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

Test generation for partially modeled state-based software components

Scientific Students' Association Report

Author
Soma Seres

Advisor
dr. Vince Molnár

November 1, 2022

Contents

Kivonat

Beágyazott szoftverkomponensek fejlesztése során a komponensek (elvárt vagy tényleges) működését pontosan és szemléletesen leírhatjuk modellek használatával. Állapotalapú komponensek esetében a mérnökök legtöbbször állapotgépeket használnak erre a célra. A modell célja, hogy egy magasabb absztrakciós szinten mutassa meg, hogyan kellene a komponensnek működnie. Ha a modell kellően precíz és teljes, felhasználható a fejlesztési folyamat támogatására, például kód, vagy akár tesztkészlet generálásával. Ezek a technikák jelentősen gyorsabbá, olcsóbbá és hatékonyabbá tehetik a fejlesztést, azonban aprólékos, lehetőleg formális modellezést igényelnek a tervező mérnökök részéről.

A gyakorlatban azonban jóval egyszerűbb és átláthatóbb egyes modellelemeket rövid szöveggel reprezentálni, melyek pontos megvalósítását a végleges kódban implementálják majd a fejlesztők. Állapotgépeknél ez jellemzően az őrfeltételeket és akciókat érinti, míg az állapotokat és állapotátmeneteket többnyire pontosan leírja a modell. Ebben az esetben a kódgenerátorok jellemzően a fő logikát megvalósító kódot készítik el, ahol a szövegesen specifikált elemek a kódban egy függvénycsonkként jelennek meg, amit a fejlesztők a megfelelő módon implementálnak. Ez a fajta részleges modellezés jól kombinálható a kódgenerálással, azonban jelentősen megnehezíti a tesztgenerálást, mivel a modell önmagában nem értelmezhető, csak a kitöltött függvények kódjával együtt.

A munkámban egy olyan módszert dolgozok ki, amely megkísérel választ adni a fentebb vázolt problémára. A megoldás három fő lépésre osztható: 1) Utak generálása a tesztekhez az állapotgép modell alapján (adott fedettségi metrikára optimalizálva); 2) Kódrészletek generálása minden egyes úthoz, melyek tartalmazzák az összes kódot, ami az út végrehajtása során lefutna; 3) Hagyományos tesztgeneráló eszköz segítségével konkrét tesztesetek generálása a kódrészletekhez.

A modellből való útgenerálás és azokra építve a kódrészletek kialakítása lehetővé teszi, hogy a fő problémát kiküszöböljük: összekötjük a kódot, ami az őrfeltételeket írja le a hiányos modellel. A konkrét tesztesetek létrehozásához egy ún. "concolic" tesztgenerátor eszközt használok, ami információt gyűjt a lefutást befolyásoló változókról, majd egy megoldó segítségével konkrét értékeket ad nekik az egyes tesztesetekben.

A dolgozatban bemutatom az általános megoldást, és ezen felül egy AUTOSAR-specifikus prototípus implementációt is. Ezt felhasználva demonstrálom a módszer működését egy autóipari esettanulmányon keresztül. Az esettanulmányban egy AUTOSAR-alapú környezetben értelmezendő szoftverkomponens belső viselkedését leíró részleges állapotgép-modelljét vizsgálom. A fentebb leírt módszert alkalmazva bemutatom, hogy a megközelítés alkalmas tesztesetek generálására az AUTOSAR szoftverkomponenshez.

Abstract

During embedded software component development, we can use models to represent the (expected or actual) behavior precisely and informatively. For state-based components engineers mostly use state machines. The goal of state machine models is to show how the component should work on a higher level of abstraction. In case the model is sufficiently precise and complete, it can be used to support the development process (e.g., by generating code or test cases). These techniques make the development process significantly faster, cheaper, and more efficient, although they require detailed and preferably formal modeling from the designing engineers.

In practice, it is often much easier and cleaner to model some parts with simple text, which will be implemented by the developers in the final code. For state machines, this usually applies to guards and actions, while states and transitions are precisely described by the model. In this case, code generators create the implementation of the main logic, whereas the elements specified by text appear in the code as function stubs, which the developers will implement in the intended way. This kind of partial modeling can be combined with code generation, but it makes test generation significantly harder, because the model cannot be interpreted in itself, only with manually written code of the function stubs.

In my work, I propose an approach that attempts to solve the above-described issue. The solution can be divided into three main steps: 1) Generating paths for the tests based on the state machine model (optimized for given coverage metrics); 2) Generating code snippets for each path that contains the code that would run during the execution of that path; 3) Using a conventional test generation tool to create concrete test cases for the generated snippets.

Using the model to create paths and then creating code snippets based on them allows us to eliminate the main issue: we connect the manually written code with the incomplete model. To create concrete test cases, I use a "concolic" test generator tool, which gathers information about the variables that affect the execution of the code and use a solver to assign concrete values to them in each test case.

In this work, I present the generic solution that can be used in different areas, as well as an AUTOSAR-specific prototype implementation to demonstrate that the method works, illustrated in a case study from the automotive industry. In the case study, I investigate a software component in an AUTOSAR-based environment, where the internal behavior of the component is described by a partially modeled state machine. I apply the proposed method to prove that it is suitable for generating test cases for an AUTOSAR software component.

Chapter 1

Introduction

In software development, it is common to use models and diagrams to represent different aspects of the product. Surveys like Akdur et al. [1] or Forward and Lethbridge [11] show that engineers tend to use models when designing software, especially for documentation, to make communication and understanding better, and in some cases to write code based on the model or to generate code from them. In the latter 2 cases, we talk about Model-Driven Development (MDD). When products are based on models, the models must be precise and complete. To achieve this we can follow standards like UML [24] by using tools that were developed to help engineers keep the rules of such standards. However, there is typically no time to be 100% precise and complete while modeling, and it is much easier to represent elements with a short descriptive text.

In embedded systems the developers mainly use Component-Based Development (CBD) due to the nature of the context – it makes sense to split the system into smaller components so they fulfill the requirements by working together. State machines are widely used to model the behavior of software components [12]. In case we want to generate the code of the component we have to follow the standards very strictly which can be challenging at times because they are sometimes hard to understand and some parts are very complex. In situations like these, it might be tempting to simplify things and create a less precise model. However, this can lead to problems regarding the code generation from the model. Take a state machine's guard as an example: language-independent logical expression should be used to express the condition, but this can lead to overcomplicated expressions that can not be interpreted quickly by glancing at the model. However, when representing these conditions with a short and meaningful text, the model can be comprehended much easier.

When using non-interpreted expressions in the model the generated code has to be completed with handwritten code. This also means that it is impossible to create test cases based solely on the model, as the information about the values determining the behavior of the component is hidden behind the short text which only gets resolved once the corresponding code is written.

In this work, we concentrate on state machine models which are used to describe embedded software components' behavior. The formalism used for this handles very simple state machines: only states, transitions, guards, and actions are used. States and transitions are precisely modeled, guards and actions, however, are only represented in the model with simple text.

1. Definition. A *partially modeled state machine* has at least one model element which is represented with short descriptive text instead of precise expressions/elements. ▪

An example of such state machine is shown in Figure 1.1. Testing these components can be very challenging as the state machine gets more complex. It can be time-consuming to provide test cases to cover each state or transition (or both) while providing guards with different values, so every combination of a logical expression is tested. To address these challenges, test generation is a promising solution.

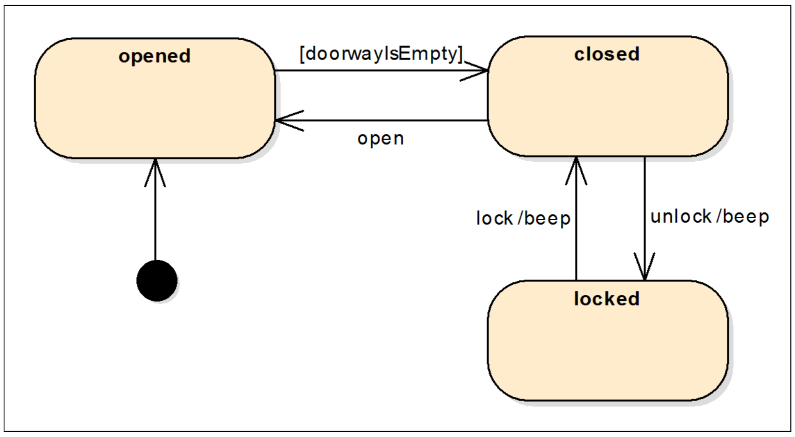


Figure 1.1: Example for a simple partially modeled state machine

Unfortunately, test generation requires precise and complete models. There are also test generation tools for source code, but these generally do not deal with state-based code very well, because they optimize for code coverage and not state coverage, which is more important for state-based behavior. In this work, a solution is proposed that combines these approaches to generate concrete test cases for partially modeled state machines.

