Qualitätssicherung intelligenter Fahrzeugfunktionen am „virtuellen Fahrzeug"

Quality Management for Intelligent Car Functions using a "Virtual Car"

Christian Basarke, Institut für Software Systems Engineering, Technische Universität Braunschweig, basarke@sse-tubs.de

Christian Berger, Institut für Software Systems Engineering, Technische Universität Braunschweig, berger@sse-tubs.de

Kai Homeier, Institut für Betriebssysteme und Rechnerverbund, Technische Universität Braunschweig, homeier@ibr.cs.tu-bs.de

Prof. Dr. Bernhard Rumpe, Institut für Software Systems Engineering, Technische Universität Braunschweig, http://www.sse-tubs.de

1 Introduction

The number of concurrently interacting software functions in modern cars is strongly increasing. Fortunately, we don't have one ECU per software function anymore, but still there are too many ECUs in each car. Today ECUs and their software functions serve a variety of aspects. They control and optimize the normal operation of various actors like the motor, wait for local device errors up to car crashes to handle like warning lights and air bag, assist the driver during normal operation, like the adaptive cruise control, or improve the driver's comfort. Intelligent driver assistance functions exhibit a strong complexity and make it increasingly important to master these complexities and ensure the quality of the result. Many innovations in a modern car strongly rely on software. The more complex the task to be solved by software the more difficult it is to develop, test and evaluate those intelligent functions.

Furthermore the aspect of sustainability is hardly addressed in common projects developing software modules. Similar software functions are redeveloped or difficult to re-use because of the lack of proper generalization of its features, modularity and documentation. Software architecture needs to be planned in a similar manner like any other complex artefact. Today, the return-of-investment could be improved because the knowledge gained is either badly or not at all documented, the solution only fits one specific configuration of sensors etc. and cannot be generalized and pieces cannot be reused individually. Synergies between multiple projects or from virtual project groups consisting of multiple companies are small though really important since they mean a real competitive advantage. Software component management is as important as ordinary component management in a product line based development department.

In this paper we propose a method for the development of intelligent car functions by means of an example function assisting the driver to pass an intersection. Therefore, we introduce some aspects of the problem space and then we deduce some ideas for the algorithm in the next chapter. After defining the requirements and design of our algorithm we explicate and discuss our concept of testing algorithms like this one, ensuring the necessary level of quality and evaluating its performance in a modern software development process. Finally, we discuss our approach with related work in the last chapters.

The autonomously driving car is called Caroline [5]. It is developed in a joint effort with the Institute of Operating Systems and Computer Networks, the Institute of Flight Guidance, the Institute of Control Engineering, the Institute of Computer Graphics and the Institute of Software Systems Engineering, each addressing one aspect of the whole challenge covering the computer infrastructure in- and outside of Caroline, the sensor system and car control, the camera based computer vision and finally the artificial intelligence. Caroline will participate in the DARPA Urban Challenge [7]. The CarOLO project [5] that developed Caroline is not only an interesting project on its own, but also an application of the testing framework developed at the TU Braunschweig.

2 Intelligent driving functions: an intersection assistant example

A car has to obey certain laws when moving around in the real world. In general a car can move backward and forward and turn the steering wheel. So we have two degrees of freedom plus a choice for speed. The allowed, respectively possible behaviour is based on physical as well as juristic laws. For those a car needs to know about its context using GPS, maps and a variety of sensors. Our high-level intelligent driving system is based on the 'DAMN' architecture [11], which is stateless and such allows describing its behaviour more easily. Turning the steering wheel results in different circle-radii that the car can use to move. Instead of the radii, it is conceptually easier to work on the inverse, a *curvature*. A curvature of 0 represents driving straight forward, while negative curvatures result in left und positive curvatures in right turns.
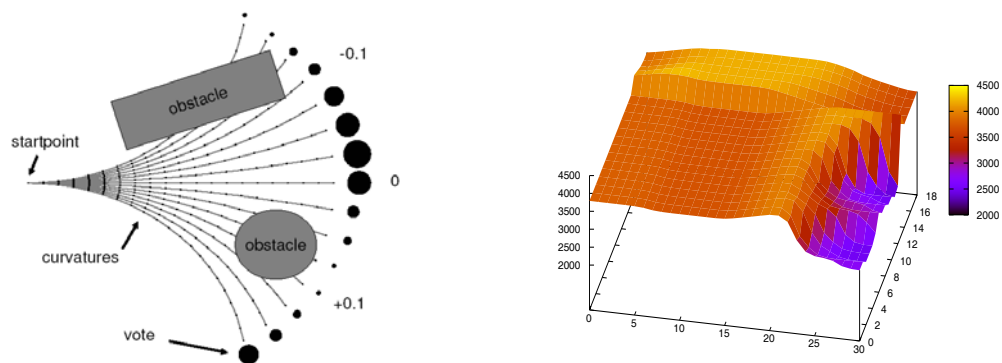
Figure 1: Left: A single curvature field, where bigger black circles represent better votes. Right: Multiple curvature fields plotted as a 3D-surface.

