On the Integration of Design and Test — A Model Based Approach for Embedded Systems*

Christian Pfaller,† Andreas Fleischmann, Judith Hartmann,
Martin Rappl, Sabine Rittmann, Doris Wild
Technische Universität München
Institut für Informatik, Chair IV: Software & Systems Engineering
Boltzmannstr. 3, 85748 Garching, Germany
{pfaller,fleischa,hartmanj,rappl,rittmann,wildd}@in.tum.de

ABSTRACT

One of the most crucial questions concerned with model-based testing is how to find "interesting" test cases. We consider test cases to be interesting if firstly, they cover the user requirements and secondly, they have a high probability to find potential errors. In this paper we introduce an approach to derive test cases along different levels of abstraction during the design phase. These levels start with services representing user requirements on the topmost level and result in models for a specific technical platform on the most concrete level. Within the presented test process we use design models of different abstraction levels as test models out of which test cases can be generated. The test cases are executed on more concrete levels and finally on the implementation. An exception is the (topmost) service level which is used for the derivation of the test case specification.

One main advantage of our approach lies in preserving the link from test cases to corresponding user requirements. Furthermore the danger of using too abstract models which do not reflect inevitable crucial aspects of the realization is avoided. Finally this yields to a front loading of quality control activities to a point as early as possible in the development process. In our work we focus on embedded reactive systems especially in the field of automotive software. Our current research targets at new kinds of test coverage criteria which reflect the systems requirements rather than structural aspects of models.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools, tracing*; C.3 [**Special-Purpose and Application-based Systems**]: *Real-time and embedded systems*

General Terms

Design, Reliability, Verification

Keywords

Abstraction levels, automotive software, model based testing, requirements coverage, services, test case specification.

*Partly supported by the Bavarian Government within the project *mobilSoft*, grant number IuK 188/001

†Project within *INI.TUM* - Ingolstadt Institutes of TU München

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AST '06 Shanghai, China
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

More than 50 % of the development costs of embedded systems are caused by testing and error correction only [11, 13]. Often this is done using copious tests in the late phases of development. Sometimes the selection of test cases is even quite arguable. Thus, testing could be more effective if two prerequisites are achieved: First, do tests as early as possible and second, focus on the interesting tests.

Considering several kinds of requirements, as for example illustrated in [2], two types of requirements must be considered in depth when it comes to testing the system under development: First —of course— the *user requirements* which define what behavior the user (more general: the system environment) expects from the system and which properties the system has to fulfill. Second, *constraints and restrictions relating to the (technical) realization* of a system are important as well, since they often reveal error-prone aspects of the system.

By applying a development process which uses design models on different levels of abstraction —more abstract ones which reflect the user requirements and more concrete ones considering more detailed aspects of the technical realization— these two dimensions (focusing on user requirements *and* technical realization) can be covered better by automatically generated model-based tests.

As for example in [5], the usefulness of classic structural coverage criteria for automated test case generation is often doubted. Instead, the demand for more meaningful coverage criteria which may serve as test case specification is demanded. We suggest to use services representing user requirements as test case specification. This finally leads to requirements based coverage criteria.

The main contribution of the paper is the integration of a model-based design process using levels of abstraction with

a test process. The advantage is that the models of the design process can be used for the testing activities as well. Therefore, the effort necessary to construct design models pays off not only for a better design process, but also for an improved testing process.

For testing, the models of the different abstraction levels are used in different ways: Services on the service level are used as *test case specification*; models on the further levels are used as *test models* from which test cases are derived by using the (service based) test case specification. The generated test cases may be applied to the models of the different levels beneath or of the implementation. In the latter case the models or the implementation, are in the role of the *test object*.

By using different levels of abstraction the models also cover realization constraints which are valuable for identifying error-prone parts in the system under test (or its models, resp.). When considering for example embedded systems in the automotive domain it is common that a system is not implemented just on one single execution platform, but that the function is realized by distributing its parts on several control units which communicate via some bus system. Implementing the bus communication of the control units is a crucial and therefore error-prone design task. Abstract models of the user-visible behavior leave out this part; but it is covered on the more concrete levels. Generating test cases from models on different levels therefore not only focuses on requirements coverage but also on error-prone design decisions.