In Chapter 2 the background of the work is described, gathering all related information needed to understand this report. After that in Chapter 3 the workflow of the proposed method to solve the described problem is presented by defining the inputs and outputs as well as the steps to take. To prove the method's usability in real life, an implementation using Eclipse-based technologies and Xtend was created, the details are described in Chapter 4. In Chapter 5 the implemented method is applied to perform a case study in the automotive industry. The subject of the case study is a state-based AUTOSAR software component that is provided as an example by thyssenkrupp Components Technology Hungary Ltd. (thyssenkrupp). In Chapter 6 I summarize my work and discuss future tasks and improvement possibilities.

Chapter 2

Background

2.1 Model Driven Engineering

Nowadays as software is becoming more and more complex, a good way to manage this complexity seems to be abstraction. Model Driven Engineering (MDE) is a popular way to achieve this by using models as a high-level description of behavior. This means the systematic use of models as primary artifacts during a software engineering process [13]. Much previous research like [1] and [16] show that adapting MDE can have positive and negative effects as well, depending on the situation. Model Driven Development (MDD) is considered to be a subset of MDE, as MDD drives only development - a sub-process of the whole engineering process.

Akdur et al. [1] show that the top three motivations of model-based development are: "improvement of the product quality", "development of functions with high complexity", and "shorter development times". To narrow down the areas, focusing on embedded systems, according to the responses in the survey, sequence diagrams and state machines/charts are the most popular diagram types. The survey also reports that documentation and code generation are the most popular reasons for using MDE.

An obvious positive effect of adapting MDE is shorter development times, especially if we are using a model-centric MDD adaptation [8], where we create code from the model (usually with generators). To be able to take this advantage, the model has to be precise and well-formed, which however can be difficult and time-consuming. This is heavily dependent on the experience of the engineers and the used tools.

Schätz et al. [23] show the two sides of MDE: although the design time (and therefore the cost as well) increased by 30-40%, code generation reduced the final cost by 40-50% and the verification/testing allowed another 40% reduction. This was possible because 60% of design errors were revealed in an early phase of the development process. The survey was done in the automotive industry with 180 participants from 14 different countries. The conclusion is that adapting MDE can be costly in the beginning, but as the engineers get more experienced and used to this process, it can be much more effective, resulting in a more efficient solution in the end.

2.2 State machines

State machines are widely used in the modeling reactive embedded systems [21]. There are different forms of state machines, which makes their semantics a complex question. The UML [24] or SysML [18] standards specify a very detailed form of state machines.

State machines have different model elements, the most commonly used are states, transitions, events, guards, and actions.

States are self-explanatory: they represent a well-definable state of the system (e.g. a door can be closed/open).

Transitions can appear between states, so they have a source and a target state. They represent that the state of the system can change from the to the target state of the transition.

Events trigger transitions. When a transition has a trigger event, it will take place when the event happens (e.g. the door will be closed when the 'close' event happens - this means we shut the door, which is an event from the door's perspective).

Guards are also used by transitions, they define conditions, which have to be true to enable a transition. A transition can only be executed if the current event processed is the trigger event of the transition and the guard on the transition is evaluated to be true.

Actions in the state machine are the way to execute commands and communicate with the environment of the system. They can also be referenced on a transition, in this case when the transition is performed, its effect (the referenced action) also gets executed. Actions can be referenced by states as well, in case we want to simplify our model. If every transition entering/leaving a certain state has the same action, it is easier to use an entry/exit action. An entry action is executed when we enter the corresponding state, and the exit action is executed once we exit the state. When we have an exit action in the source state of a transition that also has an effect (action), and the target state has an entry action, the execution of the actions follows a predefined order: first, the exit action of the source state is executed, then the effect of the transition and finally, the entry is performed.

2.3 Model-based testing

Model-based testing (MBT) is key in reducing time/cost in the development process. To test software exhaustively we need a very detailed specification and have to create complex and/or large amounts of test cases to cover all requirements. Beizer [6] shows that testing requires about 50% of development resources. In MDE, the models used can make understanding the system much easier, therefore tests can be more detailed, and they can be created much faster. In case we want to improve testing even more, MBT offers a promising technique for this: model-based test generation. If we follow standards and have the required tools, we can generate tests for different types of models, reducing the cost of testing by a lot.

When applying MBT, the input for the test cases is the model in itself, so it should be precisely designed and well-formed. This is not a simple task, but as mentioned in Section 2.1, the more experience the engineers have, the easier it is for them to create good models, resulting in higher efficiency.

2.3.1 Test terminology

When we talk about tests, we always have something we want to verify. In MBT, this is called the System Under Test (SUT), which we want to make sure behaves as expected. The requirements define what the SUT is responsible for, and the models describe how the SUT is constructed and how it behaves. Based on these we can create test cases, which are used to verify that the SUT fulfills the requirements. When applying MBT we usually do black box testing [7]. This means we think about the SUT as a black box – we know nothing about the inside, we just know what the outputs should be for the given inputs (e.g., based on the model). In this approach Test cases generally use the following structure:

- Set the inputs for the SUT
- Run/trigger the SUT
- Check the outputs

When we check the outputs, we usually use assertions. This means we assume that the output (the actual value, or the counter for a function call, etc.) is in an expected relation (e.g. equals, greater, not equals, etc.) with a previously defined value. For example, if we want to test an arithmetical system, especially its *addition* functionality, we set the two inputs for it for 3 and 2, set the operation to "add", run the system for these inputs, and check if the output is equal to 5.

2.3.2 Test generation for state-based models

As mentioned in Section 2.1 one of the main benefits of MDE is the shorter development time, which is often achieved by generators. In [16] the authors claim that the purpose of the model is often generating source code and test cases. The more formal and precise the model is, the less effort remains after the generation. When generating tests for a component we have the advantage of systematically processing the model: a generated set of tests can cover far more cases than a tester could, and this is done in a significantly smaller amount of time.

For state-based software, we can use the state machine to generate test cases. The diagram is very similar to a directed graph, therefore some techniques from that field are used. When generating test cases for state machines the first step is to find execution paths. This means the sequence of states that are accessible after each other. Path finding can be based on different coverage metrics such as state coverage (cover all states), transition coverage (cover all transitions), and transition-pair coverage (cover all combinations of adjacent transitions successively entering and leaving a given state) [22, 20, 19]. A great advantage of test generation is that it can be configured for optimizing a selected coverage metric or fulfilling multiple at once, just by adjusting the generator.

2.4 KLEE

KLEE [9] is a popular test generator tool for C. It is easy to use, open-source, and still maintained. It is based on symbolic execution. This means that the program is executed with symbolic values that are used to gather information during the execution of the program in order to create equivalence classes for these symbolic variables. These constraints are then used together to obtain one concrete value for each input using a solver. More details about symbolic execution can be found in [14].

2.5 AUTOSAR

The AUTOSAR (AUTomotive Open System ARchitecture) standard [5] is the result of a collaboration between various automotive parties. Their motivation was to create and establish an open and standardized software architecture for electronic control units (ECUs) used inside the automotive industry. Tools like AUTOSAR Architect, developed at thyssenkrupp help to adapt MDE using AUTOSAR models to provide standardized solutions when creating automotive software.

Software component-based development is commonly used in the AUTOSAR world, the final ECU software is the result of the collaboration of different SoftWare Components (SWCs). Each SWC has its own responsibilities and tasks. The SWCs can have ports, through which they can communicate with other components. To represent the behavior of the SWC we can create Runnable Entities (runnable), which are responsible for some of the SWC's tasks and responsibilities. These runnables can also communicate with each other inside the SWC in predefined, standardized ways: e.g. we can use Per-Instance Memory (PIM) or Inter Runnable Variables (IRV) for this purpose.

To identify elements in the resources unambiguously, the AUTOSAR standard's Generic Structure Template [3] Section 6.3.2 define ShortName-paths (both absolute and relative). In an AUTOSAR model, elements are identified by their shortName attribute. Some special elements (that are Identifiable, see details about this in [3]) create namespaces, inside which each element has to have a unique ShortName-path. If we are talking about an absolute ShortName-path, it can be calculated according to Section 6.3.2.1 in [3]: it begins with the '/' character, and we concatenate the shortName of each element from the containment path separated by '/' characters.

AUTOSAR uses a standardized environment to run the applications: this is called the Run Time Environment (RTE) [4]. This allows communication between components and different modules (e.g. COM, OS, etc.) using ports and interfaces, as well as scheduling of the different runnables (they are mapped to OS Tasks). When a SWC's runnable wants to access a port or an internal variable (e.g. IRV or PIM), the runnable has to explicitly declare this. The method for this depends on the type of accessed data. For accessing each data element an RTE API is available for each runnable that declares the usage of the data element. These can be found in the RTE Specification [4].

Chapter 3

Workflow of the proposed method

The goal of this work is to generate concrete test cases for partially modeled state machines. This section will cover the needed inputs and the created outputs, as well as the overview of the method.

