

Automatic Testcase Generation at the Example of a Door Controller Unit

Dipl.-Ing. Michael Brost, FKFS, Stuttgart, Automotive Mechatronics and Software
Prof. Dr.-Ing. Hans-Christian Reuss, FKFS, Stuttgart, Automotive Mechatronics

Abstract

Within the scope of a cooperation project between the Research Institute of Automotive Engineering and Vehicle Engines Stuttgart (FKFS) and a car manufacturer ways are examined how on the basis of a formal specification of ECU (Electronic Controller Unit) functions test-cases for black-box testing can be automatically generated.

An automatic test case generation needs a formal specification. The basis of that formal description is a subset of the Unified Modeling Language (UML). The structure as well as the behaviour of an ECU is modelled that way. Since the UML is not domain specific a set of modelling rules has to be implemented.

Based on the above mentioned modelling technique a heuristic is applied to generate test cases. Starting point for the test case generation are use-cases based on state- and sequence-charts.

The formal modelling of ECU functions with UML is demonstrated using a door controller unit as an example. Afterwards the heuristic is presented to determine test-cases on the basis of the formal model. Finally the advantages and limitations of that process are subsumed.

1 Introduction

Testing automotive ECUs has become a major part of quality assurance during the development process until SOP. Today's ECUs implement a wide variety of functions ranging from simple lighting functions to the very complex exhaust after-treatment and in doing so have become rather elaborate in design and potent in application. The heterogeneous landscape of the automotive industry adds another dimension to the task of testing automotive ECUs.

Automatic test execution has become the state-of-the-art way of testing ECUs over the last years. Automatic test execution eliminates error-prone manual testing, allows regression testing, is cheaper in the long run and faster than manual testing. Different manufacturers offer a wide choice of soft- and hardware to aid the development engineer in his automatic testing activities.

A totally different question is the design of the test-cases themselves. Today there's no way to automatically generate test-cases for automotive ECUs. The main reason is that automatic generation of test-cases would require some sort of formal description of the object that needs testing. There's no such formal description of ECUs in the automotive domain or even for any sort of embedded system. The goal of this paper is to introduce a process that includes a formal description of automotive ECUs, a connected strategy to automatically obtain the resulting test-cases and a way to access those test-cases through different test systems.

Based on all the informal documents used and produced by developers today (Requirement Specifications, User Specifications, different Guidelines etc.) a method is discussed to model the functions of an ECU with the Unified Modeling Language. After exporting those models to the XML-based XMI-Format (XML Metadata Interchange) a set of rules – a heuristic – is developed and implemented to obtain all resulting test-cases. Those test-cases are then stored

in a XML-based test system unspecific format which relates to the newly adopted IEEE standard 1671 also known as ATML (Automatic Test Markup Language as described in [1]).

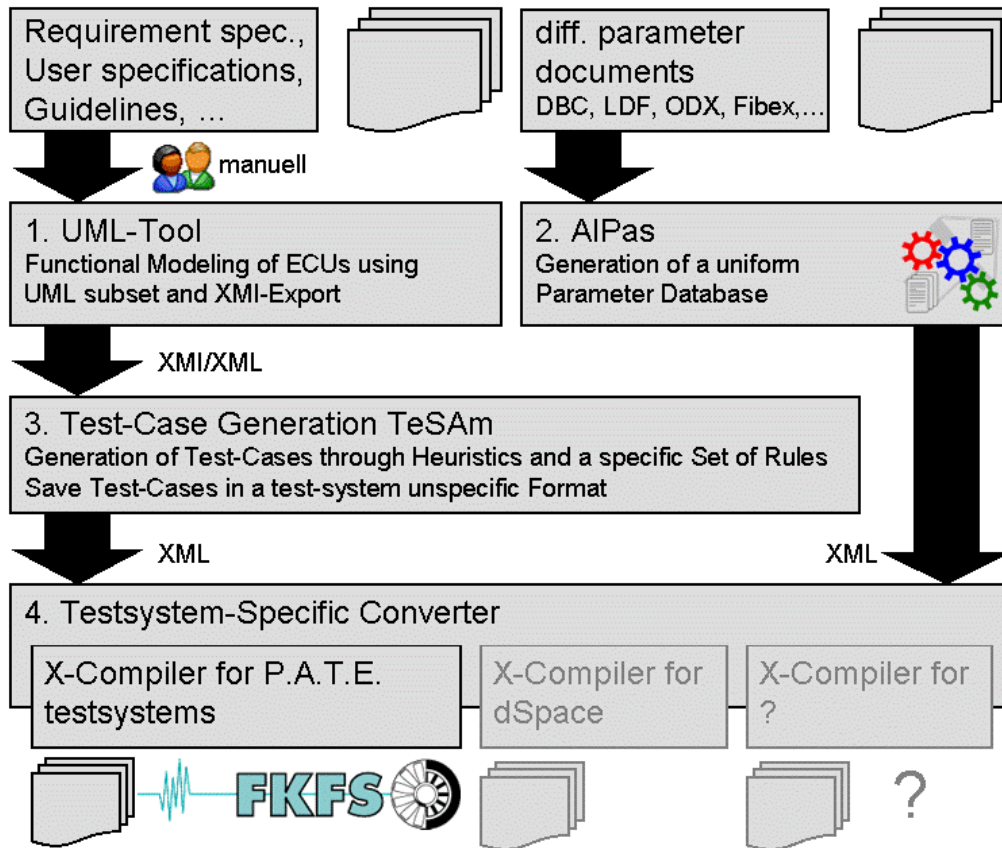


Figure 1-1: Basic outline of the automatic test-case generation process.

