Developing AUTOSAR components ... 27

3. Related work ... **29**

4. CEGAR driven test generation in AUTOSAR components ... **30**

4.1. Overview of approach ... *30*

4.2. Application of the CEGAR algorithm ... *33*

4.2.1. Terminating the CEGAR loop ... 34

4.2.2. Extracting information from an ARG ... 35

4.3. Test generation *36*

4.3.1. Symbolic execution of the abstract state-space representation 37

4.3.2. Robustness test generation for the untraversed state-space 38

4.3.3. Variable overflow in the state-space 41

4.4. Integrating formal verification in the AUTOSAR development process *44*

4.4.1. Modeling the behavior of a component 45

4.4.2. Writing verifiable requirements 46

4.4.3. Generating the verification environment 46

4.4.4. Transforming test cases 47

5. Evaluation 48

5.1. Implementation *48*

5.1.1. Generating the verification environment 49

5.1.2. Creating the CFA 51

5.1.3. Formal verification 51

5.1.4. Generating test cases 52

5.1.5. Transforming the test cases 52

5.1.6. Limitations of the implementation 53

5.2. Applying the approach to industrial code *54*

6. Conclusion 56

6.1. Future work *56*

Acknowledgment 57

Bibliography 58

Abstract

Nowadays, different kinds of software are responsible for features in safety-critical systems, like cars, airplanes, or nuclear powerplants. Often parts of the systems that used to be mechanical or hydraulical are replaced by software-driven solutions, for example, in the steering of vehicles.

These embedded software components are critical in terms of proper functioning of the system, on the one hand, however, they are quite complex on the other. It follows that certain measures have to be taken to identify and correct the faults of these systems and to prove their correctness. Testing is an efficient way of finding faults, and it is part of every major standard regulating the development of safety-critical systems. However, testing alone cannot prove the absence of errors in a program. Another approach is formal verification that takes the mathematical model of a given software and gives a mathematical proof of correctness. It is a computationally intensive task, as it needs to take all the possible states of a software into account, and even the simplest of programs can have an infinite state space. During the past two decades, researchers have achieved numerous breakthroughs in the field of formal verification; however, due to the complexity of the underlying mathematical field, it is still too early for using formal verification in the industry on a daily basis.

The goal of this paper is to combine these two different approaches in the AUTOSAR environment used heavily in automotive software development. In this paper, an algorithm is presented that uses the results of a verification process to generate tests while taking into account the uniqueness of AUTOSAR components. If the verification succeeds, either there is a mathematical proof of correctness about the software, or there is a counterexample that makes the error reproducible. However, when formal verification fails, tests will be generated using information extracted from the visited part of the state space of the program. In connection with this, multiple strategies are presented for test generation, that are efficient in finding different kinds of program errors. The algorithm will be evaluated using examples from the automotive industry.

Összefoglaló

Szoftverek egyre több kritikus feladatot látnak el biztonságkritikus rendszerekben, mint például autókban, repülőgépekben vagy erőművekben. Sokszor korábbi mechanikus/hidraulikus megoldásokat is beágyazott szoftverrel váltanak ki vagy szoftverrel támogatnak meg, például egy autó kormányművében.

Ezek a beágyazott szoftverek egyrészt kritikusak a rendszer működése szempontjából, másrészt viszont egyre összetettebbek is. Emiatt különösen fontos olyan módszereket használni, amelyek képesek ezen beágyazott szoftverek hibáit megtalálni vagy a helyességüket bizonyítani. A tesztelés hatékonyan, alacsony számítási igény mellett képes hibákat találni a meglévő rendszerekben, valamint a biztonságkritikus-rendszerek fejlesztését szabályozó szabványok alapvető elvárásként tekintenek az mélyretekintő, alaposan dokumentált tesztelésre. Azonban a tesztelés önmagában a helyesség bizonyítására nem alkalmas. Ezzel szemben a formális verifikáció a szoftver matematikai modelljét vizsgálja és matematikailag bizonyítja a különböző hibák elő nem fordulását. A formális verifikáció egy számításigényes feladat, hiszen az algoritmusnak meg kell vizsgálnia a program összes lehetséges viselkedését és állapotát, és még a legegyszerűbb programoknak is könnyen lehet végtelen nagyságú állapottere. Az elmúlt két évtizedben számos áttörést sikerült elérni a formális verifikáció területén, azonban a probléma nehézsége miatt sok esetben nem nyújtanak megoldást.

A munkám célja, hogy ezt a két különböző megközelítést alkalmazzam kombinálva autóiparban használt AUTOSAR környezetben, ötvözve a két módszer előnyeit. Munkám során kidolgozok egy olyan algoritmust, amely a verifikáció eredményeit kihasználja a teszgenerálás során, továbbá kihasználom az AUTOSAR szoftverkomponensek sajátosságait. Sikeres verifikáció esetén a vizsgált komponens helyessége eldöntött, és vagy egy bizonyítás áll rendelkezésre igazolva a helyességet, vagy egy olyan ellenpélda, aminek segítségével a hiba reprodukálható. Amennyiben a verifikáció sikertelen, teszteket generálok, felhasználva a formális verifikáció során a bejárt állapottérből kinyert információt. Ennek kapcsán több különböző tesztgenerálási stratégiát is kifejlesztettem, amelyek különböző típusú hibák megtalálására hatékonyak. A megközelítésemet egy autóipari példán is megvizsgálom és bemutatom a hatékonyságát.

1. Introduction

Nowadays, different kinds of software-driven solutions are becoming part of our lives more than ever. Almost everyone carries a smartphone in his/her pocket, household applications are gaining popularity with the smart home concept, and over the past couple of years, the demand has risen for wearable electronics. Similarly, the industry is using software-driven solutions more-and-more, as it usually tends to be more cost-efficient than the traditional electro-mechanical solutions. It follows that software components became part of almost every industrial system, even part of the so-called safety-critical systems. As a result, ensuring the safety of these systems is imperative, as a fault in them can result in significant financial loss or fatal injuries.

A textbook example for an error leading to a financial catastrophe is the European Space Agency's (ESA) Ariane 5 rocket. Ariane 5's first flight failed on the 4th of June 1996, as the rocket self-destructed 37 seconds after launch. Investigation showed that one component stored the velocity of the rocket as a 64-bit floating-point number, while another component stored it as a 16-bit integer. The conversion between these two formats failed, the rocket lost its ability to navigate, deviated from the route and self-destructed. More than a decade of development, costing about 7 billion dollars was destroyed along with cargo that was worth more than half a billion dollars.

Luckily in the previous example, there was no human casualty. However, the scenario could have been different if the error had happened to be in the central computer of an airplane or a nuclear powerplant.

Safety-critical software components must behave correctly, and ensuring their correctness is an essential factor during development. It follows that specific measures have to be taken to identify and correct the faults of these systems and to prove their correctness.

Testing is an efficient way of finding faults, and it is part of every major standard regulating the development of safety-critical systems. However, testing alone cannot prove the absence of errors in a program, only their presence.

Another approach is formal verification that takes the mathematical model of a given software and gives a mathematical proof of correctness. It is a computationally intensive task, as it needs to take all the possible states of the software into account, and even the simplest programs can have an infinite state space. During the past two decades, researchers have achieved numerous breakthroughs in the field of formal verification;

however, due to the complexity of the underlying mathematical field, it is still computationally too heavy to use formal verification in the industry on a daily basis.

As can be seen, none of the methods above is perfect, and none of them can be used on its own to prove correctness. As in the case of a safety-critical system, safety is of utmost importance, combining these two approaches is an exciting field of study.

The automotive industry has been using software-based solutions to replace the traditionally electro-mechanical parts of the vehicle. One example is the array of sensors and servo-motors that are present in the steering of the vehicle to enhance the driving experience. As the automotive industry has numerous participants, standards have been designed to help reusability and interoperability between the products of different vendors. One of these standards is the AUTOSAR standard that defines a software architecture and development methodology to design and develop automotive software.

The goal of this paper is to combine formal verification and test generation in the AUTOSAR environment. It presents an algorithm that uses the results of a verification process to generate tests while taking into account the characteristics of AUTOSAR components. If the verification succeeds, either there is a mathematical proof of correctness about the software, or there is a counterexample that makes the error reproducible. However, when formal verification fails, tests will be generated using the information extracted from the visited part of the state space of the program. In connection with this, multiple strategies were developed for test generation. The test generation strategies are efficient in finding different kinds of program errors. The approach will be evaluated using examples from the automotive industry.

2. Background

This chapter presents the necessary background to understand this paper, including the formal background and algorithms that are used, and also the AUTOSAR system supporting the development of critical automotive applications.

2.1. First-order logic

Although mathematical logic has several branches, this paper focuses on *first-order logic (FOL)* [1]. First-order logic has great expressive power; however, the satisfiability of a first-order formula is generally undecidable algorithmically. Nonetheless, there are specific *theories* [2] (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)* [3] 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 variable, and the second is an element of the domain of the variable, also called the value of the variable.

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

Specialized software, so-called *SMT solvers* [4] 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, amongst others).

> ***Example 2.1:*** *Given a first-order formula* $(x < 5 \land x \geq 3 \land y > 7)$ *where* $x, y \in \mathbb{Z}$.
>
> *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 $(4 < x \wedge x < 5)$, *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.*

2.2. Formal representation of programs

This chapter presents a formal representation of programs, upon which the formal verification and test generation 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)* [5]. The CFA is a (V, L, l_0, E) tuple, where:

- $V = \{v_0, v_1, \ldots\}$ is the set of *variables* that are present in the program. Each $v_i \in V$ variable has a D_{v_i} domain.
- $L = \{l_0, l_1, \ldots\}$ 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 a set of operations. A transition is a directed edge between two control locations, one or more *operations* labeling each of them. An operation can be:
    - A *deterministic assignment* of a variable, where the value of the right-hand side expression becomes the value of the left-hand-side variable.
    - A *non-deterministic assignment* of a variable, where the value of the variable can be anything valid based in its domain. Non-deterministic assignments are useful for modeling data coming from the user or other programs.

- 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.

```c
void main() {
  int a; scanf("%d", &a);
  int b; scanf("%d", &b);
  while(a != 0) {
    int c = a;
    a = b % a;
    b = c;
  }
}
```

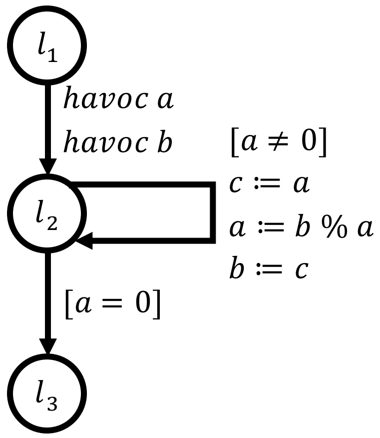

Figure 2.1: *The Euclidean algorithm written in C, and the corresponding CFA*

> ***Example 2.2:*** *On the left side of* ***Figure 2.1****, there is an implementation of the Euclidean algorithm written in C. On the right-hand side 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 ($c \coloneqq a$, $a \coloneqq b\ \%\ a$ and $b \coloneqq c$), and two examples of a guard ($[a \neq 0]$ and $[a = 0]$).*

2.2.2. The state-space of a CFA

Each program has its *state-space*, which is the set of all the possible states, the program can reach and transitions between the states. 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 $(l_i, d_1, d_2, \dots, d_n)$ tuple, where

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

As a CFA can represent a program, there needs to be a way to construct the state-space of the program from the CFA. Given the current state is $(l_i, d_1, d_2, \dots, d_n)$, 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 $v_k \coloneqq expr$, then the following state is $(l_i', d_1, \dots, d_k', \dots, d_n)$, where d_k' is the value of $expr$, in which all variables by substituted by their $d_1, \dots, d_n$ 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 unambiguous. The following state can be $(l_i', d_1, \dots, d_k', \dots, d_n)$, 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.
- If op is a guard $[cond]$, then the following state is $(l_i', d_1, \dots, d_n)$, if $cond$ evaluates to true based on the values $d_1, \dots, d_n$. 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 2.3:*** *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 $x \coloneqq 2$, 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: $\{(l_2, 3, -\infty), \dots, (l_2, 3, 0), \dots, (l_2, 3, 1), \dots, (l_2, 3, \infty)\}$, 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 to do is to determine the initial state of the state-space. The CFA has an initial location which 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#), then there is only one initial state. As the automotive industry tends to use native code, this paper uses the first approach.

2.2.3. The abstract state-space of a CFA

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 case of two 32-bit integer variables in a program, then at least $2^{32} 2^{32} = 2^{64} \approx 10^{19}$ states are needed to be represented. If the program had at least eight integer variables with 32-bit integer domains, then 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.

One possible solution is to use abstraction to remove unnecessary information from the state-space. Multiple abstraction methods are used for CFAs, like predicate abstraction [6] or explicit-value abstraction [6]. This chapter presents predicate abstraction, as the presented approach uses this particular abstraction to handle state space explosion.

The *abstract state-space* of a program is the set of abstract states and transitions between them. An *abstract state* is a set of concrete states, while a *transition* is an operation between two abstract states. One concrete state can appear in at most one abstract state, and every concrete state has to be part of at least one abstract state.

A *predicate* is a logic formula over the set of variables of a program, and it denotes certain relations between the variables. Example predicates are the following:: $p_0 = (x = 0)$ or $p_1 = (y + 2 < x)$. The set of all occurring predicates in the abstraction is called *precision*, and denoted as $P = \{p_1, p_2, \dots, p_n\}$.

If using predicate abstraction, an abstract state is a $(l_i, \widehat{p_1}, \widehat{p_2}, \dots, \widehat{p_n})$ tuple, where $l_i \in L$ is a control location, and $\widehat{p_i}$ is either p_i, $\neg p_i$ or $true$, based on whether the $p_i \in P$ predicate is present in its original form, negated form, or not present at all in the state. In short, an abstract state is a set of states, whose control location is the same, and the predicates evaluate to true on the variables in the state. The predicates that are present in the state are said to label the state.