3.1 Inputs

There are 2 main inputs for the test generation: the partial state machine model and the handwritten code describing the complex model elements. Selecting a conventional test generator tool for source code is also necessary, and the tool's output is considered as an intermediate input in the process.

3.1.1 Partial state machine model

In this method, only a specific form of state machines is used. They are partially modeled (as defined in Definition 1), and they only contain states, transitions, guards, and actions. Description of the states and transitions are modeled precisely, while guards and actions are represented with short and meaningful texts. To support this kind of modeling of state machines, a framework was created. The details and the implementation of the framework are discussed in Section 4.1.

3.1.2 Generated and handwritten code

We also need the source code generated from the state machine and the handwritten functions which implement the actions and guards in the selected programming language.

Actions have a function named the same as the shortName of the action, and as a guide, the usable RTE APIs are listed as a comment. As actions can be very diverse, we give the developers freedom by limiting them only to use the previously listed RTE APIs in the configuration. Guards are more strict, the elements listed in the configuration are automatically read using the corresponding RTE call in a generated method and passed to the stub as a parameter. The user only has to use the values and does not have to worry about using the RTE APIs, to get the data. An example of such generated stub for an action and a guard is shown in Source Code 3.1.

```c
/**
 * The callable RTE APIs from the sendSwVersionReport action are:
 * - Rte_Pim_ReportTobeSent
 * - Rte_Pim_SentDataStatus
 * - Rte_Call_TransmitSwVersionReport_send
 */
void sendSwVersionReport(){
    /* TODO */
}

/* CRCOK guard */
bool CRCOK(dtIncomingReport* IncomingReport, dtSwVersionReport* ReportTobeSent){
    return FALSE; /* TODO */
}
```

Source Code 3.1: Generated stubs for an action and a guard

3.1.3 Test generator tool

To perform the test generation we should select a test generator tool for the programming language of the previously mentioned codes. To achieve maximum efficiency it is advised to use a tool that can be run from the command line or code. In this case, the whole workflow can be automated. This tool is responsible for an intermediate input: the output of the generator is parsed and the final test cases are created from the parsed values.

3.2 Outputs

There are outputs in the process after each step, they are either usable later on their own, or are directly used in the next step of the process.

3.2.1 Instrumented code snippet

After extracting paths from the state machine model, an instrumented code snippet will be generated as the first output of the process. This code is set up to help the execution of the selected test generator. It contains all the code that would run during the execution of a path, providing information about the variables that have an impact on the guards in the path. This code is also used as an input for the test generator.

3.2.2 Output of the test generator

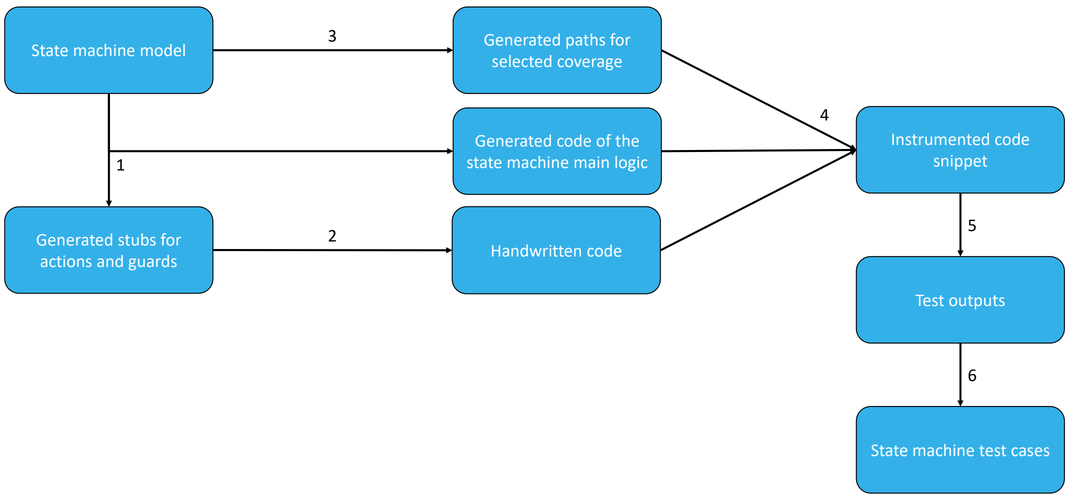
After using the test generator on the instrumented code snippets, it produces outputs. These will contain the values for the variables that were instrumented in the previous output. These outputs can be used separately to create tests or to analyze the model. The form and usability of them heavily depend on the test generator tool chosen.

3.2.3 Final test case

After parsing the outputs of the test generator the final test cases can be constructed. We know which variables get which values in each path, so we can set the inputs for the state machines, and then make assertions after each step about the state of the state machine. These final test cases can be run separately, or if we provide a function that resets the whole environment, they can be run after each other.

3.3 Modular test generation for partial state machines

To generate concrete test cases for partially modeled state machines, first, we need to create a connection between the partial model and the handwritten code parts and then generate paths and input values for the different test cases.



State machine model
3
Generated paths for selected coverage
4
Generated code of the state machine main logic
Instrumented code snippet
1
2
Generated stubs for actions and guards
Handwritten code
5
Test outputs
6
State machine test cases

Figure 3.1: Overview of the approach

Figure 3.1 provides an overview of the proposed approach. All steps are the results of this work. In the first step, we use the framework to generate the code that implements the main logic for the state machine and the stubs that will be responsible for the partially modeled elements (actions and guards). The second step is to write the code for these elements inside the stubs.

The third step is generating the paths from the state machine model. We should define what coverage we would like to optimize for in this step from the ones mentioned in Section 2.3.2 (e.g. state, transition, transition-pair). Next, we create the connection between the partial model and the handwritten code, when we create a snippet that contains all the code that would run during the execution of each path. These will also contain a call to the handwritten functions, and the snippet will be instrumented so that the chosen test generation tool can process the variables that determine the execution of each path.

Using these snippets in step five, we perform the test generation with the selected tool and parse the outputs in step six to create the final test cases.

In the end, the final test cases are just as if they were generated from a precisely modeled state machine, thanks to the connection made with the code snippets in step four.

Chapter 4

Implementation

4.1 State machine framework

To be able to build on the assumtions made in Section 3.1.1, I created a framework to support the partial state machine modeling. It is an Eclipse plugin, which means it can be integrated into any Eclipse-based tool. The framework is based on a simple metamodel (Figure 4.1) using the Eclipse Modeling Framework (EMF) [25].

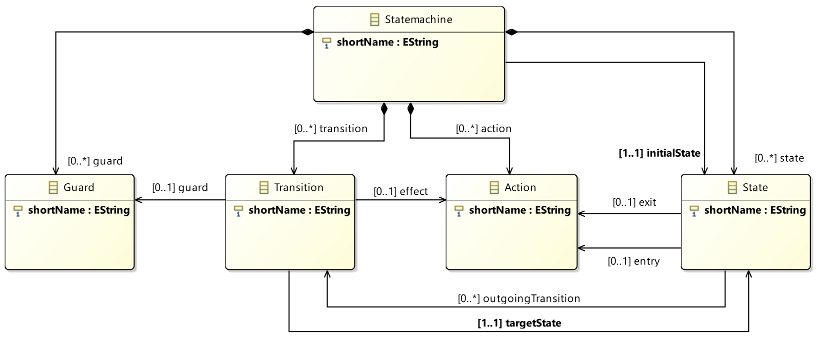


Figure 4.1: Partial state machine metamodel

This simple metamodel was enough to represent most state machines used for component internal behavior modeling at thyssenkrupp, based on research at the company. The result was that the used state machines are simple ones, no composite states and regions were used, therefore the previously listed basic elements are sufficient.

The framework is responsible for creating and validating state machine models as well as generating the code that implements them.

To integrate the framework into the AUTOSAR environment, a configuration layer was created as well. It is based on the AUTOSAR Standard's ECU Configuration [2]. A definition for the configuration was created which references state machine model elements by their shortNamePaths (to achieve this the AUTOSAR metamodel was used, and classes had to inherit from the Referrable and Identifiable classes).

This allows engineers to configure each state machine element that is not modeled precisely (actions and guards in this case), by declaring what data will be used in the function that implements the corresponding element. For example, if a guard checks a data element that is read from a port, we can add the access of this variable to the guard element's configuration, and the generated function stub for the guard will have the read data passed as a variable, so only the logical expression has to be implemented. This configuration is responsible for creating the connection between the state machine and the AUTOSAR model.

After the configuration is done, the framework generates the implementation of the main logic behind the state machine, and stubs for the partially modeled elements as well.

On top of the state machine framework, a separate plugin was implemented, that supports the workflow described in Chapter 3. The implementation was written in Xtend, which is an extended version of Java, with great template expression support that can be used when generating text (in my case code). In the following, I will describe this implementation in detail.

