Ing. Richard Schumi BSc

Automatic Test Case Generation from Formal Models

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Roderick Bloem, M.Sc. Ph.D.

Institute for Applied Information Processing and Communications
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Second Supervisor
Dipl.-Ing. Franz Röck

Graz, March 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract

Testing is a very important measure to achieve software with high quality and to comply with certain security requirements for security-critical systems. It also is very important to find ways to reduce the time consumption for testing, because thereby the development costs can be significantly reduced. Therefore, it is advantageous to use automation.

In this thesis we describe the development of a tool for automatic test case generation from formal (NuSMV) models. This tool uses graph-based and logic-based coverage criteria for the generation of trap properties, which are given to a model checker in order to produce abstract test cases. First, we evaluate and compare different graph-based and logic-based coverage criteria. Then, we explain the development process, the structure of the tool and important components like model parsing, counter example generation with a model checker and test adaption.

Furthermore, we show a case study, where the tool is applied to a real world project, in order to see if it is practicable. In this case study the model reflects the behavior of a cache system and it reproduces the actions that are necessary for the cache access. We show how the counter examples from the model checker are used as abstract test cases for several systems under test and how the test adaption can be accomplished with different abstraction levels.

Then, we analyze if the generated test cases have sufficient coverage on the model and the system under test and if it is possible to find any errors or specification problems.

Zusammenfassung

Testen ist eine wichtige Maßnahme, um Software mit hoher Qualität zu entwickeln und um bestimmte Sicherheitsanforderungen bei sicherheitskritischen Systemen zu erfüllen. Es ist auch sehr wichtig, Wege zu finden, um den Zeitaufwand fürs Testen zu reduzieren, weil dadurch die Entwicklungskosten erheblich gesenkt werden können. Daher ist es wesentlich, Automatisierung zu verwenden.

In dieser Arbeit wird die Entwicklung eines Tools zur automatischen Testfallgenerierung aus formalen (NuSMV) Modellen gezeigt. Das Tool verwendet graphenbasierte und logikbasierte Testabdeckungskriterien um Trap Properties zu erzeugen, welche dann einen Model Checker übergeben werden, um abstrakte Testfälle zu erzeugen. Erst werden verschiedene Testabdeckungskriterien ausgewertet und verglichen. Dann werden der Entwicklungsprozess und die Struktur des Tools beschrieben und wichtige Komponenten, wie Modell-Parsing, Gegenbeispielgenerierung mit einem Model-Checker und Testadaption, werden erläutert.

Des Weiteren wird eine Fallstudie durchgeführt, bei welcher das Tool für ein richtiges Projekt verwendet wird, um zu überprüfen, ob es praktisch verwendbar ist. In dieser Fallstudie bildet das Modell das Veralten eines Cache-Systems ab und die nötigen Aktionen für den Cache-Zugriff werden reproduziert. Es wird gezeigt, wie die Gegenbeispiele vom Model-Checker als abstrakte Testfälle auf verschiedenen zu testenden Systemen verwendet werden, und wie die Testadaption mit unterschiedlichen Abstraktionsebenen durchgeführt werden kann.

Danach wird untersucht, ob die generierten Testfälle eine ausreichende Testabdeckung auf dem Modell und zu testenden System haben, und es wird überprüft, ob es möglich ist irgendwelche Fehler oder Spezifikationsabweichungen, zu finden.

Contents

1 Introduction

1.1 Background

Software testing is a very important part of software engineering. Sometimes the time consumption for testing can be the biggest part of the software development process. The purpose of testing is to achieve software with high quality, to detect defects and to ensure that requirements and specifications are satisfied [26].

Software testing approaches can be categorized according to the knowledge about the system under test (SUT). Black box testing needs no knowledge about the internal structure of the SUT. It is considered as a black box and only the inputs and outputs are known. White box testing, which is sometimes also called structural testing, is a technique that inspects the internal structure of a system. The source code or other internal components are considered when the tests are created. A combination of both approaches is called gray box testing, which for example is used when only limited internal information is considered [11, p.7].

Model-based testing is a form of black box software or system testing. It uses an abstract model respectively formal specification of the behavior of the SUT and derives test cases from it. This model reflects certain states of the system, particular actions and decisions that lead to other states [30].

A model checker is a tool that takes a formal model in the form of a finite state machine (FSM) and a specification or property in the form of a formula as input and automatically checks if the model meets the specification. When the model does not meet the specification, a counter example is produced. The counter example is in the form of a trace and it shows how the property is violated on the model. This trace can be used as an abstract test case and

a test adapter executes it as concrete test case on the SUT. In this thesis trap properties are formulated with LTL. Linear temporal logic (LTL) is an extended version of classical logic and includes the classical logic operators for negation, implication, conjunction, disjunction and additionally also some special operators that can reflect the temporal connection between events or states [15, p. 2ff.], [20].

There are several assumptions on which model-based testing is based: The assumption that an adequate model exists for a SUT. This assumption deals with the abstraction level of the model. When the model is too abstract, then it may include too less details and so the coverage on the SUT could be too low. When it is too precise, it may be too complex and the validation can be difficult. Therefore, it is important to find a good balance between abstraction and precision. Another assumption is that if an adequate model is found, then effective model-based testing is possible and it is possible to reveal errors in the SUT. Furthermore, we assume that a high coverage on the model also leads to high coverage on the SUT, when an adequate model with a good abstraction level was found [29], [30].

Coverage criteria are important to ensure that test cases inspect certain model parts. Especially for security-critical systems it is important to ensure that there are no vulnerabilities that could be used from attackers to compromise the system. A model of security-critical system component can help to find weak points because it can be used to find deviations from the specification. These deviations might cause potential vulnerabilities. In order to find these deviations it is important to apply certain coverage criteria.

When it comes to testing, another important topic is automation. Test case generation is a complicated task, which requires experienced tester and much time. When it is done manually it is difficult to ensure that the tests meet all requirements, are consistent and fulfill a certain coverage criterion. For example it would be difficult to ensure that all paths of a certain length are covered.

Automatic test case generation has many benefits. It can produce a high number of test cases that are reliable and have high quality in a short time [27]. It is useful in order to save time and costs and is also usable from users with limited experience in this area. Furthermore, it is less error

prone, because an automated system does not forget to cover some model components.

1.2 Problem Description

Security-critical systems can have certain requirements with regard to testing. For example for some high risk areas like safety critical software in aircraft it is necessary to ensure that a specific test coverage criterion is achieved, in order to comply with security standards [28]. For other areas it is important to ensure that there are no vulnerabilities that could be used from attackers to compromise a system. It is necessary that the generated test cases cover certain model parts. For example, it could be required that all transitions of the model are visited with at least one test case. It is also important that the number of test cases does not become too large, so that it is still manageable.

There are already some approaches that deal with coverage criteria and have a focus on graph-based and logic-based coverage methods. For example Fraser, Wotawa, and Ammann [12] and Ammann, Offutt, and Xu [2] wrote about this topic and gave an overview of specific coverage criteria, but they showed no practical applications and also no combined approaches for graph-based and logic-based coverage criteria. Although Weißleder [32] presented a combined approach, it was only for very specific model types and this approach is not so easy to generalize for other models.

So the problem is that mostly only theoretical approaches for graph-based and logic-based coverage criteria exist and hardly any practical implementations, evaluations and approaches with combinations exist. The only approaches with a combined method, we could find were from Offutt et al. [22], Weißleder [32] and Ammann, Offutt, and Xu [2], but they only showed one combination method.

To the best of our knowledge, we could not find any work that includes different combination methods for graph-based and logic-based coverage criteria.

In this thesis we want to challenge this problem and find combinations for different graph-based and logic-based coverage criteria, which still produce a manageable number of test cases. Furthermore, we want to create an automatic test case generation tool that implements these combined coverage criteria and apply it to real world projects and we want to demonstrate that the test case generation with these combined coverage criteria can achieve a high instruction and branch coverage on the SUTs.

1.3 Solution

Initially we studied the field of model-based testing with a focus on coverage criteria. Then, we analyzed graph-based coverage criteria and after that logic-based coverage criteria.

Graph-based coverage criteria are coverage metrics that require test cases to regard certain graph components from the models. The most common ones are node coverage, edge coverage and path coverage. Node coverage is a very simple form of coverage. In order to fulfill node coverage it is only necessary that every node in the model is visited at least once. Edge coverage demands that all edges between each node pair are visited at least once. Path coverage is similar to edge coverage, but it does not only consider edges between node pairs, but also connected edges, which form a path. In contrast to node and edge coverage, complete path coverage is not possible for models with loops, but it is possible to fulfill path coverage for paths of a certain length [2], [12].

Logic-based coverage criteria are applied to logical expressions. In contract to graph-based coverage criteria they deal with predicates and clauses. In this thesis they are applied to edge conditions or guards. Very common logic-based coverage criteria are predicate coverage, clause coverage and active clause coverage. Predicate coverage is relatively simple, it requires that each predicate has to evaluate to true and false at least one time. Clause coverage is accomplished when each clause evaluates to true and false at least once. For Active Clause Coverage (ACC) it is required to test each clause under circumstances where it has an influence on the predicate value. Correlated Active Clause Coverage (CACC) is also referred to as masking

Modified Condition Decision Coverage (MCDC) and it is the most common version of ACC. MCDC is a very common code coverage criterion [2]. These terms will be defined in Section 2.7.

In this thesis we apply coverage criteria by using trap properties and an abstract model of the system. Trap properties are formulas that formulate a known property of the model in negated form. They are used to generate counter examples that verify a certain property of a model and they can be used as abstract test cases by a test adapter. In order to produce these counter examples, a model checker is needed.

Counter examples generated by a model checker have the advantage that they are in an abstract form and can be used very flexible for different systems, without the need to generate them again. They are represented as traces and are a sequence of states that shows a state change when the value of certain variables changes. For the test adaptation the traces are parsed and converted into variable assignment maps. Each of these traces represents a test case or a sequence of test cases depending on the system abstraction.

The test adaption with these abstract test cases is also possible in different ways. For example they can be used to directly interact with the SUT and give it inputs and compare the output. Another way is to generate source code for test function calls with test data and the expected output as arguments. However, the abstract test cases could be used for other adaption concepts.

The coverage criteria were implemented by a tool that automatically generates test cases according to a selected criterion. This tool uses a NuSMV model file of the SUT as input and generates test cases as output, which fulfill a certain selected coverage criterion. NuSMV and NuSMV models are defined in Section 2.4 and Section 2.5. The tool supports the generation of test cases with different coverage criteria, so that a comparison can be done.

The structure of the tool was made flexible in order to make it easy to include new coverage criteria. It consists of several components. The first component is the model parser, which uses a NuSMV file and makes an object representation, which can include one or multiple graphs. This object

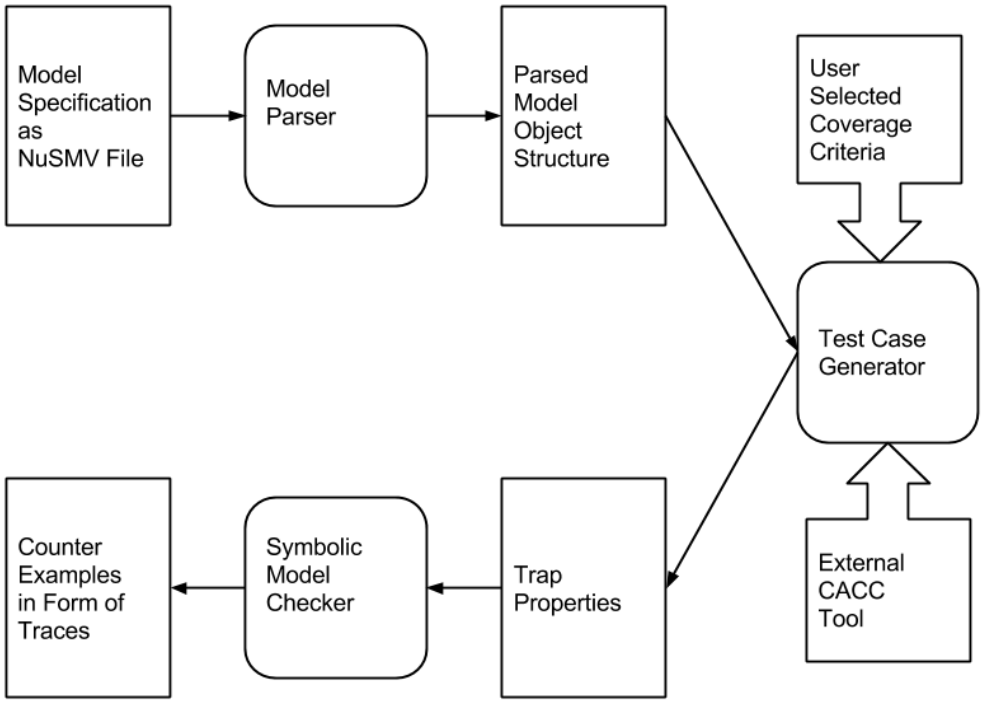

Figure 1.1: Schematic overview of the main tool components and their correlation.

representation is then used by the test case generator component, which uses it to traverse over the graphs and creates trap properties according to a certain selected coverage criterion.

When the trap properties are finished, they are used as input for a NuSMV process, which creates counter examples traces that then can be used as test cases by a test adapter.

Figure 1.1 shows an overview of the program components and how they interact.

In order to develop combination methods for graph-based and logic-based coverage we applied CACC to edge coverage and path coverage. For CACC we used a given external tool [5] that generates variable assignments for

CACC. The edge conditions are used as input for this tool and the output are different assignment sets. Each of these sets is used in combination with the trap property of the associated edge, both for edge and path coverage. The original trap property formula is combined with this generated assignments and so a new trap property is formed that considers one case for CACC.

We made several small conceptual examples in order to test the functionality of the developed tool and to show how it works. They were also used to demonstrate, how the generated counter examples could be parsed and then executed on the SUT. We used real implementations in order to show how the test adaption process works and we compared the different coverage criteria on the basis of instruction and branch coverage on these implementations.

One SUT that we used was for the triangle example as described in Section 6.1. This example could show that edge coverage could improve the code coverage compared to node coverage and it also showed what tasks of the test generation process need the most execution time. Another example was the car alarm system as described in Section 6.1. It could illustrate that the combination of graph based coverage criteria with CACC can improve the branch coverage. This improvement could be shown for edge coverage and for path coverage.

In order to check if the developed tool is usable for real world examples, we made a case study. For this case study a model of a cache implementation was developed by Franz Röck and Daniel Hein. The case study should analyze what problems can occur, which coverage criteria are best suitable for the given model, if there are certain limitations for the trap property generation or any other issues. Furthermore, the performance data and test quality was measured.

As expected, the case study showed some problems. For example the size of the model and some additional model features, which were not considered earlier, caused the necessity for some modifications to the model, to the tool and to the test adaption process. In the test adaption process the abstract test cases in the form of counter examples were parsed and converted to concrete test cases for the SUT. For the cache implementation the abstract counter examples were used to generate source code in the form of function calls to wrapper functions with test data as arguments. These wrapper functions

were used to perform the test execution and to check if the expected cache content matches the actual cache content after the execution.

Another issue was the assignment generation with the CACC tool. Define restrictions for certain variables caused assignments that were not applicable on the model. So CACC could not be applied to some problematic variables, but simple graph-based coverage like edge coverage could be used as alternative. Details about the define restrictions can be found in Section 5.9.

For the given model from the case study edge coverage could reach the best coverage on the SUT with the least number of test cases. Edge coverage combined with CACC could show the best coverage for most model parts, only for certain variables, simple edge coverage produced trap properties with a better model coverage. The model coverage was analyzed by visually highlighting the covered model parts, when a valid trap property could be produced. The coverage on the SUT was measured by line and the branch coverage. For the involved functions line coverage of 89.52% and branch coverage of 90.48% could be reached, only some conditions for error checks were not covered. Moreover, an old version of the cache implementation with a bug that was only found with a lot of manual effort was tested and the bug could be found easily.

We showed that the developed tool is useful for real world examples. Furthermore, it provides the advantage that a lot of test cases can be generated in a relatively short time, without the need to be expert for coverage criteria and with little manual effort.

1.4 Overview

Chapter 2 describes background information for the research field. Temporal logic, trap properties, NuSMV, graph-based and logic-based coverage criteria are described in detail.

In Chapter 3, the related work, which was used, is analyzed and compared.

The combination methods for graph-based and logic-based coverage criteria are shown in Chapter 4.

Chapter 5 describes how a tool was developed, how it is structured, how it can be used and what features it provides. The development of the tool is explained and also some limitations for the functionality are mentioned.

In Chapter 6, simple examples demonstrate how the tool can be used and how the generated test cases can be applied to the SUT. The examples also were used for a first evaluation of the tool and in order to test if the generation of the test cases worked as expected.

The case study is shown in Chapter 7. It describes how the tool is used for a real practical example and the problems that occur. Also the quality of the generated test cases is measured and other results are shown.

