Generation of Executable Testbenches from Natural Language Requirement Specifications for Embedded Real-Time Systems

Wolfgang Mueller[1], Alexander Bol[1], Alexander Krupp[1], Ola Lundkvist[2]

[1]University of Paderborn/C-LAB, Paderborn, Germany
[2]Volvo Technology Corp., Mechatronics & Software, Gothenburg, Sweden

Abstract. We introduce a structured methodology for the generation of executable test environments from textual requirement specifications via UML class diagrams and the application of the classification tree methodology for embedded systems. The first phase is a stepwise transformation from unstructured English text into a textual normal form (TNF), which is automatically translated into UML class diagrams. After annotations of the class diagrams and the definition of test cases by sequence diagrams, both are converted into classification trees. From the classification trees we can finally generate SystemVerilog code. The methodology is introduced and evaluated by the example of an Adaptive Cruise Controller.

Keywords: Natural Language, UML, SystemVerilog, Testbenches

1 Introduction

Since the introduction of the electronic injection control by Bosch in the 80s, we observed a rapid growth of electronic systems and software in vehicles. Today a modern car is equipped with 30-70 microcontrollers, so called ECUs (Electronic Control Units). With the acceptance of the AUTOSAR standard and its tool support there is a need for further automation in automotive systems developments, especially in the first design phases.

Currently, the model-based testing process is based on different design stages, like Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), and Hardware-in-the-Loop (HIL) tests. In this context, different testing hardware and software come into application like MTest from dSPACE which compares to the Classification Tree Method for Embedded Systems (CTM/ES) which we applied in our work. While model-based testing is well supported by existing tools, one of the major challenges still remains the transformation of requirements to a first executable specification. In practice, such requirements are typically captured as unstructured text by means of tools like Rational DOORS from IBM. Today, we can identify a major gap between requirement specifications and first implementation of the executable testbench.

This article closes this gap by introducing a structured semi-automatic methodology for the generation of test environment via UML class diagrams and CTM/ES. The first phase performs a stepwise transformation of natural language

sentences before they are automatically translated into UML class diagrams. For automatic translation, we defined a textual normal form (TNF) as a subset of natural English sentences, where classes, attributes, functions, and relationships can be easily identified. The generated class diagrams are annotated by additional information so that we can - after the definition of test scenarios - generate a testbench. In our evaluation, we applied SystemVerilog and QuestaSim and linked it with native SystemC code and C code generated from Matlab/Simulink. Though we applied SystemVerilog for the implementation of this case study, our methodology is not limited to SystemVerilog. The introduced methodology may easily adapt to other languages like e [10] as long as they support function coverage definition and random test pattern generation.

The remainder of this article is structured as follows. The next section discusses related work including CTM/ES and principles of functional verification as basic technologies. Section 3 introduces the four steps of our methodology. Thereafter, we present experimental results in Section 4. Section 5 finally closes with a summary and a conclusion.

2 Existing Work

In embedded systems design, test processes for automotive software are based on tool support with heterogeneous test infrastructures. The model-based testing process is based on different development steps like Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), and Hardware-in-the-Loop (HIL) tests. In this context, different testing environments come into application like ControlDesk, MTest, and AutomationDesk from dSPACE. Each test environment typically applies its own proprietary testing language or exchange format and we can find only very few approaches to standard languages like ETSI TTCN-3 and the OMG UML testing profile [16].

For natural language requirement specification capture and management, Rational DOORS or just MS Word or MS Excel is applied on a regular basis. In order to increase the level of automation, several XML-based formats for enhanced tool interoperabilities have been developed. For requirement captures, RIF (Requirement Interchange Format) has been defined by HIS (Hersteller Initiative Software) and is meanwhile adopted by several tools. For the exchange of test descriptions, ATML was introduced by IEEE [7] and TestML by the IMMOS project [5]. The latter provides an XML-based exchange format which supports functional, regression, Back-to-back and time partition tests where stimuli can be defined by different means like classification tree methodology for embedded systems (CTM/ES) [4], which is introduced in the next subsection.

In general, there has been early work for the formalization of text by entity relationship diagrams like [1] and multiple work for the generation of test cases from test scenarios like [13]. However, we are not aware of any work which combines those for the generation of complete test environments (i.e., testbench architectures and test cases) for real-time systems taking advantage of principles of functional verification, e.g., functional coverage and constraint based test pattern generation.

2.1 Classification Tree Method

Classification Trees were introduced by Daimler in the 90's [6]. Classification trees provide structured tree-oriented means for capturing test cases. Starting from an entry node, the test input is divided into compositions, classifications, and classes (see Fig. 1). A classification is divided into (equivalence) classes which represent an abstract set of test inputs each. A leaf class defines further an abstract set of possible input values. Columns and rows below define a combination table. Therein, in each row exactly one class of each classification is selected. Each row defines exactly one test step and compares to different states of the state machine which controls the test environment. The development of a classification is defined by the Classification Tree Method (CTM) [6], which is based on the Category-Partition-Method [14].