4.2 Overview of the architecture and code generation

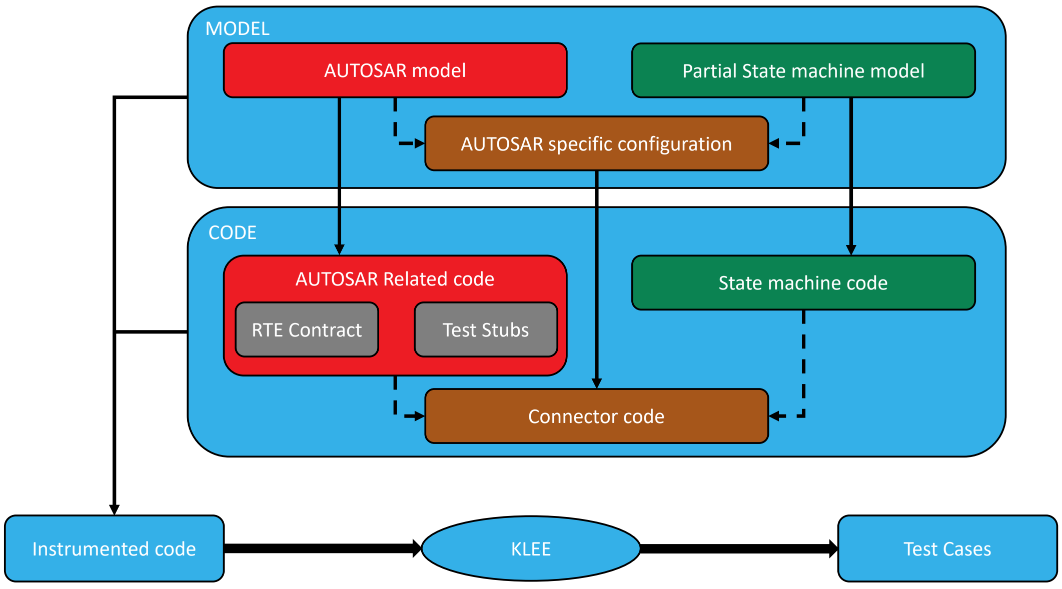


Figure 4.2: Architecture of the AUTOSAR Implementation

Figure 4.2 shows the architecture of the final implementation. 3 models are used:

- the partial state machine model to describe the behavior of a runnable
- the AUTOSAR model, which describes the Software Component (SWC) and everything else related to it
- finally, the AUTOSAR-specific configuration that connects the two (which is actually part of the AUTOSAR model, but it is visualized separately as it is a single model element that contains the connecting information between the two main models)

The dashed arrows mean that one thing is used by another. The arrow starts from the used element and ends in the user, so the configuration uses both the AUTOSAR and the partial state machine model by referencing their model elements: AUTOSAR elements by a cross-reference, state machine elements by their shortNamePath. From each model, a related code can be generated. Generation is shown with a solid thin arrow on the figure, while the workflow is shown with a thick one. The instrumented code is used by KLEE, and KLEE is used to create the final test cases.

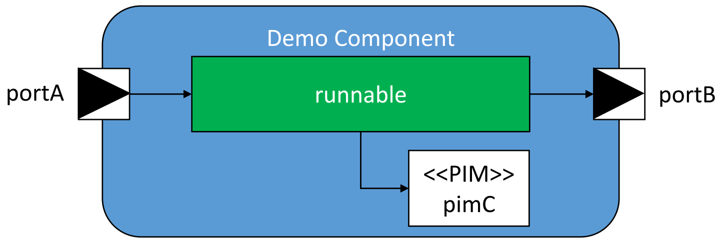


Figure 4.3: An example for an AUTOSAR Software Component

Figure 4.3 is an example of an AUTOSAR SWC. The generation of the different artifacts and the relation between them is easier to demonstrate with an example. This component is called 'Demo' and has 1 runnable called 'runnable'. The SWC has 2 ports, one provided and one required - on portA it can receive data, and on portB it can send data. The interface of the ports is not shown in the figure, portA has one data element, valA, and portB similarly has one data element, valB. The component also has a Per-Instance Memory (PIM), called pimC. The runnable can access all of the previously mentioned data elements, so the corresponding Run Time Environment (RTE) API can be used inside the runnable implementation.

From the AUTOSAR model, we can generate the RTE Contract for the SWC that most importantly contains code for the user-defined types, the declaration of functions in separate source files that will implement the different runnables and the RTE APIs. In Figure 4.4 we can see that the RTE Contract contains the demo_runnable.c file and the RTE API for all the data elements: we have a read API for portA's valA, a write API for portB's valB, and a PIM API for pimC.

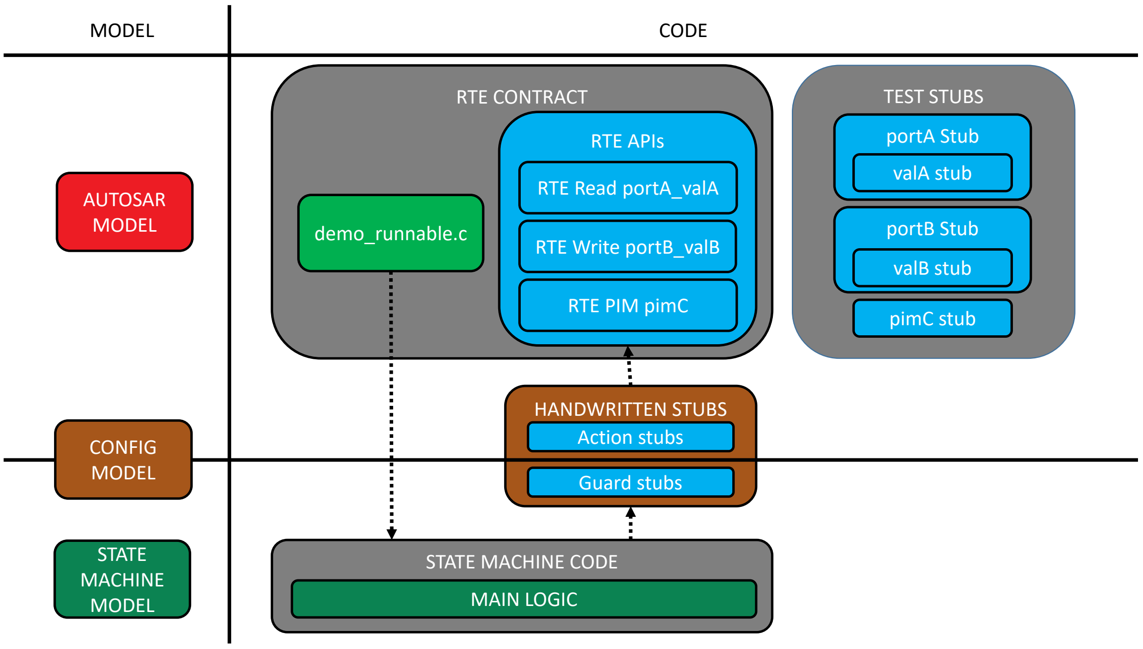


Figure 4.4: Models and codes used in the process

Test stubs can also be generated from the AUTOSAR model using thyssenkrupp's modeling tool, which allows us to set the different values on the RTE APIs used by the SWCs.

This test stub code makes testing much more convenient and will be used when creating the final test cases (which is a reasonable decision, since developers also use these when implementing test cases by hand). Figure 4.4 shows that each data element has its own stub: we can set portA's data elements through its stub, and the same applies to portB's (although it is a written port, this can be useful when testing multiple components together, but this is not the scope of this work), and PIMs have their own stub as well.

Assume the demo SWC's behavior can be modeled with a state machine and we used the framework to create one and already configured it as well, so we can access the different elements of the AUTOSAR SWC. From the partial state machine model, the code for the main logic can be generated and has to be extended with the connector code. It uses the AUTOSAR-specific configuration to create method stubs that make the developer's job easier by providing the required data for guards and by listing the callable RTE APIs for actions, so they know what APIs they're allowed and able to call. The main logic of the state machine will reference the handwritten stubs through headers (as we are in the AUTOSAR environment, the used language is C), and the handwritten stubs will use the RTE APIs. For example, if a guard is based on the value of pimC, the guard's stub will receive the value inside the PIM, and we can write a condition based on that value, or if we want to send data on portB inside an action, we will be able to call the RTE API of portB from the action's stub. The connector code uses the generated codes of the two different domains, so we include the relevant headers to connect the domains.

4.3 Path generation

Path generation for state machines is a widely researched area. I did not intend to create anything new regarding this step, but the solution needed to use some already existing algorithms.

The method is prepared so that it can generate paths depending on the preferred coverage metric. The solution uses the strategy design pattern - for each coverage metric there can be a different class that implements the path generation based on their metric goal. I have implemented 2 algorithms, but these can be extended according to the user's needs. I describe the two implemented algorithms below.

