Model-Based Quality Assurance of Automotive Software*

Jan Jürjens[1], Daniel Reiß[2], and David Trachtenherz[3]

[1] The Open University, UK
[2] Elektrobit Group Ltd, Germany
[3] Techische Universität München, Germany

Abstract. Software in embedded (e.g. automotive) systems requires a high level of reliability. Model-based development techniques are increasingly used to reach this goal, but so far there is relatively little published knowledge on the comparative benefits in using different assurance techniques. We investigate different and potentially complementary model-based software quality assurance methods (namely simulation and white-box testing vs. model-checking) at the hand of an application to the software component of a door control unit. We draw conclusions with regards to suitable application use cases.

1 Introduction

Software plays an increasingly important role in embedded systems. This applies in particular to the automotive sector, characterized by strict safety requirements to components of a motor vehicle (see [5, 16, 7, 18] for background). The constantly growing portion of software, the short development cycles and the rising complexity of software in the automobile represent high challenges to the car manufacturers, together with the limited computer resources available on the embedded components which have to be kept as cheap as possible because of the high sales numbers in this domain. This factors require an intensive use of quality assurance measures to ensure a high level of reliability, quality, and safety. Modern development procedures such as modelling and quality assurance using graphical description methods are increasingly used towards achieving this goal. Unfortunately, despite the high amount of attention that these techniques receive in the software development community, there does not seem to be much published knowledge available which would compare the effectiveness of the different and potentially complementary approaches when applied to the kind of software used in embedded and in particular automotive systems, in particular when it comes to a leading edge approach such as model-checking.

One goal of this work is to address this knowledge gap with an experience report on a field study which had the goal to compare conventional software quality assurance methods (specifically, testing) with one of the recently developed assurance approaches (specifically, model-checking). The field study took place in the context of the automobile industry. The target of the field study was the development of a door controller

* This work was performed partly when the first and second authors were at TU Munich and the third at BMW. Contact: http://www.jurjens.de/jan

unit based on a publically available textual specification [14]. The components for the door lock, the window lifter and the light control described in this specification were modelled in two CASE tools, the commercially distributed tool ASCET (which offers extensive test assistance), and the research prototype AutoFOCUS (which comes with innovative quality assurance plug-ins like model-checkers). For each of the two models constructed within these tools, different quality assurance techniques (white-box testing for ASCET and model-checking for AutoFOCUS) were applied. We then evaluated the outcomes and compared the two approaches with the goal to investigate the relative advantages and disadvantages in using them, wrt. their effectiveness in finding errors and the required time efforts. We also discuss possibilities for a complementary use.

Case study Door Control Unit: We evaluated the software quality assurance methods in the context of the development of the software component of an electronic door controller unit from [14]. This example was selected on the basis that it is representative for a specification in the automotive domain, with a particular emphasis on functionality (rather than requirements on base level software). According to the specification a door controller unit (DCU) is installed in each front door. Both electronic controller units are connected via the CAN bus and hence to the other electronic control units and intelligent sensors of the electronic comfort system. The DCUs are responsible for the settings of the front seats and the side-view mirrors, the management of user settings, the interior lighting, window lifter and the central locking system. This work concentrated on the functionality of the last three components. Error protection mechanism and on-board diagnosis were not in scope of this work. Fig. 1 gives an overview over the window lifter module as part of the door control unit. *Dispatcher* distributes the incoming signals to the concerned components over the connected communication channels[4]. *CAN_Front* and *CAN_Back* process signals for the CAN bus. *Request_Front* and *Request_Back* process signals from the local peripherals of the door control unit. The result is passed to the engine controls *FrontMotor* and *BackMotor*.

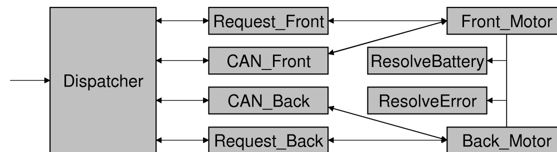

Fig. 1. DCU component *Windows* – Overview

Properties under test: Only functional errors in the model or implementation were in scope of our investigations (and thus the errors were found after the modelling resp. code generation phase). More specifically, we focussed on those properties that could be expressed as state reachability properties. Errors in or misinterpretations of the original specification were not included in the error count (but they did have an impact on the time effort needed for the modelling phase). Data collection was performed by the person who undertook the study (the second author).

2 Application in ASCET

The ASCET tool chain provided by the ETAS Group is used by automobile manufacturers and suppliers in the automotive industry. It supports an integrated software devel-

[4] A “channel” is a communication link between two modules.

opment process according to the V-Model. In conjunction with ASCET, additional tools are used to apply the desired conventional testing methods with code coverage. Within the case study, we used the tool CTE XL to graphically specify test cases in form of a matrix. They could then be exported and converted into an appropriate format to be executed in ASCET. On the other hand we measured the code coverage of the test cases using the testing framework "Test RealTime" provided by IBM Rational.