The rest of this paper is organized as follows: Section 2 gives a short introduction to model-based testing as it is considered in this paper. Section 3 states an outline of the proposed design levels for modeling automotive software. How services may be useful as test case specification and thus keep the link to user requirements is described in section 4. The process of test case generation over abstraction layers is introduced in section 5 which includes the refinement of tests on the subsequent levels. Finally we give a short conclusion and show prospects on our future work.

2. MODEL BASED TESTING

Techniques for model-based testing [4, 8] use a model as a specification of the *system under test (SUT)*. Hence the model (given in form of a state machine for example) represents the requirements the SUT must fulfill. A *test case* relating to such a model is a pair of an input sequence and the expected output sequence to these inputs. The consideration of input and output sequences (or histories) is important since the output of the system depends in general not only on the current input but on the whole history of inputs. Embedded (or reactive) systems, as they can be found for example in the automotive domain, process a potentially infinite sequence of inputs. Therefore, the set of all possible input sequences is infinite.

The aim of a test case generation out of the model is to extract a set of "interesting"' test cases from the potentially infinite set of test cases. To determine this subset of interesting test cases a *test case specification* is used. Together with the test case specification auspicious paths in the model are looked for and the according input and output histories are determined. Thus, a test case specification serves as a search strategy which limits the possible infinite search space to find the interesting test cases. The search strategy also defines *what* makes a test case "interesting". Unfortunately it is hard to state a clear definition when a test case is considered to be "interesting". The most desired definition here is that a test case should have a high likelihood to find potential errors. Since this is a far too general and vague formulation an application to test case generation is not practicable [6]. We will focus on test case specifications in more detail in section 4.2 where we show how services are used as test case specification. The generated test cases are applied to a *test object*; this can be the SUT but also a (different) model of the SUT.

As mentioned before, a test case here means not just the inputs for the SUT but also the expected outputs which is compared to the actual outputs of the SUT during the test execution. For a reasonable generation of expected outputs it is inevitable that the model is more abstract than the SUT. This means that further design decisions must be made to build the SUT. Models which are used for the automated generation of the SUT or its code can therefore not be used for test generation. If the SUT can be generated automatically from the model, test cases derived from such a model would only be able to find errors which the code generator made during code generation. However, there would be no test case which is able to find design errors in the SUT. By using more abstract models we only get abstract test cases which must be transformed to concrete tests cases when applied to the SUT. This task is done by *test drivers*. We do not consider test drivers in this paper.

In the next section the system of abstraction levels which is the basis for our approach is explained.

3. ABSTRACTION LEVELS

Our current work is based on the necessities for software engineering of automotive systems. Automotive software — especially in the field of driver assistance and comfort systems— increasingly becomes complex. Software is usually deployed on control units with a strong limitation of hardware resources. Additionally, a function often is distributed over several different control units which communicate via different bus systems. Sophisticated functionality is made up by the interplay of interconnected functions. Additionally, we face enhanced reliability needs. Therefore, to cope with this intricacy it is necessary to divide the design process into different steps which enable a stepwise modeling from an abstract point of view on the system to a concrete one.

3.1 Abstraction Levels Overview

For a structured design approach a system of four abstraction levels is presented in [12][1]. This order of abstraction is shown in figure 1. Each level abstracts from specific aspects of the system under consideration. The lowest level —the platform level— is the most concrete one: it just abstracts from the actual program code by describing the single tasks of the systems. Clusters on the next higher level —the logical cluster level— additionally abstract from the technical runtime environment. Furthermore functions on the functional level abstract from questions concerning distribution. Finally services on the service level loosen —in general— the demand of completeness of the system (in regard of possible inputs) and just focus on its characteristic behavior. The

[1] [12] focuses on automotive software but might be applicable to similar kinds of embedded systems as well