Figure 1-1 illustrates the basic outline of the automatic test-case generation process. The process is also discussed in [2].

2 A Door Controller Unit

There are some limitations to the above mentioned process. Due to the nature of the UML and its lack of a specific concept of time only quasi-static processes and event-triggered systems are considered. The testing of closed-loop control systems is not investigated.

Good examples for quasi-static and event-triggered systems are all body ECUs. As a first example a door controller unit is used as the system under test in the scope of this paper. Figure 2-1 displays the block diagram of a generic door controller unit for the driver door. The system includes different types and instances of sensors and actuators. For the door controller unit these are different electric drives for moving the window or the exterior mirror, lamps for lighting the immediate door periphery or the leg room, LEDs for status indication and buttons for user interaction with the device. It's connected to the other ECUs of the body domain via CAN bus.

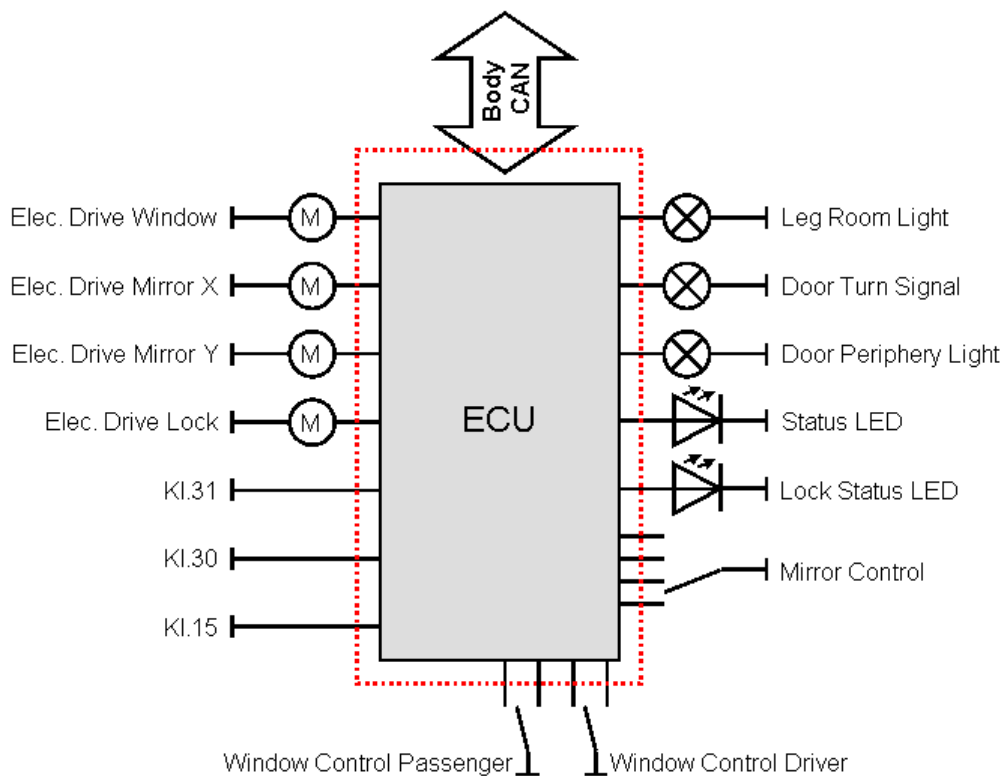


Figure 2-1: Block diagram of a generic door controller unit. The dotted line indicates the system boundary.

The door ECU fulfils different kind of functions which can be grouped as follows:

- Lighting functions: e.g.: turn on the door turn signal if activated by the driver, illuminate the space right in front of the door, if the door is opened, display current status through LED, etc.
- Moving window functions: e.g.: move window up and down if requested by driver, jam protection, close mirror if accident is imminent and detected by other system (collision mitigation), close window if driver leaves car and activates central locking system, etc.

- Moving mirror functions: e.g.: turn mirror around x-axis or y-axis on driver's request, move mirror to a stored position (memory function), fold mirrors, etc.
- Lock functions: e.g.: lock and open door if requested by driver, lock and open door as part of central locking system, etc...
- Diagnostic functions: e.g.: send ECU identification on request by diagnostic interface, read and delete DTCs, detect errors like wire malfunctions, short-circuits or CAN failure, etc...
- Communication functions: e.g.: transmit current status on the CAN bus, power down and wake up on specific CAN signals, etc.

The system boundary, indicated through the dotted line in Figure 2-1, separates the system under test from its periphery like actuators and sensors. The periphery is either simulated by the test system or real components are connected to the SUT.