3 Application in AutoFOCUS

The CASE tool AutoFOCUS is a freely available system development tool developed at TU Munich. It is based on concepts of the formal description methodology FOCUS [6] and provides interfaces to QA plug-ins such as the SMV model-checker.

Testing: AutoFOCUS can simulate the execution of a model in discrete time steps. The data flow between components and the computations of the individual components of a system and their underlying state machines (modelled as State Transition Diagrams / STDs) can be visualized in a simulator. For every time step input test data can be manually specified. Using the Validas code generator one can also generate code and then apply conventional testing methods to the generated code. Since the code is generated in a canonical way it is straightforward to transfer bug fixes back to the model.

Model-Checking: Due to the state explosion problem, we only verified components with less than ten State Transition Diagrams using SMV. The time effort to check a property varied in general between two and ten minutes for each check depending on the complexity of the components. Logical errors could thus be determined and excluded easily without interrupting the development process of the modules for too long.

Example Crush Guard: We discuss the use of model-checking on a property of the crush guard for the front window. As mentioned, the complete DCU model was not processed by the model-checker at once. Hence a modular approach was chosen to show the crush guard's correctness for the whole system: The model was divided up from a functional point of view and structured hierarchically. The goal was to minimize mutual dependencies and interface dependencies so that each block could be viewed as a "black box". The significant component *FrontMotor* is located in the *Windows*-component. The interface of this component consists of the following inputs:

- *Bottom_Front*: registers if the window pane resides at the lower block
- *MovingSensor_Front*: registers the movement of the pane
- *Top_Front*: registers if the window pane resides at the upper block
- *Battery_Front*: registers the current battery voltage
- *MotorFront*: registers a command request of the component *Request_Front*

The outputs are the following:

- *Front_Motor*: controls the voltage of the window lifter motor
- *Front_Out_ERROR_WIN*: signals an error
- *FrontLowWin*: signals low battery voltage

Using model-checking, the following property could be verified for *FrontMotor* for all system states, following the DCU specification:

```
1.  []((
2.      (Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
3.         () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
4.           () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
5.             () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
6.               () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
7.                 () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)
8.                      && not(is_Msg(MC_FrontMotor.Top_Front))
9.                      && not(is_Msg(MC_FrontMotor.MovingSensor_Front)))))))
10.   ) => (
11.     (()(()(()(()(()(()(
12.       ( (Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)
13.         $Until
14.         ( (not(is_Msg(MC_FrontMotor.Bottom_Front)) &&
15.            not(is_Msg(MC_FrontMotor.MovingSensor_Front)))
16.           || (Val(MC_FrontMotor.Bottom_Front)==F_UNTEN_Bottom))
17.       ) || (
18.         (Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down) &&
19.           () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down) &&
...           ... // 6 identical lines with NEXT operator omitted for sake of brevity
26.                         () ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down) &&
27.                           () (Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down)
28.   )))))))) ))))))) ))
```

Fig. 2. LTL formula

If the motor is on for at least the length of the startup-time (which is the time needed for the window pane to start moving) and a blockage of the window pane is registered after the expiry of this time then move the window down until:

- *the lower block is reached,*
- *the 3-second-timeout occurs or*
- *the window movement is blocked (e.g. due to a mechanical malfunction).*

The corresponding LTL formula is shown in Fig. 2 (line numbers were added for referencing the lines). In detail, the lines in the formula mean the following:

Line 1: For all system states the lines 2 to 28 apply.
Line 2 to 7: The activation of the motor occurs at least for 6 system ticks (abstraction of startup time).
Line 8 to 9: After startup, the window is at the upper block and does not move.
Line 10: If a blockage is registered (line 2 to 9) then react (line 11 to 28).
Line 11: The reaction follows at the point in time after registering the blockage.
Line 12 to 15: Move down the window (line 12) until (line 13) the window blocks (line 14 to 15) or the lower block is reached (line 16),
Line 17 to 27: or 10 system ticks have passed (abstraction of 3-seconds-timeout).

The LTL formula accesses neither subcomponents nor internal channels of *FrontMotor* but only the external interface. Thus the formulated property describes a black-box property of *FrontMotor*. The processing of a signal input in the component *FrontMotor* needs some time in form of global time ticks. Thereby messages can internally pass through a pipeline[5] of subcomponents. Hence some system ticks are needed before a result is available on the output channels. When setting up a formula in AutoFocus

[5] With "pipeline" we mean a chain of channels each leading to a communication delay.

this pipelining effect is also considered. For a given property (e.g. “if X applies, then Y arises”) the temporal delay must be expressed. The time delay becomes manifest in the change of the formula (“if X applies Y results in 2 system ticks”). Implementing these technical details results in increased formula size. Furthermore small improvements of the model can lead to great changes of the formula. On the other hand, the AutoFOCUS-model can also be modified to get a simpler and clearer formula. E.g., to avoid pipelines, a component network can be combined into a single STD.