> ***Example 2.4:*** *Let the state space of a program be* (l_1, x), *where* $D_x \in [-\infty; \infty]$. *Given the abstract state:*
>
> - $(l_1, x < 0)$, *the set of states it abstracts is* $\{(l_1, -\infty), \dots, (l_1, -2), (l_1, -1)\}$.
> - $(l_1, \neg(x < 0))$, *the set of states it abstracts is* $\{(l_1, 0), (l_1, 1), \dots, (l_1, \infty)\}$.
> - $(l_1, true)$, *the set of states it abstracts is* $\{(l_1, -\infty), \dots, (l_1, 0), \dots, (l_1, \infty)\}$.

The rules of constructing an abstract state-space based on a CFA differ slightly from the rules of constructing a concrete state-space. First of all, if there are no variables with value at the beginning of the program, all the possible initial states can be abstracted into a single abstract state, as they all share their control location (l_0), which is the initial location of the CFA.

Given that the current state is $(l_i, \widehat{p_1}, \dots, \widehat{p_n})$, and a transition (l_i, op, l_i') that leaves the control location l_i in the CFA, then the following state can be calculated based on op:

- If op is an assignment in the form of $v_k \coloneqq expr$, then the control location of the following state is l_i', and the predicates of the following states are those predicates (or their negated form) from P, which are implied by the predicates of the current state, and the assignment.
- If op is a non-deterministic assignment in the form of $havoc\ v_k$, then the control location of the following state is l_i', and the predicates of the following states are those predicates (or their negated form) from P, which are implied by the predicates of the current state and the assignment. These predicates obviously cannot contain information on v_k, as no data is available about the value except its domain.
- If op is a guard $[cond]$, then it should be first decided whether there is a contradiction between the predicates of the current state and the condition. If there is a condition, then the values of the variables cannot be chosen so that both the condition and the predicates evaluate to true, thus, the transition cannot be executed. If there is no contradiction, then the control location of the following state is l_i', and the predicates of the following states are those predicates (or their negated form) from P, which are implied by the predicates of the current state and the guard.

In practice, as long as both the predicates and the operations on the CFA can be expressed as first-order formulas, an SMT solver can be used to check for contradiction and to calculate implications [7].

> ***Example 2.5:*** *Let the current abstract state be $(l_1, x > 0, y < 4)$, where $D_x, D_y \in \mathbb{Z}$. Moreover, let a transition be (l_1, op, l_2). Based on op:*
>
> - *If op is deterministic assignment $x \coloneqq x + 1$, then the following state is $(l_2, x > 1, y < 4)$, as $(x > 0) \wedge (x \coloneqq x + 1) \rightarrow (x > 1)$.*
> - *If op is non-deterministic assignment $havoc\ x$, then the following state is $(l_2, true, y < 4)$, as no information is available about the new value of x.*
> - *If op is guard $[x > 3]$, then the following state is $(l_2, x > 3, y < 4)$, as $(x > 0) \wedge (x > 3) \rightarrow (x > 3)$.*
> - *If op is guard $[x < 0]$, then the transition cannot be executed, as there is no integer for which $(x > 0) \wedge (x < 0)$.*

> *All of the implications above are first-order formulas, so they can be fed to an SMT solver which solves them.*

2.3. CEGAR

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

2.3.1. Model checking

Given a formal model and a formal requirement (or statement), *model checking* [8] [9] 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 to be noted that the proof of unsafeness is often an example, for which the requirement fails.

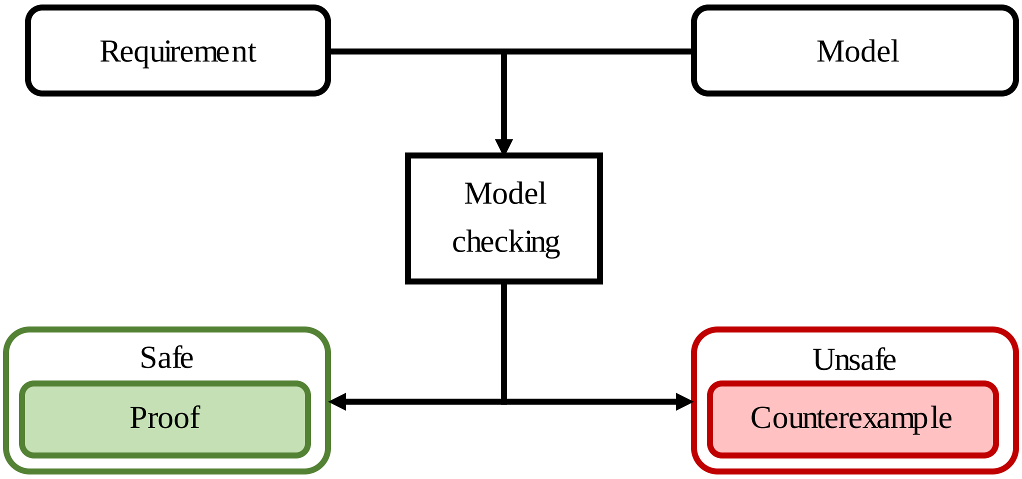

Figure 2.2: *The model checking procedure*

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 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.

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

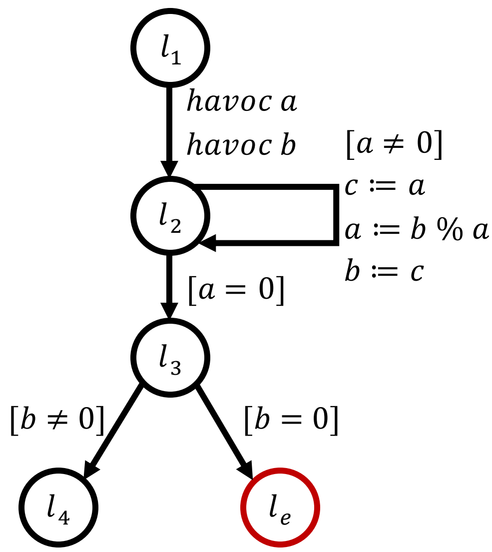

Figure 2.3: *The Euclidean algorithm with an assertion written in C, and the corresponding CFA*

> ***Example 2.6:*** *On the left side of* ***Figure 2.3****, 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 algorithm

The *Counterexample-Guided Abstraction Refinement (CEGAR)* [5] [10] [11] 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.

As the algorithm uses abstraction, it operates in the abstract state-space. 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.

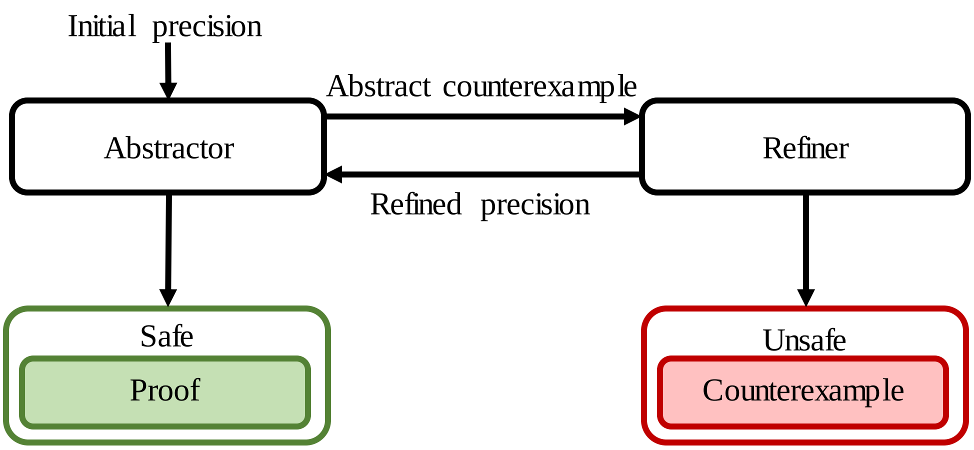

Figure 2.4: *The CEGAR-loop*

The core of the algorithm is the so-called CEGAR-loop (**Figure 2.4**) 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 and checking 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, it needs to be decided whether it is feasible or spurious. If a concrete error-state inside of it is reachable, then the abstract error-state is *feasible*, so the model fails the requirement. If a concrete error-state is not reachable, then the abstract error-state is *spurious*, the reachability of the abstract error-state is the result of the over-approximation. In this case, the abstraction needs to be refined, so that the abstract error-state does not contain the unavailable error-state. Checking the reachability of the concrete-error-state and refining is the task of the refiner part of the CEGAR-loop.

The loop keeps repeating itself until it either proves that no abstract error-state is available, thus, the requirement holds or gives an example 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 to be noted that the CEGAR algorithm does not depend on the type of abstraction. It can use predicate abstraction just as easily as explicit-value abstraction. The following sections present how predicate abstraction is combined with CEGAR.

2.3.3. Building abstraction with predicates

When using predicate abstraction [6], there is a set of predicates given, upon which the abstraction is based. The set of predicates is also called precision and denoted with P. Building the abstraction requires two operations and multiple definitions.

Expand is an operation, which given an abstract state, calculates the set of following abstract states. It takes all transitions that leave the control location of the given abstract state and forms a set from the destination of those transitions.

An *Abstract Reachability Tree (ART)* is a tree in which the nodes represent abstract states, and the edges denote the transitions between them. The root of the tree is the abstract state of the initial location of the CFA, and every state is either a leaf or the set of its children is the result of the expand operation called with the state.

Given a not yet expanded node whose abstract state is $L = (l_l, p_i, \dots, p_j)$ in the ART, and another node whose abstract state is $S = (l_s, p_k, \dots, p_l)$ for which $l_l = l_s$ and $(p_i, \dots, p_j) \rightarrow (p_k, \dots, p_l)$, where $\rightarrow$ stands for implication, then S *covers* L (or L is *covered by* S). Illustratively, it means that if a control location occurs at least twice in abstract states of the ART (let us call these nodes L and S), and one node, L is not yet expanded but its states predicates are stricter than the others, S's, that is expanded, then there is no state of the abstract state-space that is available from L, but not available from S. It also follows that L does not need to be expanded.

An *Abstract Reachability Graph (ARG)* is a directed acyclic graph, whose nodes are the nodes of an ART, and whose edges are the union of the edges of the ART, and the covering edges. A covering edge from L to S nodes denote, that L covers S. A node in the ARG is *complete* if another node covers it, or it is expanded. All other nodes are *incomplete*.

Cover is also an operation, which creates a covering edge in an ARG between L and S if L covers S.

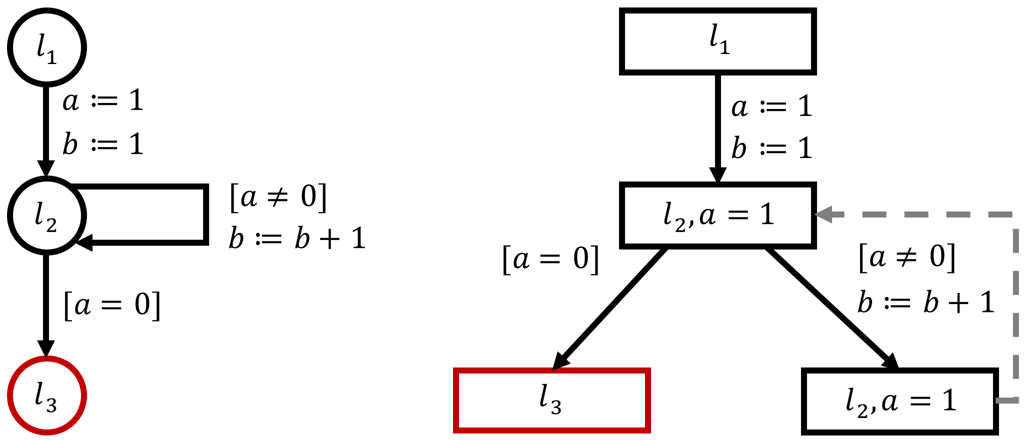

Figure 2.5: *A CFA and a corresponding ARG*

> ***Example 2.7:*** *On the left-hand side of* ***Figure 2.5*** *is a CFA, and on the right-hand side is (one of the many possible) corresponding ARGs. The result of the operation expand on the abstract state l_1 is $\{l_2\}$, as it has only one following state. On the other hand, the result of operation expand on l_2 is $\{l_3, l_2\}$, as two different transitions are possible. It can be seen, that both instances of l_2 are labeled with $(a = 1)$, and because they share the same control location, and $(a = 1) \rightarrow (a = 1)$, the latter is covered by the former, which is denoted by the dashed arrow. The state of l_3 is an abstract error state, and it is unreachable, as $(a = 1) \wedge (a = 0) \rightarrow \perp$, so it should be removed from the ARG. All of the nodes are complete because every one of them is either expanded or covered, so no abstract error-state is reachable, so the model is safe for the given requirement.*

The abstraction building procedure (**Listing 2.1**) builds an ARG. It starts with a single abstract state that represents the initial location in the CFA. In the followings, it calls the expand and cover methods systematically, until either all the nodes are complete in the ARG, or an abstract error-state is encountered in one of the nodes. In the former case, the model is safe, as it contains no error-state, while in the latter case, it needs to be determined whether the abstract error-state is feasible or spurious, but is the task of the refiner.

```
Abstractor(CFA, P):
  ARG := ARG with initial location
  FOREVER:
    s ∈ {incomplete nodes of ARG}
    IF ∄s THEN:
       ← SAFE
    ELSE IF ∃s and s is an error location THEN:
       ← COUNTEREXAMPLE(route from initial location to s)
    ELSE:
      IF ∃r ∈ {nodes of ARG}: r covers s THEN:
        cover s with r
      ELSE:
        expand s
```

Listing 2.1: *The algorithm building the algorithm*

The algorithm is highly customizable, as it can be seen in the listing above. It can apply different strategies as to how to select the next candidate for expansion or the next candidate to check for a covering relation. Tuning these two methods is an active field of study.