Chapter 8 provides an outlook. It shows how the tool could be improved, what features could be added and how known limitations could be reduced. It also shows what other potential future research could be done. Finally, in Chapter 9 there is the conclusion of the work, where the results of thesis are summarized.

2 Basics

In this chapter we explained some basics that were necessary for the understanding of the research field. First temporal logic, trap properties, NuSMV and NuSMV models are described. Then, different graph-based and logic-based coverage criteria are explained and compared.

2.1 Temporal Logic

The description in this section is based on the work from Lahtinen [20]. Temporal logic is an extended version of classical logic, which supports the expression of a temporal order of states or events without the need to add clock values. There are certain additional operators, which are used to describe this order. There are different forms of temporal logics, which can be distinguished by their time structure. However, only one of them was used in this thesis and it is described in the following section. [20]

2.1.1 LTL

Linear temporal logic (LTL) as defined by [25] is a version of temporal logic. It includes the classical logic operators for negation, implication, conjunction, disjunction and additionally there are also some special operators that can reflect the temporal connection between states or events without clock values. LTL has linear time structure, which means that it can be used to define formulas about the future of paths. The additional operators that LTL provides are the following:

- **X** Next operator: The expression which follows after the operator has to be true in the next state.
- **G** Globally operator: This operator requires that the expression, which follows after it, has to be true on all following states in the path.
- **F** Eventually or Finally operator: For this operator it is required that the subsequent expression has to evaluate to true at any time of the path in the future.
- **U** Until operator: The expression on the left side of the operator has to be true until the expression from the right side of the operator becomes true and it is also required that the right expression must evaluate to true eventually.
- **R** Release operator: Either the expression on the right side of the operator has to be always true, or at least it has to be true until the expression on the left side of the operator becomes true.

LTL formulas are structured as follows: The **X**, **G** and **F** operators are unary, so they require an expression that follows them. This expression can also include LTL operators. The **U** and **R** operators are binary and so they need an expression on the left and right side of them. Both expressions can also include LTL operators. The resulting formulas that can be built with these operators, are the following:

$$\begin{aligned} \phi := true|false| \\ \phi \vee \phi|\phi \wedge \phi|\neg\phi|\phi \rightarrow \phi| \\ \mathbf{X}\ \phi|\mathbf{G}\ \phi|\mathbf{F}\ \phi|\phi\ \mathbf{U}\ \phi|\phi\ \mathbf{R}\ \phi \end{aligned}$$

Although LTL has many application areas like definition of safety or liveness properties, in this thesis LTL is essentially used for the definition of trap properties. There are many examples in the following chapters, where LTL is applied in order to formulate trap properties [20], [18].

2.2 Trap Properties

Trap properties are formulas that formulate a known property of the model in negated form. They are used to generate counter example that verify

this known property. For example a property could be that it is possible to reach a certain state eventually and the negation would be that it is globally not possible to reach this state. The negation is needed in order to produce this counter example for the property, which needs to be checked. For the generation of the counter examples the trap properties are given to a model checker. A model checker is a tool that takes a formal model in the form of a FSM and a specification or property in the form of a formula as input and automatically checks if the model meets the specification. When the model does not meet the specification, a counter example is produced. A counter example is in the form of a trace that shows how the unnegated property is reached in the model. Additionally it can be used as an abstract test case. [15, p. 2ff.]

Trap properties should normally produce a counter example because they are the negation of a known property of the model. However, in this thesis we want to create trap property for expected properties and not for known properties because for the trap property generation we do not know all properties precisely. (This is for example the case, when there are define restrictions in the model as described in Section 5.9.) Hence, in this thesis trap properties are divided in valid and invalid types. A trap property is invalid, if it does not produce a counter example because it is true for the specification, otherwise it is valid. These invalid trap properties are recognized when they are presented to the model checker and then they are dismissed.

2.3 Test Case Concretization and Test Adapter

In this thesis, counter examples are in the form of traces that show a sequence of variable assignments for the model variables. This assignment sequence reveals how the given property becomes false for the model and it represent an abstract test sequence. This abstract test sequence can be converted by a test adapter to one or more concrete test cases for the SUT. A test adapter is a tool that parses abstract test cases, makes a connection with a SUT and executes these parsed test cases on the SUT. It uses certain

variable assignments from the trace as inputs for the SUT and also as expected outputs, which are compared with the real outputs of the SUT.

A concrete test case is an execution of the SUT with given inputs and expected outputs. The set of all test cases for a specific coverage criterion is called a test suite.

2.4 NuSMV

NuSMV is a new symbolic model checker (SMV) that allows model checking for temporal logic defined in LTL and Computational Tree Logic (CTL) and it has a robust and highly modular implementation that is well documented. Furthermore, there is the NuSMV input language that allows the description of models in the form of FSMs and to define model specification with LTL and CTL Formulas. [9]

2.5 NuSMV Model

The NuSMV input language can be used to define FSMs. The FSM structure can be descried with a tuple $M = \langle Q_0, (Q, E), P \rangle$
P : finite alphabet of variables
Q : finite set of states respectively nodes
$Q_0 \subseteq Q$: finite set of initial states
$E \subseteq Q \times \mathbb{B}(P) \times Q$: finite list of transitions respectively edges, the list is sorted by descending priority. Each edge has a source and target node and a guard $\mathbb{B}(P)$, which is a Boolean condition and can include variables from the alphabet of variables P. The priority of the edges is necessary, when a node has multiple outgoing edges. Then, the guard with the highest priority is checked first and the others are only evaluated when the guards of the edges with higher priority are evaluated to false.
Although NuSMV also supports asynchronous models, in thesis only synchronous models are used. For synchronous models, every model component performs a transition at each time instant. The main function of the model is to represent the transition relations of FSMs. These transition

relations can be formulated for every variable. The init keyword is used to mark the initial state and the next keyword is used to define how the value changes from the current value to the next value. When a variable has defined transitions with a next keyword then it is a deterministic variable, because there is a certain next value. When a variable has no defined transitions, then there are no constraints on its values and it is selected non-deterministically. For each deterministic variable a separate FSM is included in the model and these FSMs can be represented as independent graphs. The model can be seen as a composition of model components: $M = M_1 \times M_2 \times ... \times M_n$ and each of these model components is a FSM: $M_i = \langle Q_{0i}, (Q_i, E_i), P_i \rangle$. Model components and variables can be grouped into modules, which allow a logical separation and reuse [7].

2.6 Graph-based Coverage

Graph-based coverage criteria as referred from [2] are coverage metrics that require that certain graph components from the model, like nodes or edges, are covered from the test cases. They can be divided in two types: control flow and data flow coverage criteria. In this thesis the focus is on the first one because the models of the used applications are from this type. These graph-based coverage criteria are applied separately to each FSM that is included in the model [12]. For the definition of these criteria we say that an abstract test case t is a sequence of states and variable assignments and a test suite T is a set of these test cases.

In Figure 2.1 an example model is shown, which was used to demonstrate how this graph coverage criteria can be applied to a simple model. The associated NuSMV Model is shown in Listing 2.1. For the definition we assume that the models do not contain disconnected nodes.

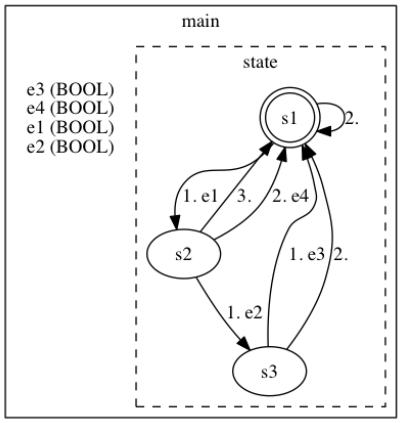

Figure 2.1: Graphical model representation of a simple model example for the explanation of the coverage criteria. (Edges for the default case, where the guard is true, have no label in order to save space.)

```
MODULE main
VAR
e1 : boolean;
e2 : boolean;
e3 : boolean;
e4 : boolean;
state : {s1, s2, s3};
ASSIGN
init(state) := s1;
next(state) :=
  case
    state = s1 & e1 : s2;
    state = s2 & e2 : s3;
    state = s3 & e3 : s1;
    state = s2 & e4 : s1;
    TRUE : s1;
  esac;
```

Listing 2.1: NuSMV test model for the illustration of coverage criteria.

2.6.1 Node Coverage

Node coverage is a very simple form of coverage. It is fulfilled, if all nodes are visited at least once during the test execution. A test suite t fulfills node coverage on a model $M = M_1 \times ... \times M_n$, iff $\forall i \in \{1...n\} \forall q \in Q_i \; \exists t \in T : q \in t$. A test case t visits a node q, when it is a counter example for the trap property $G(\neg q)$. The easiest way to fulfill node coverage is to create one trap property for each node. This method is not optimal because sometimes it is possible to fulfill node coverage with one trap property. However, it is difficult to find such trap properties automatically and it is easier to create a separate trap property for each node because so it is ensured that each node is visited at least once [12], [2].

For example when we want to create trap properties that fulfill node coverage for the model in Figure 2.1 then we only need three trap properties for the three nodes. We assert that the state cannot be reached and then the model checker computes a counterexample that shows that this specification is false because the state is reachable.

```
LTLSPEC G (state != s1);
LTLSPEC G (state != s2);
LTLSPEC G (state != s3);
```

2.6.2 Edge Coverage

Edge coverage is a coverage criterion that demands that all edges between each node pair are visited at least once. We define e_{src} as the source node of an edge, e_{guard} is the edge condition or guard, e_{other_guards} are the guards of the other edges from the source node with higher priority and e_{trg} is the target node of an edge. A test suite T fulfills edge coverage on $M = M_1 \times ... \times M_n$, iff $\forall i \in \{1...n\} \forall e \in E_i \; \exists t \in T : e \in t$. An edge is visited by a test case t if it is a counter example for the trap property: $G(e_{src} \wedge e_{guard} \wedge \neg e_{other\_guards} \rightarrow X(\neg e_{trg}))$. Edge coverage implies node coverage because when the edges are visited also the connected nodes are visited and we do not have models with disconnected nodes. In order to create a trap property for edge coverage, we need to specify the current state, where the

edge starts. We also need the guard. Then, we can use the LTL next operator and assert that the next node is not the target node from the edge. When the current node has edges with higher priority, then the trap property also has to include the negated form of the guards of these edges. This must be done, so that it is clear which outgoing edge is used [12], [2].

The following trap properties demonstrate, how edge coverage can be reached for the model shown in Figure 2.1. It is apparent that more trap properties are needed than with node coverage and that the formulas are more complex. However, the advantage is that it covers all edges and also all nodes.

```
G(((state = s2)&(e2))-> X state != s3)
G(((state = s2)&(e4)&!(e2))-> X state != s1)
G(((state = s2)&!(e2)&!(e4)) -> X state != s1)
G(((state = s1)&(e1))-> X state != s2)
G(((state = s1)&!(e1)) -> X state != s1)
G(((state = s3)&(e3))-> X state != s1)
G(((state = s3)&!(e3)) -> X state != s1)
```

Listing 2.2: Trap properties for edge coverage.

2.6.3 Path Coverage

Path coverage requires that all paths in the model are visited by at least one test case. The paths always start at an initial node. It is similar to edge coverage, but it does not only consider edges between node pairs, but also connected edges. These connected edges form a path, when they start from the initial node. In contrast to node and edge coverage, it is not possible to accomplish path coverage for most models. The reason is that when there are loops in a model then paths of infinite length occur. Therefore, it is not possible to produce all paths for such models. However, it is possible to fulfill path coverage for paths of a certain length or for paths up to a certain length. The length of a path corresponds to the number of edges that are contained in the path. For example when we want to make path coverage up to length three this would include all paths with length one, two and three. A test suite T fulfills path coverage for length k on $M = M_1 \times ... \times M_n$, iff:

$\forall i \in \{1...n\} \forall e_1...e_k \in E_i \wedge e_{1\_src} \in Q_{0i} \wedge e_{2\_src} = e_{1\_trg} \wedge ... \wedge e_{k\_src} = e_{k-1\_trg}$
$\exists t \in T : e_1...e_k \in t$
A test case t visits a path $e_1...e_k$, if it is a counter example for the trap property: $G(e_{1\_src} \wedge e_{1\_guard} \wedge \neg e_{1\_other\_guards} \rightarrow X(e_{2\_src} \wedge e_{2\_guard} \wedge \neg e_{2\_other\_guards}$ $\rightarrow X(... \rightarrow X(\neg e_{k\_trg})...)))$ In order to create a trap property for a path we start at the initial node. Then, we use the condition of the edge we want to choose and the LTL next operator for the next node. When there are edges with higher priorities we also need to include them just like with edge coverage. The difference is that we do not want to stop at the next node. The same procedure as for the first node can be used for the next node. We can use another next operator in order to connect the next edge and we continue this procedure until we reach the target node of the path. Only for the last node we need to assert that the next node is not the target node in order to make the negation for the trap property. Although this coverage criterion is powerful because it deals with many specific state sequences, it is also more complex. We have to be careful when we specify a path length that is too short, we might not reach all model parts because the paths start from the initial node. When the length is too long the number of generated paths becomes too high and impracticable. So we have to analyze the model first and find out which path length is suitable [12], [2].

The following trap properties demonstrate how path coverage is reached for the model show in Figure 2.1 with paths of length 3. For these trap properties it is important to be careful with the braces for the next operator. The variable name from the state variable was replaced in order to make the output shorter and easier to read.

```
G(((s=s1&e1)->X((s=s2&e2)->X((s=s3&e3)->X(s!=s1)))))
G(((s=s1&e1)->X((s=s2&e2)->X((s=s3&!e3)->X(s!=s1)))))
G(((s=s1&e1)->X((s=s2&e4&!e2)->X((s=s1&e1)->X(s!=s2)))))
G(((s=s1&e1)->X((s=s2&e4&!e2)->X((s=s1&!e1)->X(s!=s1)))))
G(((s=s1&e1)->X((s=s2&!e2&!e4)->X((s=s1&e1)->X(s!=s2)))))
G(((s=s1&e1)->X((s=s2&!e2&!e4)->X((s=s1&!e1)->X(s!=s1)))))
G(((s=s1&!e1)->X((s=s1&e1)->X((s=s2&e2)->X (s!=s3)))))
G(((s=s1&!e1)->X((s=s1&e1)->X((s=s2&e4&!e2)->X(s!=s1)))))
G(((s=s1&!e1)->X((s=s1&e1)->X((s=s2&!e2&!e4)->X(s!=s1)))))
G(((s=s1&!e1)->X((s=s1&!e1)->X((s=s1&e1)->X(s!=s2)))))
G(((s=s1&!e1)->X((s=s1&!e1)->X((s=s1&!e1)->X(s!=s1)))))
```

Listing 2.3: Trap properties for path coverage.

2.7 Logic-based Coverage

The description in this section is based on the work from Ammann, Offutt, and Xu [2] and the formal definitions are based on the work from Bloem et al. [5]. Logic-based coverage criteria are coverage metrics that are applied to logical expressions. In contrast to graph-based coverage criteria they deal with predicates and clauses and not with graph components. Logical expressions can occur in a variety of places in software, but for this thesis logic-based coverage criteria are applied to the edge conditions or guards of the given formal models.

A clause is an expression that can be true, false, a Boolean variable or the conjunction between two Boolean or non-Boolean variables with the following notion: $\{a > b; a \geq b; a < b; a \leq b; a = b; a \neq b\}$ It does not contain any logical operators. A predicate is a logical expression and it is evaluated to true or false. It can be a clause or the conjunction of multiple clauses or predicates with the following logic operators: $\{ \neg p; p1 \wedge p2; p1 \vee p2 \}$. For our models the predicates are the edge conditions or guards of the edges. For the definition of logic-based coverage, we introduce the following semantics: A test case t for a predicate p is an assignments for all variables $t : V \rightarrow \mathbb{D}$. For the evaluation of the truth value of a predicate p or a clause c we write $p(t)$ and $c(t)$. A clause c is part of the set of clauses of a predicate $c \in Cs(p)$.

2.7.1 Predicate Coverage

Predicate coverage is a simple form of coverage, it requires that each predicate has to evaluate to true and false at least one time. Predicate coverage is fulfilled by a test suite T, iff for each predicate: $\exists t, t' \in T : p(t) \wedge \neg p(t')$. A problem of this coverage criterion is that only the predicates as a whole are evaluated and not the clauses and so when there is a problem with a clause, this might not be covered. For example when there is a clause in the model that is not correctly mapped in the SUT, then the test cases, which are generated with this coverage criterion might not find this bug [2].