Note that getting the LTL to correctly formalize the requirement one wants to verify is non-trivial. Since the original requirement is itself not formal, it is not possible to prove that the LTL formula is correct, but one can only increase confidence in it by inspection and simulation. In particular it can happen that the verification fails because the LTL formula is incorrect. This means that if the verification fails, both the model and the LTL formula have to be checked and any of them may need to be corrected.

4 Evaluation

4.1 Evaluation of the Tools

To compare the tools AutoFOCUS and ASCET, one should keep in mind that they were developed against different backgrounds: AutoFOCUS is supposed to be as generally applicable as possible, while ASCET is more aligned to the development of electronic control units and specialized for use in the automotive domain. Also the influence of the market and the manpower of the corresponding companies differs. Therefore the following sections concentrate on the comparison of the principles behind the two tools and the applied verification methods. It is also emphasized that some restrictions in AutoFOCUS were only imposed by the constraints of model-checking, e.g. due to the state space explosion problem. The time needed for constructing the LTL formalization of the requirements to be verified is not included in the effort discussed above.

Modelling: The AutoFOCUS notation is relatively simple compared to other modelling tools such as ASCET. This results in a simpler semantics which is more easily accessible for formal verification. The approach is also relatively general: Application areas of AutoFOCUS are not limited to embedded systems in the automobile industry. However, it also makes it more difficult to deeply integrate concepts of the automotive domain, such as:

- Scheduling in OSEK using tasks and processes
- Mapping of the data types which are supported on the target hardware
- Distinction of program code and data fields in memory (e.g. for future replacement of configuration data)

Thus AutoFOCUS is primarily suitable for the description of the abstract logic of a function. The embedding into the hardware context and the precise modelling of the above mentioned concepts should not be attempted using AutoFOCUS, since this would significantly increase the size and complexity of the model (and lead to problems with model-checking because of the state explosion problem). Also the pipeline effect of the signal processing flow and associated synchronization problems due to the parallel computation of the component network get more complex with an increasing model size. In

ASCET the synchronisation of the individual components also plays an important role. There a sequential processing flow is supported by the use of process and task scheduling concepts. On the level of the component behaviour, in AutoFOCUS the selection of state machine transitions with the same priority takes place indeterministically, so the user has to enforce determinism explicitly by assigning appropriate preconditions and priorities. In ASCET the computations in STDs are deterministic because different transition priorities are mandatory. The modelling of time-controlled behaviour in AutoFOCUS requires the discretisation of time because of the global system ticks. Timer components can be realized in AutoFOCUS using integer variables. The resolution of a timeout interval is determined by a number of ticks. In ASCET time values are directly declared in the model. Restrictions in AutoFOCUS arise due to the state explosion problem when using model-checking. A limitation of the number of computation steps and of the integer values is necessary. Hence AutoFOCUS in combination with model-checking is well suited for event-controlled systems without temporal conditions.

Temporal effort for the modelling: The simple AutoFOCUS notation can be learned within a day due to its simplicity (as was demonstrated in the case-study since the user did not have any prior knowledge of the AutoFOCUS notation). The modelling of the components for the central door lock and the window lifter took place in three weeks for each. The interior light component was modelled in one week. The difficulty was in the correct interpretation and conversion of the textual specification. In particular, the time effort for constructing the AutoFOCUS model includes the effort for clarifying any unclear issues with the original specification. Further additional corrections of basic errors and the bug fixes for synchronization problems required a complex restructuring. The time effort was also a result of the restrictive notation and the missing possibility to copy whole component hierarchies. This drawback is being resolved in the ongoing development of AutoFOCUS. The need for copying hierarchies could also be avoided if the complete model was known in advance and requirements were fixed. But in practice this assumption usually does not hold since requirements can change during the development phase and a model is constructed successively during the development phase. Another week had to be invested for restructuring the *Windows* component to make it suitable for model-checking with respect to the state space explosion problem.

ASCET required a longer phase of one week to get familiar with the modelling elements. The reason for this is the vast variety of modelling elements with association to concepts of the automotive domain. The main problems concerning the interpretation of the DCU specification were already discovered due to the modelling in AutoFOCUS. Thus the time to create the model of each component in ASCET took about half the time as in AutoFOCUS. Also the liberal description languages and modelling elements facilitated the modelling. The structure of the model in AutoFOCUS was reused in ASCET and brought into a more compact form. The application of the test procedures led only to small bug fixes. The logical concept remained stable.