The first algorithm's goal is to cover all states. This was achieved by implementing an iterative version of the Depth First Search (DFS) algorithm. The algorithm is described in Algorithm 1. The main idea is that we use the state machine as a directed graph and we run a DFS from the initial state. We store the previously traversed states and whenever we reach a leaf (no more outgoing transitions are available to not visited states) we create a path, that contains the states and transitions that led us to the current leaf. Then we go back to the first state that has an outgoing transition with a target that is yet to be visited.

The other algorithm's goal is to cover all states and transitions. To achieve this I generate all possible paths between the initial state and every other state using Dijkstra's algorithm [10]. Since all possible paths are found between each state, all states will be covered using every transition at least once in a path. To avoid testing the same paths multiple times, I filter out paths that are prefixes of others. As the state machine is validated before the generation, it is guaranteed that all states are accessible in the graph, therefore this method will cover every state and transition.

These two algorithms are just examples, their purpose is to provide some basic path generation feature so the user is not forced to implement their own. There are several

Algorithm 1 All State Coverage algorithm based on Iterative Depth First Search

```
Require: $V$ is a state machine where each state is accessible
 1: Stack $S \leftarrow \{\}$, $Visited \leftarrow \{\}$, $States \leftarrow \{\}$
 2: $S.push(V.initial)$, $Visited.push(V.initial)$, $States.push(V.initial)$
 3: while $S$ not empty do
 4:     State $u \leftarrow S.pop()$
 5:     if $u \notin Visited$ then
 6:         $States.push(u)$, $Visited.push(u)$
 7:     end if
 8:     if $u.outgoing$ contains transition where target is not in $Visited$ then
 9:         for all $t \in u.outgoing$ where target is not in $Visited$ do $S.push(t.target)$
10:         end for
11:     else
12:         Create path from $States$
13:         for all $r \in States$ where r has no more unvisited neighbor do
14:             Remove $r$ from $States$
15:         end for
16:     end if
17: end while
```

other coverage criteria and algorithms, hence this part can be expanded to the users' liking.

4.4 Code snippet generation

From the previously generated paths, we want to create a code snippet that contains the code that would run during the execution of the path. This means we need to flatten the code of the state machine and provide information for the test generator about the variables that determine the execution of each step. To generate the snippet Algorithm 2 is used.

It is important to generate these snippets into well-differentiable parts (functions or even source files), so we can run the test generator separately for each snippet.

4.4.1 Instrumenting the code for KLEE

In Algorithm 2 Step 6 I mentioned instrumenting variables for the test generator tool. In the implementation, KLEE was chosen as the tool to be used thanks to the reasons listed in 2.4. When working with KLEE this means that after including the header of KLEE we can simply create a symbolic variable by calling the klee_make_symbolic function with the reference, the size, and the name of the variable. Source Code 4.1 is an example for this.

```c
uint8 myVar;
klee_make_symbolic(&myVar, sizeof(myVar), "myVar");
/* Code that uses myVar */
```

Source Code 4.1: Make a simple variable symbolic for KLEE

Algorithm 2 Code snippet generation algorithm

```
Require: $P$ is a path generated from $STM$ state machine
 1: Set $Vars \leftarrow \{\}$
 2: for all $t \in P.transitions$ where $t$ has a guard do
 3:     $Vars.push(r)$ where r is every referenced variable in $t.guard$
 4: end for
 5: for all $v \in Vars$ do                          ▷ Begin instrumented code here
 6:     instrument $v$ for test generator tool
 7: end for
 8: $State \leftarrow STM.initial$
 9: for $step \in P$ do
10:     if $step.transition$ has guard then
11:         $step.transition.guard$
12:     end if
13:     if $step.sourceState$ has exit action then
14:         $step.sourceState.exit$ call
15:     end if
16:     if $step.transition$ has action then
17:         $step.transition.action$ call
18:     end if
19:     if $step.targetState$ has entry action then
20:         $step.targetState.entry$ call
21:     end if
22:     $State \leftarrow step.targetState$
23: end for
```

In C it is common to use structures to group related variables into one place. When using structs with KLEE, it can lead to some bugs, so when instrumenting the code I made sure to break down the structs into simple variables, make those symbolic, and then set the already symbolic variables for the original struct. Structs can have other structs as their inner variables, so this had to be done recursively. Source Code 4.2 shows an example of making a nested struct symbolic for KLEE.

```c
/* Struct typedefs */
typedef struct{
    uint8 id;
    boolean processed;
} dtInfo;
typedef struct{
    dtInfo info;
    uint32 data;
} dtMessage;

/* Make struct symbolic */
dtMessage test0_message;

dtInfo test0_message_info;

uint8 test0_message_info_id;
klee_make_symbolic(&test0_message_info_id, sizeof(test0_message_info_id),
↪  "test0_message_info_id");

boolean test0_message_info_processed;
klee_make_symbolic(&test0_message_info_processed, sizeof(test0_message_info_processed),
↪  "test0_message_info_processed");
```

```c

test0_message_info = (dtInfo){.id=test0_message_info_id,
↪  .processed=test0_message_info_processed};

uint32 test0_message_data;
klee_make_symbolic(&test0_message_data, sizeof(test0_message_data),
↪  "test0_message_data");

test0_message = (dtMessage){.info=test0_message_info, .data=test0_message_data};
```

Source Code 4.2: Make a struct symbolic for KLEE

This detailed approach also makes it easier to create the final test code, because we have a separate object with a concrete value for each primitive variable.

4.5 Test generation

In order to get the final test cases we need to use the previously generated code snippets with the selected tool, in our case KLEE.

4.5.1 Running KLEE on the snippets

A disadvantage of KLEE is that it is very complicated to set up on a Windows computer, so to be able to run the tool, Windows Subsystem for Linux (WSL) had to be installed. This allows the user to use their computer as if it had a UNIX operating system. After this and installing KLEE inside the WSL, it is possible to run KLEE even from code.

KLEE uses LLVM bitcode as its input. In order to save time and resources clang is used to first compile the source files that are static through the test generation: the state machine code and the RTE APIs.

As mentioned previously the implementation is an Eclipse plugin, integrated into the modeling tool at thyssenkrupp. This means that the user can create projects, and they can create and edit AUTOSAR and partial state machine models as well as a configuration to connect the two. When the user generates the codes from the two models, they are placed in a predefined folder.

The previously mentioned predefined folders contain the source codes and the headers as well, but they also have to be extended with KLEE's headers, which are stored inside the plugin, so we don't have to copy them inside each project. The folders containing headers will be passed as an include path for the compiler and the ones containing source codes will be selected to compile all contained source codes.

After this, each test snippet is compiled alone and linked to the static parts, and KLEE is executed on the linked bitcode, which creates a folder containing the output of the test generation. Each test snippet is split into 2 source files: one contains the code for each step along the corresponding path and one contains the main function, so KLEE can be used for generation. The step functions only contain the flattened code of the state machine for the current step, these are used later to find the value combination that actually runs through the selected path. Inside the main function, first, we make every necessary variable symbolic for KLEE as described in Section 4.4.1. After that, we call each step function and check if it can be executed – if a step has a guard, it can get stuck on that guard, and we don't want KLEE to continue using those values, so we return from the main function. KLEE will handle these as a normal output, but we will filter out these value combinations later on. An example for these two files is shown in Source Code 4.3 and Source Code 4.4.

```c
/* Source containing steps: kleeInput2.c */
#include <kleeInput2.h>
SvcStateSTM testState2; /* Store the current state of the STM */
int step0(dtSwVersionReport* test2_ReportTobeSent, dtSVCIncomingReport*
↪  test2_IncomingReport){
        /* Effect for transition */
        sendSwVersionReport();
        testState2 = WAIT_RESPONSE;
        return 0;
}
int step1(dtSwVersionReport* test2_ReportTobeSent, dtSVCIncomingReport*
↪  test2_IncomingReport){
        bool res1 = incomingNotProcessed(test2_IncomingReport);
        if(TRUE == res1){
                testState2 = CHECK_CRC;
        } else {
                return 1;
        }
        return 0;
}
int step2(dtSwVersionReport* test2_ReportTobeSent, dtSVCIncomingReport*
↪  test2_IncomingReport){
        /* Entry action for FINISH */
        handleIncompatibleVersions();
        testState2 = FINISH;
        return 0;
}
```

Source Code 4.3: Source containing steps for instrumented code

```c
/* Source containing main: kleeInput2Main.c */
#include <kleeInput2.h>
int main(){
        dtSwVersionReport test2_ReportTobeSent;
        /* instrument struct test2_reportTobeSent for KLEE */
        dtSVCIncomingReport test2_IncomingReport;
        /* instrument struct test2_IncomingReport for KLEE */

        /* Perform steps in the path*/
        int s0 = step0(&test2_ReportTobeSent, &test2_IncomingReport);
        if(s0 != 0){return 1;}
        int s1 = step1(&test2_ReportTobeSent, &test2_IncomingReport);
        if(s1 != 0){return 2;}
        int s2 = step2(&test2_ReportTobeSent, &test2_IncomingReport);
        if(s2 != 0){return 3;}
        return 0;
}
```

Source Code 4.4: Source containing main function for instrumented KLEE code

4.5.2 Parsing KLEE outputs

KLEE produces ktest files, that contain information about all the symbolic variables. This includes its name, size, and concrete value. KLEE creates a ktest file for each value combination. In the previous section, it was mentioned that we need to filter out combinations that cause the path to get stuck midway. To do this, we need to gather information about the different value combinations first.