2.7.2 Clause Coverage (CC)

Clause coverage (CC) as referred from [2] is accomplished when each clause of each predicate evaluates to true and false at least once. A test suite T fulfills clause coverage, iff for each predicate: $\forall c \in Cs(p) : \exists t, t' \in T : c(t) \wedge \neg c(t')$. Although this coverage criterion considers all clauses, it has the problem that it is not guaranteed that predicate coverage is fulfilled when clause coverage is fulfilled. For example when there is a predicate with some clauses, it is possible to make two assignments, one arbitrary and one, which has each clause inverted of this arbitrary assignment. Then, these two assignments fulfill clause coverage, but not necessarily predicate coverage. This problem can be easily shown with a simple predicate $p = a > b \vee b = 5$, which can have 4 possible assignments as demonstrated in Table 2.1: For

	a	b	a >b	$b = 5$	$a > b \vee b = 5$
1	6	5	T	T	T
2	3	2	T	F	T
3	4	5	F	T	T
4	2	6	F	F	F

Table 2.1: Truth table with variable assignments for the demonstration of clause coverage.

this predicate the assignments from row 2 and row 3 would be enough to fulfill clause coverage, but these two rows would not fulfill predicate coverage. Therefore, clause coverage does not implicate predicate coverage. However, for many cases predicate coverage is reached when test cases for clause coverage are generated. For example when row 1 and row 4 would be used, then both clause coverage and predicate coverage are reached. The problem with this coverage criterion is that important branches from the SUT could be missed because not all condition cases are evaluated.

2.7.3 Combinatorial Coverage (CoC)

Combinatorial coverage (CoC), sometimes also called multiple condition coverage, requires that for each predicate all possible combination of truth values must be evaluated for the clauses. CoC is fulfilled by a test suite T,

iff for each predicate for all clauses $c_1 ... c_n \in Cs(p)$:
$\exists t \in T: \quad c_1(t) \wedge \quad c_2(t) \wedge ... \wedge \quad c_n(t) \wedge$
$\exists t \in T: \neg c_1(t) \wedge \quad c_2(t) \wedge ... \wedge \quad c_n(t) \wedge$
$\exists t \in T: \quad c_1(t) \wedge \neg c_2(t) \wedge ... \wedge \quad c_n(t) \wedge$
$\exists t \in T: \neg c_1(t) \wedge \neg c_n(t) \wedge ... \wedge \quad c_n(t) \wedge ... \wedge$
$\exists t \in T: \quad c_1(t) \wedge \quad c_2(t) \wedge ... \wedge \neg c_n(t) \wedge$
$\exists t \in T: \neg c_1(t) \wedge \quad c_2(t) \wedge ... \wedge \neg c_n(t) \wedge$
$\exists t \in T: \quad c_1(t) \wedge \neg c_2(t) \wedge ... \wedge \neg c_n(t) \wedge ... \wedge$
$\exists t \in T: \neg c_1(t) \wedge \neg c_2(t) \wedge ... \wedge \neg c_n(t).$
This is equivalent to taking all rows from the truth table of each predicate and forming a test case of each row. When we have a predicate with n independent clauses it produces 2^n test cases. Although this coverage criterion includes both clause and predicate coverage, the problem with this criterion is that the number of test cases becomes too large and therefore it is not practically usable, when the number of clauses is not very small [2].

2.7.4 Active Clause Coverage (ACC)

Active clause coverage (ACC) requires that each clause is tested under circumstances where it affects the predicate value. This is done to avoid an affect called masking, where regardless of the value of a clause the value of the predicate does not change. Therefore, it is required that the value of the predicate changes, when the value of a clause is changed. In order accomplish such an approach, the concepts of major clause and minor clause were introduced. In order to define these concepts formally, we introduce the following semantics: A clause c is a major clause, if it determines the value of p for a test case: $det(c, p, t)$, iff $p(t) \neq p[c|\neg c(t)](t)$. This means that the value of the predicate is changed, when the value of the clause is changed. The other clauses, called minor clauses c', then must have values that allow a change of the predicate value when the major clause value is changed. In order to achieve active clause coverage it is required that for each major clause and predicate, the minor clauses have to be chosen, so that the major clause determines the value of the predicate and also the true and the false case of the major clause must be evaluated [2], [5].

The advantage of this approach is that it accomplishes both predicate coverage and clause coverage and the number of test cases that are required for this approach is also manageable. ACC is very similar to earlier descriptions of modified condition decision coverage (MCDC), which is a very common code coverage criterion. However, there were some disagreements about the definition, concerning what should be done with the minor clause. Therefore, different versions of ACC are mentioned: The most general version is called general active clause coverage (GACC), which makes no limitations for the minor clauses. GACC is fulfilled by a test suite T, iff for each predicate: $\forall c \in Cs(p) : \exists t, t' \in T : c(t) \wedge \neg c(t') \wedge det(c, p, t) \wedge det(c, p, t')$. The problem with this approach is that it does not imply predicate coverage and therefore other versions of ACC, which are described in the following sections, should be preferred [2], [5].

2.7.5 Correlated Active Clause Coverage (CACC)

Correlated active clause coverage (CACC) is also referred to as masking MCDC. It requires that for each predicate and each major clause, the values of the minor clauses must be chosen, so that the value of the major clause determines the value of the predicate and the both the true and the false case of the predicate must be evaluated. CACC is fulfilled by a test suite T, iff for each predicate:
$\exists t, t' \in T : p(t) \wedge \neg p(t') \wedge$
$\forall c \in Cs(p) : \exists t, t' \in T : c(t) \wedge \neg c(t') \wedge det(c, p, t) \wedge det(c, p, t')$
Table 2.2 shows test cases for the major clause a for an example predicate $a \wedge (a \neq b \vee a > 5)$. It is apparent that the major clause $a < b$ dictates the predicate value when the $a \neq b \vee a > 5$ part is true, which is the case for two assignments. Any combination of these assignments with the true and false case of the major clause makes a test case set that accomplishes CACC. For the given example this is done by choosing a pair of test cases one from row 1 and 2 and the other from row 5, 6 and 7. This results in 6 possible combinations that fulfill CACC for the major clause $a < b$ [2], [5].

	a	b	$a < b$	$a \neq b$	$a > 5$	$a \wedge (a \neq b \vee a > 5)$
1	6	7	T	T	T	T
2	1	2	T	T	F	T
3	-	-	T	F	T	not possible
4	7	6	T	F	F	F
5	7	6	F	T	T	F
6	4	3	F	T	F	F
7	6	6	F	F	T	F
8	1	1	F	F	F	F

Table 2.2: Truth table with variable assignments for the demonstration of CACC and RACC. (It is not possible to find an variable assignment for all truth values of the clauses.)

2.7.6 Restricted Active Clause Coverage (RACC)

Restricted active clause coverage (RACC) is a coverage criterion that requires the following: For each predicate and major clause, the minor clauses have to be chosen so that the major clause determines the value of the predicate. Additionally the assignments for the minor clauses have to stay the same when the major clause is true and when it is false. RACC is fulfilled by a test suite T, iff for each predicate:
$\exists t, t' \in$: $p(t) \wedge \neg p(t') \wedge \forall c \in Cs(p) : \exists t, t' \in T : c(t) \wedge \neg c(t') \wedge det(c, p, t) \wedge$
$det(c, p, t') \wedge$
$\forall c' \in \{Cs(p) \backslash c\} : c'(t) = c'(t')$
In Table 2.2 the possible test case set for a simple example predicate $a \wedge (a \neq b \vee a > 5)$ for the major clause $a < b$ is shown. In this example there are only two row combinations that fulfill RACC (row 1 paired with 5 and row 2 with 6) [2], [5].

2.7.7 RACC versus CACC

For some formulas with dependencies for clauses for example when there are combinations of clause values that are prohibited, CACC can be reached, but it is not possible to accomplish RACC. These dependencies between clauses are called coupling. Two clauses are strongly coupled, when the

change of the value of one clause always changes the value of the other. They are weakly coupled, when the change of the value of one clause only sometimes affects the other clause [8]. The problem with RACC is that it does not allow a very flexible assignment of combinations. As it was shown in the previous simple example for RACC there were only two possible ways to make a test case set. For CACC there were six possible ways and the assignment of the minor clauses could also be selected more flexible, which makes the generation of test cases for CACC easier [2].

3 Related work

The first approach in the area of model-based testing respectively testing based on formal specifications, was presented by Bernot, Gaudel, and Marre [3]. Many other publications followed after this work: [33], [23], [22], [12].

However, many approaches only focus on a theoretical definition of coverage criteria and have no practical implementation or evaluations on real world examples. For instance Fraser, Wotawa, and Ammann [12] presented a survey of different testing methods with model checkers and showed advantages and issues. Furthermore, they gave an overview of coverage criteria, trap property definition and mutation-based test case generation. Weyuker, Goradia, and Singh [33] also showed a more theoretical approach to generate test data from specifications. Weißleder [32] gave a broader theoretical overview of model-based testing and also showed a method for combining coverage criteria, but another combination approach and also different model types were used.

Some approaches have a stronger focus on test case generation for the data flow in the program and not control flow. For example, Hong et al. [17] showed an approach in the area of data flow testing, where an extended version of temporal logic and definition-use pairs for variables were used in order to describe certain data flow coverage criteria.

Fraser [11] gave an overview of test case generation with model checkers. The focus of this work was more towards optimization possibilities for test case generation. So evaluations about the effectiveness of testing, improvement of LTL formulas, model checker performance and mutant minimization and many other optimization aspects, were shown.

Some approaches focus more on logic-based coverage criteria, for example Bloem et al. [5] showed an evaluation of CACC or MCDC and it was also

analyzed how this coverage criterion could be applied to a real practical system. Furthermore, the developed CACC tool was compared with hand-crafted tests and it was shown how much code coverage could be reached. Chilenski [8] also showed a comparison of different coverage criteria, but the focus was also only on logic-based coverage criteria.

Lahtinen [20] gives a good overview of temporal logic and about NuSMV. This work also showed how NuSMV counter examples are created and how they can be converted in a different simplified representation.

A very similar work was presented by Offutt et al. [22], which showed a test data generation technique from formal specifications with MCDC. They presented an approach, where parse trees for decisions are traversed in order to generate test cases for MCDC. They also evaluated this approach with small examples and with a case study. However, the difference is that this approach has more limitations, it only works for Boolean variables. Another difference is that the work did not show different approaches for a combination of logic-based and graph-based coverage criteria.

The closest related work we found was from Ammann, Offutt, and Xu [2]. It was also focused on coverage criteria. Many different versions of graph-based and logic-based coverage criteria are explained and compared in this work. Furthermore, it was shown how logic based coverage criteria can be applied for formal models in the form of state machines. The authors demonstrated this approach on a small example. However, the difference is that they did not show an extensive evaluation of their approach with real world examples and they did not demonstrate a practical implementation of their approach. Furthermore, they did not show how logic-based and graph-based coverage criteria could be combined in different ways.

There are other similar tools that can be used to generate test cases as well. For example, Spec Explorer also is a tool for model-based testing. However, it uses a symbolic exploration algorithm on the model in order to generate test cases [31]. Our tool uses trap properties and a model checker for the test case generation. Another example is Uppaal Cover [16]. This test generation tool also supports automatic test case generation for a dynamically selected coverage criterion. For example it offers node or edge coverage, but it does not provide logic-based coverage criteria and it also uses a different generation method.

4 Combination Method for Coverage Criteria

In this chapter we describe our combination methods for graph-based and logic-based coverage criteria. We combine edge coverage and path coverage with CACC. Node coverage was not combined with CACC, because it does not deal with edge conditions and CACC only is applied to them. The idea of the combinations is to increase the coverage of the generated test cases on the model and the SUT by applying the strengths of both logic-based and graph-based coverage criteria. With our combination methods we want to create test cases that reach a better coverage than a method that only uses graph-based or logic-based coverage. In addition, our combination methods should be practically usable and produce a manageable number of test cases for real world projects.

4.1 Combination of Edge Coverage with CACC

Our combination of edge coverage with CACC is fulfilled for a model $M = M_1 \times ... \times M_n$ by test suite $T = \{\{q_0a_0q_1a_1...\}, \{q_0a_0q_1a_1...\}, ...\}$ and $q_0a_0q_1a_1...$ is a sequence of states $q_0, q1, ... \in Q_i$ and assignments sets a, iff:
$\forall i \in \{1...n\} \forall e \in E_i$, if $p = e_{guard} \wedge \neg e_{other\_guards}$
$\exists q_j a_j q_k \in T : q_j = e_{src} \wedge p(a_j) \wedge q_k = e_{trg} \wedge$
$\exists a, a' \in T : p(a) \wedge \neg p(a') \wedge$
$\forall c \in Cs(p) : \exists a, a' \in T : c(a) \wedge \neg c(a') \wedge det(c, p, a) \wedge det(c, p, a')$

For the creation of the combined trap properties we use the trap properties for edge coverage as explained in Section 2.6.2. From each of these trap properties we use the guard and the negation of the guards of the other

edges with higher priority $e_{guard} \wedge \neg e_{other\_guards}$ as input formula to generate the assignments for CACC. For the generation we used an external tool that was presented by Bloem et al. [5]. This CACC tool uses an SMT solver to produce the variable assignments and it can handle complex input formulas very fast.

CACC produces sets of assignments for the included variables in the input formula. These assignment sets A can be used as predicates as well and we combine them with the original trap property by replacing the input formula. The original trap property has the following form:

$G(e_{src} \wedge e_{guard} \wedge \neg e_{other\_guards} \rightarrow X(\neg e_{trg}))$

For each assignment set we create a separate new trap property as follows:

$$\forall a \in A : \begin{cases} G((e_{src} \wedge a) \rightarrow X(\neg e_{trg})) & \text{if } a \text{ makes } e_{guard} \wedge \neg e_{other\_guards} \text{ true} \\ G((e_{src} \wedge a) \rightarrow X(e_{trg})) & \text{if } a \text{ makes } e_{guard} \wedge \neg e_{other\_guards} \text{ false} \end{cases}$$

CACC includes both true and false cases for the input formula. In order to create trap properties that negate a property, we have to handle the true and false cases differently. For the true cases, we can keep the negation for the target state of the original trap property. For a false case we remove the negation because when the guard is false or when the guard of a higher priority edge is true, then a different edge will be used and a different target state will be reached. Therefore, the trap property states that the next state is the target state of the edge. This is not true for an assignment set for a false case and so we have the necessary negation for the trap property.

4.2 Combination of Path Coverage with CACC

Path coverage with CACC for all path of length k is fulfilled for a model $M = M_1 \times ... \times M_n$ by test suite $T = \{\{q_0 a_0 q_1 a_1 ...\}, \{q_0 a_0 q_1 a_1 ...\}, ...\}$, iff:
$\forall i \in \{1...n\} \forall e_1 ... e_k \in E_i \wedge e_{1\_src} \in Q_{0i} \wedge e_{2\_src} = e_{1\_trg} \wedge ... \wedge e_{k\_src} = e_{k-1\_trg}$, if
$p = e_{k\_guard} \wedge \neg e_{k\_other\_guards}$
$\exists q_0 a_0 ... a_{k-1} q_k \in T : q_0 = e_{1\_src} \wedge e_{1\_guard}(a_0) \wedge ... \wedge p(a_{k-1}) \wedge q_k = e_{k\_trg} \wedge$
$\exists a, a' \in T : p(a) \wedge \neg p(a') \wedge$
$\forall c \in Cs(p) : \exists a, a' \in T : c(a) \wedge \neg c(a') \wedge det(c, p, a) \wedge det(c, p, a')$

The combination of path coverage with CACC was done similar to edge coverage, but the variable assignment sets generated by the external CACC tool were only applied to the last edge of the path. For path coverage as described in Section 2.6.3 we have trap properties in the following form:
$G(e_{1\_src} \wedge e_{1\_guard} \wedge \neg e_{1\_other\_guards} \rightarrow X(e_{2\_src} \wedge e_{2\_guard} \wedge \neg e_{2\_other\_guards}$
$\rightarrow X(... \rightarrow X(e_{k\_src} \wedge e_{k\_guard} \wedge \neg e_{k\_other\_guards} \rightarrow X(\neg e_{k\_trg}))...)))$

For our combination approach we apply CACC for the last edge e_k of each path $e_1...e_k$. As input formula for CACC we use $e_{k\_guard} \wedge \neg e_{k\_other\_guards}$, just as for edge coverage with CACC and also sets of assignments A are produced for the included variables. For each assignment set $a \in A$ we create a separate new trap property as follows:
if a makes $e_{k\_guard} \wedge \neg e_{k\_other\_guards}$ true:
$G(e_{1\_src} \wedge e_{1\_guard} \wedge \neg e_{1\_other\_guards} \rightarrow X(e_{2\_src} \wedge e_{2\_guard} \wedge \neg e_{2\_other\_guards}$
$\rightarrow X(... \rightarrow X(e_{k\_src} \wedge a \rightarrow X(\neg e_{k\_trg}))...)))$
if a makes $e_{k\_guard} \wedge \neg e_{k\_other\_guards}$ false:
$G(e_{1\_src} \wedge e_{1\_guard} \wedge \neg e_{1\_other\_guards} \rightarrow X(e_{2\_src} \wedge e_{2\_guard} \wedge \neg e_{2\_other\_guards}$
$\rightarrow X(... \rightarrow X(e_{k\_src} \wedge a \rightarrow X(e_{k\_trg}))...)))$

We apply CACC only to the last edge because otherwise the number of trap properties would be unmanageable, when all CACC assignments of one edge would be combined with the CACC assignments of other edges. The problem with this approach is that CACC might not be applied for the first edges in the path. However, it is possible to use the path coverage algorithm for all paths up to a certain length. With this method CACC can also be applied for the first edges in a path, without creating too many trap properties.