2.3.4. Refining with predicate abstraction

The procedure of refining (**Listing 2.2**) is executed for an abstract error-state and a given set of predicates, the precision.

First, it needs to be checked, whether a concrete error-state in the abstract error-state is reachable or not. If it is, then the error path is feasible, and there is an error in the model; otherwise, it is spurious, and the error is a result of over-approximation.

One way to achieve this is to form an SMT problem from the assignments and guards on the route from the root of the ARG to the concrete error-state. If this problem is satisfiable, then there is a substitution of variables to concrete values that leads to this error, so the error-state is reachable; thus, the model fails the requirement, and the example for that is the substitution.

However, if the problem is unsatisfiable, then the abstract error-state is spurious, and the precision needs to be refined. There are multiple strategies to achieve it, one of these strategies, for example, uses the proof of unsatisfiability (an interpolant) to deduce more predicates [12].

```
Refiner(Abstract Counter Example, P):
  R ≔ concrete route from initial location to error state
  s ≔ R as SMT-problem
  IF s is satisfiable THEN:
    ← UNSAFE(R)
  ELSE:
    P′ ≔ P ∪ {new predicates}
    ← REFINED PRECISION(P′)
```

Listing 2.2: *The refiner algorithm*

As a final step, the states that are unreachable need to be removed from the state-space and the ARG; in other words, the ARG has to be *cut back*. After finishing the refinement, the abstraction needs to be rebuilt based on the new precision.

2.4. Testing

Testing is a generic method to check the validity of computer programs. The industry widely uses it, and it is required by almost all the standards regulating the development of safety-critical systems. However, the state-space of a program is usually huge, while one test case is only an arbitrary choice of inputs, denoting a path in the program. So there needs to be a method to choose those inputs that lead to an error with the highest possibility, also called testing methods.

2.4.1. Basics of testing

Testing [13] is a complicated method. This section presents a simplified approach that fits the goals of this paper. The program is tested by a single *test suite* that consists of multiple test cases. This program is called *software-under-testing (SUT)*. A *test case* is tuple of:

- Pre-conditions
- Input values and actions to take
- Target values
- Post-conditions

When executing a test case, first, the list of pre-conditions is checked, and the test case is only executed if they hold. After the set of input, values are given to the SUT, then the actions required to be carried out are run. After execution, the output of the program is checked whether it matches the expected target values, and the post-conditions are checked if they hold. The result of each test case can be:

- *Successful*: the run of the program is consistent with the expected results and post-conditions
- *Failure*: the run of the program is inconsistent with either the expected results or post-conditions
- *Inconclusive*: the run of the program is inconsistent with both the requirements of a successful run and a failed run. One example is that it is given a set of both successful and failing outputs, but the output of the program is not an element of any of them.
- *Error*: there was an unexpected error while executing the test, so it cannot be decided.

Coverage metrics often measure the quality of the individual test-suites. These metrics measure that which lines of codes, or which branches of codes are executed during the test suite. It is often a requirement by safety-critical standards that the test-suites have a near 100% coverage.

2.4.2. Black box testing

A black box testing technique is a method that derives test cases from solely the specification of the program. There are several black box techniques, so only those are presented that are used use in this paper.

One example is the *equivalence partitioning* [13]. The domain of each variable is split up to multiple intervals, or multiple sets of values so-called equivalence partitions. For each interval or set, the program indeed behaves the same way for every value in it (assuming that all the other variables remain unchanged). When using equivalence partitioning, one test case is derived for every interval by selecting a random value from it.

Another common technique is *boundary value analysis* [13]. It assumes that the faults in the program happen more often around the boundaries because often, unique code is required to handle them. Each variable has a domain, which in turn have a minimum and a maximum value (the other values are called nominal values). There are multiple methods as to what values are usually useful for testing, but usually, the following values get to be chosen:

- A bit below the minimum value (MIN-)
- The minimum value (MIN)

- A bit above the minimum value (MIN+)
- A nominal value (NOM)
- A bit below the maximum value (MAX-)
- The maximum value (MAX)
- A bit above the maximum value (MAX+)

With these seven test cases, the domain of a variable is tested for all the possible kinds of values, assuming the software behaves the same for all nominal values.

However, usually, the two methods above are combined and used hand-in-hand. First of all, there are variables with discrete domains (for example, enumerations), for which boundary value analysis cannot be used, only equivalence partitioning. Moreover, an equivalence partition is tested better, if its boundaries are also checked, so usually, five test cases are generated for each of them: MIN, MIN+, NOM, MAX-, MAX+. The other two test cases are only needed if there are gaps between the partitions, so a MIN- or a MAX+ value does not belong to any of the partitions.

> ***Example 2.8:*** *A food delivery service allows the users to order up to 10 portions of food. However, their website, on which the order is placed, the input filed accepts any integer numbers. An additional code component checks whether the number of portions is in the right range, and this component is tested with equivalence partitions and boundary value analysis.*
>
> *The set of integers can be split into three different equivalence partitions, based on the requirement:*
>
> - *Invalid 1:* $[-\infty; 0]$*, the order fails to complete*
> - *Valid:* $[1; 10]$*, the order is successful*
> - *Invalid 2:* $[11; \infty]$*, the order fails to complete*
>
> *Applying boundary value analysis on the valid partitions results in the following inputs:* 1 *(MIN),* 2 *(MIN+),* 5 *(NOM),* 9 *(MAX-),* 10 *(MAX).*
>
> *Additionally, at least two other test cases are recommended from the invalid partitions,* 0 *(MIN-), and* 11 *(MAX+). Of course, other invalid values supposed to be tested as well.*

2.4.3. White box testing

White box testing techniques know the internal structure of the SUT and derive test cases solely based on the structure, ignoring the semantics of the variables. On the one hand, this could prove to be a disadvantage; however, these kinds of methods can usually be automated, so another tool generates the test cases, without human interaction. This way, many more tests can be generated and executed, compared to merely human written tests.

One of the most widely known white box technique is *symbolic execution* [14] [15]. It executes the program, but instead of remembering the exact values of the variables, it records symbolic values. All the expressions and operations are evaluated, then with these symbolic values, rather than concrete ones.

A symbolic value represents a mathematical constant, that can have the value of anything in the domain of the variable. Symbolic execution maintains two distinct data structures:

- A symbolic state (δ) that maps the variables of the program to their current symbolic value.
- A path constraint (π), that is a first-order formula over symbolic values and decodes the path the execution is taking.

Symbolic execution also takes a CFA as a starting point. In the beginning, the execution starts at the initial location, and $\delta = \{\}$ and $\pi = \top$. After that the algorithm takes all the transitions from that location, using depth-first search and if in (l_i, op, l_i') transition:

- op is non-deterministic assignment $havoc\ x$, then the value of x in the symbolic state must be a never before used symbolic value x_i, so $\delta(x) = x_i$.
- op is deterministic assignment $x \coloneqq expr$, then the symbolic state must be updated with the new value, so $\delta(x) = expr'$, where $expr'$ is expression $expr$, with all the variables substituted for their symbolic value.
- op is guard $[cond]$, then the path constraint needs to be updated, so $\pi' = \pi \wedge cond'$, where $cond'$ is expression $cond$, with all the variables substituted for their symbolic value.

The execution continues until it reaches the end of the execution. After that, by solving the path constraint with an SMT solver, the result is either satisfiable or unsatisfiable. In the former case, the input values of the program that guide the execution on the path described by the path constraint can be extracted from the solution, and a test case can be created that tests this path. In the latter case, this path is unfeasible. After that, the algorithm backtracks and tries other paths (for example, other branches of an if-then-else structure), also solving the path constraint at the end.

This way, a test case can be derived for all possible paths that the execution can take. However, this number can quickly become huge, even infinite, in case of cycles. This phenomenon is called *path-explosion* and renders symbolic execution on its own rather useless against industry software. Path explosion is usually tacked by tricky heuristic techniques that cut back the possible paths in case of a cycle, like pruning redundant paths or interleaving random and symbolic execution [15].

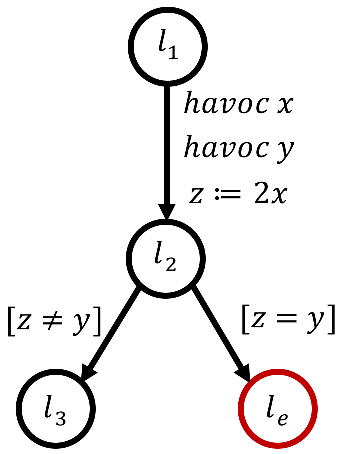

Figure 2.6: *A CFA with a branch*

> ***Example 2.9:*** *Let us apply symbolic execution on the CFA in* ***Figure 2.6****. At the start* $\delta = \{\}$, $\pi = \top$, *and the execution is at* l_1. *Then:*
>
> - *Symbolic execution moves to* l_2 *and executes the operations on the transitions. As a result of the two non-deterministic assignments, two new symbolic values will be introduced,* x_1 *and* y_1. *As a result of the deterministic assignment, the variable will be associated with the symbolic value of the expression on the right-hand side. At the end* $\delta = \{x \to x_1, y \to y_1, z \to 2x_1\}$, $\pi = \top$.
> - *First, the left branch is taken, and the symbolic execution takes* l_3. *As there is only one guard, only the path constraint is updated:* $\pi = (2x_1 \neq y_1)$. *The path ends here, so the path constraint should be given to an SMT-solver. One possible model is* $\{(x_1 = 1), (y_1 = 1)\}$. *So giving* x *the value* 1 *and* y *the value* 1, *the execution follows this path.*
> - *Next, the algorithm backtracks to the nearest decision and executes the other branches, which is* l_e *in this case. As there is only one guard, only the path constraint is updated:* $\pi = (2x_1 = y_1)$. *The path ends here, so the path constraint should be given to an SMT-solver. One possible model is* $\{(x_1 = 1), (y_1 = 2)\}$. *So giving* x *the value* 1 *and* y *the value* 2, *the execution follows this path.*
> - *Then the algorithm backtracks, but there are no additional branches, so it terminates while emitting two test cases.*

2.5. AUTOSAR

Automotive software development is a diverse industry with many participants. To improve interoperability and reusability, multiple interested parties, like BMW, Bosch, or Volkswagen, founded a development partnership in 2003. This partnership created the *Automotive Open System Architecture (AUTOSAR)* [16], which is open, and more importantly, standardized software architecture for automotive *electronic control units (ECU)*. Besides the architecture, it also sets goals for reusability, availability, safety, and maintainability reducing the costs of research and development.

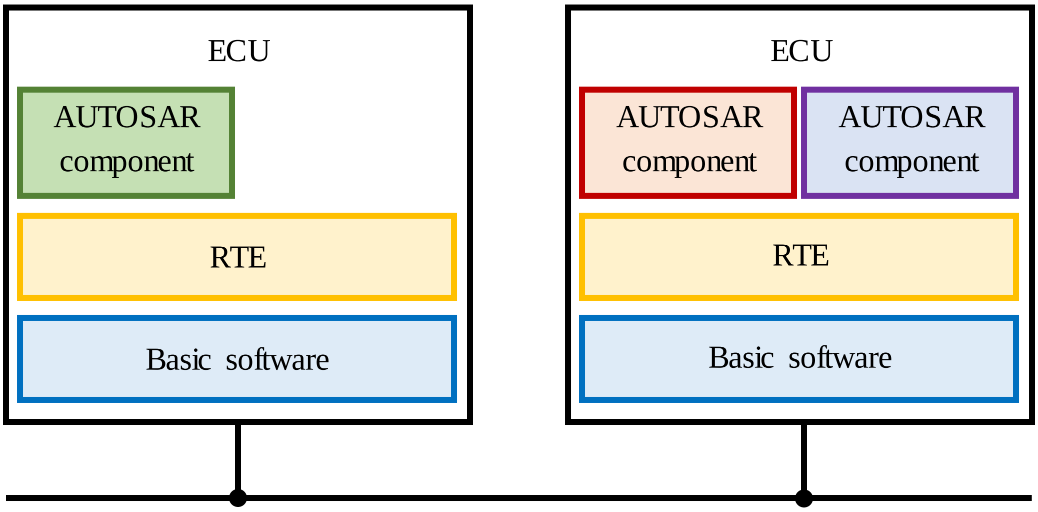

Figure 2.7: *The layers of AUTOSAR*

Over the past two decades, the standard had multiple revisions and has been used all over the world. AUTOSAR defines a software architecture with three layers (**Figure 2.7**):

- Basic Software (BS): it consists of standardized software components, that provide functionalities to the upper layers
- Runtime Environment (RTE): it is a middleware that abstracts the hardware topology, providing connections between application components disregarding if they share the same ECU or not. It also provides an interface to BS components.
- Application Layer: these are application software components, that provide unique functionality, and the focus of this paper.

The following sections present only the subset of the AUTOSAR standard, which is required by this paper.

2.5.1. Application Software Components

An *application software component* or *AUTOSAR component* is a piece of software that has a standardized interface and standardized inner-structure. This chapter presents only the part of the basic structure of a component that is used in the paper.

AUTOSAR components communicate with the rest of the world through well-known ports, that encapsulate interfaces, ensuring type-safety across components. There are multiple types of ports, the two most important are:

- *Client-server ports* define a set of operations that can be invoked. Client-server communications consist of a server component that defines the operations to be invoked and multiple client components that invoke the functionality of the server. It is worth to be noted that data can flow in both directions when invoking an operation. This kind of communication is synchronous, as the clients wait for the answer.
- *Sender-receiver ports* define an asynchronous type of communication between components, where the sender port sends a message, and multiple receiver port receives it.

Each component can have *parameters* that contain data that can be configured but does not change during the lifetime of the component. Thereby parameters help to create reusable code, as the same code or algorithm can be easily reused, but still configured according to individual needs.