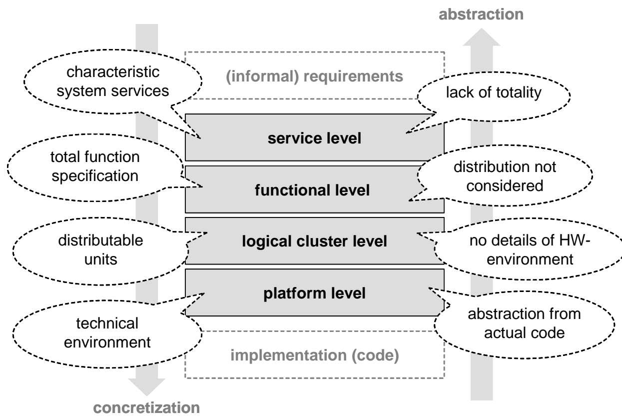

Figure 1: Abstraction levels

following subsections give a short overview over the focus of each abstraction level and its design decisions.

3.2 Service level

The aim of the service level is the consolidation and precise specification of those requirements (user requirements) that describe the characteristic system behavior. In order to achieve this the user requirements have to be formalized in terms of services. The design decisions on this level consist of both the specification of the system boundaries and the specification of the inputs and outputs (=actions). Both are described in a quite abstract way - that is to say: as they are interpreted by the users. The behavior therefore is specified by a black box view. The complete definition of the overall system behavior is not aim of this level. The system behavior is only partially defined by single services, i.e. the behavior is not defined for each possible input sequence.

In the presented testing method we will use services on this level for the *test case specification* when generating test cases (see section 4) later. We do not use services to directly get test cases from them.

3.3 Functional level

So far, the focus lied on a consistent, but not necessarily total description of the characteristic behavior. When transforming the services to the functional level, the single services are integrated to the overall functions of the system. At this point undesired interactions are solved as well. On the functional level, the behavior is totalized (defined completely) and made deterministic (i.e. the system provides a well defined predictable output for each possible input sequence). On this level the assumption holds that the complete system does not have to be distributed over different runtime environments. It is ideally assumed that there exists a monolithic runtime model.

In our testing approach the models on this level are used as input models for test case generation from which tests are derived (see sections 5 and 5.2.1 later).

3.4 Logical cluster level

Software of embedded automotive systems typically runs in asynchronous tasks and typically is distributed over different runtime platforms. On the functional level the assumption still holds that the system runs on a single synchronous runtime platform. On the logical cluster level we now switch to an asynchronous runtime model. The aim of the logical cluster platform is to partition the system into distributable units (=cluster). This has to be done in an adequate manner so that these distributable units can be a usable basis for the physical distribution. Major design decisions that have to be made on this level concern questions at what points the catenation between functions has to be disconnected. Additionally, the determination of the timing which defines when the units are called is of interest.

Like the models on the functional level, the models on this level are also used as input models for test case generators to retrieve test cases which address the specific design decisions made on the cluster level (see section 5.2.2).

Figure 2: Black Box View on Service Level, the *WindowControl* example

3.5 Platform level

On the previous levels, the system is developed widely independently from hardware issues. This is motivated by reuse issues. On the platform level the hardware independence is abandoned and the system is adapted to specific hardware. Design decisions that are typically made on this level comprise the mapping of clusters to controllers or tasks and their scheduling. The concrete technical data types, the signals, and bus messages are defined.

Moreover, the platform level models serve as input models to generate test cases to cover constraints and restrictions given by the specific platform (see also section 5.2.3).

4. TRACING USER REQUIREMENTS BY TEST CASE SPECIFICATION

In our approach services handled on the service level (see section 3.2) serve as test case specifications. In the following we show why we use services to derive the test case specification. Therefore we first explain the role of test case specifications in our testing method in more detail.