This curvature is the most important element to control and is selected by an *arbiter*. This architecture evaluates each input information, e.g. about obstacles and driving lanes allow individually defined behaviour functions the possibility to vote for possible and optimal curvatures. This modular architecture allows adding more kinds of behaviour to the system, which makes it very modular and extendable. Here's a list of kinds of behaviour which we currently need and which has been developed for our autonomously driving car Caroline [8], [9]:

- Follow waypoints: This behaviour simply drives the car from point to point as found in a Road-Network-Description file, containing GPS-Waypoints.

- Stay in lane: This behaviour votes for a curvature that keeps the car in the lanes, detected by cameras available.

- Avoid obstacles: This behaviour votes for curvatures, that keep us as far away from obstacles as possible and forbids curvatures leading us directly into them.

- Stay in driveable area detected by texture matching.
- Avoid holes and small obstacles detected by laser scanners.

All votes are collected and weighted resulting in an overall vote that finally selects behaviour and curvature to take. The weights shown in **Fehler! Verweisquelle konnte nicht gefunden werden.** are not fixed; they depend on factors like distance to an intersection, presence of lanes, etc.

A second arbiter controls the speed. Each of the implemented kinds of behaviour votes for a certain speed, which can accommodate comfort, sport driving or winning a challenge. According to the arbiter's strategy it selects a safe value for the speed to take. A driveable corridor is then iteratively found through a list of curvatures. These are further processed by the path planner.

Because we have to deal with more complex situations than the DAMN-architecture is designed for, our software detects situations where a special treatment is necessary. For example if we reach a stop line or a position in which a u-turn is required, we set an interrupt point, at which the arbiter gives the control to a special mechanism temporarily surpassing the DAMN-control flows to handle that situation.

After the car has come to a complete stop at the stop line recognized by the cameras, we wait until our *intersection observer* gives its okay to move on. This observer initially knows where intersections are by reading the navigation map. Knowledge about the concrete layout and traffic situation at the intersection is continuously updated when it gets in the focus of the cameras. When reaching the stop line the intersection observer uses fused data from various sensors to decide to give way to another car or to proceed. When the intersection is clear, the control is given back to the arbiters to continue in normal driving mode.

Besides the development of the correct software modules for arbiters, voters and kinds of behaviour, it is a hard task to find optimal weights to choose the right situations. These weights are not formally part of the software, but of its parameterisation. And there are many parameters to calibrate in sensors, actuators as well as the intelligent autonomic driver. Finding the optimal parameterisation thus becomes a complex problem on its own. To allow for efficient finding and optimization of those parameters as well as for testing the correctness of the development software functions, we have decided to setup a "virtual car". A virtual car does mimic the sensors and actuators of the real car, but only consists of a subset of the ECU allowing to efficiently and autonomously running automated tests. To gather data for those tests we designed a training-mode in which a human driver steers the car and collects all data that may influent the AI's decisions. We then pre-process the data to find a combination of votes, which fits best to the human way of driving.

3 Automatic system tests for intelligent software functions

As indicated in the last chapter modern software functions used in driving assistant components consist of multiple modules. A central module however is the algorithm that implements the intelligence, where all previously collected and aggregated information is used to decide which actions to be taken. As discussed in the previous section, this algorithm consists of highly parameterized strategies for suggestion, voting and selection of the appropriate actions to take. The main questions for quality assurance and performance measurement are therefore to define the appropriate goals and their underlying metrics for the software quality in the context of automotive software engineering. With this in mind, the definition itself is an abstract goal for implementing and testing the software that makes it necessary to explicitly define functional and non-functional requirements for the software. In this chapter we concentrate on quality metrics for functional parts of software to show an appropriate infrastructure to assess and understand quality of software.