The main elements of the internal structure are the *runnables*. These are pieces of code that realize the functionality of the component. Each runnable must be associated with at least one *event*, that runs the runnable as an action when triggered. The trigger of an event can be (amongst many other):

- Timed, which can trigger the event periodically, or after a specific time
- Calling an operation of a client-server port

AUTOSAR also requires to declare all the memory that stores data for persistency, which is called *per instance memory*. It is required because, in safety-critical environment, dynamic memory allocation is forbidden, so everything has to be declared that cannot be stored on the stack because, for example, it needs to persist data between two executions of a runnable.

Moreover, AUTOSAR expects the developer to annotate the component with metadata that store (amongst others), which runnable can access which ports, parameters, and per instance memories and the domain of every variable in the interface. This information can be useful for verification if used correctly.

2.5.2. Runtime Environment

AUTOSAR components communicate with other components through their ports. It is the task of the *RTE* as a middleware to connect communicating components and to provide access to the functionality of the BS if needed. However, the main task of the

RTE is to hide the hardware-dependability of the communication. It also hides whether the communicating parties share the same ECU or not.

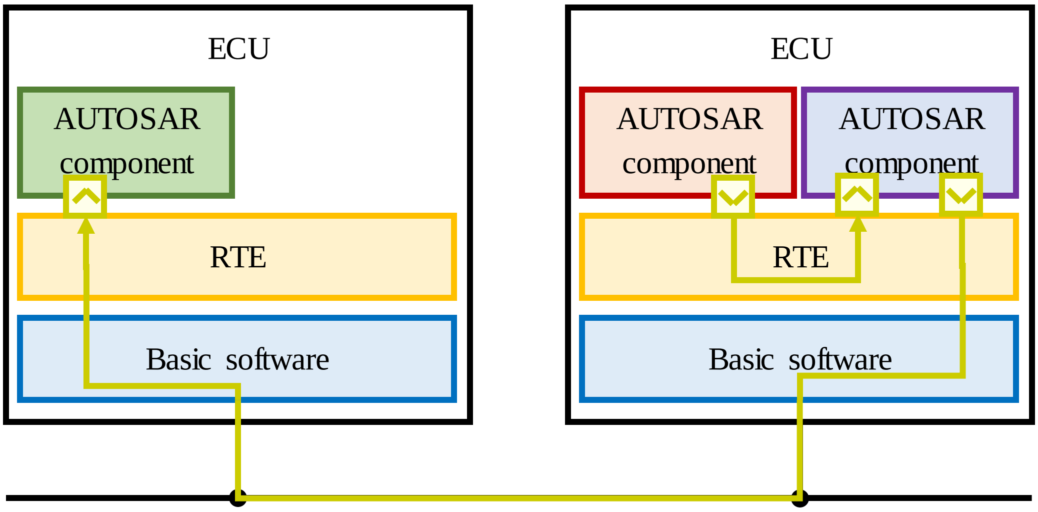

Figure 2.8: *The communication paths between components*

This notion is called the *Virtual Function Bus (VFB)*, that in short, is responsible for connecting the communicating parties. There are two kinds of connections between components, that the VFB abstracts:

- Intra-ECU: the same ECU runs the communicating components, so the components share the CPU and memory.
- Inter-ECU: different ECUs run the communicating components, so the components do not share CPU or memory

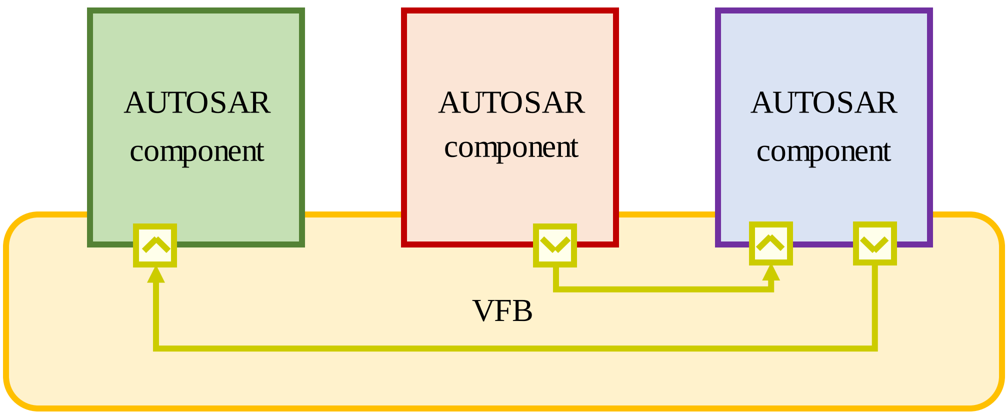

Figure 2.9: *The virtual function bus*

The communication methods of components are depicted in **Figure 2.8**. In the case of Intra-ECU communication (between red and purple components), the RTE of that particular ECU is responsible for connecting the communicating parties. However, in the case of Inter-ECU communication (between purple and green components), the RTE forwards the communication to the BS, which puts it on the bus connecting the communicating ECUs. The BS of the other ECU parses the communication from the bus, and the RTE of the other ECU forwards it to the correct port.

The RTE completely masks this difference; the components perceive only the VFB, which forwards all communication. This phenomenon is portrayed in **Figure 2.9**.

2.5.3. Developing AUTOSAR components

The development of an AUTOSAR component usually follows a rigid waterfall or V-model methodology, as safety-critical systems often do. It has a distinct requirement design and model design phase, then coding and testing.

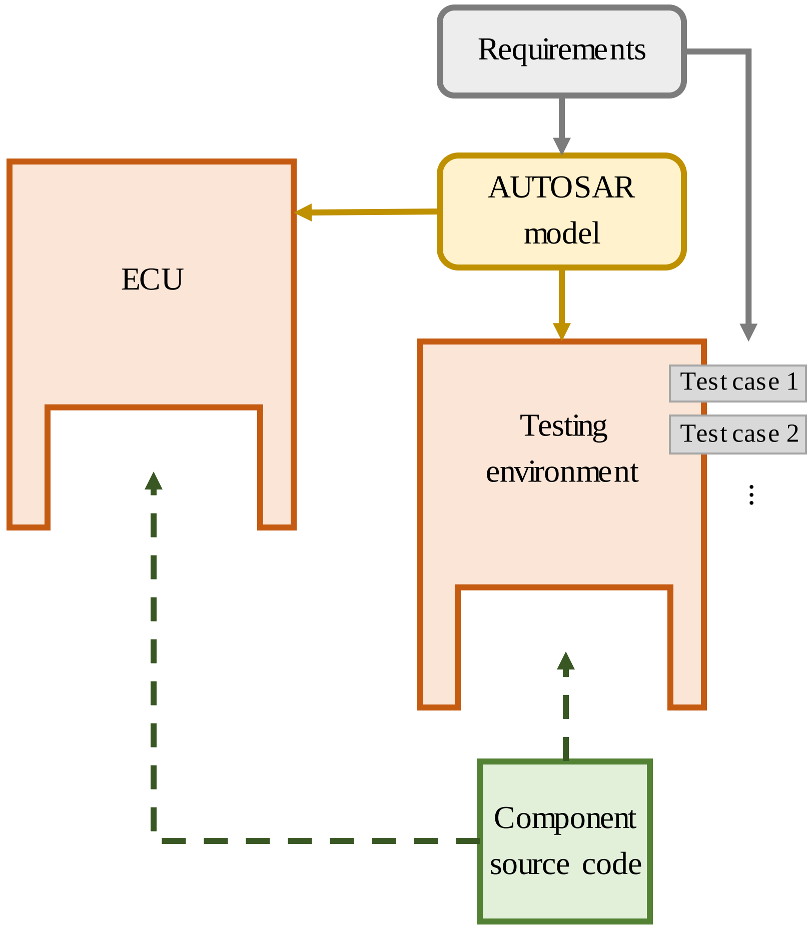

Figure 2.10: *The (simplified) development process of an AUTOSAR component*

The development process starts with creating an AUTOSAR model. The model describes the defined ports, parameters, per instance, memories, events, runnables, system and ECU configurations, and other metadata of the component.

Next, the source code of the runnables can be written in native C code. After that, the testing phase can begin. Testing requires a testing environment, that can mock de behavior of the RTE, by making the developer able to set and check the values of ports, parameters, amongst others. This environment can easily be generated based on the AUTOSAR model, so only the source code of the tests has to be written.

After the testing phase is finished, the component is compiled and deployed to an ECU, where additional testing takes place. Additional code that is required to configure the ECU can also be generated from the model.

The development process can be seen in **Figure 2.10**. It shows that both the model, and the test cases are derived from the requirements, the ECU and testing environments are generated from the AUTOSAR model, and both environments run the same source code.

3. Related work

The algorithm proposed in this paper can be approached from different perspectives. First, it is an attempt to combine formal verification with test generation, and second, it is an attempt to apply formal verification tools for automotive software.

This paper is mainly based on the author's Scientific Students' Association Report in 2018 [17]. That approach presented a working solution for combining formal verification and test generation. Compared to that approach, this paper presents improved test generation methods, especially in terms of variable overflow.

One of the attempts by Maria Christakis et al. in 2016 [18] tried guiding dynamic symbolic execution towards unverified program paths and achieved impressive results. In contrast, this paper uses symbolic execution rather than dynamic symbolic execution, as the latter often requires special instrumentation. Moreover, that approach did not focus on the type of verification algorithm.

Another approach was published by Mike Czech et al. in 2015 [19] that combined formal verification with testing. Their approach tried running a formal method on a program than tried to generate another program, that only represented the unverified part of the state-space. Later on, the newly generated software was fed to test generation tools. In contrast to that approach, this paper does not generate intermediate software, as it possibly could lead to losing information about the state-space. Instead, it uses the state-space representation directly to generate tests, requiring new kinds of test generation methods.

In terms of verifying automotive software, there are numerous attempts [20] [21] [22]. The main drawbacks mentioned by these attempts is that an automotive system is a massively distributed, concurrent system, which causes significant difficulties during verification. However, the approach of this paper significantly simplifies the underlying problem, as it tries to verify only one component.

4. CEGAR driven test generation in AUTOSAR components

This chapter presents a method that combines formal verification and test generation. It also elaborates the algorithms and methods required for that, such as the test generation methods that use the formal representation acquired by the formal verification. This chapter also presents how it fits into the development process of an AUTOSAR component.

4.1. Overview of approach

In the real world, the cost of an algorithm is an imperative factor. The cost in this context consists of the time it needs to complete, and the computational power it requires. The budget allocated to determine the correctness of a software is always finite, and as a result, it needs to take the costs of every algorithm into account.

Model checking is an approach that can formally decide whether a given requirement holds on a given model. Although the previous sentence is correct in terms of mathematics, it tends to fail in practical application. The phenomenon of state-space explosion causes the model checking algorithm to examine an enormous state-space, and even when using abstraction, the worst-case is to traverse the whole state-space. However, this will not work for software with potentially infinite state space.

Having a finite budget, and an algorithm whose runtime cannot be predicted, a model checking algorithm rather have three different outputs in practice (**Figure 4.1**), in opposite to the two possible outputs in theory (**Figure 2.2**). The possible practical outputs are:

- Safe, where the requirement provably holds.
- Unsafe, where the requirement provably fails.
- Undecided, when it cannot be determined under an assigned cost budget (and using the given algorithm), whether the requirement holds or not.

However, if the result is undecided, the computations performed during the verification usually go to waste. The novelty of this approach that it saves the state-space representation of the verification and uses it to focus the test case generation.

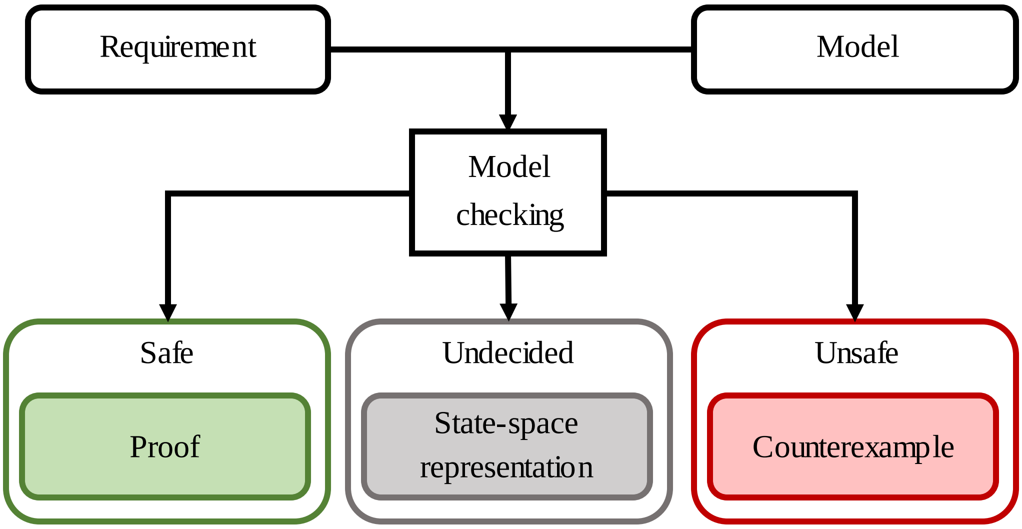

Figure 4.1: *The practical method of model checking*

The CEGAR algorithm introduced in **Chapter 2.3** is a model checking algorithm that uses abstraction to handle state-space explosion. It takes a CFA as its input, where the model is the CFA itself, and the requirement is that no error locations are reachable. It also has the three possible outputs mentioned above. In case the model is safe, it can yield a proof, in case it is unsafe, it can emit a counterexample. Additionally, if the algorithm is terminated early, and the result is undecided, the abstract state-space representation can be extracted in the form of an ARG. Later on, the test generation methods are using the ARG.

If the result of CEGAR is undecided, additional measures have to be taken to ensure safety. The obvious choice is testing, which can decide if the SUT contains errors. Tests can be generated using traditional test generation methods, however, using the abstract state-space representation left over by the model checker, more precise tests can be generated, that traverse the untraversed part of the state-space, and apply the test generation methods.