3 Formal Description using UML

In the world of object-oriented software design the Unified Modeling Language UML offers a way to specify, implement and document software. It has been standardized by the Object Management Group (OMG). As mentioned before, the UML is not domain specific and can be extended and adapted to other domains. It's adapted to automotive ECUs in this project, as it has been discussed in [3] and [4].

3.1 Using UML subset

The UML in its current release 2.1 offers 13 different types of diagrams. Some of those diagrams include similar information and can be interchanged without significant loss of data like state and activity diagrams. Other diagrams are hard to apply to the automotive domain respectively to embedded systems. If one also keeps in mind that the proposed process is to be used by engineers without or only sparse knowledge of object-oriented software engineering, it is advisable to reduce the number of used types of diagrams to model ECUs. Other approaches in modelling with UML (compare e.g. [5]) include a reduction in diagram types as well.

The used diagrams in the scope of the automatic test-case generation process are:

- class diagrams
- use cases
- state diagrams
- sequence diagrams

Class diagrams are the architectural backbone of UML modelling (see [5]). Class diagrams describe both the structure and the capabilities of the modelled entities. (see section 3.2)

Use cases introduce an abstract level of modelling ECU functions. They serve as an intuitive starting point for the modelling process and are capable to accommodate cross-relations between different use cases such as inclusions and extensions (see figure 3-2). Each use case serves as origin of a specific behaviour which is then modelled in more detail by state and sequence diagrams.

State diagrams are based on advancements in deterministic finite automata to model object behaviour (see [6]). They are well defined and an intuitive mean to model the behaviour of embedded systems. They offer sophisticated algorithms in test-case generation as well.

Sequence diagrams are well suited to model activities that consist of the succession of operations. Diagnostic operations like reading the ECU identification are good representatives of such activities.

3.2 Structural Modelling

Class diagrams are used to model the structure and capability of the system or parts of it. The system consisting of the ECU, sensors, actuators and CAN bus is divided into self-contained entities that are represented by different classes. Object-oriented concepts like inheritance are used in modelling those entities.

Figure 3-1 illustrates the class diagram of an abstract class *illuminant* and the derived classes *blinkerlight*, *LED* and *lamp*. The abstract class *illuminant* offers the functions *On()* to turn a light on, *Off()* to turn a light off and *GetStatus()* to query its current status. The attribute *mState* indicates the current state of the lamp, true meaning the lamp is turned on and false meaning the lamp is turned off. The classes *blinkerlight*, *LED* and *lamp* inherit those attributes and functions. The class *blinkerlight* extends the functionality by using the two attributes *mOffTime* and *mOnTime* which indicate the time in ms the lamp is activated and deactivated while being turned on.

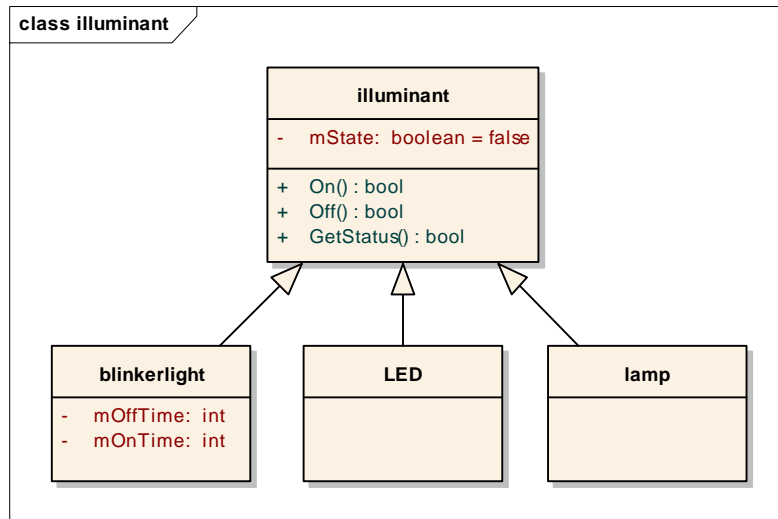


Figure 3-1: Class diagram *illuminant* with the derived classes *LED* and *lamp*.

3.2 Behavioural Modelling

Use cases represent self-contained ECU functions. They are used to model functions on an abstract level and to show connections between the different use cases. Such connections represent e.g. the fact that one function includes another function or is extended by another function. Actors that make use of the modelled functions can be other systems, the driver or the connected periphery.

Figure 3-2 displays a section of the use case diagram of the door controller unit. The three actors are members of the ECU periphery as shown in Figure 2-1. The use case *Move Window* can be utilised both by window control buttons and via body CAN bus. An example of moving the window through data on the CAN bus are functions like collision mitigation or as part of the central locking system functions. The function or use case *Move Window* includes the two functions *Jam Protection*, which e.g. detects a child's hand in its path, and the *EDrive Thermal Protection*, which deactivates the electric drive to protect it from thermal destruction through overload.