4.1 Example: WindowControler

To illustrate our use of services as test case specification we introduce a simple example from a window control system in cars. For the system given in figure 2, a *switch*, a *sensor*, and the *motor* are the "users" in the environment. The given *WindowControler* must fulfill the following user requirements. Actions are indicated in **bold** text. For simplification we consider the operations for opening the window only:

(a) After action **motor:start** has occurred
an occurrence of **sensor:isDown**
must lead to **motor:stop**

(b) After occurrence of **switch:inPosDown1**
the window must move down (**motor:startDown**)
until **switch:inPos0**, then **motor:stop**

(c) After occurrence of **switch:inPosDown2**
the window must move down: **motor:startDown**

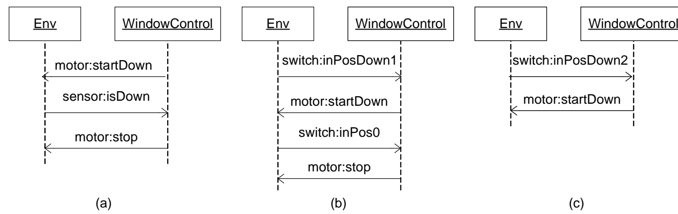

Figure 3: Services for the example requirements of *WindowControl*

The first requirement states that when the window reaches its end position (signalized by the sensor) the motor always must stop. The difference between (b) and (c) is that in (b) the window only moves as long as the switch is in position *Down1*, and when the switch is released (in *Pos0*) the motor should stop. In the case of (c) the window goes down completely after the switch is once set to *Down2*, the motor then stops according to requirement (a).

In figure 3 the services representing these three requirements are given as message sequence charts (MSC). Since we just consider the interactions between the system and its environment, a distinction between the different users is not necessary. Therefore the *switch*, the *sensor*, and the *motor* are accumulated to the environment *Env*.

4.2 Test Case Specification

We follow the definition of a test case specification as it is given in [7] where a test case specification is the formal representation of a test suite. In contrast, the informal description of a property which is to be tested is considered a *test purpose*. Obviously, it is impossible to state precise and formal rules for the transformation of an informal test purpose into a formal test case specification. But it must be possible to derive a test suite (a set of test cases) from a test case specification in a well-defined way. It is important to note that we use the test case specification in order to generate the test cases out of a behavior model which is given as a state machine (see section 2). There the test case specification is used to limit the possible infinite search space for traces in the model.

Many approaches on test case generation use structural coverage criteria like state coverage, condition coverage, or MC/DC coverage [6, 9, 10] as test case specification. There the aim of the test case generation is to find a test suite which fulfills the selected coverage criteria. Another kind of test case specification is a stochastic specification, where test cases are generated at random. Since the lack of expressiveness of stochastic specification is obvious and also the significance of structural test case specification must be doubted [5], there is a demand for functional test case specifications which refer to the users requirements on the SUT.

In our case the test purposes are the user requirements. Clearly, there is no well-defined way to derive a formal test case specification from informally given user requirements. But in our design approach based on abstraction levels requirements are formalized as services. These services are useful as test case specification; it is possible to define an algorithm for deriving a test suite from a model of these services. In the presented way of test case derivation one advantage lies in the fact, that the trace between test cases and user requirements is preserved.

4.3 Using Services as Test Case Specifications

On the service level modeling is restricted to a black box view of the system. Only the interaction between the system and its users and environment is described. On this level input and output actions of the system are identified. Now we explain in more detail why services are useful for test case specification. (In more detail a service defines the test case specification not completely; we focus this point at the end of this section and in section 5.1).

In the example above we have shown how services reflect the user requirements. Trivially a simple execution of one of the MSCs in figure 3 may also be performed on the system as test case. Hence, message sequence charts are used to describe test cases for a long time, as for example in [3] as "test purpose" where MSCs even comprise a larger set of test cases which can be directly expanded from an MSC. Since we generate the test cases mainly from additional behavioral models of the SUT, we use services in a different way: The services (test case specification) serve as search strategy within the model of the SUT, but the content of a test case is determined in detail by the system model. On the service level (see section 3.2) we consider a set of single, independent services which describe the *characteristic* system behavior; e. g. services state exemplary action sequences. The integration of the single services to an integrated model is done when moving from the service level to the functional level. The integration of services requires the solution of undesired interaction.