However, testing cannot prove that the SUT is safe. If no test in a test suite founds an error, then the answer in terms of safety is still undecided. On the other hand, different coverage indicators can reflect on how well the test suite checks the state-space, which is an assurance of the quality and exhaustiveness of the testing. Safety-critical standards also require to achieve high coverage during testing.

If the test suit finds an error in the SUT, then an example is given for which the program is faulty so that it can be fixed later. It is also worth to be noted that the counterexample yielded by the model checker can also be used to generate a test, which will obviously fail, but it makes it executable in the testing environment.

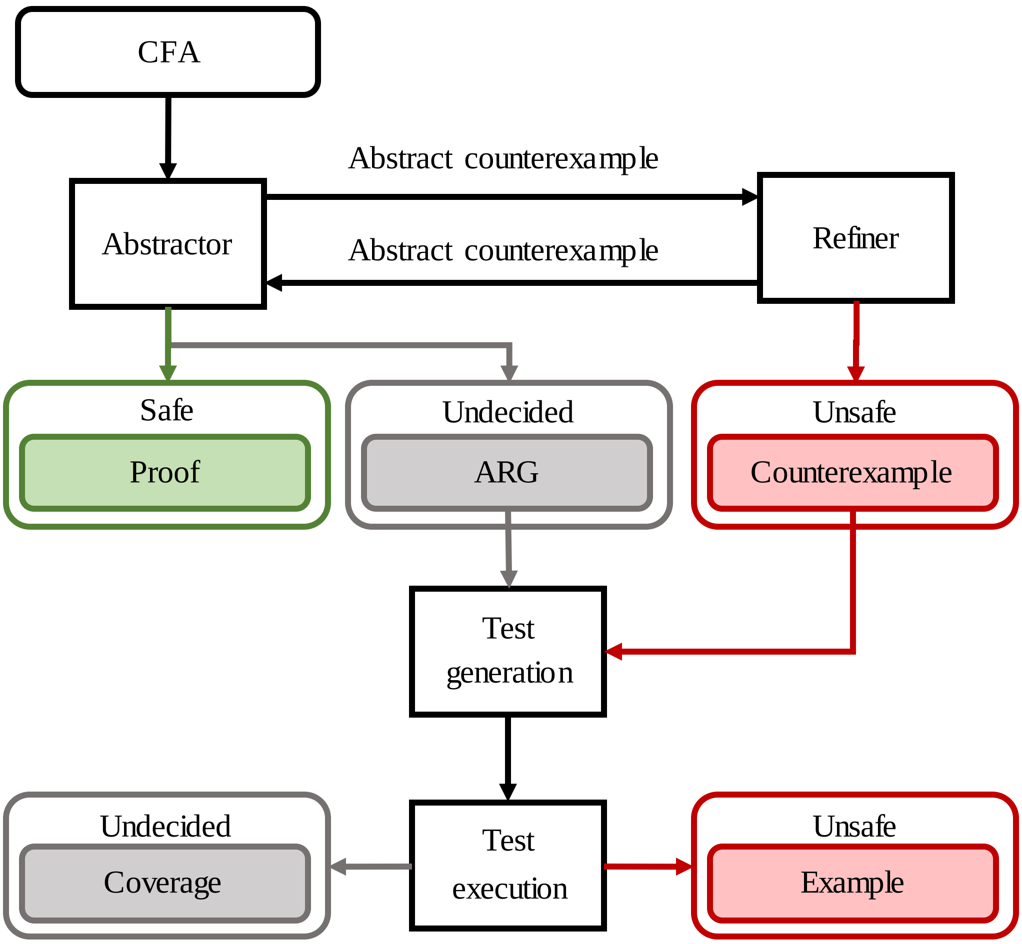

Figure 4.2: *Combining CEGAR and test generation*

The method described above is depicted in **Figure 4.2**. The CEGAR algorithm has three possible outputs. If the verification cannot succeed, the test generation method generates a test suite from the abstract state-space representation of the verifier. On the other hand, if the result is unsafe, the test generation method generated a simple test case based on the counterexample, that shows the error. While executing the test suite, either an error is found, or coverage is calculated at the end.

The traditional approach of AUTOSAR component development requires the developers to write test cases by hand. However, when using the method described above, the test cases can be the result of generation after the formal verification. Moreover, if the program is unsafe, and either testing or the formal verification provides a counterexample, a test case can be derived from it, that the testing environment can execute to show the fault.

To support formal verification, another environment needs to be developed that provides an interface to the formal verification tool, and mocks the behavior of the RTE so that a formal method could verify the component in question for a given requirement, which also needs to be formalized. **Figure 4.3** describes the AUTOSAR component development methodology that includes formal verification as well. The requirements for the formal method come from the requirements of the component, while the test suite of the testing environment is the result of the test generation if the model checker did not yield safe.

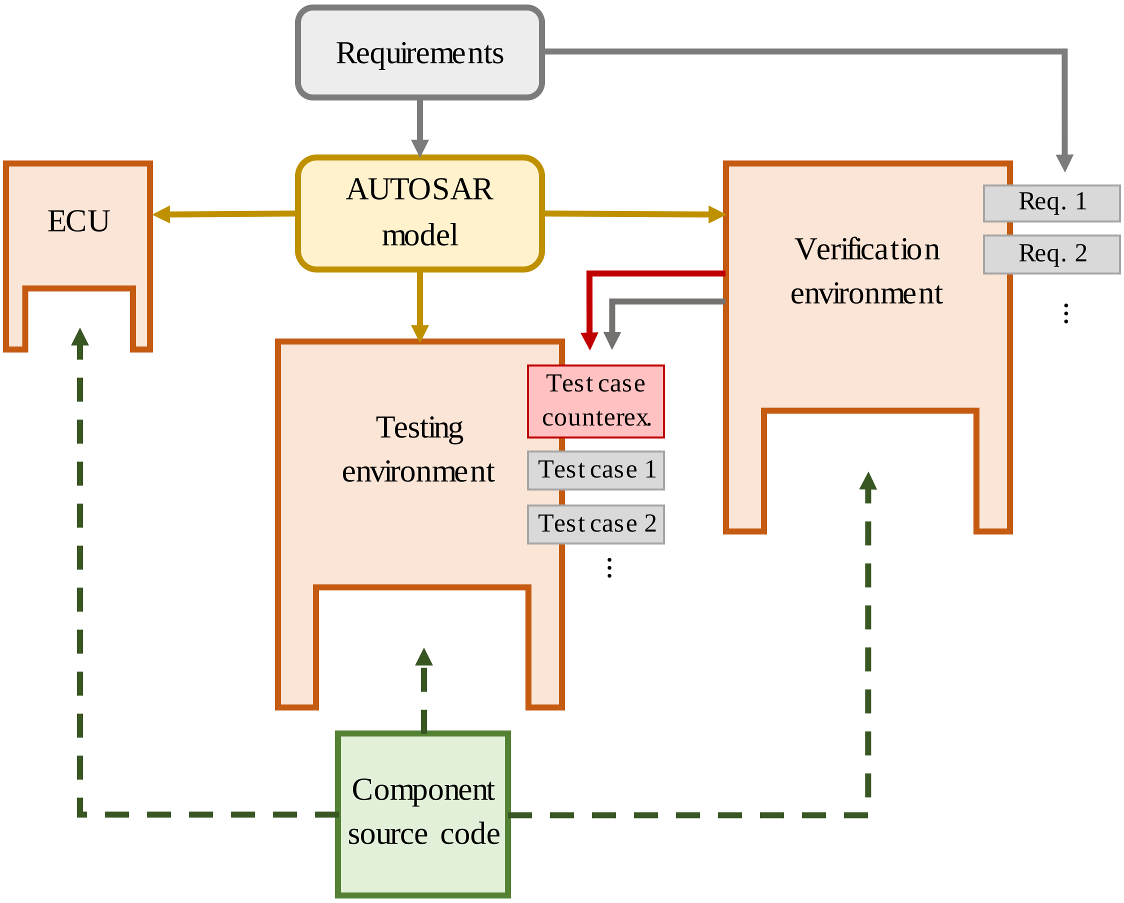

Figure 4.3: *The improved methodology of AUTOSAR component development*

4.2. Application of the CEGAR algorithm

This section introduces the combination of formal verification and test generation to find software errors efficiently. This section details how the CEGAR algorithm is used to generate test cases. First, it is modified to terminate if given conditions hold, then the information is extracted from the abstract state-space representation it has built.

4.2.1. Terminating the CEGAR loop

As the time required for the termination of a model checking algorithm is not predictable, the algorithm needs to be stopped in a state where it produces a consistent state-space representation.

```
Abstractor(CFA, P, TERM):
  ARG := ARG with initial location
  FOREVER:
    s ∈ {incomplete nodes of ARG}
    IF ∄s THEN:
       ← SAFE
    ELSE IF ∃s and s is an error location THEN:
       ← COUNTEREXAMPLE(route from initial location to s)
    ELSE IF TERM(ARG) THEN:
       ← UNDECIDED(ARG)
    ELSE:
      IF ∃r ∈ {nodes of ARG}: r covers s THEN:
        cover s with r
      ELSE:
        expand s
```

Listing 4.1: *The modified CEGAR algorithm*

The CEGAR algorithm terminates in two cases: either the abstractor builds an abstract state-space representation that cannot be expanded further and has no abstract error-state, or the refiner proves that an abstract counterexample is feasible. However, to terminate the algorithm before either happens, it needs to be modified accordingly.

The algorithm of the abstractor part of the CEGAR loop does computationally-heavy operations while it builds the abstract state-space. Moreover, it contains a cycle, which repeats itself while either all the nodes in the built ARG are complete, or an abstract error-state is encountered. However, neither of the previous conditions are predictable, in terms of how much iteration a cycle needs to do so.

Let us introduce a third condition, which upon being true, exits the cycle of the abstractor, and terminates the CEGAR algorithm with undecided as a result. This third condition should be a predicate function that takes the ARG as a parameter and returns true if the algorithm should terminate, as any information about the state-space representation can be extracted from the ARG. The modified version of the algorithm can be seen in **Listing 4.1**.

4.2.2. Extracting information from an ARG

In case, the result of the CEGAR algorithm is undecided, it yields the ARG, as the state space representation. Information can be extracted from it that can be useful when generating test cases. The goal of the generated test cases to find errors in the program so to navigate through an error-state.

Nodes of an ARG		
Unreachable	Reachable	
	Complete	*Incomplete*

Table 4.1: *The different types of nodes in an ARG*

In an ARG, each node is reachable or unreachable. If a node is unreachable, it means that there are no such input values, so the execution path goes through an unreachable node. The presence of unreachable nodes in an ARG is the result of using abstraction, as it overapproximates the reachable state-space. For example, abstract error-states are only reachable if the program is unsafe. As no execution goes through unreachable nodes, they can be removed from the ARG.

If a node is reachable, it is safe to assume, that it is not an abstract error-state, because if it were, the refiner would have concretized it, and it would have led to an unsafe termination of CEGAR. If a node is reachable, it can be either complete or incomplete. Concerning the reachability of error-states from them, a complete node is either:

- Expanded: in this case, every error-state reachable from this node is reachable through one of its children; or
- Covered: in this case, the set of the reachable states from this node is a subset of the set of reachable states from another node; it follows, that all error-states reachable from this node are reachable from the other node.

It leads to the conclusion that in terms of reachability of error-states, no complete nodes need to be examined, as every reachable error-state can be reached from another node as well.

The only remaining nodes are the incomplete nodes. The node is incomplete because the CEGAR algorithm was terminated before it could either expand or cover them. In other words, these have not yet been traversed. As a result, they act as a doorway to the untraversed part of the state-space, every state and error-state of the state-space are reachable through one of the incomplete nodes.

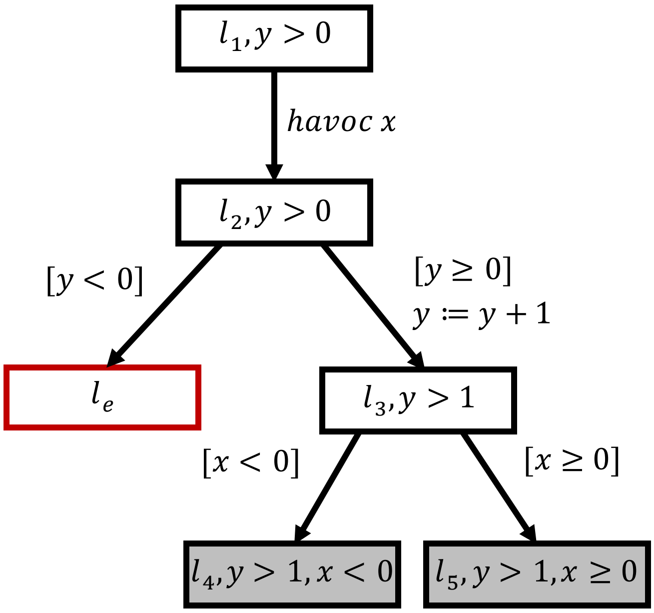

Figure 4.4: *Part of an Abstract Reachability Graph*

Additional information can be extracted from the ARG. The edges in an ARG describe operations, and the predicates, the nodes are labeled by the contained information about the already traversed state-space.

> ***Example 4.1:*** *In* ***Figure 4.4****, an ARG can be seen. Only one node, l_e is unreachable (as $(y > 0) \wedge (y < 0) \rightarrow \perp$). The rest of the nodes are reachable. The nodes l_1, l_2 and l_3 are complete as all of them are expanded. The remaining nodes, l_4 and l_5 (denoted by a grey background) are incomplete, because they are neither expanded nor covered.*

4.3. Test generation

This chapter presents the novel test generation approach introduced in the paper. It starts with symbolic execution, then applies black box testing techniques, such as boundary value analysis, and checks for variable overflows in the program.