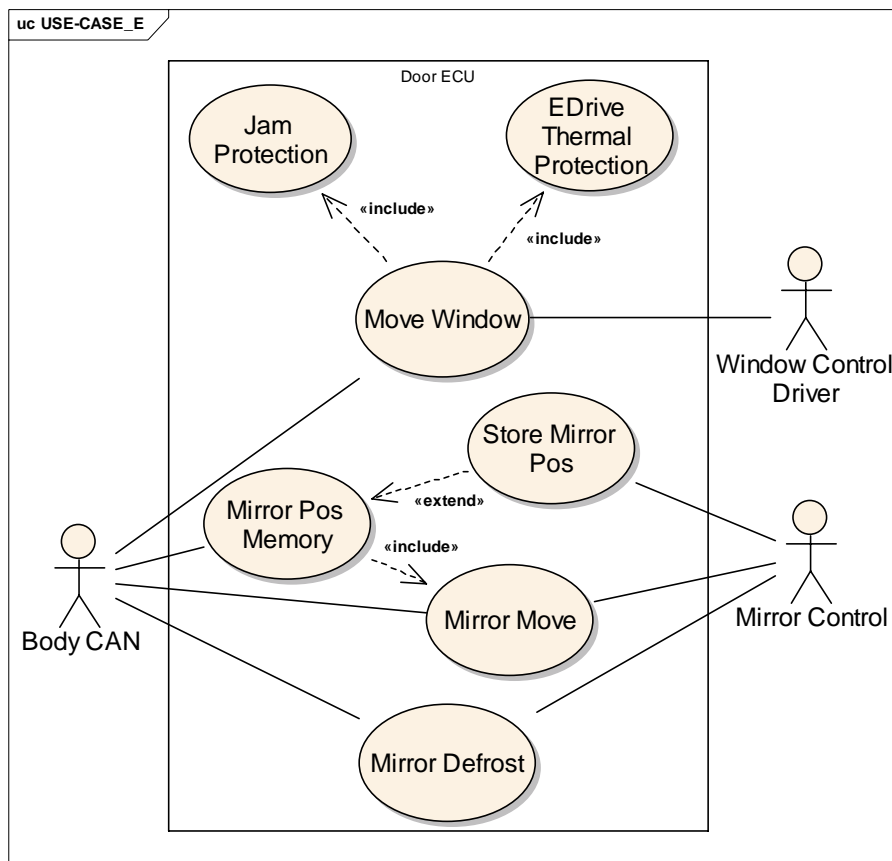


Figure 3-2: Use case diagram with selected use cases of the door controller unit.

The function *Mirror Pos Memory* is responsible for moving the exterior mirrors to the stored positions once a valid key is detected. It includes respectively makes use of the *Mirror Move* function, that can be triggered both via the mirror control panel or CAN bus if applied to the passenger side exterior mirror.

The *Mirror Defrost* function is responsible for defrosting the exterior mirror on driver's request through the mirror control panel or via CAN bus if another system triggers the function. The function could be triggered via CAN bus e.g. if another system activates the rear window heating function.

Each use case must be supported by a more detailed description of the described behaviour. The more detailed descriptions are represented by state and sequence diagrams. Figure 3-3 displays that connection schematically. A nesting of diagrams is possible.

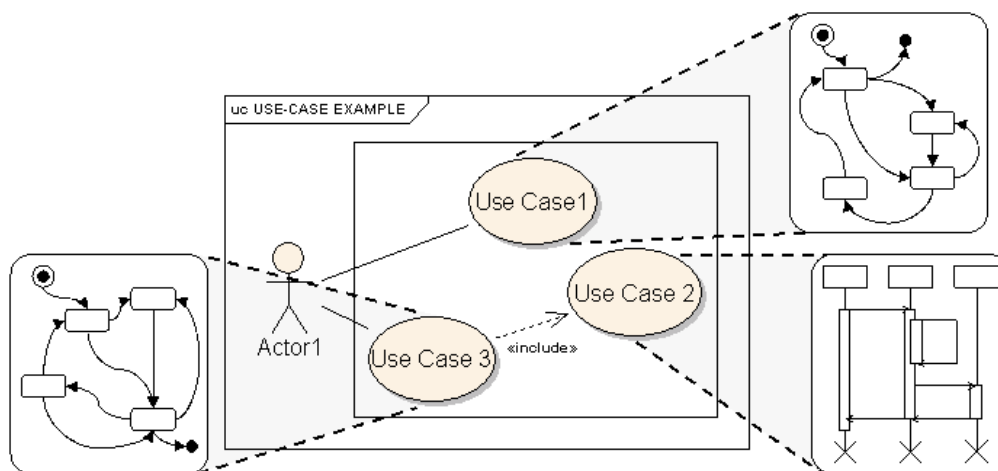


Figure 3-3: Example use case diagram with connected and more detailed behavioural diagrams.