5 Tool Development

In order to show that the described coverage criteria present a compelling method to create abstract test cases and also to show that they are practically usable, we developed a Java Tool. This tool uses a given NuSMV model file as input and generates test cases as output, which fulfill a certain selected coverage criterion. These test cases can be used by a test adapter in order to test the functionality of a SUT. Java was used for the development because it is platform independent and also because there were already existing projects, which are described later. We designed our tool with a component-based structure in order to make it easily extendable and to allow the integration of further coverage criteria. Furthermore, we included a feature for process handling in order to make it possible to use external applications, for example for model checking or for the assignment generation for CACC.

In this chapter the development process of the tool, the component structure, included external applications, algorithmic design and configuration options are explained. Furthermore, additional features like model visualization and known limitations of the tool are mentioned.

5.1 Component Structure

We developed the tool with object-oriented programming and separated it in different components for the different tasks of the generation process. The first component is the model parser, which takes the given NuSMV file and makes an object representation, which has class instances for the model components. The model contains at least one deterministic variable, which is represented as a graph. The states are represented as node objects and

the transitions as edge objects. The representation is then used by the test case generator component, which uses it to traverse over the graphs and creates trap properties according to a certain selected coverage criterion.

Instead of including edge conditions in the trap properties, it is also possible to include variable assignments. These assignments are generated by an external tool that takes the edge condition as input and produces test data for CACC. We write the formulas from the edge conditions in an input file. Then a process is started for the CACC tool, which reads the input file and produces assignment for the variables that are part of these conditions. These assignments are then used for the trap property definition. The development and analysis of the CACC tool was described in [5].

After all trap properties are created, a NuSMV process is started, which uses them and the NuSMV model file in order to create counter example traces. These traces represent abstract test cases and a test adapter can execute them on the SUT.

Figure 1.1 shows an overview of the tool components and the inputs and outputs of each component are displayed. Furthermore, the correlation respectively the sequence, in which the components are used, is shown and also where the configuration arguments from the user are applied and where the given external tool is used.

5.2 Model Parsing

The first important part of the tool is the model parser. This component uses a given NuSMV file as input and generates an object structure. The model parts that are relevant for this object structure are the variable definitions and state transitions and initial states for deterministic variables. The variable definitions include the name of the variable, the type and the options for enumeration type variables. The state transitions include the source and the target state and also a condition that has to be fulfilled so that the transition can occur. In order to extract information from the given NuSMV model file and convert it in a form that is processable, it was necessary to find a suitable parsing solution.

Due to the fact that model or file parsing is a common task, this part was not implemented from scratch. A few projects were analyzed in order to find a usable solution, which already performs parsing of NuSMV files. An Xtext project called NuSMV-Tools [14] was found, which accomplished this task properly.

5.2.1 Xtext

Xtext is an open source framework for the development of programming languages and domain specific languages and it is mostly used in combination with the Eclipse integrated development environment (IDE) [24]. It provides the possibility to develop language extensions for Eclipse and easy ways to accomplish features like syntax tree generation, syntax highlighting, editor tools, parsing and so on. For this project only the parsing process was relevant.

5.2.2 Conversion of the Parsed Object Structure

After the NuSMV file is parsed with the given parser, the produced object structure needs some conversions. This is necessary because it includes unnecessary information and complicated representation forms. Some parts, for which a string representation is enough, are stored in a complicated abstract form. In order to produce a usable object structure, many parts are stored in different classes. For nodes, edges and variables separated classes are used with associated attributes. Nevertheless, for some model parts the generated structure is already very convenient. For instance the edge conditions or guards are already stored in a recursive tree structure, which makes it easy to process these formulas.

5.2.3 Parsed Model Structure

In Figure 5.1 the UML class diagram of the model classes is shown. These classes are for the object structure of the parsed model. The attributes of

a model are mapped in a project object, which consists of one or more modules. Each of these modules can have a number of variables and they are divided in two groups. There are deterministic and nondeterministic variables, which are represented with the same base class. The difference is that the deterministic variables include a graph representation and have classes for nodes and edges.

5.3 Model Visualization

Visualization is a useful feature, in order to verify manually that the parsing was done correctly and that the model was designed as expected. It also helps to provide a quick general view of the model and this makes it possible to check easily, if there are any flaws. For example, it can be used to check, if there are no missing edges or if an edge is misplaced between the wrong nodes.

To visualize the object structure, each object has a toString method and the combined output of these methods is a DOT string. DOT from Graphviz [10] is a textual description language for graphs. It offers a very simple way to describe a graph in a textual form and there are various tools that are able to generate a visual image of the graph from this textual representation. The great advantage of the DOT description language is that it is not required to deal with aspects like the positioning of the nodes or how the edges are placed. These settings are automatically chosen from the generation tools and most of the time the generated graph is formatted neatly. However, there are many styling options that are useful to change some default design decisions or to change the formatting or styling features [19], [13].

Listing 5.1 shows the generate DOT description of the model of an access policy system and in Figure 5.2 the generated visual image, representation from the described graph, is shown. The model of this example demonstrates a simple conceptual access policy system, which regulates the access to the file system. In this example the access to the file system should only be granted to users with the required permissions. For example a user, which has super user permissions should be allowed to access the file system. Furthermore, it should not be allowed to open files that are password

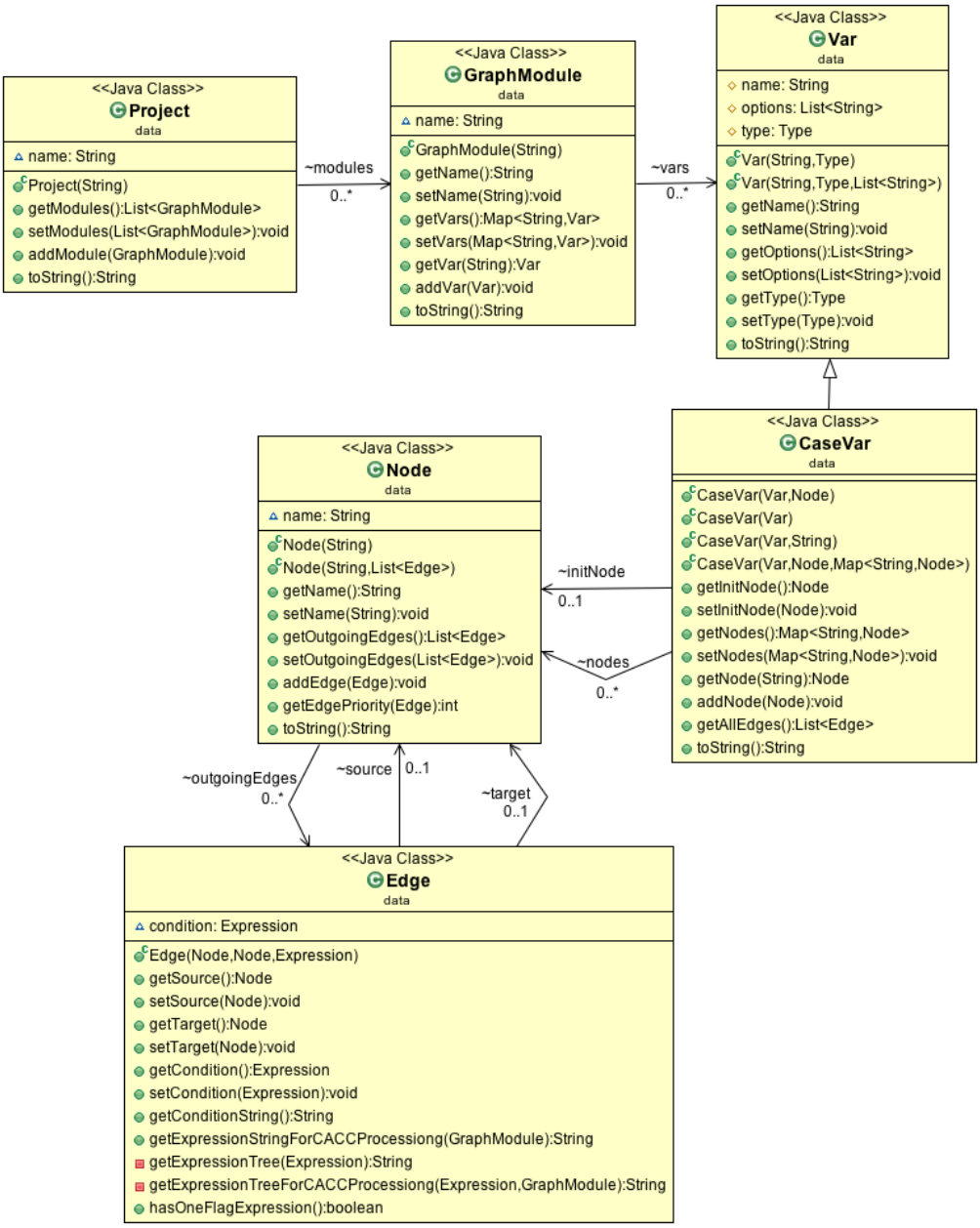

Figure 5.1: UML class diagram of the classes for the model representation after the parsing was done.

protected. When one of the access rules is violated, the system goes in the locked state, otherwise it remains in the idle state.

```
digraph AccessPol{ subgraph clustermain{
  label="main";
  vars [shape = none label="fileOpen(BOOL)\n
                            isSU(BOOL)\n
                            pwProtection(BOOL)\n
                            command(ENUM)\n
                            reset(BOOL)\n
                            fileExists(BOOL)\n"];
    subgraph clusterstate { style = "dashed";
                            label = "state";
      idle [shape=doublecircle];
      locked;
      locked -> idle [label="1. reset"];
      idle -> idle [label="1. isSU"];
      idle -> locked [label=
      "2. command = open & pwProtection"];
      idle -> locked [label="4. "];
      idle -> idle [label="3. fileExists"];
      locked -> locked [label="2. "];}
  }
}
```

Listing 5.1: DOT output of a small access policy example.

5.4 Test Case Generation

For the test case generation we used the object structure, which was created by the model parser. According to a user selected coverage criterion, the graphs from the object structure are traversed in order to create trap properties. These trap properties are given to a symbolic model checker, which checks if the given formulas are true or false for the model. For a valid trap property the formula must be false and then the model checker produces a counter example, which can be used as an abstract test case.

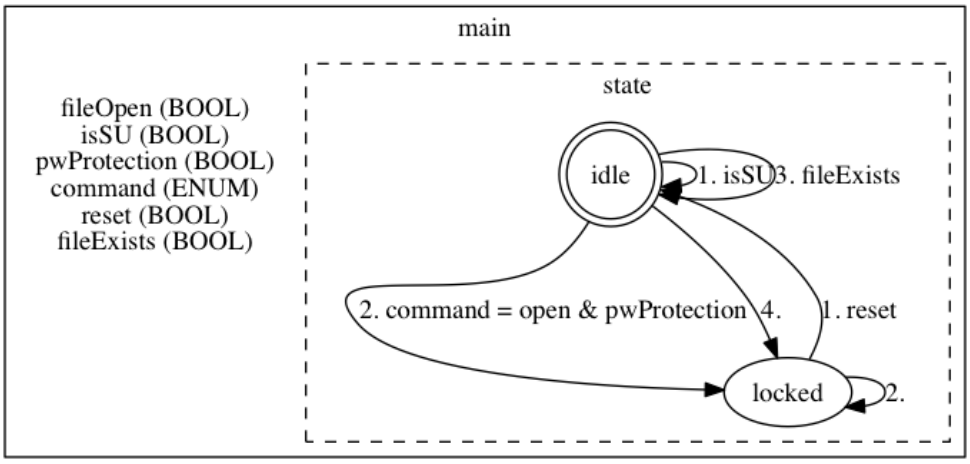

Figure 5.2: Generated visual representation from DOT description for an access policy example.

These abstract test cases are converted by a test adapter to concrete test cases for the SUT.

5.4.1 Trap Property Generation

A model is represented by a project object that includes one or more modules. Each module consists of one or more variables. For the trap property generation the focus is on the deterministic variables because they have a graph structure, which is important for the trap properties. The nondeterministic variables are not so important because for most coverage criteria only their name is needed and they only occur in the edge condition. For each deterministic variable the corresponding graph is traversed. Depending on the coverage criterion, different graph components are relevant. For example, for node coverage all nodes are traversed and for edge coverage all edges. During this process the object attributes like the name of the nodes or the edge condition are used in combination with logic and LTL operator in order to create the formulas for the trap properties. How this is done was

already described in Section 2.6 and further implementation details will follow in Section 5.5.

5.4.2 Counter Example Generation

The generated trap properties from the test generator component are used as input for a symbolic model checker. In this implementation NuSMV was not only used in order to specify the model, but also as symbolic model checker. So a NuSMV process is started, then the model is loaded and the formulas for the trap properties are given to the process one by one. If a trap property is valid the symbolic model checker returns a counter example in the form of an execution trace, otherwise it states that the formula is true for the given model and no counter example is generated. Each of these counter examples is then written to the output file for the test cases. After the test case generation process is finished, a test adapter can use this output file. So a test adapter can parse the abstract test cases from the file and then execute them as concrete test cases on the SUT. Due to the fact that each SUT needs an individual test adapter, we described it for each illustrated SUT separately.

Listing 5.2 shows a counter example for a trap property for edge coverage, applied to the model shown in Figure 5.2. It can be seen that the NuSMV process states that the formula from the trap property is false. Furthermore, a LTL Counterexample in the form of a trace is shown. This trace demonstrates how a change of the nondeterministic variables influences the deterministic variables.

```
-- specification  G ((state = locked & reset)
                        -> X state != idle)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 5.1 <-
  isSU = TRUE
  pwProtection = FALSE
  fileExists = FALSE
  fileOpen = FALSE
  command = open
  reset = FALSE
  state = idle
-> State: 5.2 <-
  isSU = FALSE
  pwProtection = TRUE
  fileExists = TRUE
-> State: 5.3 <-
  pwProtection = FALSE
  fileExists = FALSE
  reset = TRUE
  state = locked
-- Loop starts here
-> State: 5.4 <-
  isSU = TRUE
  reset = FALSE
  state = idle
-> State: 5.5 <-
```

Listing 5.2: Counter example in the form of a trace from NuSMV process.

5.5 Implemented Coverage Criteria

In this section we describe the actual coverage criteria, which are supported from our tool, and we highlight how they were implemented. First, we illustrate the graph-based coverage criteria and then our combination approach for logic-based and graph-based coverage criteria.

5.5.1 Graph-based Coverage Criteria

The implementation of graph-based coverage criteria was done according to the definitions in Section 2.6 and by using the data from the parsed model. For node coverage it was required to iterate over all node objects. During this iteration the name of the node and the concerning variable name was used in order to create the formulas for the trap properties. The implementation of edge coverage required an iteration loop over all edge objects. For each edge the guard or edge condition, the name of the target node and the source node were used for the trap property formulas. For edges that did not have the highest priority, it was also important to include the negated guards of the higher priority edges in the formula. Therefore, the guards of the previous iteration steps were stored so that they could be used later in negated form for the formulas for lower priority edges. Details for the creation of the formulas can be found in Section 2.6.

The implementation of trap properties for path coverage was more complex than for node or edge coverage, because an algorithm with iteration was not enough to build the requested paths. Therefore, a recursive algorithm was designed, which goes to each outgoing edge of a node and continues to call itself again for the target nodes of the outgoing edges. During this process, the data that is needed for the trap properties is obtained from the nodes and edges and the process stops when the path has reached the length that was specified from the input parameter.

When the user specifies that all paths up to a certain length are needed, then this algorithm for path coverage is simply executed several times for the different lengths.

5.5.2 Combination of Graph-based Coverage with CACC

As explained in Section 5.1, an external tool [5] for the generations of variable assignments for CACC coverage was used. In order to integrate this tool, it was started as a separate process. We used the Java process builder, which provides an easy way to manage a process and to interact with this process. In order to communicate with the process of the CACC tool, we write the formula for the edge conditions in an input file. (This is done for each edge separately.) Then, the process runs and uses the formula from this input file to generate an output file with a number of assignment sets for the variables that occur in the edge conditions. Each of these sets has assignment values for a separate test case and it contains values for all variables that were included in the input formula. The assignment sets are used in combination with the trap property of the associated edge. The combination was done as described in Section 4.1.