To parse the outputs of KLEE, the program uses their original parser tool called *ktest-tool*[15]. To use it we need to provide a ktest file produced by KLEE. Each path has its own corresponding output folder produced by KLEE. The implementation is built into AUTOSAR Architect, which is an Eclipse-based application, so I stored the script of the parser inside my plugin and used the Eclipse Platform to load the script. For each previously generated path, I entered the corresponding folder where KLEE placed its outputs, and ran the script for every ktest file. The scripts print out the information about all the variables in a human-readable way, this can be easily parsed from Xtend. An example of such output is shown in Source Code 4.5. The name of the variable was test2_ReportTobeSent_SwVersionCrc, it was 4 bytes and its value was 0.

```
object 0: name: 'test2_ReportTobeSent_SwVersionCrc'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
```

Source Code 4.5: Output of ktest-tool for a variable

4.5.2.1 Selecting the correct value combination

For the final tests, we only want to use the value combination where the path could be fully executed. For this a separate source file is created, that contains the main function for helping determine the incorrect combinations and functions that set all the related inputs to the corresponding value. The main function has an array of function pointers, that stores the step functions for the current path. It also receives a command line argument, that tells the program which value combination should be used to initialize the inputs. After these, each function pointer is called and we check if the return value is different from 0, which means the state machine got stuck in the current step, which implies this combination is wrong, so we return from the main function with the number of steps completed. An example of such file is shown in Source Code 4.6.

```c
#include <stdio.h>
#include <kleeInput2.h>
#include <SVC_TestStubs.h>
#include <valuesPath2.h>
int (*steps[3]) (dtSwVersionReport* test2_ReportTobeSent, dtSVCIncomingReport*
↪  test2_IncomingReport);
extern SvcStateSTM testState2;
void svcPath2Test0(void){
    TESTSTUB.Rte_Pim_SVC_IncomingReport.ReportProcessed = 0U;
    /* Setting the rest of the inputs for the corresponding value */
}
void svcPath2Test1(void){
    TESTSTUB.Rte_Pim_SVC_IncomingReport.ReportProcessed = 255U;
    /* Setting the rest of the inputs for the corresponding value */
}
int main(int argc, char **argv){
        if(argc != 2) return -1;
        char* c = argv[1];
        switch(*c){
                case '0': svcPath2Test0();break;
                case '1': svcPath2Test1();break;
                default: return -1;
        }
        steps[0] = step0;
        steps[1] = step1;
        steps[2] = step2;
        for(int i = 0; i < 3; i++){
                int resI = steps[i](Rte_Pim_ReportTobeSent(), Rte_Pim_IncomingReport());
                if(resI != 0){
                printf("Stuck in %s\n", svcStateToString(testState2));
                        return i+1;
                }
        }
        return 0;
}
```

Source Code 4.6: Source containing main function for instrumented KLEE code

In the example, the path had 3 steps, so the function pointer array has 3 elements: step0, step1 and step2 referenced from the correct header (kleeInput2.h). 2 combinations were generated, therefore 2 functions were created for each, these are the svcPath2TestX functions. Inside these we use the test stubs generated from the AUTOSAR model, setting the ReportProcessed element of the IncomingReport PIM for 0 and 255, depending on the combination used. In the main function, a for loop is used to call each step, and we check if the return value is not 0, and in that case, we return the number of the step that the state machine got stuck on. If all steps could be performed correctly with the values, we reach the end of the array and return 0, which means we found the correct path.

To determine the right combination we compile each above-described source file with the related codes and headers (AUTOSAR + partial state machine) with GCC, run the executable and check the return value of the program. If the return value is different from 0, the state machine got stuck, in case it is equal to 0, the combination is the right one and we can save this setup for generating the final test case for this path.

4.5.3 Generating final test code

After parsing the outputs and finding the correct combination of values we have a concrete value for each input variable used during the execution of a given path. In the final test code, we assign these values to the corresponding variables using the generated test stubs. When a structure is needed to be assigned, we can not set the whole structure data with one command, instead, we have to set the inner values one by one - the previously made efforts while generating the instrumented code comes in handy now, as we have access to all these sub-variables separately.

After setting the inputs for the path, we simply call the function that implements the state machine as many times as needed to complete the path, and after each call, we assert the current state of the state machine. An example of such a final test case is shown in Source Code 4.7

```
#include <svc.h>
#include <valuesPath2.h>
#include <TKP_ASSERT.h>
```

```c
extern SvcStateSTM svcCurrentState;
SvcStateSTM expectedStates[3] = {WAIT_RESPONSE, CHECK_CRC, FINISH}
void SVC_testCase2(){
    /* Initialize inputs with the correct values */
    svcPath2Test0();

    for(int i = 0; i < 3; i++){
        svc();
        TKP_ASSERT_FATAL(expectedStates[i] == svcCurrentState);
    }
    return 0;
}
```

Source Code 4.7: Example for a final test case for a path

For the assertions, the thyssenkrupp-provided macros are used, but any library for C could be used for this purpose.

Chapter 5

Case study in AUTOSAR environment

5.1 Software Version Checker component

The subject of the case study is an AUTOSAR SoftWare Component (SWC) called 'Software Version Checker'. The main requirement for this component is that it has to send the CRC for the software version stored on the Electronic Control Unit (ECU) this component is running on, receive a CRC from another ECU as well, and verify the same software version is installed on both ECUs.

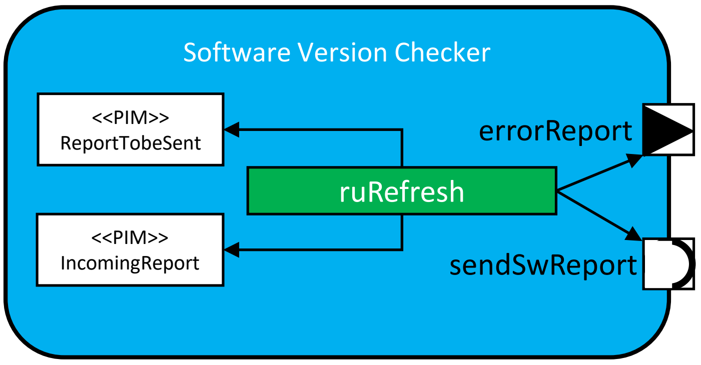


Figure 5.1: Software Version Checker SWC

The checking process is modeled with a state machine, where the guards and actions are represented with simple texts. The state machine can be seen on Figure 5.2 and the SWC (with only the parts relevant to the state machine shown) on Figure 5.1. The ruRefresh runnable is responsible for implementing the checking functionality, which will be based on the state machine.

The initial state is called SEND_CRC, from which we automatically enter the WAIT_RESPONSE state. During this transition, we call the sendSwVersionReport action, which is an existing C function, that uses the sendSwReport server port to call an operation that sends the content of the ReportTobeSent PIM.

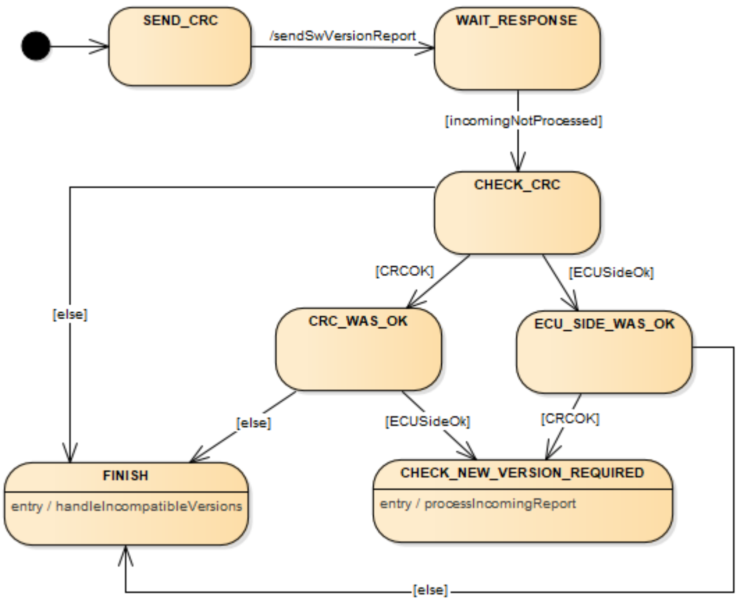


Figure 5.2: State machine representing the behavior of SVC

The incomingNotProcessed guard checks if the report stored in the IncomingReport PIM(which is handled by another runnable) is processed already. If it is not yet processed, we continue to check the software version. The check has 2 steps: checking the incoming software version's CRC and the ECUSideOk flag.