To model pre- and postconditions, which e.g. guard transitions in state diagrams, the object constraint language (OCL, see [7]) is used. The OCL is a modelling language that can express relationships and properties of modelled elements according to [8]. The pre- and postconditions are invoked before or after a method respectively function of the modelled entities or systems is called. The functions are part of the particular class diagrams. E.g.: A lamp modelled by the class diagram in figure 3-1 is to be turned on as part of a behaviour modelled in a state diagram. A state defines the calling of the lamp function *On()*. In order to turn the lamp on a button has to be pressed. The transition to that state is then guarded by an OCL precondition like:

```
button.state == 1
```

While using all kinds of signals and parameters during modelling it's is more intuitive and more efficient to use names as distinctive attribute. The test system which later executes a resulting test case needs additional information. To access a CAN-Signal parameters like start bit, bit length, scaling factor or offset are needed. This additional data is obtained by the test execution machine respectively the connected X-compiler (see figure 1-1) in the uniform pa-

parameter database generated by the AIPas tool, which has been developed in the course of this project.

A syntax comparable to the C++ namespace notation is used to allocate a specific meaning to the used names. The namespace or parameterspace CAN for example is used to identify a CAN-Signal. The SIG parameterspace is assigned to ECU signals, like ground or sensor supply. Other parameterspaces like LIN for LIN-Signals, FLX for Flexray-signals or DIA for diagnostic data exist.

3.3 Example

The following example models the external mirror movement function activated by the mirror control panel (manual activation). The state diagram in figure 3-5 displays the behaviour. The diagram contains five states: one idle state and two states for each axis of movement, where each state corresponds to a rotational direction. Figure 4-4 illustrates the movement of the mirror pane around its x- and y-axis.

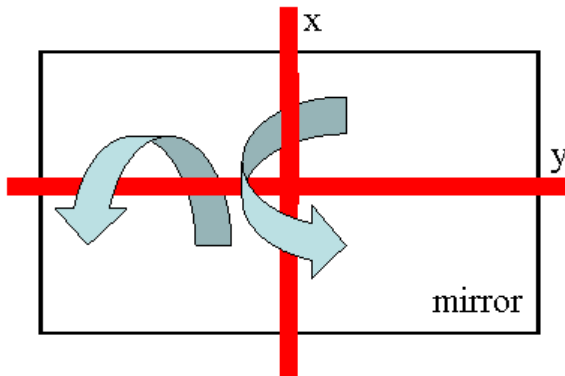


Figure 4-4:

The exterior mirror pane can be moved around its x- respectively y-axis, to adjust to different drivers.

There are ten transitions between those states. Connected to those transitions are constraints expressed in object constraint logic which are very similar to C statements. For example the constraint:

```
SIG::MirrorControlIn >= PAR::MoveMirrorX-LowerLimit &&
SIG::MirrorControlIn <= PAR::MoveMirrorX-UpperLimit
```

guards the transition between states *idle* and *moveX-* as a precondition, meaning that the transition is only executed if the value of the ECU signal *MirrorControlIn* is in the range limited by the parameters *MoveMirrorX-LowerLimit* and *MoveMirrorX-UpperLimit*.

Other constraints define states. For example the constraint:

```
SIG::EDriveSupply > PAR::EDriveSupplyLowerLimit &&
SIG::EDriveCtrlY > PAR::EDriveCtrlLowerLimit
```

connected to the state *MoveY+* indicates that while this state is valid the ECU signals *EDriveSupply* and *EDriveCtrlY* have to be above their respective thresholds. What those thresholds are and whether it's e.g. a value measured in volts or amperes can be identified in the uniform parameter database.

An example of behavioural modelling using a sequence chart can be found in [2].