The following observations are important when services are concerned as means for test case generation:

1. Each single service represents one specific requirement.
2. In general, a service represents an infinite set of actual input sequences (and matching output sequences) but only comprises a subset of all possible input sequences of the SUT.
3. The system behavior, as defined by one service, may be "violated" by actual correct system runs: In general, a single service does not consider interaction with other services. In case of desired feature interaction, the correct overall system behavior is not described by the single service.
4. The set of all services and the integration of services may not define the complete system behavior in the sense of totality: I.e. it is not possible to determine for every input sequence the expected outputs. (The reason lies in the fact the we only focus the so called *characteristic* system behavior on service level).

From the first statement follows: using services independently from each other would lead to tests for each single

Figure 4: Use of test cases along abstraction levels

user requirement, but there would not be any guarantee that the combination of user requirements is reflected in the tests.

The second observation shows that on the one hand a service is a useful means to define a test case specification (because the set of relevant system traces is restricted), but on the other hand a service alone does not fulfill the requirements for test case specification as given in [7]. Hence it still comprises an infinite set of system traces which may be possible test cases.

Considering statement three, it is obvious that a service is in general not useful when directly used as test case. Due to the effects of desired feature interaction, a service used as test case may detect errors in a system run even if the run is correct with regard to the integration of all services.

The fact shown in the last point above implies that there are different kinds of dependencies in the complete (total) system model: Dependences which are contained in the set of user requirements (those are considered when integrating the single services) and dependencies which are necessary to give a full system specification—remember: Not until the model on the functional level a complete model in sense of a total function is required in the presented modeling approach (see section 3.3).

These considerations show that a service is valuable for test case specification: It restricts the set of system traces and it is linked to the actual user requirements. Since test cases are generated from an integrated model the interplay between requirements is taken into consideration as well. As a service may also specify a large or even infinite set of execution traces of a system, further restriction criteria of the possible amount of test cases are necessary to gain a complete test case specification in the sense of [7]. Therefore structural test case specification criteria may be used —the observations shown above will be retained.

5. TEST CASE DERIVATION ON SUBSEQEUNT LEVELS

In this section we first give an overview how the generation of test cases on the subsequent design levels works in general. More details for each specific level are shown in section 5.2. In figure 4 the modeling approach and the tests of the respective design levels and their application are illustrated.

The test cases of the functional, the logical cluster, and the platform level are always applied to the more concrete levels underneath and finally to the implementation. Of course test cases of an abstract level may not be applied

Figure 5: Example: Test case generation on functional level

to the following level only but can be applied to all of the more concrete levels as well (provided a suitable test driver for transforming the test cases). For this method one must be aware that the transformation from models on one level to the next more concrete level always requires additional design decisions. Here testing is about finding errors, caused by this new design decisions. Since the models of the more concrete levels address technical realization constraints, test cases derived from this model comprise these aspects as well.

The reader may note that we omit questions about the validation of informal requirements to services since these may not be able to automate. Additionally we omit questions about the correct integration of the single services to the complete function since a violation of a specified service in the integration to a function may be a realization of both desired and undesired feature interaction. In the proposed modeling approach these questions are tackled during the design decisions when building function models from services.

5.1 General Process on Each Level

In figure 5 the general process is shown exemplary for the test case derivation on the functional level. The modeled services are used as test case specifications. With help of these service models, a test case generator derives test cases out of the functional model (cluster and platform model, resp.). These test cases are applied to artifacts on the more concrete levels.

The first models which are used to generate test cases are the models on the functional level, since this is the most abstract representation of the complete specification of the system. In figure 6 a simple automaton is shown which realizes the services specified for the *WindowControler* example of section 4.1 before.

From the models of the functional level, the logical cluster level test cases and platform level test cases are generated according to the identified services. For using these test case specifications, it is necessary to consider the functional models from the point of view of each service. This is done by cutting only those parts out of the complete model which are touched by the service. In a formal way, as for example described in [1], this means a restriction of the function