The assignment sets have one limitation. It is not clear if a given assignment set represents a true or false case for the edge condition. CACC includes both true and false cases for the input formula and the output of the CACC tool provides no way to distinguish them. Hence, we create a second trap property for each CACC assignment set, which has no negation of the expected next state. This is done because for a false case the next state is not the expected target state from the edge, but could be any target state that is reachable with an outgoing edge. Therefore, the trap property states that the next state is the target state from the edge. This is not true for an assignment set for a negative case and so we have the necessary negation for the trap property.

Listing 5.3 shows the generated CACC assignment sets for the following edge condition:

```
!doClose & doOpen & lock != locked
```

This edge condition is from the door variable of the car alarm example, which is explained in Section 6.2 and illustrated in Figure 6.5. In order to fulfill the requirements for the CACC tool, the formula needs conversions, which are demonstrated in the following example:

```
(~(doClose)) & (doOpen) & (lock_CONV_ENUM != 1)
```

It can be seen that the enumerative type variable in this condition is marked with a placeholder "_CONV_ENUM" and that the value is replaced with an integer. This is done, because the CACC tool does not directly support enumerative type variables. The assignment sets that are generated for this input formula are shown in Listing 5.3. After the assignment sets are generated, the replacements for the enumerative type variable is converted back, so that it can be used for the trap properties. The trap properties that were generated for the assignment sets are shown in Listing 5.4.

The combination of path coverage with CACC as described in Section 4.2 was done similar to edge coverage. The only difference is that the variable assignment sets generated by the external CACC tool were only applied to the last edge of the path. The implementation for edge coverage with CACC could be reused and no major changes were necessary.

```
>> Test Nr. 1: (makes the input formula false)
doClose = 1
doOpen = 1
lock_CONV_ENUM = 0

>> Test Nr. 2: (makes the input formula true)
doClose = 0
doOpen = 1
lock_CONV_ENUM = 0

>> Test Nr. 3:  (makes the input formula false)
doClose = 0
doOpen = 0
lock_CONV_ENUM = 2

>> Test Nr. 4: (makes the input formula false)
doClose = 0
doOpen = 1
lock_CONV_ENUM = 1
```

Listing 5.3: Generated test case assignments from the CACC tool.

```
G(((door = closed)&(doClose = TRUE & doOpen = TRUE &
   lock = unlocked))-> X door != open)
G(((door = closed)&(doClose = TRUE & doOpen = TRUE &
   lock = unlocked))-> X door = open)
G(((door = closed)&(doClose = FALSE & doOpen = TRUE &
   lock = unlocked))-> X door != open)
G(((door = closed)&(doClose = FALSE & doOpen = TRUE &
   lock = unlocked))-> X door = open)
G(((door = closed)&(doClose = FALSE & doOpen = FALSE &
   lock = unlocked))-> X door != open)
G(((door = closed)&(doClose = FALSE & doOpen = FALSE &
   lock = unlocked))-> X door = open)
G(((door = closed)&(doClose = FALSE & doOpen = TRUE &
   lock = locked))-> X door != open)
G(((door = closed)&(doClose = FALSE & doOpen = TRUE &
   lock = locked))-> X door = open)
```

Listing 5.4: Trap properties for the CACC assignments.

5.6 Configuration Options

For the execution of the tool at least three arguments are needed: The NuSMV input file with the model specification, a path for an output file, where the generated test cases should be stored, and a coverage criterion. The available options for the coverage criterion are: ”Node”, ”Edge”, ”Path”, ”EdgeWithCACC” and ”PathWithCACC”. Details about these criteria are described in Section 5.5. When path coverage is chosen, the user can also specify the path length or if all paths up to a specific length starting at the initial node should be generated. Additionally there is an option to specify a path, where a dot representation of the model should be stored. When the tool is started with invalid arguments the user is informed that the configuration was wrong and a description is shown, which states how the input arguments should look like. This output is shown in the next block.

```
Options:
-i, --input     path to the NuSMV file with the model
-o, --output    path to the output file, where the generated test cases
should get stored
-c, --coverage  test case generation method or coverage
options: [Node, Edge, Path, EdgeWithCACC, PathWithCACC]
-p, --path-conf path length or "upto" plus length  (eg. upto3)
-d, --dot-file  path to the file, where the dot representation
should get stored
```

5.7 Additional Features

During the development and test of the tool we discovered that some NuSMV notations and some models required adjustments for the tool. In this section these issues are described and it is shown how it was possible to handle them.

Define Symbol Parsing

In NuSMV the define part can be used to create symbols that work like macros. These symbols are defined without the need for a new variable, so that the state space is not enhanced, and the computation complexity does not increase substantially. During execution these symbols are simply substituted by their assignment [7], [6].

First, we thought that the define part could be ignored during the parsing process, because it is not necessary for the trap property generation and is only relevant when the model checker checks the trap properties on the model. However, we discovered that the type of these symbols was relevant in order to process the generated assignments of the used CACC tool. When the CACC tool produces an assignment for a variable or a symbol, the type must be known, because the used model checker does not allow assignments for integers and Boolean variables in the same way. For example a one must be replaced with true for a Boolean symbol in order to

create a valid assignment that the model checker accepts. The problem with defined symbols is that they have no predefined types, but the type can be determined by inspecting the assignments. Hence, it was necessary that the define part of the model was also considered during the parsing process. This could be done similar to the variable parsing. The difference was that the type was not known at the beginning and so the assigned values were analyzed in order to determine the type and also to obtain the possible options for enumeration type symbols.

Output and Performance Optimization

The size of some models also caused some problems. When there are many edges, then the computation of path and edge coverage takes much more time, because the model checker needs a lot of time to compute all counter examples. The initial implementation did not include a progress notification, so it was not possible to see if the tool still worked properly. Hence, an output message was added that tells the user, how many trap properties are already processed, and how many are still left.

Another issue was the memory usage. First, the generated counter examples from the model checker were simply stored in a list, which was placed in the memory. For small models this was not a problem because mostly there were less than hundred counter examples. However, larger models could produce some thousand trap properties even for simple coverage criteria like edge coverage with CACC. Therefore, the storage of all the counter example traces in memory was not reasonable anymore. Thus, a modification was made so that the generated counter example traces are directly stored in the output file, after they are generated, so that there is no need to buffer them all in memory and save them all at once.

Multiple Initial Nodes

Initially we only considered deterministic variables with one initial state but NuSMV supports models that can have deterministic variables that have multiple initial states, so that the initial value is selected non-deterministically.

These non-deterministically selected values can only be used as inputs by a test adapter and not as expected outputs because a comparison with the real output of the SUT is not possible [7].

The following example shows a variable initialization with multiple initial states.

```
init(CacheIdx) := {0,1,2,3};
```

So the parsing process needed modifications in order to support these cases and also the representation objects for the models needed adjustments. Thus, the variable definition was changed so that a list, which supports one or more initial values respectively nodes, was used. The trap property generation only needed adjustments for path coverage because the other criteria don't use the initial nodes, but paths start at initial nodes. Therefore, the generation was changed in order to generate path coverage trap properties for each of these initial nodes. However, this change increases the number of possible paths and so this coverage criterion is less practical for the given model or for other models with many initial nodes.

Multiple Target Nodes

NuSMV models can have multiple target states for a single transition case. So the next state can be any of the states of the set of target states, and the decision is made non-deterministically. A test adapter can use these non-deterministically selected states only as inputs for the SUT and not as expected outputs because a comparison with the real output of the SUT is not possible. An example of this feature and how it is used in a model is shown here:

```
next(CacheIdx) := case
...
state = idle & ... : {-1,2,3};
...
esac;
```

The changes for the parsing process and the object representation structure were similar to the modifications that were necessary for the support of

multiple initial nodes. So a single reference to a target was replaced with a list of targets and the modification to the parsing process was also similar. The trap property generation needed more changes. For edge coverage and for path coverage it was necessary to not only consider one possible target node for each edge, but to iterate over the list of targets and generate trap properties for each individually. So when an edge has multiple target nodes the trap property generation works like when there would be multiple edges, with the same source node, guard and priority, but with a different target.

Target Variables

Instead of multiple target nodes, NuSMV also supports variables as target state. In this case the next state is determined by the value of the target variable and any value that can be assigned to the variable is the possible next value respectively state. The following example shows this feature and how it is used in a model:

```
next(cacheo) := case
...
state=accessCacheBlock&CacheIdx=0: cache1;
...
esac;
```

The changes that were necessary in order to support this feature, where similar to the changes for the support for multiple target nodes, but with the difference that the trap property generation was more complex. So the Y-operator was used in order to check what the variable value of the previous state was and then it is assumed that this value does not match the value of the associated value. This assumption is made for all possible variable values, as shown in the following example:

```
... -> X !((cacheo = Mx & Y(cache1) = Mx) |
(cacheo = Dx & Y(cache1 )= Dx) ...
```

5.8 Known Limitations

NuSMV models can have rather complex features, which can be seen in the manual [6]. For most models these features are not needed. Therefore, we did not implement all of them. For example case distinctions with integers are not fully supported, because there could be rather complex case distinctions with mathematical operations and it would be difficult to handle them.

Another unsupported feature is case differentiations that contain multiple source states in one formula. For example when the formula for one case contains a disjunction of multiple source states, it is difficult to extract all of them and create edges for each associated node. Nevertheless, this feature is not important, because the same functionality can also be achieved when such a complex case formula is written in multiple parts.

Due to the fact that the CACC generation tool only supports Boolean operators and basic arithmetic comparisons, not all mathematical operations are supported. The tool was extended for simple cases like plus and minus of two variables, because these operations were important for an example. Most other complex mathematical operations are also not implemented from this tool, but they work for all other coverage criteria, where the tool is not used.

Another limitation is the computation effort. When the number of nodes or edges is to high, then the resulting number of trap properties can become unmanageable because the model checker would need too long to compute them all.

5.9 Define Restriction

The trap property generation process only considers certain model parts. So when there are restrictions from other parts, this can cause a higher number of invalid trap properties. For example the define part can cause restrictions for certain variable assignments or make some transitions impossible. A define restriction occurs, when there is a symbol in the define part that is

dependent on the values of one or more variables and therefore the symbol can only have certain values depending on the values of these variables. It also can occur, when there are multiple symbols with dependencies. For example in the case study we found a define restriction that prevented some symbols from having the same value because they were dependent on the same variable. These define restrictions are not a big problem for simple graph-based coverage criteria, because they do not use specific variable or symbol assignment for the edge conditions. However, they are an issue for the CACC assignment generation. When certain assignment combinations are not applicable for the model, but the input formulas for the CACC generation do not include these limitations, then many generated assignments are not usable. A solution for this problem is to identify the model components that have problems with the CACC assignment generation and use other coverage criteria without CACC for them.

6 Tool Evaluation

In this chapter simple examples are shown that were used to evaluate the developed tool. Different coverage criteria are compared, the functionality of the tool is tested and it is shown how the tool works. These simple example models are functional equal to the SUTs. An abstraction was not needed, because the behavior of the SUT could be modeled nearly completely. The only difference is that the SUTs include input and error handling.

For each example there is a general description and a visual representation. Furthermore, an illustration of a program run can be seen and the results are evaluated. It is shown how the generated counter example in the form of traces are parsed and then used by a test adapter to run them on the SUT. Therefore, real implementations were made in order to show, how the interaction with the test adapter and these implemented systems works. They also were used to compare the coverage criteria on the basis of instruction and branch coverage.

6.1 Triangle Example

In this section a simple triangle example is shown that was illustrated by Aichernig [1]. This example takes three numbers, which represent side lengths of a triangle, and calculates if it is an equilateral, an isosceles or a scalene. The example is particularly interesting, because it has edge conditions or guards that have multiple parts with dependent variables. So it is good to evaluate logic-based coverage because the edge conditions are not too simple.

6.1.1 Functional Description

The Example works as follows, first three numbers are given as input. Each of these numbers represents the length of a triangle side. Then, according to these lengths it is calculated, if a valid triangle can be formed. When it is possible, the output is the kind of triangle (equilateral, isosceles, scalene) that is formed with the given lengths.

In Listing 6.2 the NuSMV model for this triangle example is shown. The input lengths are modeled as nondeterministic integer variables (a, b, c) that only have a limited range from 0 to 100. This range was used because it was enough for all interesting tests for the SUT, otherwise if it was too big, the model checker needed too much time for the computation. When the limit was bigger only the execution time of the model checker increased, but the test cases remained the same. For the representation of the output, a deterministic enumeration type variable was used, which indicates the type of the triangle. When the lengths of two sides are smaller or equal to the length of the third side, then it is not possible to form a triangle. Then, the output variable has the value ”notriangle”, which indicates that no triangle could be formed.

6.1.2 SUT

The implementation of the SUT was quite straightforward. Basically the three side lengths of the triangle were given as input parameters and then there is a case distinction in order to determine which type of triangle is formed, or if it is not possible to form a valid triangle. The result of this distinction is printed as output. Listing 6.1 shows the Java implementation of the SUT. It can be seen that this implementation is straightforward and short. It only consists of input handling and of a case distinction for the output determination.

```
public class Main {

public static void main(String[] args) {
if (args.length < 3) {
System.err.println("Invalid Input args! Please provide
   3 Integers for the triangle side lengths.");
return;
}
int a = Integer.parseInt(args[0]);
int b = Integer.parseInt(args[1]);
int c = Integer.parseInt(args[2]);

if (a <= c - b) {
System.out.println("notriangle");
} else if (a <= b - c) {
System.out.println("notriangle");
} else if (b <= a - c) {
System.out.println("notriangle");
} else if (a == b && b == c) {
System.out.println("equilateral");
} else if (a == b) {
System.out.println("isosceles");
} else if (b == c) {
System.out.println("isosceles");
} else if (a == c) {
System.out.println("isosceles");
} else {
System.out.println("scalene");
}
}
}
```

Listing 6.1: SUT for the triangle example.

```
MODULE main
VAR
a : 0 .. 100;
b : 0 .. 100;
c : 0 .. 100;
state : {notriangle, equilateral, isosceles, scalene};
ASSIGN
init(state) := notriangle;
next(state) :=
  case
    a <= (c - b) : notriangle;
    a <= (b - c) : notriangle;
    b <= (a - c) : notriangle;
    a = b & b = c : equilateral;
    a = b  : isosceles;
    a = c  : isosceles;
    b = c  : isosceles;
    TRUE : scalene;
  esac;
```

Listing 6.2: NuSMV model for the triangle exmaple.

6.1.3 Graphical Representation

In Figure 6.1 the model is visualized. This representation was made with the DOT visualization feature, which was described earlier. It can be seen that the graph, which represents the case distinction, has many edges. The reason is that the model is not state-based and in each state all transitions can occur, because they have no source state respectively node. When a transition has no source node it is possible in all states and therefore in the graph representation an edge is added to each node for this transition.

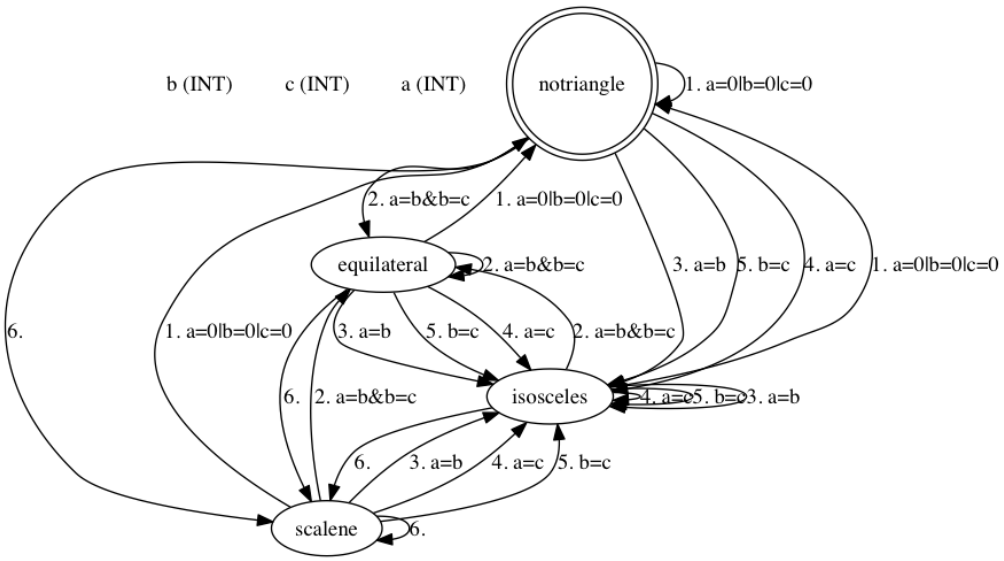

Figure 6.1: Graphical model representation of the triangle example, which was generated by the DOT visualization feature.

6.1.4 Test Adapter

As explained in Section 2.3, a test adapter is a tool for that takes abstract test cases as input and converts them into concrete test cases for the SUT. The test adapter for the SUT of the triangle example was implemented with two separated components. The first was used to parse the counter examples and crate an object representation. The second component made a connection with the SUT and used this object representation in order to execute concrete test cases on the SUT. The counter examples for the first component are generated by the model checker. They are in the form of traces, which show how the change of the variable assignments affects the state course.