4.3.1. Symbolic execution of the abstract state-space representation

When CEGAR cannot verify the requirement and terminates early, abstract state-space representation can be extracted from it. This ARG describes the already traversed part of the state-space and denotes the doorways to the untraversed part. It follows that the start of every possible path the program execution might take is described in the ARG.

The goal of symbolic execution is to traverse all possible execution paths in the program. The main issue is that because of the path-explosion, it is usually impossible to generate a test for every path under a finite budget and time. However, the abstract state-space yielded by the formal method has finite size and eliminates the branches as well, so path-explosion does not occur.

The ARG excluding the covering edges is a tree, in which the path from the root of the tree (the initial location) to one of the leaves describes a unique path of execution. The number of these paths can be reduced if those are excluded that traverse through an unreachable node, or end in a complete node. Paths ending in complete nodes can be eliminated because no error-state is reachable from them that is not reachable from at least another node, and the task of testing is to find errors.

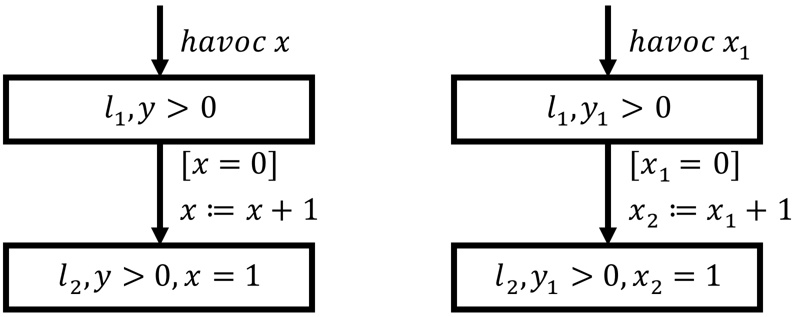

Figure 4.5: *Symbolic execution of a path excerpt*

It follows that only those paths should be focused on when generating tests that end in an incomplete node. A path from the root to an incomplete node is a series of nodes and transitions, which contain operations. A symbolic state (δ) and a path constraint (π) must be maintained to apply symbolic execution. The symbolic state requires each variable always to have an associated symbolic value, which is used in the expressions.

Starting symbolic execution, $\delta = \{\}$, $\pi = \top$. Given two adjacent nodes, n_i and n_j, and operation op between them, the algorithm is:

- For every predicate p of n_i, and for every variable v in p, v must be substituted with $\delta(v)$.

- If op is a deterministic assignment $x \coloneqq expr$, different rules apply for the left and the right-hand side:
  - Every variable v in $expr$ side must be substituted with $\delta(v)$, leading to $expr'$.
  - The variable x on the left-hand side must be replaced by a new symbolic value that has never been used before, and the symbolic state updated accordingly, so $\delta(x) = x_i$, where x_i has never been used before.
  - $\pi' = \pi \wedge (x_i = expr')$
- If op is a non-deterministic assignment $havoc\, x$, then x must be replaced by a new symbolic value that has never been used before, and the symbolic state updated accordingly, so $\delta(x) = x_i$, where x_i has never been used before.
- If op is a guard $[cond]$, then for every variable v in $cond$, v must be substituted with $\delta(v)$, leading to $cond'$. Also $\pi' = \pi \wedge cond'$.

> ***Example 4.2:*** *An example of the algorithm described above can be seen in* ***Figure 4.5****. The left-hand side depicts a path, while the right-hand side depicts the same path but with the variables substituted.*

In the end, the path constraint is a first-order formula, that can be fed to an SMT-solver, which will yield a model. The result cannot be unsatisfiable, as only reachable nodes are part of the path. The values of the non-deterministic assignments or inputs can be extracted from the model, and based on them, a test case can be generated that executes the exact path the path constraint describes. The method above can be repeated for all paths, resulting in the test suite.

4.3.2. Robustness test generation for the untraversed state-space

The robustness of a program is its ability to handle errors during execution. It contains the ability to cope with erroneous or unexpected inputs or generally a wide range of inputs. There are multiple methods that support robustness testing, such as equivalence partitioning and boundary value analysis.

When using equivalence partitions, the domain of every input variable is split into multiple partitions. A test case takes a partition for each input variable and chooses a value from them. When boundary value analysis is applied, the values taken from the partitions are systematically the MIN-, MIN, MIN+, NOM, MAX-, MAX, MAX+ values.

The untraversed part of the state-space can be thought of as a black box, whose input is modifiable, and whose output is observable, but its inner workings are not transparent. It follows that black box testing techniques can be applied. Black box techniques require a specification, which should be the specification of the program refined by the data gathered during the symbolic execution.

BoundaryValueAnalysy(ARG **):**
  $T \coloneqq \{\}$
  FORALL $P \in \{$possible combinations of equivalence partitions$\}$ **DO:**
    FORALL $n \in \{$incomplete nodes of $ARG\}$ **DO:**
      $P \coloneqq$ path from root to n
      $C \coloneqq$ path constraint of P
      FORALL $v \in \{$non-deterministic variables in $C\}$ **DO:**
        $C \coloneqq C \wedge ($domain of v in $P)$
      FORALL $v \in \{$non-deterministic variables in $C\}$ **DO:**
        $M_1 \coloneqq$ model of SMT-problem C while optimizing for $\min(v)$
        $M_2 \coloneqq$ model of SMT-problem C while optimizing for $\max(v)$
        $T \coloneqq T \cup \{$test cases based on M_1 and $M_2\}$
  $\leftarrow T$

Listing 4.2: *The algorithm generating test cases with boundary value analysis*

Taking a symbolic executed path, the end of the path denotes a doorway into the untraversed state-space. The untraversed state-space has two kinds of input values: first, the non-deterministic assignments inside, second the variables that have value in the doorway, the so-called entry-variables.

However, the entry-variables are not input variables of the program, only of the black box, so the specification of the program does not contain information regarding them. To calculate their boundary, the symbolic execution algorithm should be modified. The modified algorithm is depicted in **Listing 4.2**.

First, the path constraint should be calculated, as described in the previous chapter. Then information about the equivalence partitions should be inserted. Based on the domain of x in a partition, for which non-deterministic assignment $havoc\ x$ is present on the path, and the corresponding symbolic value is $\delta(x) = x_1$:

- If $D_x = [a; b]$, then $\pi' = \pi \wedge (a \leq x_1) \wedge (x_1 \leq b)$.

- If $D_x =]a; b[$, then $\pi' = \pi \wedge (a < x_1) \wedge (x_1 < b)$.

The steps above should be repeated for every non-deterministic assignment. Then, as boundary value analysis requires one variable to be on minimal or maximal value, while others are on nominal, one of the variables must be chosen. Following, the SMT-problem must be fed to an SMT-solver with an optimization constraint. This constraint should specify that the given variable should have a minimal or maximal value. As a result, such a model is returned from the set of possible models for the SMT-problem, in which the value of that particular variable is minimal or maximal, while the other variables have a possible (not necessarily nominal) value.

The method above should be repeated for all the non-deterministic variables systematically, resulting in a set of test cases.

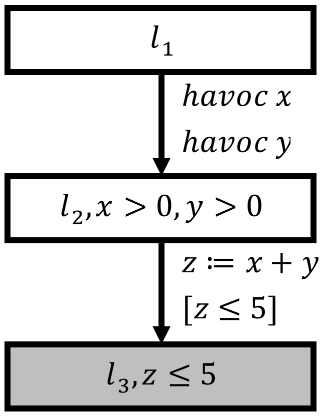

Figure 4.6: *Path in an ARG*

These test cases differ from a traditional boundary value analysis presented test suite because they are more precise. First, the formal method proves that the program is safe for some part of the domain while undecided for another part. The path constraint removes the safe part from the possible values, so in the resulting model, the minimal value is the lowest possible value, the untraversed state-space is reachable with on that path, while the maximal value is the highest possible.

> ***Example 4.3:*** *In* ***Figure 4.6****, there is a path in an ARG. The path goes from the initial location l_1 to the incomplete node l_3 The domain of both its input variables is $[0; 15]$, so a 4-bit unsigned integer, and there are no equivalence partitions.*
>
> *The path constraint derived from the path is $\pi = (x_1 > 0) \wedge (y_1 > 0) \wedge (z_1 = x_1 + y_1) \wedge (z_1 \leq 5)$. The variables in non-deterministic assignments are x_1 and y_1. Adding the domain of variables to the path constraints yields $\pi' = \pi \wedge (x_1 \geq 0) \wedge (x_1 \leq 15) \wedge (y_1 \geq 0) \wedge (y_1 \leq 15)$. With two input variables, four optimization constraints can be formed:*

> - $min(x_1)$: *one of the models is* $\{(x_1 = 1), (y_1 = 1)\}$
> - $max(x_1)$: *the model is* $\{(x_1 = 4), (y_1 = 1)\}$
> - $min(y_1)$: *one of the models is* $\{(x_1 = 1), (y_1 = 1)\}$
> - $max(y_1)$: *the model is* $\{(x_1 = 1), (y_1 = 4)\}$
>
> *Based on this information, four test cases can be generated, which are the four models listed above.*

4.3.3. Variable overflow in the state-space

Variable overflow is an exciting topic in formal verification because the SMT-solvers usually work with mathematical variables with infinite domains. On the other hand, the variables in programs are represented on a finite number of bits, so their domain is also finite.

There are multiple methods to circumvent this phenomenon, for example:

- Define every arithmetical operation as an operation over bit-vectors. Although it works, it has a significant drawback on the performance.
- Define every arithmetical operation as a modulo operation. This method has a lesser drawback on the performance; however, this way, it cannot be determined later that overflow occurred.
- Test for overflow after verification.

Although testing overflow does not prove its absence, this paper uses this approach, because AUTOSAR development requires compliance with safety-critical standards, such as MISRA C, and they always forbid using code that overflows. This rule eliminates option two from the previous list, while the significant performance loss the first, leaving only the third approach.

The overflow might occur in two situations: either in the traversed or in the untraversed part, which requires different approaches. Overflow always occurs as a result of arithmetical operations.

If the overflow happens in the traversed part, it means that the result of an arithmetic variable is outside the domain of the target variable. Fortunately, this can easily be tested by an SMT-solver, as done by the algorithm in **Listing 4.3**.

OverflowInTraversed(ARG)**:**
  $T \coloneqq \{\}$
  FORALL $e \in \{$edges of ARG containing arithmetic operation$\}$ **DO:**
    $x \coloneqq$ target variable of arithmetic operation in e
    $P \coloneqq$ path from root to destination of e
    $C \coloneqq$ path constraint of P
    FORALL $v \in \{$non-deterministic variables in $C\}$ **DO:**
      $C \coloneqq C \land ($domain of v in $P)$
    $C_1 \coloneqq C \land (\delta(x) >$ maximum of its domain$)$
    IF SMT-problem C_1 is satisfiable **THEN:**
      $\leftarrow$ **OVERFLOW**
    $C_2 \coloneqq C \land (\delta(x) <$ minimum of its domain$)$
    IF SMT-problem C_2 is satisfiable **THEN:**
      $\leftarrow$ **OVERFLOW**
  $\longleftarrow$ **NO OVERFLOW**

Listing 4.3: *The algorithm checking overflow in traversed part if the state-space*

First, every operation must be located that uses arithmetic operation. These are the variables where overflow might occur. For each operation, a path must be generated that leads from the root to the destination of that operation. Assignments on this path could cause an overflow. Then the path constraint should be calculated, and the domain of every non-deterministic variable should be added to the formula, similar to the method in the previous section.

In the next step, a clause must be added to the path constraint that states that the value of the variable (can be extracted from the symbolic state) is greater than the top part of its domain. This modified SMT-formula can be only true, if the value of that variable is outside of its domain, so an overflow occurs. This check can be repeated for a lower bound check, as well.

The other case, when the overflow is in the untraversed part, is slightly more complicated than the first scenario. However, it is worth to be noted, that overflow usually occurs when one or more variables are near their upper limit and do arithmetic operations. Based on this observation, a method very similar to the one described in the previous section can be designed to test the untraversed part for overflows.

```
OverflowInUntraversed(ARG):
  T ≔ {}
  FORALL n ∈ {incomplete nodes of ARG} DO:
    P ≔ path from root to n
    C ≔ path constraint of P
    FORALL v ∈ {non-deterministic variables in C} DO:
      C ≔ C ∧ (domain of v in P)
    FORALL v ∈ {variables valid in n} DO:
      M1 ≔ model of SMT-problem C while optimizing for
      min(v)
      M2 ≔ model of SMT-problem C while optimizing for
      max(v)
      T ≔ T ∪ {test cases based on M1 and M2}
  ⟵ T
```

Listing 4.4: *The algorithm generating test cases to test overflow in untraversed part*

The algorithm (described in **Listing 4.4**) should navigate to each incomplete node, as in the previous case. Also, the path constrained must be constructed, and the domain information should be added. However, the optimization constraint should be the minimization and maximization of each variable that is valid in the current incomplete node. This way, the variables at the entry of the untraversed state-space have their lowest or highest possible value and are likely to overflow if they indeed do.

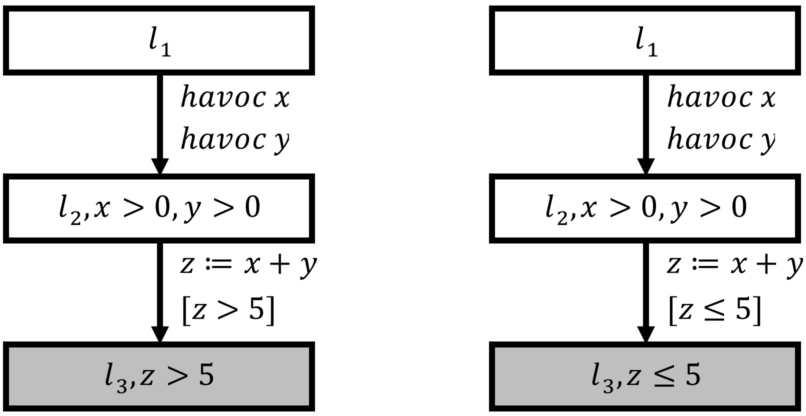

Figure 4.7: *Two paths of an ARG*

> ***Example 4.4:*** *On both sides of* ***Figure 4.7****, there is a path in an ARG. The paths go from the initial location* l_1 *to the incomplete node* l_3.