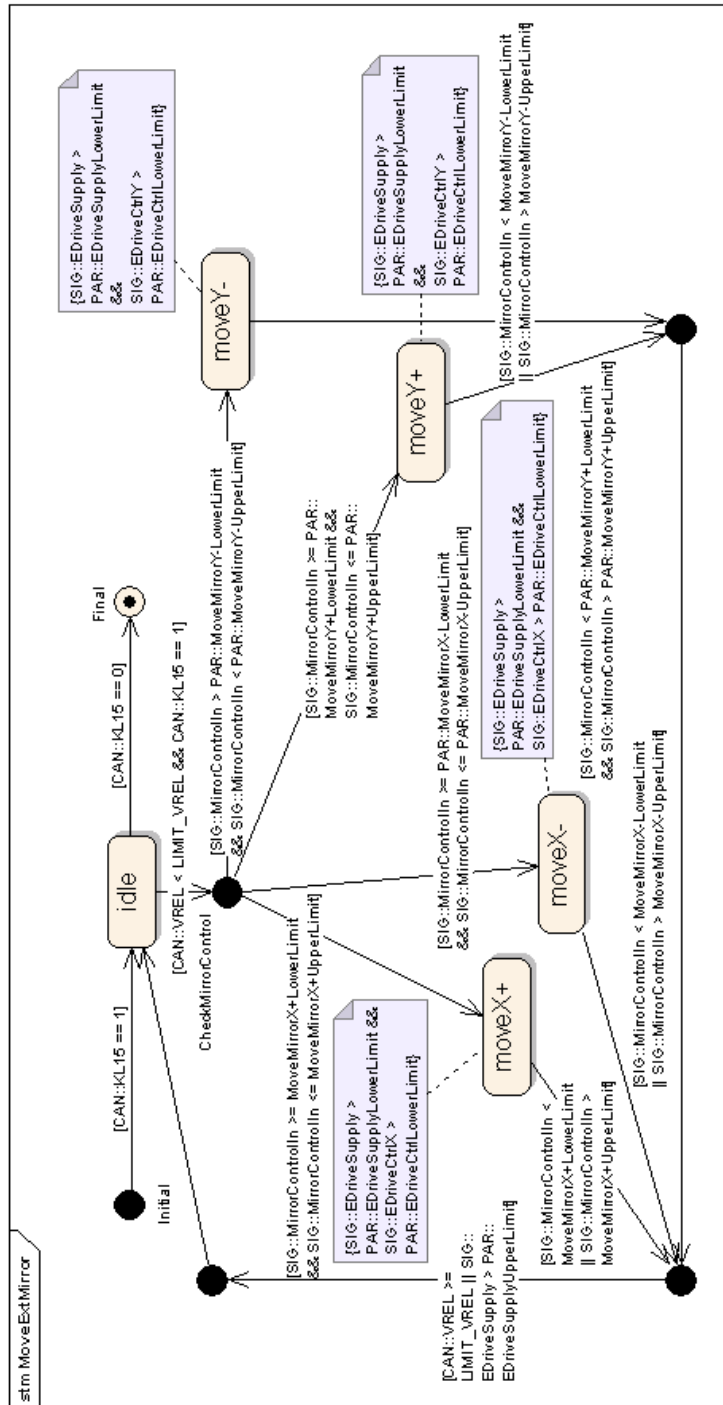


Figure 3-5: Simple state diagram modelling the manual external mirror moving function of the door controller ECU. For readability reasons the state diagram is not fully specified.

4 XMI Ex- and Import

Since UML is only a graphical notation a standardized interchange format is needed for automatic data processing. The XML-based XMI (XML Metadata Interchange) format fulfils this requirement. It's is like UML an OMG (and ISO) standard and is designed for metadata interchange. It can be used for any metadata whose metamodel can be expressed through the Meta-Object Facility (MOF). Usage as an interchange model for UML is its most common use. A detailed description of the XMI standard can be found in [9].

The modelling of automotive ECUs is done through commercial UML tools. Most of those tools include some versions of XMI export. To access the UML information the resulting XML document must be parsed and transformed to an adequate internal data representation.

A XMI document in version 2.1 consists of two parts. One part is all the information needed for graphical representation. It includes all the data to draw a specific diagram. The other part includes the logical setup of the UML model. For example it includes the configuration of a specific state diagram, its assigned states and which states are connected to each other. This part is the part which is of interest for the test-case generation process.

The import process can be divided into three steps.

1. Open the XMI-file and validate it against the corresponding schema.
2. Extract the model information and exclude all other data from further processing. Close the file.
3. Create a hierarchical tree-oriented presentation of the gathered data.

In practice there are some limitations to the interchangeability of XMI documents. Different tools differ slightly in the implementation of their import- respectively export-filters. Differences range from spelling discrepancies to the way packages inside a project are arranged. Thus an adaptation of the XMI import-filter of the test-case generation tool (see figure 1-1) to the specific UML tool used is necessary. The UML tool used in this work is Enterprise Architect by Sparx Systems.

5 Test-Case Generation

The test-case generation is based on different strategies depending on the type of behavioural modelling. The following points need to be addressed.

The modelled functions only include the stimulation or the input vector of the system. Beside the generation of a particular input sequence the monitoring of system outputs is vital in testing. The heuristic or strategy needs to deduce what system outputs are to be measured at what time against what reference. The state *moveX+* in the example illustrated in figure 3-5 is defined by the two OCL statements:

```
SIG::EDriveSupply > PAR::EDriveSupplyLowerLimit &&
```

```
SIG::EDriveCtrlX > PAR::EDriveCtrlLowerLimit
```

meaning that if the state is active the signal *EDriveSupply* and *EDriveCtrlX* have to be above their respective thresholds indicating activation. The test-case generation strategy will then introduce steps to activate the state and e.g. start a measurement operation to check whether the current of *EDriveSupply* is above the indicated threshold.

5.1 Test-case generation based on state charts

Test-case generation based on state charts have been extensively discussed in various sources. Binder [10] offers a good starting point for object-oriented systems. He introduces several strategies to obtain test cases from state charts. Some of those strategies are put to use by the heuristic developed in this project.

„All Transitions“: By executing every transition in a state chart it is guaranteed that all states are visited. This strategy identifies corrupt and missing but not additional transitions. It cannot identify corrupt states.

„All Round-Trips“: The successful execution of all transition sequences identified by the N+-strategy respectively the Round-Trip-Test proves the correctness of the behaviour model. Corrupt and missing transitions are identified. Corrupt states cannot definitely be identified.

The N+-strategy is made up by the Round-Trip-Test and the detection of secret paths, which are transitions that are not explicitly modelled. This test is only necessary if the state chart is not fully specified (see [6] for explanation).

„Round-Trip-Test“: The Round-Trip-Test creates a transition tree. Each resulting test-case describes a transition sequence starting at a particular starting state (root) and ending in a final state (leaf). After executing the sequence it's checked whether the resulting state corresponds to the defined leaf state.