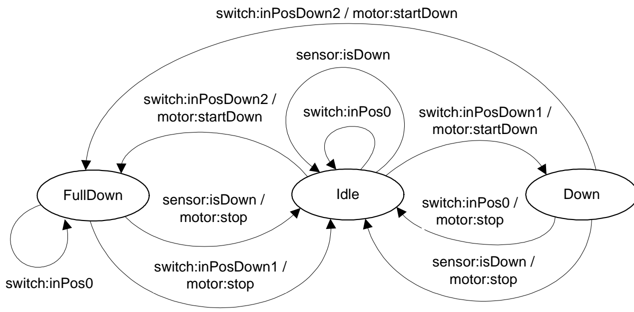

Figure 6: *WindowControl* example on the functional level

(which can be seen as a total component) to the respective service domains. For the actual test case generation, an existing test case generator as mentioned in section 2 might be used. The procedure can be described by the following algorithm:

```
TestSet generateTestCases(Model fullModel,
                          ServiceSet setOfServices) {
  TestSet testSuite;
  foreach s in setOfServices do {
    Model selectedModel = restrictModelToService(fullModel, s);
    TestSet testsForService = callTestGenerator(selectedModel);
    testSuite.insert(testsForService);
  }
  return testSuite;
}

Model restrictModelToService(Model m, Service s) {
  // returns a model which only represents
  // the traces of the model m
  // which fulfill service s
}

TestSet callTestGenerator(Model m) {
  // some known test generator (see section 2) is called
  // which for example uses coverage criteria to further
  // select test cases in the restricted model m
}
```

The key point in the algorithm is that the complete model is restricted to an appropriate extract for each service before the classic test case generation is done. The complete test suite contains in the end the union of all test sets generated for each service. In this way we get every user requirement (which is represented through a service) tested. Additionally, a promising set of test cases is generated rather than just one test case for each service sequence. In general possible methods for the restriction of the model are as follow:

- eliminate all states in m which have no outgoing transition caused by an action of s
- eliminate all states in m which have no incoming transition emitting an action of s
- eliminate all transitions in m which are not caused by an action of s
- eliminate all transitions in m which do not emit an action of s

The detailed formulation and evaluation of the algorithm is still part of our ongoing research.

5.2 Refinement of Tests Along Design Levels

Besides the above mentioned general methods to restrict models, specific techniques can be applied to the single modeling levels. In the following sections we therefore take a closer look at the single levels. When using the different abstraction levels to generate tests out of them, the resulting test suites contain more and more refined tests from each level. This refinement of test cases is shown in the following.

5.2.1 Functional Level

Models on the functional level, as introduced in section 3.3, integrate all single services *and* give a complete system specification in the sense of a total function. But still, these models remain quite abstract from the implementation and focus only on the users' view on the system.

When applying the functional models to the test case generation algorithm stated in the section above, the distinction between dependencies of services is valuable: As already noted in section 4.3 two kinds of dependencies of services exist in the functional models: First, there are interactions of services which result from the integration of single services to map desired feature interaction. A second class of dependencies comes from the totalization of the function, e. g. when missing transitions are added in the state machine. These distinction of dependencies is used to restrict the model in the `restrictModelToService(...)` method:

- *Feature interaction dependencies* should be followed in model paths, since these are important from the users view.
- *Totalization dependencies* should not be followed in model paths, since these have quite less significance towards the users' expectations on the SUT.

The advantage of getting tests from the functional level lies in both the immediate representation of the users view on a system in the test and the high usefulness for re-use of test cases. Since the retrieved test cases are independent from almost all realization constraints, the same test case may be used for different implementations of the function. Consider for example a function which has been implemented on a single control unit and must now be deployed on several control units communicating over some bus architecture in a new technical environment.

But this is also a backtrack of tests from this level: The actual technical realization, as for example distribution to different control units, identifies error-prone model elements (*"Has the bus communication been realized correctly?"*). The tests from the functional level cannot specifically address such questions. But tests on the further levels serve as adequate means therefore and will refine the test set as described in the next sections.

5.2.2 Level of Logical Clusters

As explained in section 3.4 models on the logical cluster level consider mainly the distribution of embedded systems on different runtime platforms and in asynchronous tasks. On the functional level these design aspects are neglected but for testing information about distribution of the system may be quite useful. Connections between distributed units (e. g. realized by some communication bus) are often a source for errors. Therefore it is imported to check if the

defined communication relations will not be violated in the implementation and/or on the subsequent modeling levels.