Table 1 gives the time effort spent for the modelling in both tools. Note that the time effort for AutoFOCUS also included the preliminary work on the specification that did not need to be repeated for ASCET, so these effort values are not meant to be used to draw any conclusions about the relative ease of using the two tools but just to give an idea about the size of the case-study that was performed. Unfortunately it is difficult to

	ASCET	**AutoFocus** (Incl. spec.)
Training	1 week	1 day
Modelling: door lock	1.5 weeks	3 weeks
Modelling: window lifter	1.5 weeks	3 weeks
Modelling: interior light	0.5 weeks	1 week

Table 1. Time effort.

	ASCET	**AutoFocus**
Experiment / simulation	10	5
Model-checking	-	5
White-box tests	5	-

Table 2. Number of errors found.

determine in a reliable way which fragment of the time effort needed for constructing the AutoFocus specification was needed to understand the specification, because these two activities cannot be separated when constructing a first model of a specification (because many clarification questions about the specification only arise while working on the model). Therefore, we do not attempt to adjust the data given in Table 1 to account for this difference. However, we estimate that the conversion and interpretation of the specification constituted nearly half of the time used for modelling in AutoFocus (and this was not repeated in ASCET since one could benefit of the experience previously acquired when building the model in AutoFocus). Thus one may hypothesize that in fact approximately the same time effort was necessary for the modelling in both tools (apart from adaptations for model-checking that became necessary in addition).

4.2 Evaluation and comparison of the quality assurance methods

Both testing and model-checking were tool-supported and used automation available in the tools (in particular producing input data in CTE XL and exporting them for testing, as well as automated model and property export from AutoFocus for model-checking). Thus we believe the experimental arrangements for both methodologies to be fairly comparable. The errors found when comparing the two quality assurance approaches can be considered to originate from modelling resp. code generation and to be independent of possible misinterpretations of specification: Both models as well as their quality assurance were performed based on the same interpretation of the system requirements specification. Thus the quality assurance for the two models reflected the same specification interpretation as implemented in the models and just determined whether the implementation was correct with respect to this interpretation of the specification. Therefore the comparison results should not be influenced by possible misinterpretations of specification. Validation of the requirements specification is out of scope of this paper, as the focus of this work lies on comparing different approaches to quality assurance.

Error Finding: Table 2 compares the number of bugs found using both tools. It lists the bugs which were uncovered after the completion of the modelling of the components by applying the testing methods. Furthermore it must be noted that for the listed bugs the simulation always preceded model-checking and white-box testing. Hence the majority of the bugs listed were already found during the simulation.

The errors could be classified as follows:

- errors in interpreting the specification
- specification errors (e.g. incorrect timing information)

- oversights (e.g. forgetting a transition)
- conceptual errors (specification correctly interpreted but incorrectly transformed to design)
- obscure errors (only occurring under obscure side conditions, e.g. race conditions, sporadic errors or timing errors, e.g. due to pipeline effects)

In ASCET discrepancies with the DCU specification were found by simulating the model. These errors were due to a wrong implementation: For example setting the wrong priority for a transition of a state machine or forgetting to reinitialize the trigger of a transition. There were also deadlocks or incorrect transition preconditions which led to a wrong state machine behaviour. In the code generation phase, omitted cases in case distinctions can be detected. Missing initializations or type errors should already be detected by the tool during the modelling phase and thus were not relevant for our study. White-box testing with code coverage analysis detected the remaining bugs like careless errors (e.g. transmission of a wrong message value) or unreachable code due to missing transition preconditions, or errors originating on the conceptual level (unless these were caught earlier if they lead to incompatible interfaces).

In AutoFOCUS the simulator was quite helpful in order to expose basic errors (e.g. wrong prioritization of transitions). Bugs which were hard to detect (e.g. concerning pipeline synchronization and boundary conditions) could easily be found by the model-checker. We did not notice any errors that would have been introduced by changing the model in order to make it amenable to model-checking (although in larger projects involving more people this could certainly be expected). However, some kinds of errors (e.g. pipeline synchronization for time-triggered communication) were due to the time-triggered communication semantics of the AutoFOCUS modelling notation, while other errors (e.g. boundary conditions and incorrect prioritization of transitions) occurred for both modelling techniques. Such bugs would take an substantially longer time to be found using conventional testing methods because of the difficulty in determining the corresponding test input data.

Some errors were found with both approaches. For example, states that will not be reached (e.g. because of a missing transition or a wrong transition priority) can be found using simulation or testing by transversing the relevant transition path, or using model-checking against a suitable reachability property. Conversely, showing that a certain state will *not* be reached can be easier using model-checking because it ensures that all possible paths are investigated.