In the following a short example will illustrate the usage of the Round-Trip-Test strategy based on the example shown in figure 3-5. Figure 5-1 displays the transition tree generated based on the given state chart. Figure 5-2 outlines the pattern of the first test-case based on the transition tree.

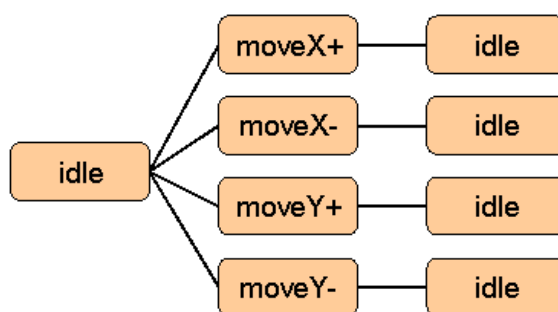


Figure 5-1:
The transition tree of the example state chart depicted in figure 3-5. This transition tree is the basis of the Round-Trip-Test algorithm.

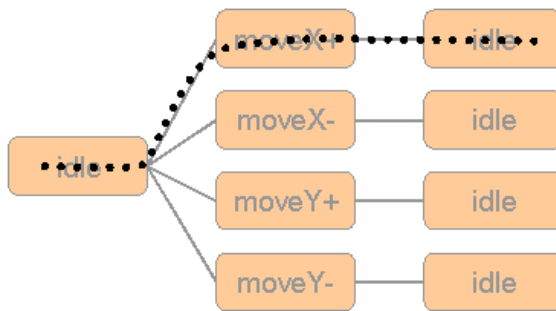


Figure 5-2:

The pattern of the first test-case generated based on the transition tree pictured in figure 5-1. The test-case starts out in *idle* state and moves the mirror counter-clockwise around the x-axis. It returns to *idle* state in the end.

The pattern results in one or multiple test-cases depending on the guarding preconditions expressed in OCL. For the pattern in figure 5-2 four test-cases are determined. The guards of the first transition from state *idle* to state *moveX+* are connected by a logical „and“ and therefore have to be satisfied. The second transition from states *moveX+* to *idle* features four guarding OCL statements which are connected by a logical „or“. Each of those guards has to be tested resulting in three different test-cases. Figure 5-3 displays the first of those three test-cases in the editor of the P.A.T.E. test system. This test-case is based on fulfilling the guard expressions:

```
SIG::MirrorControlIn < MoveMirrorX+LowerLimit ||
SIG::MirrorControlIn > MoveMirrorX+UpperLimit
```

which from the user point of view means that the button to move the mirror around its x-axis is no longer pressed.

Figure 5-4 shows the editor-view of the test-case based on the guarding OCL expression:

```
CAN::VREL >= LIMIT_VREL
```

which from the user point of view means that the speed of the vehicle is greater than or equals the defined threshold for activating the mirror moving function. According to the specification the mirror mustn't move after crossing that threshold. The state of the electric drive that moves the mirror is checked by the two finishing measurement operations: one measuring the current of the supply circuit and the other measuring the voltage of the control circuit.

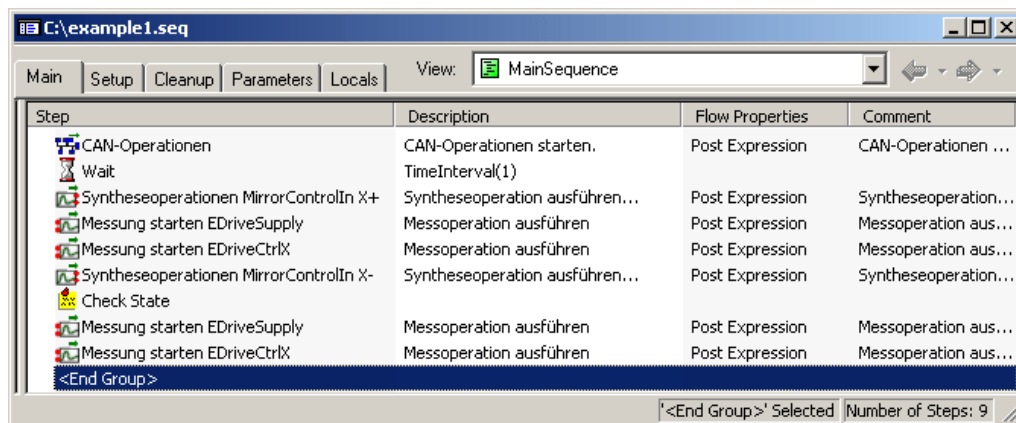


Figure 5-3: First test-case for the moving exterior mirror function in the editor of the P.A.T.E. test system.