The main difficulties in today's software engineering process for embedded software modules especially in the research and development in the automotive industries are often the late phase of integration [14]. Today, in the automotive industry software components are still developed separately and are combined late. The difficulties are for example incompatible interfaces between complex software components, misunderstandings in earlier defined requirements or many smaller problems that potentially delay a project.

In Chapter 2 we discussed the infrastructure of an intelligent software function for assisting the driver passing an intersection. For testing the algorithm to find bugs or to evaluate its performance one normally needs at least two cars: The test car itself and another car for creating a realistic intersection situation. But even that simple example leads to many problems. Firstly, one needs employees to drive the car, observe the test and safety and finally to record the test results for later evaluations. Secondly, tested situations are usually not reproducible as it is hard to reproduce the exact situation with the same timing. Furthermore, extensive testing of various situations is extremely tedious. This inspired us to create a virtual car running on a virtual test course. This exhibits major advantages. A first one is the simplicity in execution: One only needs the software to be tested (herein denoted by *"sut": software under test*) and the virtual test environment similar to well understood unit test frameworks [3], [12], [13].

But the difference to unit test environment is the kind of system under test. In our unit tests we deal with low level methods or functions. For validating its correctness a comprehensible number of representative tests cover the different program execution paths. This approach of course is very helpful for simple and modular functions and for more complex, but deterministic and controllable system tests.

An intelligent software function has a complex decision part based on various information sources like sensor data or GPS, where data is not necessarily confident and adaptive weight functions that assist votes. In such an environment decisions are often not clearly to determine. So instead of deterministic tests, we define forms of "quality attributes" for the behaviour of the intelligent function under test. This includes e.g. that an intelligent function does not provoke a "virtual accident", drives the car such that it is comfortable for the driver, does not suddenly stop, etc. Therefore, we developed a framework that on the one hand can interac-

tively train the intelligent algorithm with the help of the developers. On the other hand, it can be parameterised in a very flexible way, such that it can precisely and fully automatically repeat a given situation to understand whether the function behaviour is still working (regression test) or has even improved (optimization). In our case, we can train and repeat specific intersection situations as often as we want, without human resources or even the real car involved.
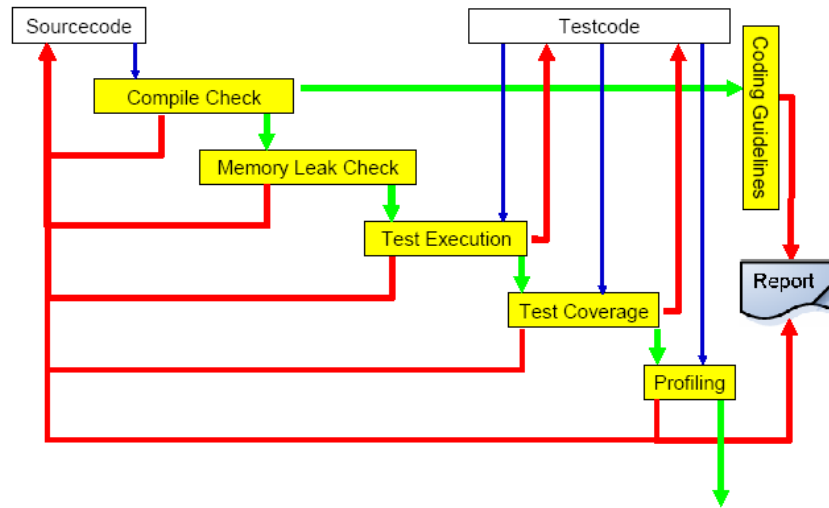

Figure 2: Multi-level test process.