This can be done in arbitrary order: this is why we have states for each check. If any of the checks fail, we go to the FINISH state, where we handle the difference by reporting an error on the errorReport port. If both checks were successful, we go to the CHECK_NEW_VERSION_REQUIRED state, where we check if the software should be updated and do so in the processIncomingReport action.

5.2 Using the method to generate test cases

After creating the state machine, we have to connect the elements to the AUTOSAR model using the configuration layer. As mentioned before the incomingNotProcessed guard uses the IncomingReport PIM to check whether the IncomingReport has already been processed. To make this connection we reference the IncomingReport PIM in the configuration for the guard. We create configurations for the other guards as well applying the same logic: for the CRCOK guard, we reference the IncomingReport and the ReportTobeSent PIMs, for the ECUSideOk guard we only reference the ReportTobeSent.

After creating the connection with the AUTOSAR model we can generate the code that implements the state machine. The generated code will contain the main logic for the state machine and stubs for the actions and guards.

As the legacy code of the SWC does not use the state machine framework, we have to copy the implementation for each guard and action into the corresponding stubs. At this point, we are done with the implementation of the state machine.

Before we want to use the method, we have to generate the RTE contract and the test stubs for the component. After this step, we have everything ready to use the method.

As the implementation described in Chapter 4 is fully automated, we just have to run the command responsible for generating test cases. After the generation is done, we can find the final test cases for testing the state machine in the project's root folder. If we want to use the intermediate in/outputs for debugging, we can find them in the 'kleeInputs' folder. This contains the instrumented code that was used by KLEE, the outputs of KLEE, and the source files used for finding the correct value set, as well as the compiled version of them. Examples for each file are presented below.

The generated paths all begin with the following subpath: SEND_CRC→WAIT_RESPONSE→CHECK_CRC→... The final test cases cover the following paths (only listing the part after the common subpath):

1. ECU_SIDE_WAS_OK→CHECK_NEW_VERSION_REQUIRED
2. CRC_WAS_OK→CHECK_NEW_VERSION_REQUIRED
3. ECU_SIDE_WAS_OK→FINISH
4. CRC_WAS_OK→FINISH
5. FINISH

By looking at the paths and the state machine model, we can verify that these paths satisfy the all-state and all-transition coverage.

```c
#include <kleeInput4.h>
SvcStateSTM testState4;
int step0(dtSwVersionReport* test4_ReportTobeSent, dtSVCIncomingReport*
↪  test4_IncomingReport){
        /* Effect for transition */
        sendSwVersionReport();
        testState4 = WAIT_RESPONSE;
        return 0;
}
int step1(dtSwVersionReport* test4_ReportTobeSent, dtSVCIncomingReport*
↪  test4_IncomingReport){
        bool res1 = incomingNotProcessed(test4_IncomingReport);
        if(TRUE == res1){
                testState4 = CHECK_CRC;
        } else {
                return 1;
        }

        return 0;
}
int step2(dtSwVersionReport* test4_ReportTobeSent, dtSVCIncomingReport*
↪  test4_IncomingReport){
        /* Entry action for FINISH */
        handleIncompatibleVersions();
        testState4 = FINISH;
        return 0;
}
```

Source Code 5.1: Steps for path5 from the generated paths

```c
#include <kleeInput4.h>
int main(){
        dtSwVersionReport test4_ReportTobeSent;

        dtSwVersionCrc test4_ReportTobeSent_SwVersionCrc;
        klee_make_symbolic(&test4_ReportTobeSent_SwVersionCrc, sizeof(test4_ReportTobeSent_SwVersionCrc), "test4_ReportTobeSent_SwVersionCrc");

        boolean test4_ReportTobeSent_EcuSideOk;
        klee_make_symbolic(&test4_ReportTobeSent_EcuSideOk, sizeof(test4_ReportTobeSent_EcuSideOk), "test4_ReportTobeSent_EcuSideOk");

        boolean test4_ReportTobeSent_SwVersionRequest;
        klee_make_symbolic(&test4_ReportTobeSent_SwVersionRequest, sizeof(test4_ReportTobeSent_SwVersionRequest), "test4_ReportTobeSent_SwVersionRequest");
        test4_ReportTobeSent = (dtSwVersionReport){.SwVersionCrc=test4_ReportTobeSent_SwVersionCrc, .EcuSideOk=test4_ReportTobeSent_EcuSideOk, .SwVersionRequest=test4_ReportTobeSent_SwVersionRequest};
        dtSVCIncomingReport test4_IncomingReport;

        dtSwVersionReport test4_IncomingReport_ReceivedReport;

        dtSwVersionCrc test4_IncomingReport_ReceivedReport_SwVersionCrc;
        klee_make_symbolic(&test4_IncomingReport_ReceivedReport_SwVersionCrc, sizeof(test4_IncomingReport_ReceivedReport_SwVersionCrc), "test4_IncomingReport_ReceivedReport_SwVersionCrc");

        boolean test4_IncomingReport_ReceivedReport_EcuSideOk;
        klee_make_symbolic(&test4_IncomingReport_ReceivedReport_EcuSideOk, sizeof(test4_IncomingReport_ReceivedReport_EcuSideOk), "test4_IncomingReport_ReceivedReport_EcuSideOk");

        boolean test4_IncomingReport_ReceivedReport_SwVersionRequest;
        klee_make_symbolic(&test4_IncomingReport_ReceivedReport_SwVersionRequest, sizeof(test4_IncomingReport_ReceivedReport_SwVersionRequest), "test4_IncomingReport_ReceivedReport_SwVersionRequest");
        test4_IncomingReport_ReceivedReport = (dtSwVersionReport){.SwVersionCrc=test4_IncomingReport_ReceivedReport_SwVersionCrc, .EcuSideOk=test4_IncomingReport_ReceivedReport_EcuSideOk, .SwVersionRequest=test4_IncomingReport_ReceivedReport_SwVersionRequest};

        boolean test4_IncomingReport_ReportProcessed;
        klee_make_symbolic(&test4_IncomingReport_ReportProcessed, sizeof(test4_IncomingReport_ReportProcessed), "test4_IncomingReport_ReportProcessed");
        test4_IncomingReport = (dtSVCIncomingReport){.ReceivedReport=test4_IncomingReport_ReceivedReport, .ReportProcessed=test4_IncomingReport_ReportProcessed};
        int s0 = step0(&test4_ReportTobeSent, &test4_IncomingReport);
        if(s0 != 0){return 0;}
        int s1 = step1(&test4_ReportTobeSent, &test4_IncomingReport);
        if(s1 != 0){return 1;}
        int s2 = step2(&test4_ReportTobeSent, &test4_IncomingReport);
        if(s2 != 0){return 2;}
        return 0;
}
```

Source Code 5.2: Instrumented main function for path5

```c
#include <stdio.h>
#include <kleeInput4.h>
#include <SVC_TestStubs.h>
```

```c
#include <valuesPath4.h>
int (*steps[3]) (dtSwVersionReport* test4_ReportTobeSent, dtSVCIncomingReport*
↪  test4_IncomingReport);
extern SvcModStateSTM testState4;
void svcModPath4Test0(void){
        TESTSTUB.Rte_Pim_SVC_ReportTobeSent.SwVersionCrc = 0U;
        TESTSTUB.Rte_Pim_SVC_ReportTobeSent.EcuSideOk = 0U;
        TESTSTUB.Rte_Pim_SVC_ReportTobeSent.output.value.returnValue.SwVersionRequest =
↪  0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReceivedReport.SwVersionCrc = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReceivedReport.EcuSideOk = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReceivedReport.SwVersionRequest = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReportProcessed = 255U;
}
void svcModPath4Test1(void){
        TESTSTUB.Rte_Pim_SVC_ReportTobeSent.SwVersionCrc = 0U;
        TESTSTUB.Rte_Pim_SVC_ReportTobeSent.EcuSideOk = 0U;
        TESTSTUB.Rte_Pim_SVC_ReportTobeSent.SwVersionRequest = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReceivedReport.SwVersionCrc = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReceivedReport.EcuSideOk = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReceivedReport.SwVersionRequest = 0U;
        TESTSTUB.Rte_Pim_SVC_IncomingReport.ReportProcessed = 0U;
}
int main(int argc, char **argv){
        if(argc != 2) return -1;
        char* c = argv[1];
        switch(*c){
                case '0': svcModPath4Test0();break;
                case '1': svcModPath4Test1();break;
                default: return -1;
        }
        steps[0] = step0;
        steps[1] = step1;
        steps[2] = step2;
        for(int i = 0; i < 3; i++){
                int resI = steps[i](Rte_Pim_ReportTobeSent(), Rte_Pim_IncomingReport());
                if(resI != 0){
                printf("Stuck in %s\n", svcModStateToString(testState4));
                        return i+1;
                }
        }
        return 0;
}
```

Source Code 5.3: Code to find the correct value combination for path5

The final test case for path5 is shown in Source Code 5.4.

```c
#include <svc.h>
#include <valuesPath4.h>
#include <TKP_ASSERT.h>
extern SvcStateSTM svcCurrentState;
SvcStateSTM expectedStates[3] = {WAIT_RESPONSE, CHECK_CRC, FINISH}
void SVC_testCase2(){
    /* Initialize inputs with the correct values */
    svcPath4Test1();

    for(int i = 0; i < 3; i++){
        svc();
        TKP_ASSERT_FATAL(expectedStates[i] == svcCurrentState);
    }
    return 0;
}
```

Source Code 5.4: Example for a final test case for a path

Similar test cases were generated for each path, running these will test the behavior of the SWC. Thanks to the fulfilment of the previously mentioned metrics, 100% coverage is achieved, as required for safety-critical software.