Note that in principle, all errors found using model-checking could also be found using automated testing because at least theoretically it would usually be thinkable to achieve a complete test value coverage (i.e. to test every possible sequence of input values and every possible sequence of assignments to local variables). In practice, a complete test value coverage is usually not feasible given the available time and money. However, this means that one cannot characterize errors that can be found using model-checkers but not using automated testing on a logical level, but this always depends on the practical application situation (in particular economic aspects).

Example: As an example one can consider the component *FrontMotor* responsible for driving the window pane motor. Its subcomponent *DispatchCommand* handles the current state of the pane movement. *DispatchCommand* is connected to a further

```
[]((( Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
     ()  ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
       ()  ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
         ()  ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
           ()  ((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up) &&
             ()  (((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)
                   && not(is_Msg(MC_FrontMotor.Top_Front))
                   && not(is_Msg(MC_FrontMotor.MovingSensor_Front))) &&
                ()((Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Up)
                   && not(is_Msg(MC_FrontMotor.Top_Front))
                   && not(is_Msg(MC_FrontMotor.MovingSensor_Front)))))))
  ) => ( (()(()(()(()(()(()( Val(MC_FrontMotor.Front_Motor)==F_MOTOR_Down ))))))) ))
```

Fig. 3. LTL formula for *FrontMotor*

subcomponent *UpError* responsible for detection of a window jam condition. The LTL-formula in Fig. 3 has to be verified for *FrontMotor*. It can be interpreted in the same way as shown for the LTL-formula in Fig. 2 and tells that a window jam has to be acted on. Since external signals are forwarded by other subcomponents until they reach *DispatchCommand* the jam condition has to be checked twice after the startup time (pipelining effect). But this also imposes a synchronization problem as it could be shown by a counter example of the SMV model-checker: A certain signal input sequence leads to a window jam signal which will not be processed - the window pane continues moving up instead of down. The reason for this is a reset input signal for *DispatchCommand* which overlaps the jam input signal and is followed by a window close input signal.

Neither in AutoFOCUS nor in ASCET did bugs occur due to a wrong interpretation of the textual DCU specification. Inconsistencies in the specification (e.g. different interpretations of the values for the CAN message *ZV_SCHL_X* in different sections in the document) were resolved before implementing the corresponding functionality.

Intellectual effort: The main effort for testing in ASCET was particularly the interpretation of the test case output and checking its consistency regarding the DCU specification. Generally it is possible to reduce this effort with an increasing degree of automation. The difficulty here lies in the collection of dependencies between the requirements and their transformation into a computable form. There seems to be so far no tool which automatically transforms requirements from a textual form into a machine processable one without human interaction. The existence of a specification on a formal basis would support this process.

Model-checking for AutoFOCUS models was used to examine properties for small component networks. Since the knowledge of the internal structure of the components was available (white-box assumption), simple properties (e.g. absence of deadlocks) could already be formulated and checked during the development phase. The reason for an error found by model-checking could be either a bug in the model, an error due to the interpretation of the specification, or an error in the specification. The difficulty in model-checking is the formulation of correct properties which are consistent with the specification. One is forced to closely inspect the specification to avoid introducing consistency errors. The formulation of properties without any knowledge of the internal data flow of a component (black-box assumption) is difficult in AutoFOCUS. For example, time delays can be expressed by concatenating NEXT-operators according to the time discretization in AutoFOCUS. The resulting properties are generally complex due

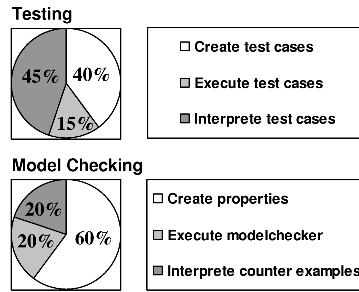

Fig. 4. Time distribution

Procedure	Time effort (hours)	Number of bugs
Experiment (ASCET)	24	10
Simulation (AutoFOCUS)	24	5
Model-checking	80	5
White-box testing	56	5

Table 3. Found bugs in relation to time effort.

to internal interactions and pipeline effects. With increasing complexity of the formula it is also often hard to convince oneself of the correctness of the formalization of the property itself. However, the added redundancy forced by formalizing the verification properties allows one to find errors that may be undetected by straightforward testing. Where possible, both approaches should be used both in parallel to the development itself by the developers (model-checking for an incremental development of model, and unit testing for quick feedback on the ongoing development), but also by an independent QA group (who may have additional specific expertise in applying them, and a different motivation and point of view that facilitates finding different kinds of errors).

Time effort: Nearly the entire time for conventional testing was spent on creating test cases to get a good code coverage and for the interpretation of the test case outputs. This part could be reduced if appropriate automation tools are available, for example automated oracles on the basis of state invariants. For model-checking nearly the whole time had been spent to formulate the properties in accordance to the model. A run of the model-checker to show or contradict a property for small components with a small state space lasted not more than five minutes on average. However, such a relatively short time cannot always be assumed. Depending on the model size several hours or days can be spent. Thus a compromise has to be made between model accuracy and the time effort, depending on the assurance level needed. Fig. 4 shows the estimated effort distribution within the testing and model-checking activities.