Listing 6.3 shows a counter example, which was generated from the following trap property:

```
G ((state=notriangle & a <= c - b)->X state !=
    notriangle)
```

This trap property was made for edge coverage for one edge of the model in Figure 6.1. The important parts of this trace are the variable assignments and the state transitions. We have three input variables: a, b and c for the lengths of the triangle and one output variable called state, which represents the type of the triangle. The state transitions are marked with arrows and have an ascending number.

It can be seen that each state shows values for the specific variables of the model, but the values are only shown when they are not the same as in the previous state. The reason for this is that the model checker creates the traces in a compressed form to prevent overhead and when a value does not change it is not necessary to show it again until a new value is assigned.

In order to parse this trace a linked listed of state objects was made. Each state contains an assignment map for the variable values and a flag to mark if a state is the beginning of a loop. The state objects are part of a test case object, which contains the information about the trap property. This information can be used to find a deviation of the SUT from the model, when a test case fails.

For the connection of the test adapter with the SUT, the SUT was executed as external process, which was integrated in the runtime environment of the test adapter. The test adapter starts this process and specifies, which variables are input variables, and which are output variables for the SUT. The input variables are used as arguments when the SUT is executed and then the response is compared with the value of the output variable, which was parsed from the counterexample trace.

When the output value from the SUT is equal to the expected output value from the trace, then the test case was successful, otherwise there was a deviation of the SUT from the model. Another important aspect was that input variables of the current state affect only the output variables of the next state. Therefore, the input arguments for the SUT were taken from the current state and then the output from the next state of the trace was compared with the actual output of the SUT.

```
-- specification  G ((state = notriangle & a <= c - b)
   -> X state != notriangle)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  a = 7
  b = 7
  c = 3
  state = notriangle
-> State: 1.2 <-
  b = 3
  state = isosceles
-> State: 1.3 <-
  a = 3
  c = 7
  state = notriangle
-- Loop starts here
-> State: 1.4 <-
  a = 7
  b = 7
  c = 3
-> State: 1.5 <-
  b = 3
  state = isosceles
-> State: 1.6 <-
  b = 7
  state = notriangle
```

Listing 6.3: Counter example trace for the triangle example.

6.1.5 Results

The example was useful because it showed limitations for CACC coverage with edge conditions that contain integer calculations. For example the external CACC tool had problems with the minus operator. Therefore, this example was only used to evaluate graph based coverage criteria. However, the example showed good result for the invested development efforts. The model could be covered completely with the graph based coverage criteria because there were no invalid trap properties. The coverage on the SUT was measured with instruction and branch coverage. Instruction cover is a measuring method that shows the percentage of the covered instructions from the source code of the SUT by a test suite. Branch coverage is similar, but the difference is that it uses branches instead of instructions.

Table 7.2 shows an overview of the coverage criteria, their test number, code coverage and run time. It can be seen that node coverage produces only four test cases and has instruction coverage of 67.1%. Path coverage with all paths with length one produces eight test cases and shows a better instruction coverage of 86.3%. Edge coverage and path coverage with length two and three all reach the maximum instruction coverage of 89.0% and branch coverage of 94.4%. (100% cannot be reached because this SUT has an error test at the beginning, which checks if the number of arguments is sufficient for this system.) However, edge coverage only needs 32 test cases to reach the same coverage as path coverage with length two and three, which need 64 and 512 test cases.

Coverage criterion	Tests	Instr. coverage	Branch coverage	Run time
Node	4	67.1%	50.0%	1617ms
Path Length 1	8	86.3%	88.9%	4495ms
Path Length 2	64	89.0%	94.4%	62520ms
Path Length 3	512	89.0%	94.4%	1252853ms
Edge	32	89.0%	94.4%	12346ms

Table 6.1: Overview of the coverage criteria, their test number, code coverage and run time.

Figure 6.2 shows a comparison of the instruction coverage of test cases from edge coverage and random testing. For random testing 100 test cases

were created with random numbers between 0 and 100, which is the same range that was used in the model. These random numbers were used as values for the side lengths. It can be seen that random testing only reaches instruction coverage of 72.2%, which is significantly lower than the 89% that are reached with edge coverage. For example random testing could not cover some condition cases and therefore it could not cover all instructions, edge coverage could cover all instructions except the ones for the error handling.

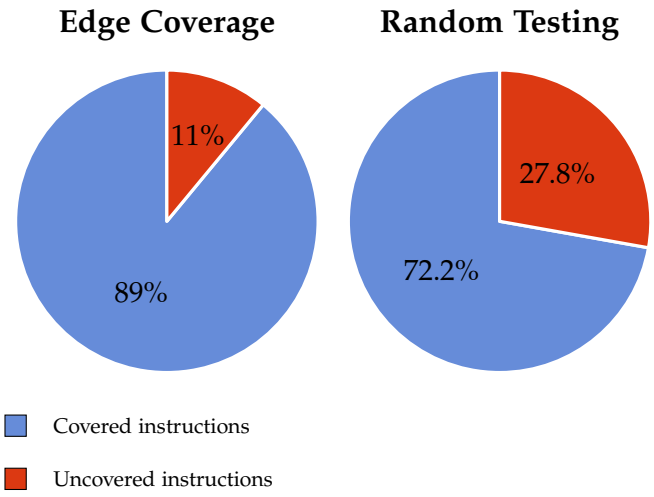

Figure 6.2: Instruction coverage comparison of edge coverage with random testing with the same number of test cases.

Figure 6.3 shows an overview of the time consumption of the tool execution steps for different coverage criteria. It can be seen that the time consumption for model parsing is constant for all coverage criteria and the time for the trap property generation is so small that it is insignificant compared to the other parts. Furthermore, the figure illustrates that the time for computations of the model checker increases, when the number of test cases becomes bigger. Only when the number of test cases is very small, model checking is not the major time consumption, otherwise it is so high that the other parts are insignificant.

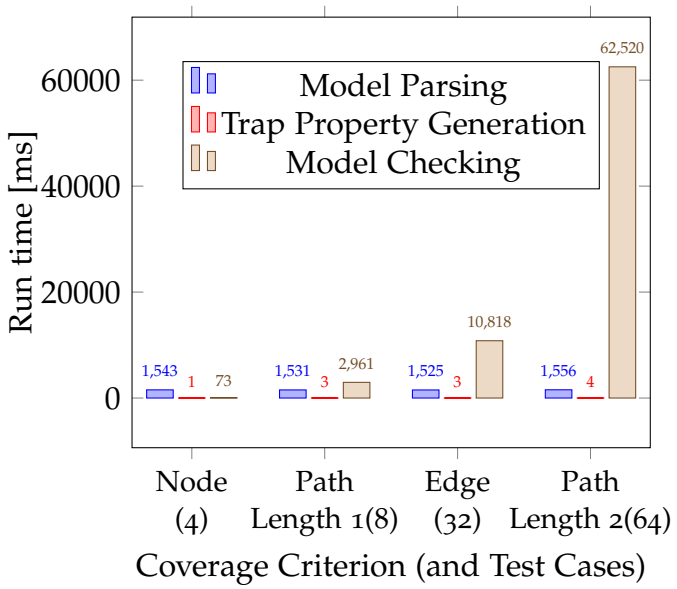

Figure 6.3: Time consumption of the tool execution steps for different coverage criteria.

6.2 Car Alarm Example

This car alarm system is based on the example state machine model which was presented by Aichernig [1]. The state machine model is shown in Figure 6.4. We made some modifications to the original model because not all complex features were necessary for a simple example. For instance the time constraints are not included because they are not relevant for the models in this thesis. However, this example is interesting because it has several deterministic variables and it has many transitions with conditions that are dependent on other deterministic variables and it should be tested if the tool has problems with such features.

6.2.1 Functional Description

The car alarm example represents the security system of a car, which should be able to detect a break-in, when a door is opened unjustified. In order to activate the protection of the car, all doors need be closed and the car

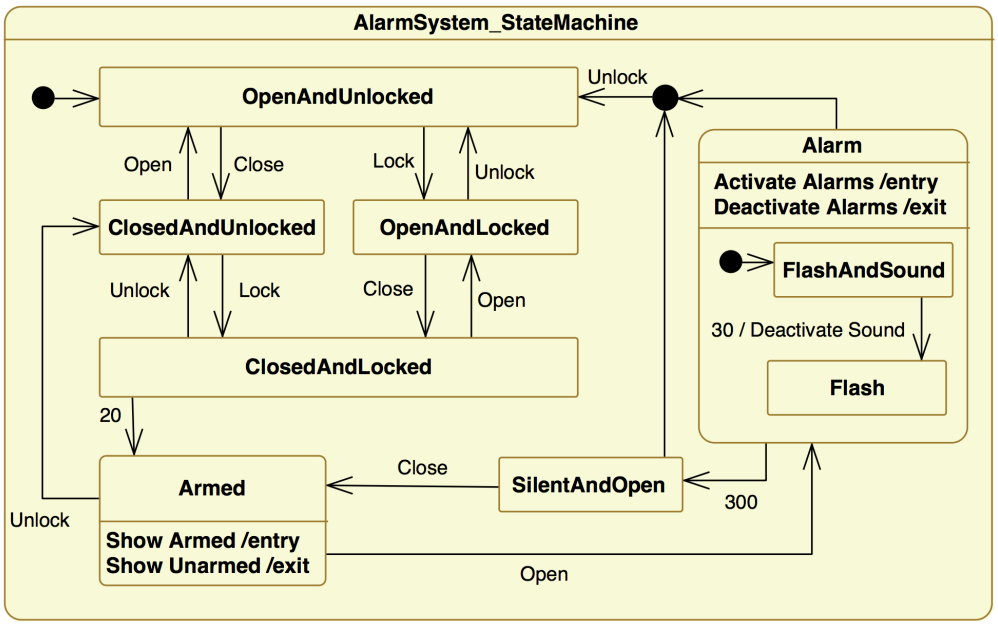

Figure 6.4: State machine model in UML of a car alarm system [1, p. 4].

must be locked. Then, the alarm system is armed. In this state, an alarm is triggered when someone opens a door. The system has different alarm modes. First, there is a sound alarm and afterwards there is a flash alarm. During these modes, the alarm can be deactivated by unlocking the car, which also makes the alarm system armed again. When the alarm is not deactivated, the alarm system becomes silent after the alarm modes run out. In this state, the car is not protected. This behavior is represented with three deterministic variables with case distinctions. One variable that indicates whether the door is open or closed, one that reflects whether the car is locked or unlocked and one is for the state of the alarm system, which is used for the silent, armed and the alarmed states. The actions of closing, opening, locking and unlocking are represented by simple Boolean variables [1].

Listing 6.4 shows the described model. It can be seen that this model has more components and more complex than the first example, because there are more deterministic variables and they are interconnected with each other.

6.2.2 SUT

The implementation of the SUT required several case distinctions and an input handling method. The case distinctions were used to reflect the state transition of the alarm system. Additionally, the SUT required variables to preserve the state. These variables were necessary to store the current state of the variables that represent the FSMs, so that the case distinctions only influence the output of the next state. This is important because the case distinctions are dependent on the variable values of the previous state. Therefore, the value must not change in the middle of the case distinctions.

```
MODULE main

VAR
doClose: boolean;
doOpen: boolean;
doLock: boolean;
doUnLock: boolean;

door: {open, closed};
lock: {unlocked, locked};
alarmSystem: {unarmed, armed, soundAlarming,
   falshAlarming, silent};

ASSIGN

init(door) := open;
next(door) :=
  case
    door = open & doClose & !doOpen: closed;
    door = closed & !doClose & doOpen & lock != locked:
        open;
    door = open: open;
    door = closed: closed;
  esac;

init(lock) := unlocked;
next(lock) :=
  case
    lock = unlocked & doLock & !doUnLock & door =
       closed : locked;
    lock = locked & doUnLock & !doLock : unlocked;
    lock = unlocked: unlocked;
    lock = locked: locked;
  esac;
```

```
init(alarmSystem) := unarmed;
next(alarmSystem) :=
  case
    alarmSystem = armed & doOpen & !doClose:
       soundAlarming;
    alarmSystem = soundAlarming & !doLock & doUnLock:
       silent;
    alarmSystem = soundAlarming & !doUnLock:
       falshAlarming;
    alarmSystem = falshAlarming & !doLock & doUnLock:
       silent;
    alarmSystem = falshAlarming & !doUnLock: armed;
    alarmSystem = silent & door = open & lock =
       unlocked: unarmed;
    alarmSystem = silent: silent;
    door = closed & lock = locked : armed;
    TRUE: unarmed;
  esac;
```

Listing 6.4: NuSMV Model for the car alarm system example.

6.2.3 Graphical Representation

Figure 6.5 displays the graphical representation of the NuSMV model for the car alarm system, which is shown in Listing 6.4. This model contains three graphs, which represent the case distinctions of the deterministic variables for the system state, the lock and the door, as described in the last section. Furthermore, it can be seen that the edge conditions include access to the other deterministic variables, which means that the graphs are not independent from each other. The variable for the door is dependent on the value of lock variable because it is included in the guard and vice versa. Moreover, the variable for the alarm system state is dependent on both the door and the look variable.

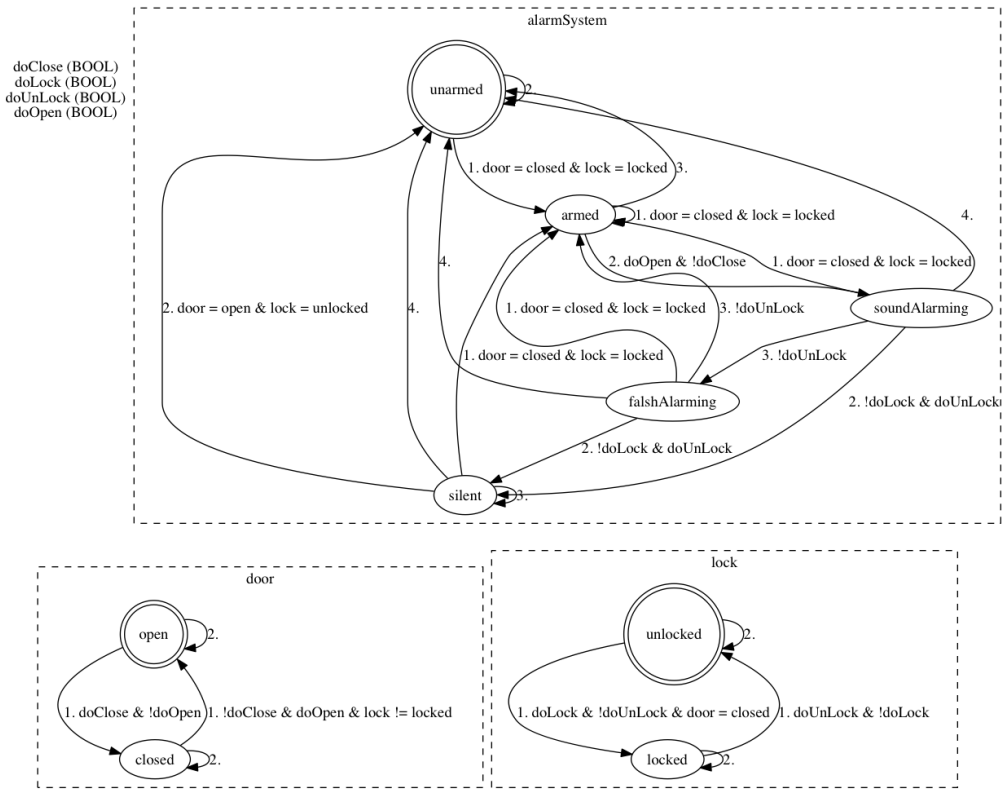

Figure 6.5: Graphical model representation of the car alarm system example, which was generated by the DOT visualization feature.

6.2.4 Test Adaptation

The test adapter was similar to that of the triangle example. The first component of the test adapter, which was used to parse the traces from the counter examples, could be reused without the need for any changes. It did not need any changes, because the structure of the traces is always the same for all generated counter example from NuSMV, only the variables are different, but the traces can be parsed independent from the included variables. The second component for the test execution could not be reused. In contrast to the SUT of the triangle example, the car alarm system is a reactive system, which continuously reacts to input and sends output. Therefore, it was necessary to change the interaction with the process of the SUT, so that a continuous communication for a test sequence was possible.

6.2.5 Results

The example was useful to evaluate if the combination of graph based coverage criteria with CACC shows an advantage. Furthermore, it provided the opportunity to evaluate the path coverage criterion with different path lengths because it has a model that includes a transition graph with sufficient size. Table 7.2 shows an overview of the coverage criteria, their test number, code coverage and run time. Node coverage could achieve instruction coverage of 97.9% with nine test cases. Edge coverage reaches 100.0% with 23 test cases and also path coverage with length four and five reach this percentage. However, for this example the differences of branch coverage are more interesting.