The domain of both its input variables and z is $[0; 15]$*, so a 4-bit unsigned integer.*

The path constraint derived from the left-hand side of the path is $\pi = (x_1 > 0) \wedge (y_1 > 0) \wedge (z_1 = x_1 + y_1) \wedge (z_1 > 5)$. *The variables in non-deterministic assignments are* x_1 *and* y_1. *Adding the domain of variables to the path constraints yields* $\pi' = \pi \wedge (x_1 \geq 0) \wedge (x_1 \leq 15) \wedge (y_1 \geq 0) \wedge (y_1 \leq 15)$. *Assuming that z is under- or overflowing, the SMT-solver is fed with the following problems:*

- $\pi' \wedge (z_1 < 0)$*: it is unsatisfiable, so z does not underflow*
- $\pi' \wedge (z_1 > 15)$*: it is satisfiable, so z overflows (for inputs* $\{(x_1 = 15), (y_1 = 15)\}$*)*

Based on this information, one test case can be generated, which causes the program to overflow.

On the other hand, the path constraint derived from the right-hand side of the path is $\pi = (x_1 > 0) \wedge (y_1 > 0) \wedge (z_1 = x_1 + y_1) \wedge (z_1 \leq 5)$. *The variables in non-deterministic assignments are* x_1 *and* y_1. *Adding the domain of variables to the path constraints yields* $\pi' = \pi \wedge (x_1 \geq 0) \wedge (x_1 \leq 15) \wedge (y_1 \geq 0) \wedge (y_1 \leq 15)$. *Aiming for the overflow of* z*, the SMT-solver is fed with the path constraint, with the following optimization constraint:*

- $min(z_1)$*: the model is* $\{(x_1 = 1), (y_1 = 1)\}$
- $max(z_1)$*: one of the possible models are* $\{(x_1 = 3), (y_1 = 2)\}$

Based on this information, two test cases can be generated, which might lead to overflow.

4.4. Integrating formal verification in the AUTOSAR development process

AUTOSAR already has development environments to build and test components. To develop a verification environment, first, a verification environment must be generated. This environment must mock the behavior of the RTE and model the behavior of the component as well. After the verification and test generation is finished, the test cases emitted by the test generator must be transformed so that they can be fed to the testing environment.

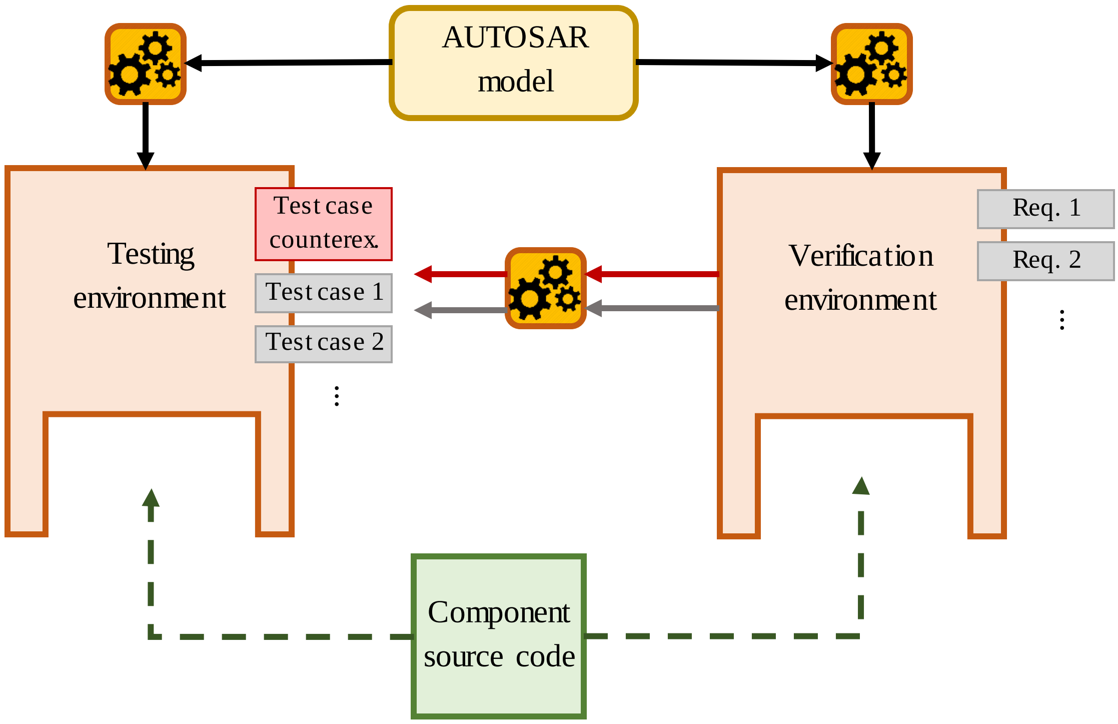

Figure 4.8: Integrating verification in the AUTOSAR methodology

The methodology described above can be observed in **Figure 4.8**. It depicts three different operations. First, the operation that generates the testing environment from the model, which is assumed to be developed and fully functional. Second, the operation that generates the verification environment from the model. Finally, the operation that transforms a test case emitted by the verification environment to fit the needs of the testing environment.

4.4.1. Modeling the behavior of a component

An AUTOSAR component runs on a single ECU, which has implications. It is single-threaded, so no two operations using the same memory can overlap in time. The RTE also buffers all the messages, requests to the component, so every message is handled when all other operations are finished. It leads to the realization that an AUTOSAR component can be modeled as a statechart with only one state. The structure of the statechart is:

- It has only one state, which denotes that the component is waiting for an input or an event.
- The initial location and the state are connected by a transition, whose action initializes the parameters and per instance memories of the component.

- For every input sender-receiver port, a loop transition is created, that reads the value of the port. These reads are non-deterministic assignments, as there is no information on what the result will be.
- For every providing client-server port, a loop transition is created, that reads the value of the input parameters, fires the corresponding event and runnable, and writes the value of the output parameters.
- For every timed event, a loop transition is created, that fires the corresponding runnable.

This statechart can easily be transformed into a C code that interacts with the implementation of the component.

4.4.2. Writing verifiable requirements

To run verification, the requirements must be entered to the verifier as well. As it was described in earlier chapters, in case of software components, the easiest way to do so is to write assertions, and the requirement is that the assertions never fail.

These assertions can be placed in the code by the developer, similarly to how fault injection is usually handled [23].

4.4.3. Generating the verification environment

The verification environment mocks the behavior of the RTE and the behavior of the component. To mock the RTE, an implementation must be generated that has the same standard interface that is required, but its inner workings are compatible with the verification tool.

First of all, a method should be devised for the RTE. The component using the mock-RTE should be able to read from ports and write to ports, should be able to read parameter data, should be able to read and write per instance memories, and the component must handle if the RTE fires events.

The AUTOSAR standard fixes how the RTE should interface to the source code of the component: it specifies functions for each scenario, where the name of the function can be derived from the name of the component and the port; and the parameters of the functions are specified by the data that is passed in that scenario. This fixed interface is called the contract of the component and can be generated from the AUTOSAR model.

The implementation of the contract consists of function definitions, which can also be generated. In case of a port, the corresponding functions should provide persistent storage of the value of the port, and the functions should be able to read and write the data. The same is true for the per instance memories. However, in case of events, the implementation should fire the runnables it is bound to.

> ***Example 4.5:*** *Given a component with name SampleComponent. It has:*
>
> - *Input sender-receiver port named InPort, with payload named inData with type dInData*
> - *Output sender-receiver port named OutPor, with payload named outData with type dOutData*
> - *Runnable named SampleRunnable*
> - *Timed event named SampleEvent, which fires SampleRunnable*
>
> *The contract and the RTE has the following functions defined:*
>
> - `Std_ReturnType Rte_Read_SampleComponent_InPort_inData(dInData* data)`
> - `Std_ReturnType Rte_Read_SampleComponent_OutPort_outData(dOutData const* data)`
> - `void SampleComponent_SampleRunnable(void)`
> - `void SampleComponent_SampleEvent(void)`

4.4.4. Transforming test cases

The test cases outputted by the verification environment must be transformed so they can be fed to the testing environment. While the input of the transformation heavily depends on the formal model and the structure of the generated test cases, the output heavily depends on the format required by the testing environment.

Fortunately, the testing environment uses the contract of the model as well to provide mocking functionality of the RTE, so the assignments of the test case have to be matched to function parameters and the order in which the functions are called.

Another exciting aspect is the coverage of the test cases. Although the testing environment can measure the coverage of the test cases, some part of the code will not be covered. The reason is that it has been proved to be safe by the formal verification, thereby no test case was targeting that part of the state-space. There are different approaches as to how to measure coverage during formal verification [24] [25]. The result of one such metrics can be merged by the coverage of testing, resulting in a unified coverage indicator, but this process is not the target of investigation of this paper.

5. Evaluation

This chapter presents the evaluation of the methods described in this paper, using a custom implementation. The implementation is tested on industrial software provided by thyssenkrupp Components Technology Hungary Kft.

5.1. Implementation

The implementation of the algorithm is based on many other software. This section presents the interaction between these software to apply formal verification on, and generate tests for AUTOSAR components. The overview of the implementation can be seen in **Figure 5.1**.

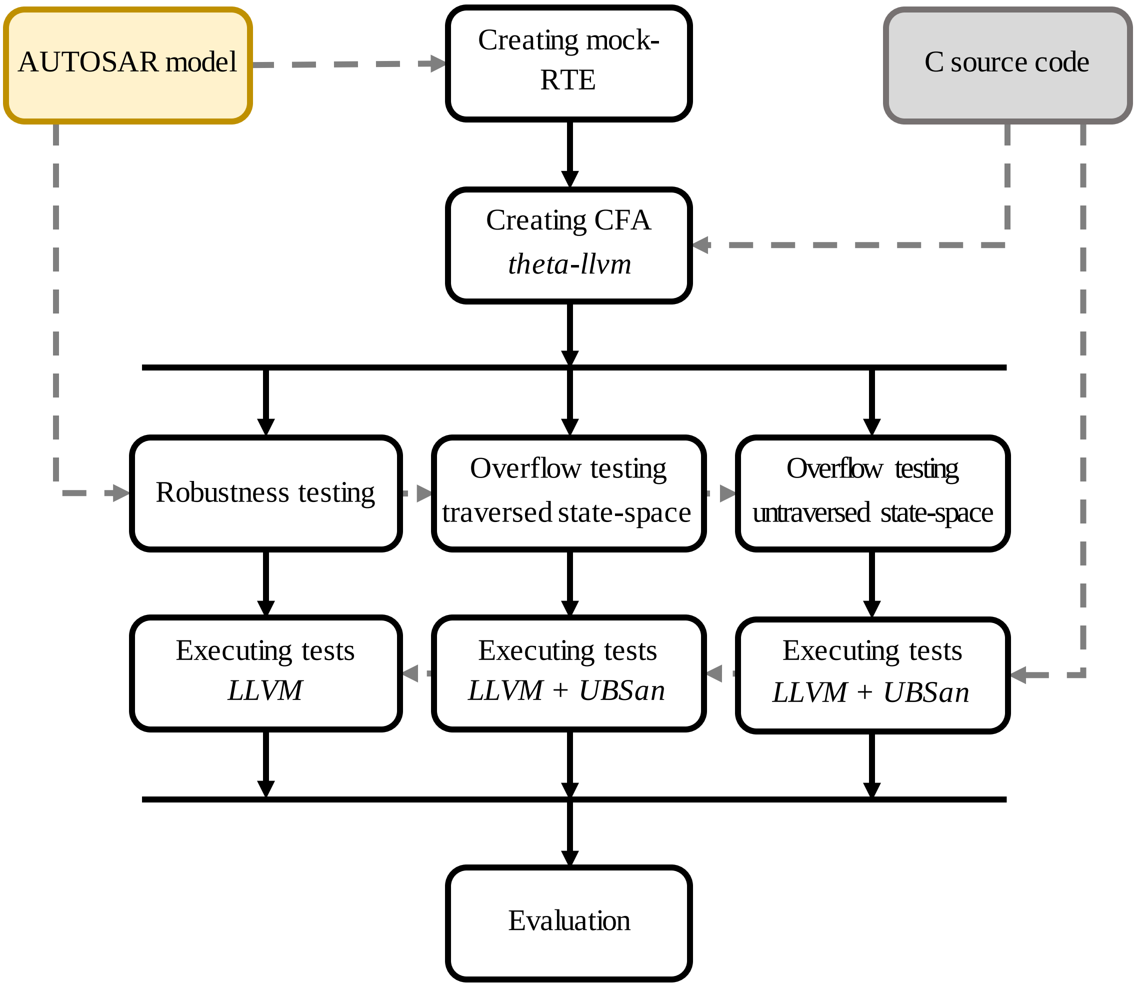

Figure 5.1: *The overview of the implementation*

The AUTOSAR environment chosen was AUTOSAR Architect, as a courtesy of thyssenkrupp, while the verification environment selected is Theta [26], while the underlying C compiler framework is the LLVM framework [27].

5.1.1. Generating the verification environment

The verification environment can be generated based on the AUTOSAR model. It contains information on the parameters, ports, per instance memories, events, and runnables of the component. Each parameter, port, and per instance memory has a type, and the type has an associated range info, which describes the domain of that value.

AUTOSAR Architect represents the AUTOSAR model as an EMF model [28], and thereby it can be processed in an Eclipse environment using Java or Xtend. Based on that, the implementations of the contract can be generated easily.

In terms of the C code, the persistent behavior of parameters, ports, and per instance memories is realized with global, static variables.