To handle all levels of tests, we have designed a multi-level test process shown in **Fehler! Verweisquelle konnte nicht gefunden werden.** that can be executed by every developer or by an automatic server system. The input data for this process are the source- and the test code as well as all input to replay virtual car tests at this level. In the first step the source code is compiled to check for syntactic errors. Then, the code is inspected for potential memory leaks that lead to unforeseen errors during execution. Therefore, the source code is compiled and executed using the test code. The result of this step is among others a report about the misuse of memory and the well known unit test results. We also take these results to measure the quality of the test code by deducing the test coverage rate for the source code. In a next step, the algorithms are evaluated for finding time consuming parts. Additionally, the source code is analysed for compliance with previously defined coding guidelines. All these information is collected and aggregated into a human readable report, which is made available to all developers and the project management.

In a next stage, we run the much more complex automatic system level test framework for the "virtual car tests". This framework is based on clearly defined interfaces between the software modules and an abstract simulation of the hardware components of a car. It is somewhat tricky to provide these simulations, as the component vendors do neither provide these abstractions on their own, nor do they tell enough about the expected behaviour (physically, sensory, etc.) This leads to the necessity that the hardware components need partially to be modelled on their own. Further approaches are to just model the output e.g. of radar or camera sensors through a generator or to replay pre-recorded data. A sophisticated version of the generators would even

create artefacts to check the quality of fusion and consolidating algorithms. However, for checking the intelligent software function it is also sufficient to start testing the fused and extracted information after the sensor handlers, which is far more efficient and easier to accomplish. The resulting framework can be seen as a multi-functional simulator. Its overall architecture is shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**.
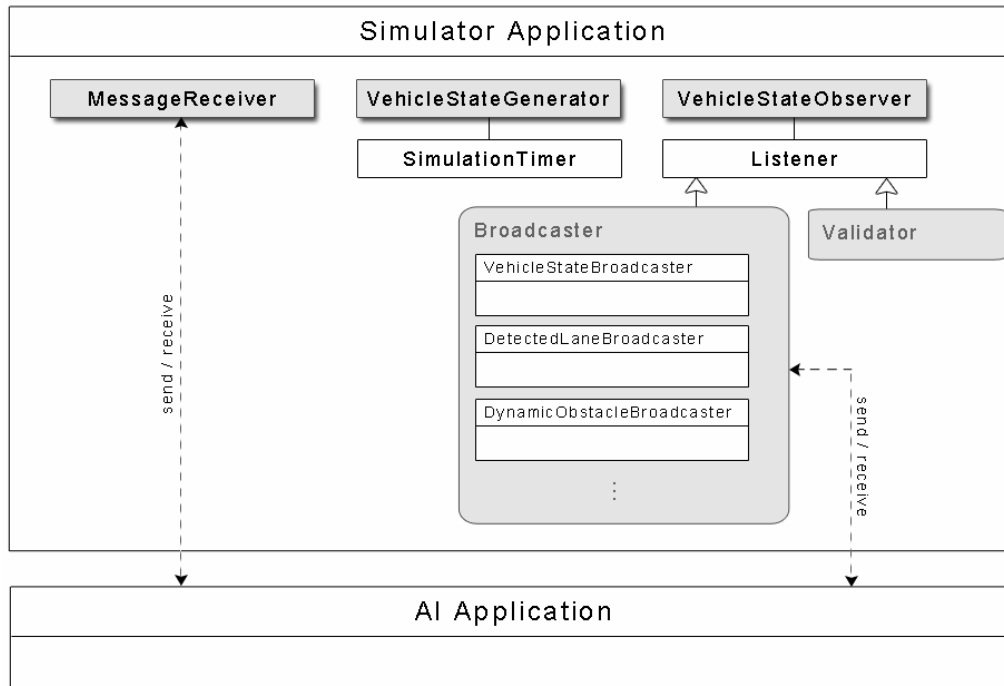
Figure 3: System architecture of the simulator.

To supply the data for the modules under test, the simulator application emulates necessary parts of the real world such as sensor data fusion or image processing algorithms. The simulator consists of three central components, a *message receiver*, a *vehicle state generator* and a *vehicle state observer*. The message receiver picks up messages created by the AI application. Data such as the planned trajectory and the intended velocity is received and processed by the simulator. This information together with the virtually elapsed time during two simulation steps is used by the vehicle state generator to compute the next vehicle state. The vehicle state is defined by the current position, orientation, yaw rate and a number of other parameters. The new vehicle state is passed to all listener modules via the vehicle state observer, which holds a group of broadcasters and a group of *validators* both derived from the listener interface.