Figure 6.6 shows a comparison of the branch coverage of node coverage and edge coverage with and without CACC. The figure illustrates that node coverage only has branch coverage of 85.1% and edge coverage is substantially better with 95.9%. Furthermore, it shows that the combination of edge coverage with CACC also creates a significant improvement. The combined version reaches branch coverage of 100.0% with 104 test cases.

In Figure 6.7 a comparison of the branch coverage of path coverage criteria with and without CACC for different lengths is shown. It can be seen

Coverage criterion	Tests	Instr. coverage	Branch coverage	Run time
Node	9	97.9%	85.1%	1741ms
Edge	23	100.0%	95.9%	1871ms
Path Length 2	13	88.9%	73.0%	1675ms
Path Length 3	30	99.0%	95.9%	1774ms
Path Length 4	70	100.0%	98.6%	2309ms
Path Length 5	164	100.0%	100.0%	5063ms
Edge with CACC	104	100.0%	100.0%	3471ms
Path Length 2 w. CACC	13	93.1%	83.8%	1675ms
Path Length 3 w. CACC	130	99.0%	97.3%	5941ms
Path Length 4 w. CACC	303	100.0%	98.6%	24190ms
Path Length 5 w. CACC	716	100.0%	100.0%	181162ms

Table 6.2: Overview of the coverage criteria, their test number, code coverage and run time.

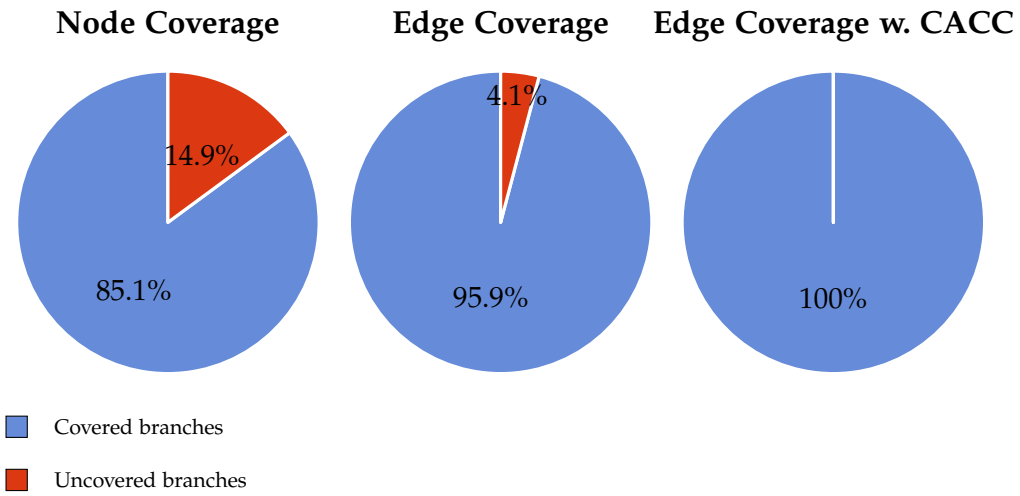

Figure 6.6: Branch coverage comparison of node coverage and edge coverage with and without CACC.

that the combination with CACC also makes an improvement for path coverage with length two and three. For path coverage with length two the combination with CACC made an improvement from 73.0% to 83.8% and for length three from 95.9% to 97.3%. For length four and five there was no change, but they already showed branch coverage of 98.6% and 100.0%.

There was not much scope for improvement.

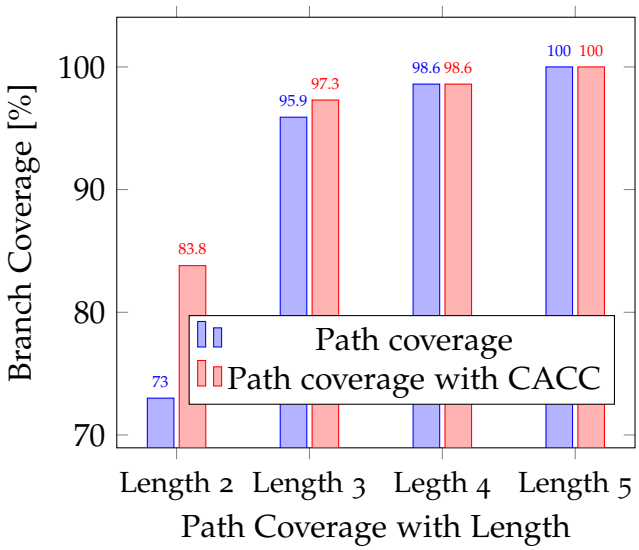

Figure 6.7: Branch coverage of path coverage with and without CACC for different path lengths.

The example was able to show that the combination of graph based coverage criteria with CACC can significantly improve the branch coverage. For the SUT edge coverage with CACC showed excellent results with the least test cases because it could reach full instruction and branch coverage with only 104 test cases.

```
-- specification  G ((alarmSystem = unarmed & (door =
   closed & lock = locked)) ->  X ((alarmSystem = armed
    & (doOpen = FALSE & doClose = FALSE)) ->  X
   alarmSystem = soundAlarming))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 24.1 <-
  doClose = FALSE
  doOpen = FALSE
  doLock = FALSE
  doUnLock = FALSE
  door = open
  lock = unlocked
  alarmSystem = unarmed
-> State: 24.2 <-
  doClose = TRUE
-> State: 24.3 <-
  doLock = TRUE
  door = closed
-> State: 24.4 <-
  doClose = FALSE
  doLock = FALSE
  lock = locked
-- Loop starts here
-> State: 24.5 <-
  alarmSystem = armed
-> State: 24.6 <-
```

Listing 6.5: Counter example trace for the car alarm system example.

7 Case Study

In this a case study we used our tool in order to generate test cases for a file cache implementation. The aim of this case study was to check if our tool is usable for real world examples. It was intended to check what problems occur, which coverage criteria are suitable for the given model, if there are certain limitations for the trap property generation or any other problems. Furthermore, we will show performance data for the different coverage criteria and how we measured the test quality. We will show how long the generation of the test cases takes and what coverage can be reached for certain model and implementation parts.

7.1 SUT

The file cache that was used for the case study was implemented for a Secure Block Device (SBD) by Daniel Hein and C was used as programming language. A SBD is a software component that uses symmetric cryptography to ensure data confidentiality and integrity for persistent data storage. In order to increase the access speed to the secure data, the SBD splits the file data into blocks and stores a configurable number of them in RAM. In RAM the blocks are unencrypted, but for the security goals of the SBD, a cryptographic nonce and an integrity tag are necessary. The SBD stores cryptographic nonces and integrity tags bundled for a number of data blocks in separate blocks called management block. These management blocks contain the security meta data for a specific range of data blocks. A data block can only be written or read, when the corresponding management block is also in the cache. Therefore, the cache implementation ensures that for each data block that is in the cache the corresponding management block must also be in the cache. When a data block is added to the cache or

accessed, then also the according management block must be in the cache or must be added. A management block can only be removed from the cache, when there are no dependent data blocks.

In order to make future requests faster, a cache stores certain blocks that are accessed more often longer than blocks that are not used often. There are many different approaches that can be used in order to decide which blocks should be stored and how they are organized. This cache implementation uses the least recently used (LRU) algorithm. For this approach the blocks are sorted according to their access frequency and the block that was used least is on the last position. When a block is accessed and the block already is in the cache, then its position in the cache is swapped with the position of the block in front of it. When there is an access to a block that is not in the cache, then one block must be removed so that this new block can be added. The LRU algorithm removes the block that was accessed least. This block is on the last position. Although many actions happen in the background in order to accomplish this behavior, the cache implementation details are beyond the scope of this thesis.

7.2 Model Description

For this case study a model of the file cache implementation was given. This model was developed by Franz Röck and Daniel Hein. The primary aim of the model is to reproduce the access behavior of the cache implementation, so it should reflect what happens when a certain cache position is accessed or when a block is searched that is not in the cache.

For the abstraction of the SUT we decided to limit the number of cache positions to four. The size of the cache can be configured on the SUT, but a limited number is enough to test the access behavior. Another abstraction is that only the type of a block is represented in the model, because modeling the actual data is not feasible. A block can be a data or a management block and both of them can be out of three ranges (x, y, z). A range represents a set of data blocks that have a common management block. For the SUT the size of the ranges can be configured, but we decided to limit it for the

model. In total, we have six different types: Dx, Dy, Dz for the data blocks and Mx, My, Mz for the corresponding management blocks.

The model of the cache implementation is divided in seven components respectively deterministic variables: One is used for the representation of the actions that can occur on the system, four are used to reflect the content of each cache position. Another one is used to select the cache position and the last one is used to decide what type of entry should be added, when there is a cache miss. These variables are explained in more detail in the following paragraphs.

The first variable is shown in Figure 7.1. (The guards are not shown in this figure because else the representations would be too big and therefore not viewable on a page.) This variable is the major component of the model, because it controls all actions. For example there are states that indicate that a new entry is added or that a cache block is accessed.

When the system is started only random data blocks are in the cache, which were selected with a non-deterministic init statement. The cache is started with this content because so it is not necessary to handle empty cache positions separately. For the initialization of the cache there is an InitMNGT state, which loads the management blocks for these data blocks. After the cache is initialized, the system goes in the idle state. In this state, it is possible to access the cache. This can be done with a state called accessCacheBlock. When there is an access to block that already is in the cache then the system returns to the idle state, otherwise a new block must be loaded. For this action there is the fechNextD state, in which it is decided what type of block should be loaded. This decision is made by a separate deterministic variable that is explained later. After this decision the system goes in a stated called processD, in which it is checked if the corresponding management block for the data block is in the cache and if the block on the last position can be removed or not. If there is a management block on this position, which has dependent data blocks, then this block cannot be removed and it is repositioned. This action is done in the bump state. When the block on the last position is free to drop, then the actual insertion happens in a state called insertD. In order to signal that a block was successfully added, there is the insertDsuccess state. After this state the system returns in the idle state. When the corresponding management block for the data block that

needs to be added is not in the cache then the management block needs to be added first. The insertion of a management block is similar to the insertion of a data block. It also happens in three states (processM, insertM, insertMsucsses). The only difference is that after the management block was successfully inserted the system does not go back in the idle state, but in the processD state so that the according data block can be added. The error state is only the default case. The system would go in this state, when no other transitions are possible, but normally this state cannot be reached.

The content of the cache positions and the required cache movements and replacements for the different cache states are modeled by four deterministic variables. Each of them represents the block type of one cache position. Together they control how the cache content is moved or added for the different states. When there is an access to a cache position, then its content is swapped with the content of the position in front of it.

The decision which cache position will be accessed is made by variable called CacheIdx when the system is in the idle state. This decision is only limited by the fact that only data blocks can be accessed, because management blocks are not usable by the user they only provide security information for the data blocks and are used in the background. In order to find out on which position there are data blocks, the variables for the cache content are checked. The index of one position with a data block is selected non-deterministically or minus one is selected. When this variable is set to minus one, it represents a cache miss. So then a block that is not in the cache is accessed and it must be loaded into the cache.

Another variable called newentry was used to decide, which type of entry will be added, when there is a cache miss. The choice is made when the system is in the fetchNextD state and then the value stays the same until the system is in this state again. Only data blocks can be chosen, because management blocks cannot be directly accessed and the possible values are: Dx, Dy and Dz.

Table 7.1 shows an example for the access behavior of the cache. When the cache is started there are random data blocks in the cache. In InitMNGT state the management blocks are loaded for these data blocks. After the initialization several accesses are made to the cache, some of them produce a cache miss and then new entries are loaded in the cache. The states between

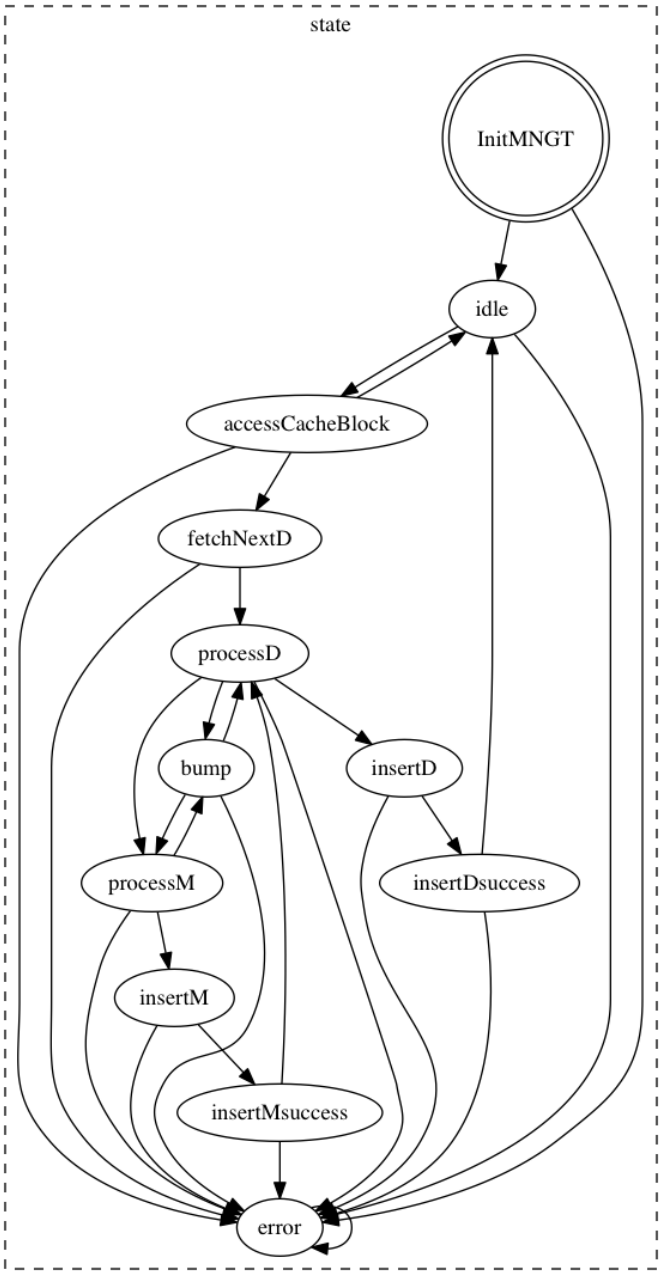

Figure 7.1: Cache model variable for the global system state, which shows what action is currently in progress.

the cache accesses are not shown in this table because this would produce too much overhead, but it can be seen how the cache content is changed when an access occurs or when new entries are loaded.

action	cache0	cache1	cache2	cache3
when started	Dx	Dx	Dx	Dx
InitMNGT	Dx	Dx	Dx	Mx
accessCacheBlock CacheIdx = -1, newentry = Dz	Dx	Mx	Dz	Mz
accessCacheBlock CacheIdx = 0	Mx	Dx	Dz	Mz
accessCacheBlock CacheIdx = 1	Mx	Dz	Dx	Mz
accessCacheBlock CacheIdx = -1, newentry = Dx	Dx	Mz	Mx	Dz
accessCacheBlock CacheIdx = 0	Mz	Dx	Mx	Dx
accessCacheBlock CacheIdx = -1, newentry = Dx	Dx	Mx	Dx	Dx
accessCacheBlock CacheIdx = 0	Mx	Dx	Dx	Dx

Table 7.1: Example for the access behavior of the cache. The actions with the required parameters and the resulting cache content are demonstrated in this table.

7.3 Test Adapter

The test adapter implementation was different from the previous test adapters from the example projects. Only the parsing process of the counter example traces was the same, but the test cases were not directly executing after they were parsed. Instead, we used the parsed traces to generate source code in the form of function calls to wrapper functions with test data as arguments. These wrapper functions were used to perform the test execution and to check if the expected cache content matches the actual cache content after the execution. The source code was injected in a test class of the SUT, which includes these wrapper functions. Listing 7.1 shows an example of a generated test case. The example was already used to demonstrate the access behavior of the model, which is shown in Table 7.1

and explained in Section 7.2. In the first line, a variable called tpErrorMsg is set with the trap property that was used to generate this test sequence. This variable is important to identify the cause, when there is a deviation of the SUT from the model. In the second line, the first wrapper function is used for the initialization of the cache. It takes the block types as input in order to set each cache position. After the cache content is initialized, a wrapper function for the cache access is called multiple times with different arguments. The arguments include the cache index, the block type that should be added when there is a cache miss and the expected contents after this access. This function uses these arguments in order to perform the access and check if the expected cache content matches the actual cache content after this operation. The input values for a test case are the initial cache content, the cache index and the type of a new entry and the result respectively output, which is compared, is also the content of the cache.

```
tpErrorMsg = "G ((newEntry = Dy & state = fetchNextD)
   -> X newEntry != Dz)  is false";
initCacheWrapper(Dx,Dx,Dx,Mx);
findAndCheckWrapper(Dz,-1,Dx,Mx,Dz,Mz);
findAndCheckWrapper(Dz,0,Mx,Dx,Dz,Mz);
findAndCheckWrapper(Dz,1,Mx,Dz,Dx,Mz);
findAndCheckWrapper(Dx,-1,Dx,Mz,Mx,Dx);
findAndCheckWrapper(Dx,0,Mz,Dx,Mx,Dx);
findAndCheckWrapper(Dx,-1,Dx,Mx,Dx,Dx);
findAndCheckWrapper(Dx,0,Mx,Dx,Dx,Dx);
printSuccess();
```

Listing 7.1: Generated test case in the form of function calls.