Chapter 6

Summary

Model-Driven Engineering has many advantages, one of the most important is making the development process shorter and more efficient. However, the industry is always in a rush, engineers rarely have enough time to create fault-free and 100% precise models. Reducing the complexity can help them to address these challenges, but it can lead to less support regarding automated processes.

An example of this is modeling state machines using simple texts to represent some model elements. In my work, I proposed a method that can handle such partially modeled state machines and generate test cases for them based on the model and the additional handwritten code that is needed to interpret the textually modeled elements. To verify the method I created an implementation in an Eclipse plugin that was integrated into an automotive modeling tool developed at thyssenkrupp. This way I could test the proposed method in the AUTOSAR world, modeling the internal behavior of AUTOSAR Software Components (SWC).

The method uses a conventional test generator, after extracting paths from the state machine model and creating instrumented code snippets that contain the code that would run during the execution of that path. The test generator is used on these snippets to get concrete values for the variables that determine the execution of each path. The output is then parsed and the values are used in the final test case to set the variables' values. Then we assert the current state of the state machine after calling the function that implements its main logic.

An implementation of the method was created in an Eclipse-plugin, so I could integrate it into the Eclipse-based modeling tool, which is used at thyssenkrupp to help engineers model AUTOSAR SWCs. Using this implementation I did a case study on a state-based component, creating tests automatically using the proposed method.

Future work

While KLEE has a lot of advantages, it also has disadvantages. One of these is that floats are not handled well by the tool, so components that used floating point numbers had to be avoided for now. KLEE uses symbolic execution that relies on constraint solvers, which cannot handle floating-point numbers well. If it encounters a float, KLEE casts the symbolic variables to an integer and assigns 0 to it as a value. This leads to incomplete and often incorrect value combinations.

KLEE has a fork that can handle float as well, it is called KLEE-FLOAT [17]. This version requires a lot more effort to install and the time limitations did not allow experimenting with this fork, but this can be a promising continuation of this work.

Another point of potential extension comes from the fact that currently, the used state machines are really simple, containing only a small subset of the possible elements. This subset could be extended, for example, with composite states or regions. These make the code generation for the state machine itself much more complex. Handling the different kinds of events in AUTOSAR (e.g., change events on variables) can bring another partially modeled aspect into the environment. Depending on practical need, some of these extensions could broaden the application possibilities in more use cases.

Acknowledgements

I would like to thank thyssenkrupp Components Technology Hungary, especially Gergely Sisak and Bence Limbay, for providing countless industrial examples for my work and helping me professionally as well, regarding AUTOSAR and embedded software engineering.

I would also like to thank my advisor, Vince Molnar for helping me overcome the difficulties by proposing great inspirational ideas all the time I felt stuck.

Last but not least, I would like to thank my friends and family, especially my girlfriend for being patient with me during the time I spent writing this report.

Bibliography

[1] Deniz Akdur, Vahid Garousi, and Onur Demirörs. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, 91:62–82, 2018. ISSN 1383-7621. DOI: `https://doi.org/10.1016/j.sysarc.2018.09.007`. URL `https://www.sciencedirect.com/science/article/pii/S1383762118302455`.

[2] AUTOSAR. AUTOSAR Standard - Classic Platform - Specification of ECU Configuration, 2016. URL `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TPS_ECUConfiguration.pdf`. [Online; accessed: 28-October-2022].

[3] AUTOSAR. AUTOSAR Standard - Classic Platform - Generic Structure Template, 2016. URL `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TPS_GenericStructureTemplate.pdf`. [Online; accessed: 28-October-2022].

[4] AUTOSAR. AUTOSAR Standard - Classic Platform - Specification of RTE Software, 2017. URL `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_RTE.pdf`. [Online; accessed: 31-October-2022].

[5] AUTOSAR. AUTOSAR Standard - Classic Platform, 2022. URL `https://www.autosar.org/standards/classic-platform/`. [Online; accessed: 25-October-2022].

[6] Boris Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., USA, 1990. ISBN 0442206720. DOI: `10.5555/79060`.

[7] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., USA, 1995. ISBN 0471120944. DOI: `10.5555/202699`.

[8] Alan W. Brown. An Introduction to Model Driven Architecture - Part 1; MDA and Today's Systems. 2004.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association. URL `http://www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf`.

[10] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[11] Andrew Forward and Timothy C. Lethbridge. Problems and Opportunities for Model-Centric versus Code-Centric Software Development: A Survey of Software Professionals. In *Proceedings of the 2008 International Workshop on Models in Software*

Engineering, MiSE '08, page 27–32, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580258. DOI: `10.1145/1370731.1370738`. URL `https://doi.org/10.1145/1370731.1370738`.

[12] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. DOI: `https://doi.org/10.1016/0167-6423(87)90035-9`. URL `https://www.sciencedirect.com/science/article/pii/0167642387900359`.

[13] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 471–480, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. DOI: `10.1145/1985793.1985858`. URL `https://doi.org/10.1145/1985793.1985858`.

[14] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, jul 1976. ISSN 0001-0782. DOI: `10.1145/360248.360252`. URL `https://doi.org/10.1145/360248.360252`.

[15] KLEE. ktest-tool, 2021. URL `https://github.com/klee/klee/blob/master/tools/ktest-tool/ktest-tool`. [Online; accessed 26-October-2022].

[16] Liebel, Grischa and Marko, Nadja and Tichy, Matthias and Leitner, Andrea and Hansson, Jörgen. Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain. In Dingel, Juergen and Schulte, Wolfram and Ramos, Isidro and Abrahão, Silvia and Insfran, Emilio, editor, *Model-Driven Engineering Languages and Systems*, pages 166–182, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11653-2. DOI: `10.1007/978-3-319-11653-2_11`.

[17] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zahl, and Klaus Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 601–612, 2017. DOI: `10.1109/ASE.2017.8115670`.

[18] Object Management Group (OMG). OMG Systems Modeling Language, Version 1.6. OMG Document Number formal/19-11-01 (`https://www.omg.org/spec/SysML/1.6/`), 2019.

[19] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In Robert France and Bernhard Rumpe, editors, *«UML»'99 — The Unified Modeling Language*, pages 416–429, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-66712-4. DOI: `10.1007/3-540-46852-8_30`.

[20] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13:25–53, 03 2003. DOI: `10.1002/stvr.264`.

[21] Reggio, Gianna and Leotta, Maurizio and Ricca, Filippo. Who Knows/Uses What of the UML: A Personal Opinion Survey. In Dingel, Juergen and Schulte, Wolfram and Ramos, Isidro and Abrahão, Silvia and Insfran, Emilio, editor, *Model-Driven Engineering Languages and Systems*, pages 149–165, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11653-2. DOI: `10.1007/978-3-319-11653-2_10`.

[22] Yasir Dawood Salman, Nor Laily Hashim, Mawarny Md Rejab, Rohaida Romli, and Haslina Mohd. Coverage criteria for test case generation using UML state chart diagram. *AIP Conference Proceedings*, 1891(1):020125, 2017. DOI: 10.1063/1.5005458. URL https://aip.scitation.org/doi/abs/10.1063/1.5005458.

[23] Bernhard Schätz, Manfred Broy, Sascha Kirstan, and Helmut Krcmar. *What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?*, volume 1, pages 310 – 334. IGI Global, 01 2011. DOI: 10.4018/978-1-4666-4301-7.ch017.

[24] Bran Selic, Conrad Bock, Steve Cook, Pete Rivett, Tom Rutt, Ed Seidewitz, and Doug Tolbert. OMG Unified Modeling Language (Version 2.5), 03 2015. URL https://www.omg.org/spec/UML/2.5.1/PDF.

[25] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885. DOI: 10.5555/1197540.