Each broadcaster is responsible for emulating certain aspects of a simulated scenario. Depending on the current vehicle state, broadcaster specific data is generated and sent to the AI application. For example one broadcaster supplies the AI with detected lanes or intersections and another one generates dynamic obstacles. In order to be able to check the behaviour of the artificial intelligence for their interaction with dynamic obstacles the dynamic obstacle broadcaster uses individual route and mission files for each obstacle involved. Arbitrary scenarios at

intersections or even more complex traffic scenarios can be constructed and replayed by this architecture.

A timing module is used to vary the execution speed of the simulator, making it possible to run the simulation slower or faster than real time. This allows the simulated car functions to live in a virtual time as well. To determine the travelled distance and acceleration of our vehicle and of the dynamic obstacles the simulated virtual time is used. Other than the real time, the virtual time does not progress continuously, but may jump. This allows for a variable execution speed. It considerably speeds up testing, as time a test needs to run is considerably lower than the time it needs to drive a car through test scenarios. This is a main enabler for improved testing strategies.

Currently our simulator is designed to supply idealized information. Its intention is not (as yet) to exactly emulate a real environment, but merely to provide the components under test with the data necessary to precisely run predefined scenarios. It is possible to construct various test scenarios by specifying particular sets of configuration files. Thus manual and automated system tests can be conducted and easily repeated. This is also the basis for automated regression tests.

To manually check and observe the behaviour of the developed algorithms in the intersection scenario discussed above, the simulator and the AI application can be launched and configured by hand. Previously created test scenarios can be evaluated directly. In addition, the simulator architecture makes it possible for the developer to interactively control a dynamic obstacle (e.g. a moving car). Thus it is easy to take appropriate action in order to simulate various constellations.
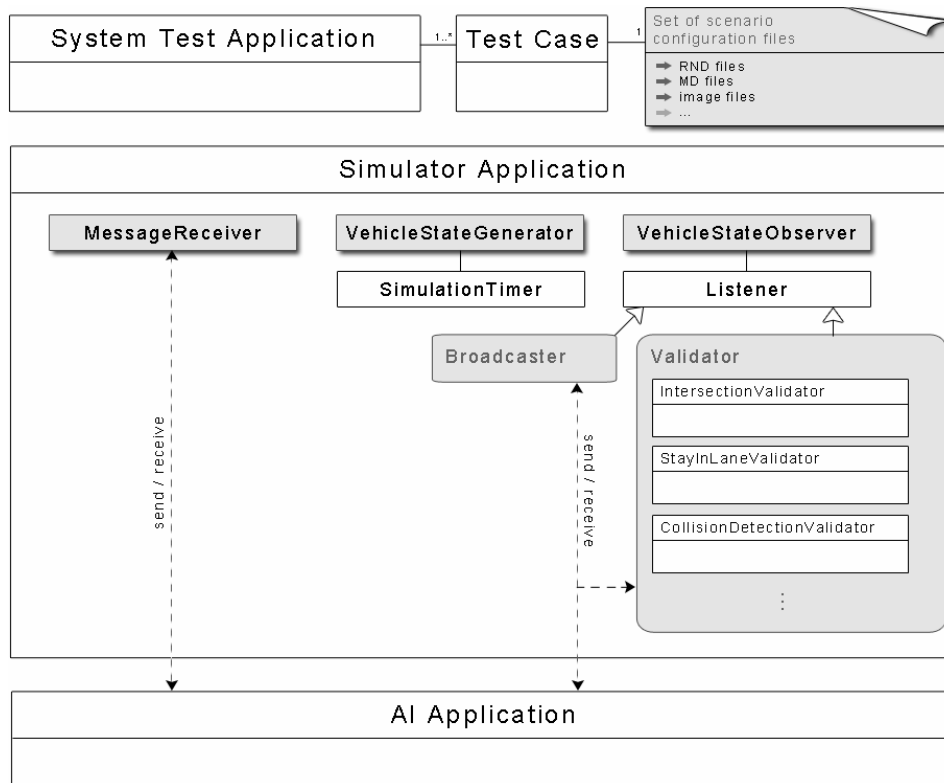
Figure 4: Architecture for automated system tests.