When building the model an alternation of model-checking and resulting model changes will sum up the time effort spent for model-checking. So even if one model-checking pass lasts only some minutes the overall amount of time could be considerable. If the existence of a bug was shown by the model-checker its localization also takes some time. A counter-example must be executed and interpreted which can be an intuitive task without automation. It is possible to accelerate this process by bounded model-checking. Apart from the runtime of the model-checker bounded model-checking will also reduce the length of the counter example. Table 3 sets in relationship the time effort for each quality assurance method and the number of bugs found. Again it is noted that simulation preceded model-checking and white-box testing.

In the context of this case study the same number of bugs were found by model-checking as for code coverage testing. However it must be noted that the errors found by model-checking were essentially of logical nature: For example when the result of

a parallel computation of components had to reach a receiving component at the same time tick, the model-checker could simply tell the user if the computation of one of the preceding components delayed and failed to deliver its result in time. Mainly synchronization errors in a component network were detected easily by model-checking.

Careless errors like having forgotten to specify a channel, priority or using an incorrect data type were also in the scope of the bugs found by model-checking but most errors of this type were found in AutoFOCUS by static interface checking and simulation (preceding the model-checking). In ASCET testing with code coverage and simulation detected mainly careless errors (e.g. using the wrong priority of a transition). Here the amount of synchronization errors was smaller since ASCET does not use global time ticks where some signal has to reach a component at an exact point in time. Errors of logical nature were deadlocks due to missing preconditions of transitions.

The relative amount of time needed for finding a particular bug in comparison between testing and model-checking depends on the error at hand: Simple errors can be quickly found using simulation (and would take more effort to be found using model-checking because there one first needs to formalize the relevant requirement in LTL), while intricate errors can be found more quickly and reliably using model-checking. Testing is thus best for quickly ruling out a large number of simple errors, while model-checking is best for reliably ruling out certain intricate errors. Typical examples for this observation are the following:

– State machine dead-locks because a transition priority is defined incorrectly. Found using the ASCET simulator. Time effort: few minutes.
– Data arrives in the wrong order at a module because of the pipelining effect. Found using the AutoFOCUS simular. Time effort: few minutes.
– Race condition causes a window to continue closing although the sensor has recognized that it is blocking. Found with model-checking using AutoFOCUS. Time effort 1-2 days (to determine and formalize the relevant property in LTL).

5 General comparison of the two methodologies

From the mere testing point of view, more bugs are found the more successful the tests are (although of course in a larger context, it is rather that the more successful the development is the fewer errors are found during testing). The probability to find errors increases with the number and meaningfulness of the test cases. Many large test cases in the CTE-Tool lead to large matrices which do not really contribute to the clarity of the interpretation. It is generally not possible to get a 100% input coverage. The number and time of possible test cases is limited in practice.

The same applies to model-checking: Generally it is not possible to prove a 100% correctness of a model under all circumstances. Only the correctness of the specifically formulated properties can be proven. The coverage of the model thus depends on the number of properties that were investigated. Ideally for each requirement of the specification the appropriate property is proven. For safety reasons also the corresponding negated form of requirements should be investigated (and shown not to hold) to avoid system failures due to incorrect usage. Thus also for model-checking revealing a bug in

a complex system is simpler than the proof of its overall correctness. And even showing the correctness of a property is relativized by the following possible problems:

- Initially defined assumptions do not hold (e.g. due to hardware problems)
- The textual specification is erroneous or its interpretation incorrect.
- The proof system is incorrect (e.g. due to programming bugs).

If quality assurance is performed by the developers of the software themselves, model-checking has the advantage that it forces the users to think in a way that is much different from the writing of the software, because they have to create the model and the formalized of the property under investigation. In comparison to that, conventional testing, when performed by the developers themselves, is much closer to the programming itself. For example, the code may have actually been constructed with a certain set of test cases in mind, so that it by construction it will satisfy these test cases, but may not satisfy others then, which however the developer may then not think of.

In practice, model-checking generally requires a significant effort when applied to complex projects. A simplification of the model as had to be done for the DCU in AutoFOCUS to reduce the state space would pose additional problems in practice. The developers are usually concerned with problems of control systems and hardware problems and normally do not have the time to cope with the theoretical challenging of model checking due to a short production schedule. Forcing modelling guidelines to achieve a model-checkable system by default would impair usability. Therefore, similar to the necessity of separate testing groups for conventional testing verification, specialists should accompany the development process of complex projects when model-checking is used.