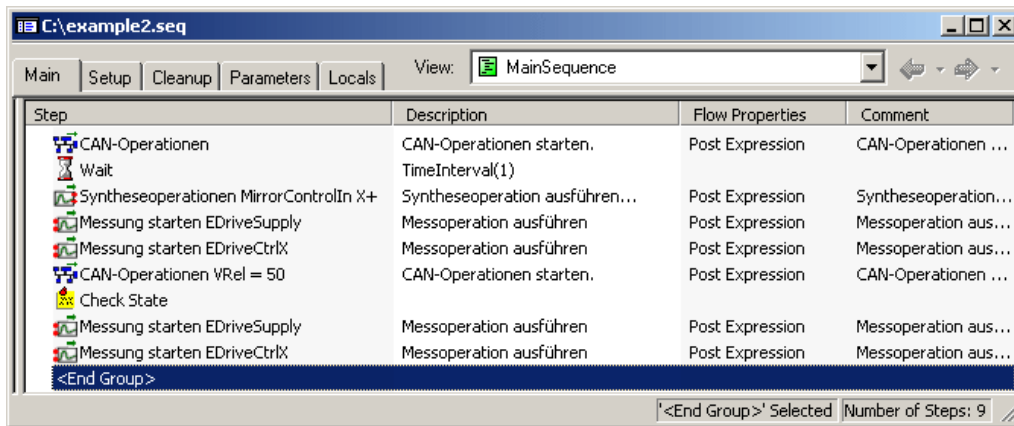


Figure 5-4: Second test-case for the moving exterior mirror function in the editor of the P.A.T.E. test system.

5.2 Test-case generation based on sequence charts

Test-case generation based on sequence-charts is not as well discussed as it is for state charts. One strategy is comparable to the above introduced „Round-Trip-Test“. First step is to create a control flow chart based on the sequence chart. Each path through the control flow chart corresponds to one test-case. The evaluation is done by checking the resulting state with the corresponding leaf state.

See [2] and [10] for an example of the test-case generation algorithm based on sequence charts.

6 Adapting to a specific test system

After creating test-cases and storing them in a test system unspecific format they have to be transformed so they can be executed by a real test system. This step is performed by a test system X-compiler as depicted in figure 1-1.

The X-compiler needs to access the test-cases and create sequences of operations that are readable by the target test system. Using a system-specific profile optimizes the resulting test-cases. E.g. the first target system allows to bundle up to 10 CAN- and LIN-operations, the X-compiler uses that information to generate more compact test-cases.

Each operation of the target system needs a particular set of parameters. E.g. CAN-operations need the scaling and the offset of the specific CAN-signal, diagnostic error codes need a verbal

description etc. The X-compiler uses the uniform parameter database to obtain those parameters (see figure 1-1).

The first test system integrated in the process is a P.A.T.E. system for the body domain. It is an open loop tester for quasi-static operation modes. All ECUs of the body domain can be tested by the system. Parallel testing of two ECUs is also supported. Figure 6-1 displays the system in its current expansion state. The test-cases are generated by using the COM-server of the deployed test execution machine.



Figure 6-1:

The P.A.T.E. test system for the body domain. The test system features parallel testing of two ECUs for distributed functions. It's a universal test system that allows the testing of all ECUs within the scope of its hardware configuration. It is designed as an open-loop tester for quasi-stable operating conditions.

7 Outlook

Current work is focused on the test-case generation algorithm and its automation. At the moment the implemented tool exports test sequences for National Instruments Teststand which is the test execution engine for the above mentioned P.A.T.E. test system. The next step will be to implement an XML-based export filter, which exports files relating to the IEEE 1671 (ATML) standard.

Optimization and development of the test-case generation algorithm will include the modelling of mode ECU functions of the door controller unit and the modelling of functions implemented by other ECUs. Those functions will be utilized to further optimize the algorithm.

The resulting test-cases will be compared to the test-cases that were generated manually during the development process of the corresponding ECUs. It will be assessed how good the batch of automatically generated test-cases fits the manually generated one. Criteria will be the amount of test-cases generated and coverage.

8 References

- [1] IEEE: „IEEE Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML“, Piscataway, NJ, 2006
- [2] Brost, M.; Reuss, H.C.; Zöller R.: „Automatische Testfallgenerierung aus einer formalen Funktionsbeschreibung“, 1.Autotest, Stuttgart, 2006
- [3] v.d. Beeck, M.: „Eignung der UML 2.0 zur Entwicklung von Bordnetzarchitekturen“, Tagungsband „Modellbasierte Entwicklung eingebetteter Systeme II“, Informatik Bericht TU Braunschweig, 2006
- [4] Götze, M.; Kattaneck, W.: „Erfahrungen mit der UML beim Entwurf von Kfz-Steuerungen“, ITG/GI/GMM Workshop „Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen“, Meißen, 2001
- [5] Rumpe, R.: „Modellierung mit UML“, Springer Verlag, Heidelberg, 2004
- [6] Hopcroft, J.; Ullman, J.: „Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie“, Pearson Studium, Reading, 2002
- [7] Object Management Group: „Object Constraint Language V2.0“, Needham, Massachusetts, 2001
- [8] Warmer, J.; Kleppe, A.: „The Object Constraint Language: Precise modelling with UML“, Addison Wesley, Massachusetts, 1998
- [9] Object Management Group: „MOF 2.0/XMI Mapping Specification V2.1“, Needham, Massachusetts, 2004
- [10] Binder, R.: „Testing Object-Oriented Systems – Models, Patterns, and Tools“, Addison Wesley, Massachusetts, 2000