Based on the publicly available unit test framework CxxTest [6] for C and C++ applications, test cases are described and executed. This is done within a test application that is compiled separately and that starts the relevant and distributed functions such as the simulator and the intelligent driving functions. **Fehler! Verweisquelle konnte nicht gefunden werden.** shows the software architecture for automated system tests. Every test case can use an individual setting of configuration files. Appropriate route data is specified by these settings and exactly defines a test scenario. To be able to compare the expected with the actual behaviour, additional listeners called *validators* are registered in the simulator application. These validators receive the current vehicle state and check it for compliance with previously defined reference behaviour such as staying in a predefined lane or avoiding collisions with obstacles.
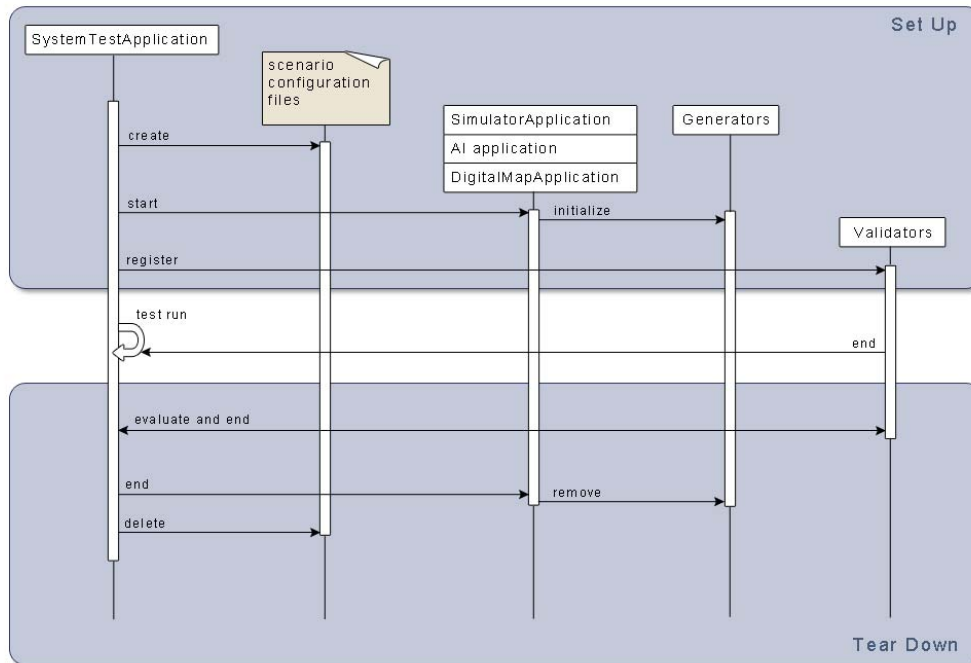
Figure 5 Sequence chart for automated system tests.

Fehler! Verweisquelle konnte nicht gefunden werden. shows an overview how a system test is conducted. During the setup phase as a first step configuration files are created and necessary applications are started. Corresponding to the scenario the simulator initializes required generators and the test application registers validators which will be utilized later. The scheduled test runs until at least one validator reports an erroneous behaviour or the whole test run is evaluated as being successful. In the following tear down phase the monitored behaviour is summarized in a test report. Finally running applications are stopped and previously created configuration files are deleted to be able to rerun this or another test without any behaviour chance and thus have deterministic tests with repeatable results.

4 Conclusion

Our automatic system test environment simplifies and elaborates the test process by primarily allowing us to use the same programming language for development and testing. A sophisticated framework and system of configuration files allows for an easy and efficient definition of test cases. These test cases can be executed automatically, e.g. over night, allowing the developers as well as the management to understand and control the progress of the project and quality of the results. Special techniques, like mock-objects [3] substitute parts of the real

system, are used where necessary and simulated time is entirely disconnected from the time really needed for testing. With these techniques, we have been able to build a "virtual car" that for the system under test looks quite similar as if it would run in its real environment inside the car. The *virtual car* is becoming an important element of an efficient development and testing strategy.