7.4 Results

Table 7.2 shows the results of the trap property generation of the different used coverage criteria. The number of valid trap properties in this table also equates to the number of test cases. First, the simple graph-based coverage criteria are shown and then the combined criteria with CACC. The simple graph-based coverage criteria produce a quite fast result with a small

number of invalid trap properties. Nevertheless, the combined coverage criteria with CACC have really a lot of invalid trap properties and only a limited number of valid once. The cause of this problem was analyzed. It was measured, which model components have transitions that were not covered. It turns out that especially the variables for the cache content have problems. These variables have basically the same transitions and therefore the same problems. The assignment generation with the CACC tool produced a high number of assignment combinations that are not applicable on the model, because they are prohibited from define restrictions as explained in Section 5.9. However, the other model components worked well with CACC coverage. For the problematic variables it might be better to use a simple graph-based coverage like edge coverage.

Coverage criterion	all TPs	valid TPs	invalid TPs	Run time
Node	46	45	1	3min 13s
Edge	530	357	173	18min 39s
Path Length 1	330	248	82	10min 37s
Path Length 2	15629	451	15178	27h 31min
Edge with CACC	7584	1328	6256	1h 49min
Path Length 1 with CACC	5474	1108	4366	57min 56s

Table 7.2: Overview of the coverage criteria, their trap properties (TPs) and run time.

Figure 7.2 shows a comparison of the runtime of the used coverage criteria. The major time consumption is caused by the model checker, the runtime for the other tasks is neglectable. This was explained in Section 6.1.5. It can be seen that node coverage only takes about three minutes and generates 45 test cases. Path coverage created 248 test cases in about 10 minutes. However, path coverage with length one is not interesting because it cannot reach most model states and with length two it already took over 27 hours. Path coverage with a usable length was impracticable for this model, because it includes many variables with a high number of edges and multiple initial and target nodes, which makes the number of theoretically possible paths and thereby the computation time impracticable. Edge coverage took about 19 minutes to generate 357 test cases and the combined version with CACC took about one hour and 11 minutes to generate 1328 test cases.

For the analysis of the test quality the percentage of line coverage and branch coverage were measured on the SUT. The gnu coverage tool gcov [4] was

used to perform this measure for the involved functions. Figure 7.3 shows a comparison of the coverage criteria and their line and branch coverage. Node coverage reached line coverage of 87.1% and branch coverage of 90.48% on the SUT with 45 test cases. Edge coverage could improve the line coverage to 89.52% and used 357 test cases. Path coverage with length two also showed line coverage of 89.52% and branch coverage of 90.48% with 451 test cases. The combination of edge coverage with CACC produced 1328 test cases and showed also line coverage of 89.52% and branch coverage of 90.48%. So node coverage was not as effective as edge coverage, but edge coverage and edge coverage with CACC showed no difference regarding line and branch coverage.

The model coverage was analyzed by visually highlighting the covered model parts, when a valid trap property could be produced. For example after the trap property generation was finished a model visualization with colored transitions was produced. In this visualization color was applied to highlight the transitions, for which it was possible to generate a test case. We compared the highlighted transitions and states for edge coverage with and without CACC. This comparison was especially helpful to find the cause of the high number of invalid trap properties with CACC, but it also showed that CACC worked well for the other model parts.

Overall, edge coverage reached the best coverage on the SUT with the least number of test cases. Edge coverage combined with CACC showed a better coverage for the edge conditions of most variables, only for certain variables, simple edge coverage produced trap properties with a better model coverage. Furthermore, an old version of the cache implementation with a bug was tested. The bug occurred when a data block was added and the corresponding management block was in the last position. When a management block is in the last position and there are no corresponding data blocks in the cache, then this block can be dropped. However, when the new block that needs to be added is dependent on this management block, then it is wrong to delete this block first because a data block always needs its corresponding management block in the cache. This bug, which was only found after weeks of manual testing, could be found immediately by the generated test cases. If the tool would have been used earlier, it could have saved a lot of debugging effort.

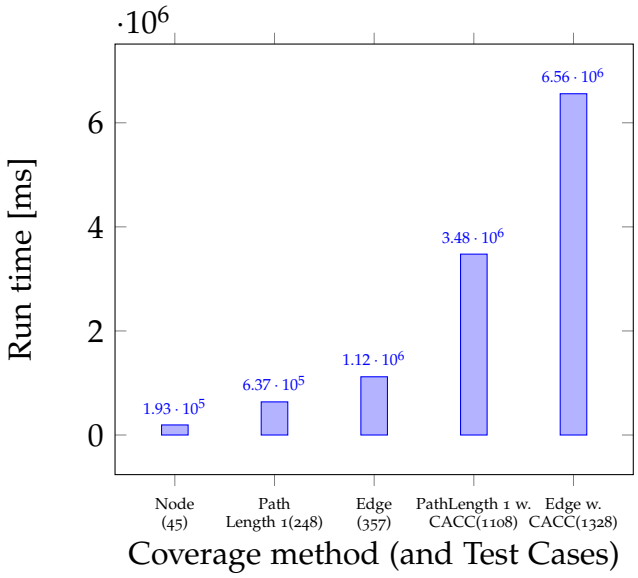

Figure 7.2: Run time comparison of the different coverage methods.

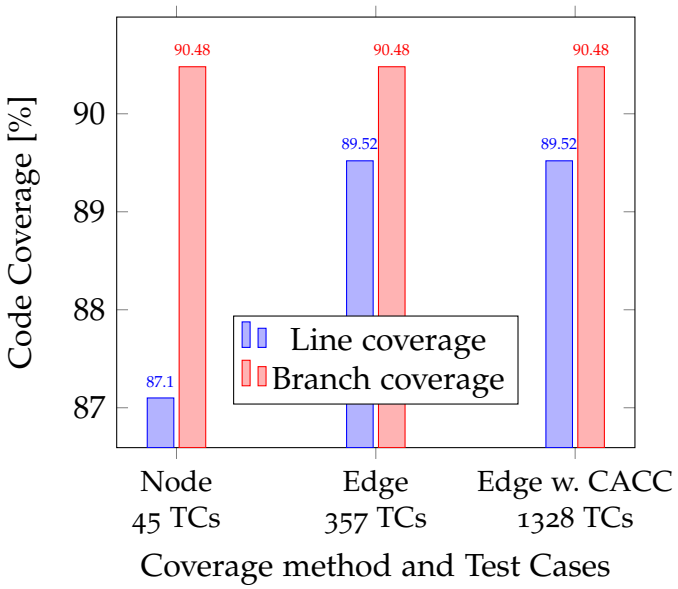

Figure 7.3: Comparison of the coverage criteria and their line and branch coverage.

8 Outlook

There are many features and possible extensions for the developed tool, which could provide further research topics. The tool could be extended to support models that include further features, for instance array support, more mathematical operations for the CACC tool, like multiplication and division, and full integer support would be options.

Moreover, an interesting extension would be the implementation of more coverage criteria. For example there are a few more logic-based criteria like RACC that could be combined with the implemented graph-based coverage criteria. Different combination methods of coverage criteria could be considered and evaluated and it could be checked if they show better results.

Another issue is the performance of the test case generation with the tool. Although the trap property generation works quite fast, the generation of the counter examples for the trap properties with the model checker needs a lot of time, when a model has many transition. For example path coverage with length two took about 27 hours for the model from the case study and also the other coverage criteria took some time to compute the trap properties, which can be seen in Table 7.2. There are optimizations that could be done in order to improve the performance. For example, the trap properties could be formulated in a different way, for example with CTL, or another model checker could be used and there are many other approaches as described by Fraser [11].

As already described in the case study due to the define restrictions and the limitation for the distinction of the CACC assignments, many counter examples are invalid and therefore not useful to generate test cases, but they also contribute to the computation time. There are certain actions that can reduce the number of invalid trap properties. For example, the external

CACC tool could add a mark to the test assignments that make the input formula true in order to distinguish them the assignments for a false case. Then, it would be possible to reduce the number of trap properties, because it would not be necessary to create them for both cases.

Another improvement would be parallelization of the counter example generation with the model checker. When multiple model checker processes are used for the application of the trap properties then depending on the given hardware it would also make the generation faster.

An extension for online testing would also be interesting. The difference between normal (offline) and online testing is that the test cases are immediately executed on the SUT when they are generated. It is not necessary to store all test cases and execute them after the generation is finished, but the execution is directly combined with the generation. The advantage of this approach is that the storage of test cases does not become a bottleneck and also it is possible to receive a direct feedback for the current test cases. When there is a problem we can find it faster, because we do not have to wait until the generation of all test cases is finished [21, p. 8off.].

Another possible research topic could be test suite minimization. The generated test cases from the tool are not free from overlap. For example, when a test case was generated for a certain graph components, then it might also include steps that could be used to test other components. This can make other test cases redundant. If it is possible to find these test cases, it can increase the performance of the tool and reduce the number of required test cases. Test case minimization is a similar approach that also can increase the performance. The difference is that not the number of test cases is reduced, but the length of the test cases. NuSMV creates counter example traces that contain some overhead because it tries to find loops for the traces. A possible way to minimize the test cases respectively traces is to cut the traces of, when the property that needs to be checked is reached in the trace.

9 Conclusion

In this thesis an automatic test case generation approach was presented that combines graph-based coverage criteria with logic-based coverage criteria and a tool was developed that implements this approach.

First, different coverage criteria were evaluated and implemented to test how the perform for number of example models.

In order to automatically generate test cases a tool was implemented that takes a NuSMV model as input, then computes trap properties according to a selected coverage criteria, and uses a model checker to compute abstract test cases for these trap properties. These abstract test cases were parsed by a test adapter and executed as concrete test cases on the SUT. Then, combination methods were developed by including an existing tool for CACC test data generation and applying this test data to graph-based coverage criteria.

These combination methods and the tool were analyzed with several small test projects. The triangle example could reach statement coverage of 89.0% and branch coverage of 94.4% with edge coverage. For the car alarm system it could be shown that the combination of edge coverage with CACC increased the branch coverage from 95.9% to 100% and also path coverage could be improved with this combination.

Then, the approach was evaluated with a case study, where it was applied to a real world project with a bigger model. It could be shown that edge coverage reaches line coverage of 89.52% and branch coverage of 90.48% for the SUT with the least number of test cases. Edge coverage with CACC showed the same coverage for the SUT and could increase the coverage for most model parts. However, for some model parts CACC could not increase the coverage, because variable limitations in the define part restricted some

generated variable assignments for CACC. Other models without such limitations could even show better results, but the generated test cases still showed a high code coverage and were able to find a bug in an older version of the implementation. Furthermore, the automated tool provides the advantage that many test cases can be generated in a relatively short time, without the need to be expert for coverage criteria and with little manual effort.

List of Figures

Listings

Bibliography

[1] Bernhard K Aichernig. "Model-Based Mutation Testing of Reactive Systems." In: *Theories of Programming and Formal Methods*. Springer, 2013, pp. 23–36 (cit. on pp. 49, 58–60).

[2] Paul Ammann, Jeff Offutt, and Wuzhi Xu. "Coverage criteria for state based specifications." In: *Formal methods and testing*. Springer, 2008, pp. 118–156 (cit. on pp. 3–5, 14, 16–24, 26).

[3] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. "Software testing based on formal specifications: a theory and a tool." In: *Software Engineering Journal* 6.6 (1991), pp. 387–405 (cit. on p. 25).

[4] Steve Best. "Analyzing code coverage with gcov." In: *Linux Magazine* (2003), pp. 43–50 (cit. on p. 75).

[5] Roderick Bloem, Karin Greimel, Robert Koenighofer, and Franz Roeck. "Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall." In: *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*. 2013, pp. 1–6 (cit. on pp. 6, 19, 21–23, 25, 28, 31, 40).

[6] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *Nusmv 2.4 user manual*. 1998 (cit. on pp. 43, 47).

[7] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, and Marco Roveri. "NuSMV 2.2 Tutorial." In: *ITC-irst-Via Sommarive* 18 (2004), p. 38055 (cit. on pp. 14, 43, 45).

[8] John J Chilenski. *An investigation of three forms of the modified condition decision coverage (MCDC) criterion*. Tech. rep. DTIC Document, 2001 (cit. on pp. 24, 26).

[9] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. "NuSMV: a new symbolic model checker." In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425 (cit. on p. 13).

[10] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. "Graphviz—open source graph drawing tools." In: *Graph Drawing*. Springer. 2002, pp. 483–484 (cit. on p. 33).

[11] Gordon Fraser. "Automated Software Testing with Model Checkers." PhD thesis. Institute for Softwaretechnology Graz University of Technology, Oct. 2007 (cit. on pp. 1, 25, 78).

[12] Gordon Fraser, Franz Wotawa, and Paul E Ammann. "Testing with model checkers: a survey." In: *Software Testing, Verification and Reliability* 19.3 (2009), pp. 215–261 (cit. on pp. 3, 4, 14, 16–18, 25).

[13] Emden R Gansner. *Drawing graphs with Graphviz*. Tech. rep. GraphViz WebSite, 2009. URL: `http://www.graphviz.org/pdf/libguide.pdf` (cit. on p. 33).

[14] Siamak Haschemi. *NuSMV-Tools (Tools for the model checker NuSMV)*. Apr. 2012. URL: `https://code.google.com/a/eclipselabs.org/p/nusmv-tools/` (cit. on p. 32).

[15] Mats PE Heimdahl, Sanjai Rayadurgam, and Willem Visser. "Specification centered testing." In: *Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification*. 2001 (cit. on pp. 2, 12).

[16] Anders Hessel and Paul Pettersson. "Cover-A test-case generation tool for timed systems." In: *Testing of Software and Communicating Systems* (2007), pp. 31–34 (cit. on p. 26).

[17] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. "Data flow testing as model checking." In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE. 2003, pp. 232–242 (cit. on p. 25).

[18] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004 (cit. on p. 11).

[19] Eleftherios Koutsofios, Stephen North, et al. *Drawing graphs with dot*. Tech. rep. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991 (cit. on p. 33).

[20] Jussi Lahtinen. *Simplification of NuSMV Model Checking Counter Examples*. 2008. URL: `http://www.tcs.hut.fi/Studies/T-79.5001/reports/laht08simcex.pdf` (cit. on pp. 2, 10, 11, 26).

[21] Kim G Larsen, Marius Mikucionis, and Brian Nielsen. *Online testing of real-time systems using uppaal*. Springer, 2005 (cit. on p. 79).

[22] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. "Generating test data from state-based specifications." In: *Software Testing, Verification and Reliability* 13.1 (2003), pp. 25–53 (cit. on pp. 3, 25, 26).

[23] Thomas J. Ostrand and Marc J. Balcer. "The category-partition method for specifying and generating fuctional tests." In: *Communications of the ACM* 31.6 (1988), pp. 676–686 (cit. on p. 25).

[24] Jan Köhnlein Peter Friese Sven Efftinge. *Build your own textual DSL with Tools from the Eclipse Modeling Project*. Apr. 2008. URL: `http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html` (cit. on p. 32).

[25] Amir Pnueli. "The temporal logic of programs." In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 46–57 (cit. on p. 10).

[26] Martin Pol, Tim Koomen, and Andreas Spillner. "Management und Optimierung des Testprozesses." In: *Heidelberg: dpunkt. verlag* (2000) (cit. on p. 1).

[27] M Prasanna, SN Sivanandam, R Venkatesan, and R Sundarrajan. "A survey on automatic test case generation." In: *Academic Open Internet Journal* 15.part 6 (2005) (cit. on p. 2).

[28] Special C. of RTCA. *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. 2011 (cit. on p. 3).

[29] Jan Tretmans. "Model based testing with labelled transition systems." In: *Formal methods and testing*. Springer, 2008, pp. 1–38 (cit. on p. 2).

[30] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches." In: *Software Testing, Verification and Reliability (STVR)* 22.5 (Aug. 2012), pp. 297–312. ISSN: 0960-0833. DOI: 10.1002/stvr.456 (cit. on pp. 1, 2).

[31] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. "Model-based testing of object-oriented reactive systems with Spec Explorer." In: *Formal methods and testing*. Springer, 2008, pp. 39–76 (cit. on p. 26).

[32] Stephan Weißleder. "Test models and coverage criteria for automatic model-based test generation with UML state machines." PhD thesis. Humboldt University of Berlin, 2010 (cit. on pp. 3, 25).

[33] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. "Automatically generating test data from a Boolean specification." In: *IEEE Transactions on Software Engineering* 20.5 (1994), pp. 353–363 (cit. on p. 25).