Additionally, a main function must be generated, that mocks the behavior of the component as a statechart, and references the implementation through the contract.

> ***Example 5.1:*** *An example of the result of the main function generation can be seen in* ***Listing 5.1****. The component in question (SampleComponent) has one input sender-receiver port (InPort), one output sender-receiver port (OutPort), one parameter (SampleParameter), and one timed event (SampleEvent), with an associated runnable.*
>
> *The core of the program is the infinite while cycle. Before that, all the values are initialized, and the parameters are assigned a non-deterministic value.*
>
> *In the cycle, first, a non-deterministic assignment decides which event occurs. In this case, there are only two possibilities: either the input port receives a new message, or the timed event is fired.*
>
> *If a new value is received, non-deterministic assignments model the new value. On the other hand, if the timed event is fired, the corresponding function is called.*
>
> *The assertions are placed in the source code of the runnable that the event invokes. The assertions should reference the static variables that store the data. For example, if the requirement is that the output port always has a value greater that 0, then the following assertion should be placed at the end of the runnable:*
>
> ```
> assert(VAR_SampleComponent_OutPort_OutData > 0);
> ```

```
/* Variable declarations */
dInData VAR_SampleComponent_InPort_InData;
dOutData VAR_SampleComponent_OutPort_OutData;
dParamData VAR_SampleComponent_SampleParameter;

/* Event declarations */
extern void SampleComponent_SampleEvent();

int main(void) {
  /* Initial values */
  VAR_SampleComponent_InPort_InData = 0;
  VAR_SampleComponent_OutPort_OutData = 0;

  /* Parameter values */
  VAR_SampleComponent_SampleParameter =
  __theta_nondet_int();

  while(1) {
    int event = __theta_nondet_int();
    switch(event) {
      case 0: {
        /* Input variables */
        VAR_SampleComponent_InPort_InData =
        __theta_nondet_int();
        break;
      }
      case 1: {
        /* Firing event */
        SampleComponent_SampleEvent();
        break;
      }
      default:
        break;
    }
  }
}
```

Listing 5.1: *A generated main function*

5.1.2. Creating the CFA

The formal verification method requires a formal model of the software to verify. This formal model is the CFA, so there needs to be a way to convert an AUTOSAR component to a CFA. The source code of a component is written in C, so a tool is needed that can convert the between C code and CFA. The tool chosen was theta-llvm [29], which relies on the LLVM framework, and describes a two-step process:

- First, the software source code must be compiled to LLVM bytecode. Since the language used is C, the clang compiler can achieve this.
- Next, the LLVM bytecode can be converted to a CFA using the theta-llvm software.

It is worth to be noted, that the more assertion in a code is, the harder it is for the verification algorithm to prove correctness. As a countermeasure, formal methods often rely on slicing, which leaves only one assertion in the code to verify, but emits multiple CFAs, one for each assertion. The tool theta-llvm supports slicing, and it is used in the implementation.

The tool also defines three special functions:

- `int __theta_nondet_int()`: It denotes a non-deterministic assignment of a 32-bit integer.
- `void __theta_assert(bool)`: It denotes an assertion.
- `void __theta_exit(int)`: It denotes an exit point in the program.

These functions are heavily used in the source code: the verification environment models all non-deterministic assignments using `__theta_nondet_int()`, while the developer should write the assertions using `__theta_assert(bool)`.

5.1.3. Formal verification

The formal verification environment used by the implementation is the Theta framework [26]. The framework has a highly customizable CEGAR algorithm that let the developer choose the type of abstraction, refinement, and termination condition.

The implementation uses predicate abstraction, interpolant based refinement, and multiple termination conditions: it is possible to run the verification up until a given time, the number of nodes in the ARG, or the depth of the ARG. The SMT-solver used in the background is Microsoft’s Z3 [30].

5.1.4. Generating test cases

Test generation methods were implemented in Java, relying on formalisms provided by the Theta framework. The SMT-solver used was also Z3 [30], as it supports optimization constraints as well.

The methods use the abstract state-space representation emitted by Theta and generate tests based on the algorithms described in this paper. It is worth to be noted that the different methods, the boundary value analysis, and the overflow detection in both the traversed and the untraversed part of the state-space are independent from each other so that they can be executed concurrently.

It is worth to be noted that the domain information required by the generation methods can be extracted from the AUTOSAR model as well.

5.1.5. Transforming the test cases

The test cases emitted by the test generators should be transformed into the format required by the testing environment. However, due to limitations described later, this part is not yet implemented, an LLVM based testing environment was used instead.

The LLVM environment requires an executable program, so the source of the AUTOSAR component must be made executable. As the source has already been compiled to generate the CFA, the sources only need to be linked to make them executable.

However, linking requires an implementation behind all three special functions that theta-llvm uses:

- `int __theta_nondet_int()`: The implementation should return the results of non-deterministic assignments in the correct order. This behavior can be achieved by a static array that contains the values in order.
- `void __theta_assert(bool)`: If its parameter evaluates to false, it should signal a failing test.
- `void __theta_exit(int)`: It should terminate the test, and signal a successful test.

This implementation only drives the test case to the doorway to the untraversed part; however, it does not specify what should happen if another non-deterministic value is required in the untraversed part. Since the main function uses an infinite cycle at its core, this scenario is bound to happen. Thereby the implementation provides random values if additional non-deterministic values are required until a certain number of requests are made, then terminates the test and marks it successful (unless there was a failing assertion).

After linking, an executable can be created and executed as a test case. To catch overflows in test cases, the LLVM framework provides the Undefined Behavior Sanitizer (UBSan) [31], which is capable of capturing overflows. The executable is compiled and linked using the UBSan as well.

5.1.6. Limitations of the implementation

This implementation depends on multiple software components, which also have limitations. First of all, only a part of the AUTOSAR standard is supported, namely:

- Sender-receiver ports
- Client-server ports
- Parameters
- Per instance memories
- Timed events
- Client-server port bound events
- Runnables

The tool theta-llvm has its limitations as well:

- It supports function invocations by inlining only, so recursion is not supported.
- Only boolean and integer numbers are supported as primitive types (no floating-point numbers).
- Pointer arithmetic is not supported.
- Bitwise operations are not supported.
- Only pointers to primitive values are supported (no pointer to structs or unions).
- No debugging information is preserved.

The second to last limitation has other consequences. As the contract uses pointers to structs whenever passing objects to functions, the transformation of test cases is not possible to the testing environment, because the structs had to be flattened, and replaced by only numeric fields, which violates the contract.

Also, the last limitation is a severe hindrance as well, as it makes it impossible to transform the test cases. When the test generation tool emits the list of values to pass the program, the test case transformer cannot determine which value belongs to which parameter or port.

5.2. Applying the approach to industrial code

The proposed algorithm and the implementation were tested using AUTOSAR components provided by thyssenkrupp Components Technology Hungary Kft. They provided two components, one simpler, and one a bit more complex.

ComponentA, is the simpler component: it has multiple parameters, multiple sender-receiver ports, and a timed event that fires the single runnable. This component does not contain per-instance memory, so no persistence needed.

ComponentB, on the other hand, is a bit more complicated, as it has multiple parameters, sender-receiver ports, provides multiple client-server ports, has per-instance memory, has one timed event, multiple events for the client-server ports, and multiple runnables.

The features of the components that violated one of the limitations of the tool were removed. The removed features were mainly bitwise operations.

The requirements given to them can be categorized +according to what kind of program structure is required to realize them. The categories were the following:

- Simple sequential or conditional calculation: to realize the feature, no cycle is needed.
- Calculations using deterministic cycles: the iteration count of every cycle is deterministic; it does not depend on input.
- Calculations using non-deterministic cycles: the iteration count of at least one cycle depends on non-deterministic value (input value).

	ComponentA			**ComponentB**		
	require-ments	*verified requirem ents*	*test cases*	*require-ments*	*verified requirem ents*	*test cases*
Sequential, conditional	4	4	-	5	5	-
Deterministic cycles	3	3	-	2	2	-
Non-deterministic cycles	0	-	-	3	1	45

Table 5.1: *The results of applying the algorithm on AUTOSAR components*

The algorithm was run on several requirements. The formal method was limited to running at most one hour. The results can be seen in **Table 5.1**. As it can be seen, the formal method could easily handle the situations when only simple sequential or conditional code was required to realize the feature. It also was able to complete the verification if it relied on deterministic cycles. However, when the result depended on calculations done in non-deterministic cycles, the formal method mostly failed, and tests were generated. These tests did not find any error, which is not surprising, given that these components are used daily.

6. Conclusion

This paper presented an approach to support formal verification guided test generation in AUTOSAR components.

The algorithm used an abstraction based formal method to verify requirements and used the information extracted from the state-space representation of the verifier to generate test cases based on if the verification failed to complete. The test generation methods were based on symbolic execution, and used boundary value analysis, and checked the robustness of the software as well.

Moreover, methods were devised to apply formal verification on safety-critical AUTOSAR components, heavily used by the automotive industry. By doing so, the test cases were generated by an algorithm, rather than written by a developer, which could lead to shortened product-to-market time.

An early implementation was developed to test whether this approach is functional, and it was tested on AUTOSAR components used by the automotive industry, diagnosing the shortcomings of the implementation.

All in all, a novel algorithm was developed that successfully verified automotive codes, and generated tests, and this algorithm was proved to be working on industrial code.

6.1. Future work

In the future, I plan to solve the limitations of the implementation and examine more test generation methods.

- First of all, the support of bitwise operations must be developed, as it is heavily used in the source code of the automotive industry.
- Then the theta-llvm tool must be developed to support the subset of C used in AUTOSAR components fully.
- Multiple abstraction techniques, particularly product-abstraction methods, should be examined, whether they provide a better verification result.
- More test generation strategies should be examined.
- The test generation methods should be evaluated, and compared to traditional source code-based test generation methods.

Acknowledgment

I would like to thank thyssenkrupp Components Technology Hungary Kft. For supporting my work and providing me with the resources necessary for my research, including tools and components, to test my algorithms on.

Also, I would like to thank Ákos Hajdu, and Tamás Tóth personally for the professional support I got from them during my work, and the knowledge I learned from them in the fields of formal verification.

The research was supported by the EFOP-3.6.2-16-2017-00013 grant of the European Union, co-financed by the European Social Fund.

Bibliography

[1] "First-Order Logic," in *The Calculus of Computation*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 35–68.

[2] T. Biere, A. and Biere, A. and Heule, M. and van Maaren, H. and Walsh, *Handbook of Satisfiability*. 2009.

[3] L. De Moura and N. Bjørner, "Satisfiability modulo theories," *Commun. ACM*, vol. 54, no. 9, p. 69, Sep. 2011.

[4] "SMT-LIB The Satisfiability Modulo Theories Library." [Online]. Available: http://smtlib.cs.uiowa.edu/solvers.shtml. [Accessed: 27-Oct-2019].

[5] D. Beyer and S. Löwe, "Explicit-State Software Model Checking Based on CEGAR and Interpolation," in *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, Springer-Verlag, 2013, pp. 146–162.

[6] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," in *Proceedings of the 9th International Conference on Computer Aided Verification*, 1997, pp. 72–83.

[7] D. Beyer and M. Dangl, "SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms," Springer, Cham, 2016, pp. 181–198.

[8] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, 2008.

[9] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to Model Checking," in *Handbook of Model Checking*, Cham: Springer International Publishing, 2018, pp. 1–26.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.

[11] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik, "A Configurable CEGAR Framework with Interpolation-Based Refinements," 2016, pp. 158–174.

[12] K. L. McMillan, "Applications of Craig Interpolants in Model Checking," Springer, Berlin, Heidelberg, 2005, pp. 1–12.

[13] International Software Testing Qualifications Board, "Foundation Level Syllabus." 2018.

[14] J. C. King and J. C., "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[15] C. Cadar and K. Sen, "Symbolic execution for software testing," *Commun. ACM*, vol. 56, no. 2, p. 82, Feb. 2013.

[16] "AUTOSAR standard." [Online]. Available: https://www.autosar.org/standards/. [Accessed: 28-Oct-2019].

[17] M. Dobos-Kovács, A. Vörös, and Á. Hajdu, "Modellellenőrzés és tesztelés: egy kombinált megközelítés szoftverek verifikálására," 2018.

[18] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, 2016, pp. 144–155.

[19] M. Czech, M.-C. Jakobs, and H. Wehrheim, "Just Test What You Cannot Verify!," Springer, Berlin, Heidelberg, 2015, pp. 100–114.

[20] J. Botaschanjan *et al.*, "Towards verified automotive software," in *Proceedings of the second international workshop on Software engineering for automotive systems - SEAS '05*, 2005, vol. 30, no. 4, pp. 1–6.

[21] M. H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti, "Formal verification of an automotive scenario in service-oriented computing," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008, p. 613.

[22] S. Beyer *et al.*, "Towards the formal verification of lower system layers in automotive systems," in *2005 International Conference on Computer Design*, pp. 317–324.

[23] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer (Long. Beach. Calif).*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[24] Y. Hoskote, T. Kam, Pei-Hsin Ho, and Xudong Zhao, "Coverage estimation for symbolic model checking," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pp. 300–305.

[25] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage Metrics for Formal Verification," Springer, Berlin, Heidelberg, 2003, pp. 111–125.

[26] T. Toth, A. Hajdu, A. Vorcos, Z. Micskei, and I. Majzik, "Theta: A framework for abstraction refinement-based model checking," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 176–179.

[27] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," Springer, Berlin, Heidelberg, 2005, pp. 15–16.

[28] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[29] G. Sallai, T. Tóth, Á. Hajdu, and A. Vörös, "Development of a Verification Compiler for C Programs," 2016.

[30] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," Springer, Berlin, Heidelberg, 2008, pp. 337–340.

[31] The Clang Team, "UndefinedBehaviorSanitizer — Clang 10 documentation." [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Accessed: 27-Oct-2019].