Nowadays, much of the software of electronic control units is written in a conventional programming language like C (also in systems supporting AUTOSAR (see http://www.autosar.org for more information). It includes the basic software modules (e.g. operating system and diagnosis) with their hardware dependent functionality (communication bus and microcontroller interface). The use of models is more popular only on the higher application layers. It is however often difficult to provide an abstract interface to these by hiding hardware aspects (e.g. interrupts, workarounds for hardware bugs). In AUTOSAR, one tries to cope with this situation using a layered approach including a virtual function bus and a run-time environment.

Limitations for model-based development and model-checking in embedded (in particular automotive) systems in practice thus include the following:

- Limitations of computing power or time.
- Only software components of limited complexity can be abstracted sufficiently to become model-checkable.
- Model-based approaches work well only on high levels of abstraction: Time dependencies and interrupts which occur in the base software are hard to formalize in the model without compromising its usability and the feasbility of its model-checking.
- Memory constraints do not allow generating arbitrarily complex code (unless optimized for space).
- Lack of existing experiences in practice.
- Ongoing developments in model-checking research which make it difficult for practitioners to keep track.

6 Related Work

Only few case-studies on testing or model-checking automotive or more generally embedded software are published. An example is [2], which presents the application of model-checking based verification tools to STATEMATE-based specification models of automotive control units. [17] reports on a case study of an automotive network controller to assess different test suites in terms of error detection, model coverage, and implementation coverage, where the suites were generated automatically or manually in various ways. Automatically and manually derived model-based test suites detected significantly more requirements errors than hand-crafted test suites directly derived from the requirements. [12] appears to conclude that the significant progress in terms of productivity and quality made in recent years in automotive software development has been achieved without direct usage of formal methods. This raises the question whether further improvements could be reached *with* the use of formal methods (to some extent suggested by our case-study, although we did not aim to perform a full empirical study on this). A comparative description of ASCET, AutoFOCUS and further CASE tools can be found in [19], with a focus on methodology. There has not been much work comparing formal verification and traditional quality assurance so far. [1] compares directed testing, random testing and model-checking, but treated hardware verification rather than embedded software. [4] gives a detailed study on the usage of testing, runtime analysis, model checking and static analysis techniques for NASA Rover Software, although design model verification was not considered. [13] reports on a controlled experiment that investigates the impact of using statecharts for testing class clusters that exhibit a state-dependent behaviour. [8, 15] report on experiences on UML model analysis, but do not consider model-based testing. [3] presents an assessment framework for symmetrical comparison and evaluation of testing and formal analysis with respect to debugging and more specifically bug detection. Other experience reports on software verification include [9] (on using automated theorem provers) and [10] (on tool-supported inspections).

7 Conclusion

We performed a field study in the automotive domain to compare traditional testing with model-checking. We implemented an industrial specification in two different model-based development tools. We performed quality assurance using simulation in both tools, testing in ASCET, and model-checking in AutoFOCUS. We evaluated the number and type of errors that were found and compared the effort and the effectiveness of the quality assurance methods. An interesting result is the effort distribution within each of the two methods: while in the case of testing the evaluation of the test results required the largest part of the effort needed, in the case of model-checking the formulation of properties to be checked turned out to be most time-consuming. Note that it was not our goal to compare ASCET and AutoFOCUS regarding the relative effort they require compared to each other (in particular, the two modelling phases were performed subsequently which means that the second modelling task using AutoFOCUS may have been facilitated by insights gained during the first using ASCET). The presented comparison

Testing	**Modelchecking**
Examines a physical or concrete system	Examines an abstract model
In-the-loop-tests take place in an environment near to the real one	Cheap and early verification (without setting up complex in-the-loop-test environments)
No proof of correctness of properties possible	Proof of correctness of properties possible
Uses often many, superficial test cases	Uses selected user specific properties

Table 4. Conventional testing vs. model-checking.

does not aim to be complete or statistically significant as we did not perform a comprehensive empirical study with a high number of experiment participants. The result is thus mostly a qualitative assessment of the effort and results of traditional and formal verification. Our outcome was that model-checking needs some more time effort compared with testing, but was worth the effort because some types of errors could rather be found by model-checking than by testing, especially errors that developers may not look out for specifically during testing (such as synchronization error or boundary conditions). A combination of both methods might be optimal because both of them have their strengths in finding different types of errors.