Similar to the functional level services are used again as test case specification; again we want to preserve the link to the user requirements. On the level of logical clusters we are able to identify subsets of clusters which contribute to accomplish the single services. Thus we get the following rules for the `restrictModelToService(...)` method:

- Connections between clusters which are related to a specific service should be followed in model paths.
- Connections to clusters which do not affect a specific service must not be followed.

Unlike the models on the functional level, the logical cluster level is not only used for test case generation. The models on this levels are also used as test objects: Tests generated from the functional level are applied to these models.

5.2.3 Platform Level

The usage of models on the platform level is quire similar to the usage of models on the logical cluster level. Unless the levels before restriction specific to the chosen hardware platform are also considered in the models on this level. Thus test cases can be retrieved which also comprise these hardware depended requirements. In our ongoing research we will consider the platform level in more detail.

6. RELATED WORK

Test case generation with structural criteria is for example described in [4, 6, 8, 9, 10]. Unlike these approaches we base our work mainly on user requirements for test case generation. Using sequences for test case generation is known for long as for example introduced in [3] where test cases are derived directly from sequence charts. In contrast, our work focuses on the integration of the before mentioned model-based generation techniques by the use of services (which may be represented by sequence charts). Furthermore we integrate the test process in a design process which is based on different abstraction levels as introduced in [12]. Similar to [5] we see the need for better coverage criteria and layout a basis for the definition of an adequate requirements coverage.

7. CONCLUSION AND FURTHER WORK

In this paper we introduced a testing approach which is based on an order of abstraction levels. Here the user requirements are formalized as services which are used as test case specification. Test cases are derived from the models on the subsequent levels. By this we keep the trace between generated test cases and user requirements. Since the different levels focus on different design decisions we also get test cases which cover realization constraints and identify error-prone parts of the system design.

We believe this method will be a valuable basis for the definition of new coverage criteria for test case generation which are based on user requirements rather than on structural aspects. In our current work we focus on the refinement of the test case generation algorithms and on a more detailed specification of the modeling approach. Currently an architecture description language (called CAR-DL) is being developed which includes the relations between the introduced abstraction level. We consider this also as helpful for the definition of test drivers which are needed to instantiate the abstract test cases to make them executable on the test object.

8. REFERENCES

[1] M. Broy. *Engineering Theories of Software Intensive Systems*, chapter Service-Oriented Systems Engineering: Specification and Design of Services and Layered Architectures, pages 47–81. Springer, 2005.

[2] A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, and D. Wild. Concretization and formalization of requirements for automotive embedded software systems development. In K. Cox, J.L. Cybulski et.al, editor, *Proc. of the Tenth Australian Workshop on Requiremements Engineering (AWRE)*, pages 60–65, Melbourne, 2005.

[3] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by mscs. In O. Faergemand and A. Sarma, editors, *SDL'93 - Using Objects*, North-Holland, Oct. 1993.

[4] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 261–270, 2004.

[5] M. Heimdahl, G. Devaraj, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 178–186, 2004.

[6] A. Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme.* PhD thesis, Technische Universität München, 2003.

[7] A. Pretschner and M. Leucker. *Model-Based Testing - A Glossary*, volume 3472 of *Lecture Notes in Computer Science*, chapter 20, pages 607–609. Springer-Verlag, Berlin, Heidlberg, July 2005.

[8] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2 - 3):140–157, Mar. 2004.

[9] S. Rayadurgam and M. Heimdahl. Coverage based test-case generation using model checkers. In *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, pages 83–91, 2001.

[10] S. Rayadurgam and M. Heimdahl. Generating mc/dc adequate test sequences through model checking. In *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, pages 91–96, 2003.

[11] R. Whiting. Take the black magic out of software development. *Electronic Business*, 18(9):34–44, 1992.

[12] D. Wild, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, and S. Rittmann. An architecture-centric approach towards the construction of dependable automotive software. In S. of Automotive Engineers, editor, *Proceedings of of the SAE 2006 World Congress (to appear)*, Detroit, 2006.

[13] M. Zelkowitz, A. Shaw, and J. Gannan. *Principles of Software Engineering and Design.* Prentice-Hall, Englewood Cliffs, 1979.