But, it is also to be kept in mind that our goal is neither the complete substitution of the real car in every single development step nor the complete substitution of real car tests. We aim for a process that minimizes the test time where developers or testers need to be paid for, but increases the number of test cases and the number of test executions. The overall quality management strategy must include all kinds of tests, consisting of unit tests for single methods, classes up to more complex, but still automated tests for software subsystems, system tests and finally tests in the real. Actually our experience is that we do not get a hierarchy of tests, but two hierarchies: Software modules and their integration need to be tested, but also a software module and the controlled hardware component need to be tested early.

Overall, we lower the test complexity by using our sophisticated test framework. As mentioned one major benefit is the possibility for running unattended und automatic test suites consisting of multiple test cases each describing a special task for the intelligent software function. Every developer and even an overnight quality check system can automatically execute these test suites. They can even be run on every single change that becomes visible to the versioning system. Test results are distributed by an email and via a detailed summary on a web site.

Although in the beginning, setup of such a test environment cost some work. It furthermore adds to the labour of developers to come up with a testable architecture and define tests together with their software functions. Unfortunately, this effort does not pay off immediately, but only after a while. That is why we need well trained software developers, knowing that such a process needs to live from the beginning and cannot be handled later (when the project is already caught in the "testing trap").

There is a variety of work going on to deal with these kinds of testing systems. First, we want to considerably reduce work necessary to setup a testing infrastructure. Second, we need to further elaborate the architecture necessary to decouple software functions from their controlled hardware, class libraries, frameworks, operating systems etc. This work will both be helpful for upcoming AUTOSAR development projects [1] as well as be simplified by using the abstraction and decoupling mechanisms that AUTOSAR is promising to provide. Further work is needed to push the simulators further towards embedded parts of the system such as sensors and actors input and output. This includes that some physical models have to be implemented and statistically spread artefacts in sensor data have to be understood and produced for the simulator.

Finally, we can note that autonomously acting cars are currently a big issue. However, many automotive companies are currently trying to develop one autonomic function after the other. We however believe that it is necessary to decouple the question of what sensors to use from the development of intelligent functions. Only if we are able to do this decoupling, we will be able to independently and quickly come up with general intelligent functions.

References

[1] The AUTOSAR Project Website, http://www.autosar.org

[2] Basarke, C. et al.: DARPA Urban Challenge 2007 – Team CarOLO. Technical Report as part of the competition.

[3] Beck, K.: Test-Driven Development. By Example. Addison-Wesley, 2002.

[4] Binder, R.: Testing Object-Oriented Software. American Programmer, 7(4), pp. 22-28, April 1994.

[5] The CarOLO-Project Website, http://carolo.tu-bs.de/

[6] CxxTest Website, http://cxxtest.sourceforge.net/

[7] The DARPA Urban Challenge Website, http://www.darpa.mil/grandchallenge/

[8] Form, T. et al.: *Caroline - Ein autonom fahrendes Fahrzeug im Stadtverkehr.* AAET 2007, Automatisierungs-, Assistenssysteme und eingebettete Systeme für Tran, 2007, Braunschweig.

[9] Form, T., Effertz, J. and Wille, J.: Urban Challenge 2007 – Autonomes Fahren in dynamischen Umgebungen. In: IQ-Journal VDI/VDE Bezirksvereine Braunschweig e.V., Februar 2007, Braunschweig

[10] M. Pellkofer, Verhaltensentscheidung für autonome Fahrzeuge mit Blickrichtungssteuerung (behavior decision for autonomous vehicles with gaze control), Ph.D. dissertation, Institut für Systemdynamik und Flugmechanik, Universität der Bundeswehr München, January 2003.

[11] Rosenblatt, J., Damn: A distributed architecture for mobile navigation, Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, January 1997

[12] Rumpe, B.: Agile Test-based Modeling. In: Proceedings of the 2006 International Conference on Software Engineering Research & Practice. SERP'2006. CSREA Press, USA, June 2006.

[13] Rumpe, B. und Krahn, H. und Berger, C.: Softwaretechnische Absicherung intelligenter Systeme im Fahrzeug. In: Proceedings der 21. VDI/VW-Gemeinschaftstagung - Integrierte Sicherheit und Fahrerassistenzsysteme. 12.-13. Oktober 2006, Wolfsburg.

[14] Schäuffele, J. und Zurawka, T.: Automotive Software Engineering. Vieweg-Verlag, Wiesbaden, 2003.