Table 4 compares general aspects of the examined quality assurance methods. We could show that there exists no overall winner and that each verification methods has its own specialties: One deficit of the model-checking method is the problem of the state space explosion. However, model-checkers are still subject to ongoing improvements, and at least in the case of event-controlled systems at a higher abstraction level, they are already applicable. Regarding automotive applications, the use of formal verification techniques should not mean the absence of conventional test procedures. Rather, formal verification provides complementary quality assurance techniques which could lead to synergy effects if applied along with conventional testing methods. For example, a base model for successive development steps and test procedures could be created by providing verified models for generating test cases for Back-to-Back tests. Note that "complementary" here does not necessarily mean that certain errors can be found *only* with one of the two techniques (except of course errors that cannot be found using model-checking because that works on a higher level of abstraction). Instead, it means that to detect large classes of simple errors, testing is more time-effective, while model-checking is more effective to find specific classes of particularly intricate errors.

Although some of the results that were found during the case-study may confirm existing intuitions (e.g. that model-checking may find additional bugs but takes more time to learn than testing), we are not aware of any other literature comparing model-checking and model-based testing, in particular not regarding embedded or automotive software. Although we target our discussion e.g. in Sect. 5 mostly to the automotive domain, we believe that it generalizes to other embedded domains that are both safety-critical (which justifies the use of sophisticated assurance techniques such as model-checking) and at the same time are characterized by large product numbers (forcing hardware cost reduction and therefore optimized code) as well as a relatively short development cycle (which presents time pressure in the development).

Lessons learned regarding the experimental setup include that it may have been beneficial to have the two QA techniques applied by two different users in parallel,

rather than by the same sequentially. Disadvantages would however include that this will be comparable only if the two users have the same expertise, and that this would significantly increase the effort needed for performing the experiment. Since the person performing the study did not have any prior knowledge in the two tools that were used, we however believe the experiment to be repeatable by others.

Details of experimental results omitted for space reasons can be found at [11].

Acknowledgements: Constructive feedback from the anonymous referees that led to a significant improvement in the presentation of the paper is gratefully acknowledged.

References

1. M. G. Bartley, D. Galpin, and T. Blackmore. A Comparison of Three Verification Techniques. In *DAC*, pages 819–823. ACM, 2002.
2. T. Bienmüller, J. Bohn, H. Brinkmann, U. Brockmeyer, W. Damm, H. Hungar, and P. Jansen. Verification of Automotive Control Units. In *Correct System Design*, pages 319–341. 1999.
3. J. S. Bradbury, J. R. Cordy, and J. Dingel. An empirical framework for comparing effectiveness of testing and property-based formal analysis. In *PASTE*, pages 2–5, 2005.
4. G. Brat, D. Drusinsky, D. Giannakopoulou, et al. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.
5. M. Broy. Challenges in automotive software engineering. In *ICSE*, pages 33–42. ACM, 2006.
6. M. Broy and K. Stolen. *Specification and Development of Interactive Systems*. Springer, 2001.
7. B. Cheng, F. Houdek, and S. Kawana, editors. *Workshop on Automotive Requirements Engineering (AuRE)*. IEEE, 2006.
8. B. H. C. Cheng, R. Stephenson, and B. Berenbach. Lessons learned from automated analysis of industrial UML class models (an experience report). In *MoDELS*, pages 324–338, 2005.
9. E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *Int. J. on Artif. Intell. Tools*, 15(1):81–108, 2006.
10. M. Halling, S. Biffl, and P. Grünbacher. An experiment family to investigate the defect detection effect of tool-support for requirements inspection. In *IEEE METRICS*, pages 278–285, 2003.
11. J. Jürjens, D. Reiss, and D. Trachtenherz. Model-based quality assurance of automotive software: Experimental data. http://mcs.open.ac.uk/jj2924/publications/experiments/autoqa, Apr. 2008.
12. T. Kropf. Software bugs seen from an industrial perspective or can formal methods help on automotive software development? In *CAV*, volume 4590 of *LNCS*, page 3. Springer, 2007.
13. S. Mouchawrab, L. C. Briand, and Y. Labiche. Assessing, comparing, and combining statechart- based testing and structural testing: An experiment. In *ESEM*, pages 41–50, 2007.
14. B. Paech and F. Houdek. The door controller unit – an example specification. Technical Report 002.02/D, Fraunhofer IESE, 2002.
15. O. Pilskalns, A. A. Andrews, A. Knight, S. Ghosh, and R. B. France. Testing UML designs. *Information & Software Technology*, 49(8):892–912, 2007.
16. A. Pretschner, M. Broy, I. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *ICSE, Future of Softw. Engin.*, pages 33–42. ACM, 2007.
17. A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE*, pages 392–401. ACM, 2005.

18. A. Pretschner, C. Salzmann, B. Schätz, and T. Stauner. ICSE Workshop on Software Engineering for Automotive Systems. In *ICSE Companion*, page 146. IEEE, 2007.
19. B. Schätz, T. Hain, F. Houdek, W. Prenninger, M. Rappl, J. Romberg, O. Slotosch, M. Strecker, and A. Wißpeintner. CASE Tools for Embedded Systems. Technical Report I0309, TU Munich, 2003.