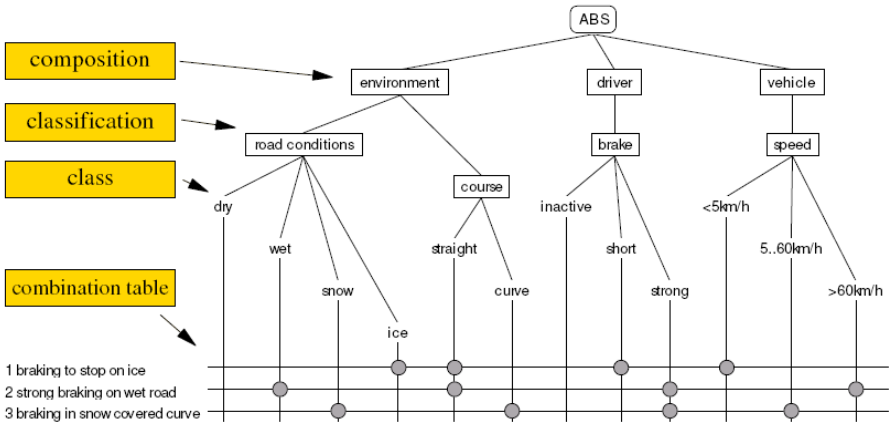

Figure 1. Classification Tree.

In its first introduction, classification trees described different variations of input stimuli of a System Under Test (SUT). As embedded automotive systems testing is based on sampling rates and time-based relationships between stimuli, Conrad has extended CTM to the Classification Tree Method for Embedded Systems (CTM/ES) [4]. As such, a classification tree is derived from the interface of the SUT and classifications from SUT inputs. The input domain is partitioned into different input interval classes like safety-critical ranges and corner cases. This compares to the definition of bins in the definition of a SystemVerilog functional coverage definition. For the management of more complex test suites, test cases are additionally divided into test sequences. Finally a timeline is assigned to each test sequence. That timeline is typically related to the sampling rate in automotive systems testing. Each horizontal line defines the inputs of a SUT for a specific time period where a time point stands for the activation or synchronization point of a test step. Therefore, an additional transition functions, e.g., step, ramp, sinus, has to be assigned to a synchronization point, which defines the transition between values of different synchronization points. In the combination table, different transition functions are indicated by different line styles between transition points (see Fig. 6).

2.2 Functional Verification

Our final verification environment based on the principles of functional verification. The notion of functional verification denotes the application of assertions, functional coverage, and constrained random test pattern generation in ESL and RTL designs. Such technologies are based on the application of Hardware Verification Languages like the IEEE Standards SystemVerilog [8], PSL [9] and e [10]. They support the formal and reusable definition of system properties for functional verification. Standardized APIs like the SystemVerilog DPI additionally support multi language environments and provide an interface to adapt proprietary test environments. Meanwhile, there exist several libraries and methodologies for additional support like VMM [2] and OVM [15].

3 Generation of Testbenches from Requirements Specifications

Our methodology for the derivation of executable SystemVerilog testbenches applies four different phases:

1. Formalization of Requirements
2. Transformation of Class Diagrams
3. Definition of Test Scenarios
4. Generation of the Testbench

We first provide a stepwise manual transformation of unstructured natural language English sentences into short structured English sentences. The latter can be seen as a first formal version of the requirements as they directly correspond to UML class diagrams to which they can be automatically translated. After some simple transformations of the class diagrams they are further translated into compositions, classifications, and classes of a classification tree for embedded systems. Concurrently, in Phase 3, test scenarios have to be developed. We propose the application of UML sequence diagrams though related means can be applied as well. The test scenarios compose the test sequences of the classification tree. Finally, we can automatically generate a testbench from the classification tree. In our case, we generate SystemVerilog code [8]. However, we can apply any comparable Hardware Verification Language which supports random test pattern generation and function coverage specification.

In the next paragraphs, we outline the four phases in further details. For this we apply an industrial case study from the automotive domain, i.e., an Adaptive Cruise Controllers (ACC) [3]. The ACC is a cruise controller with a radar-based distance control to a front (subject) vehicle. The ACC controls the cruise speed and the distance to the front vehicle in the same lane with the desired speed and a desired distance as input.

3.1 Formalization of Requirements

Starting from a set of unstructured English sentences, they are stepwise manually formalized into a Textual Normal Form (TNF) which is composed of unambiguous sentences which can be automatically transformed into a UML class diagram. Table 1 gives an example of some sentences before and after the transformation.

Table 1 Transformation of Unstructured Sentences.

Unstructured Sentence	Transformed Sentences
The ACC system shall include a long-range radar sensor capable of detecting data about moving objects travelling in the same direction as the driven vehicle. If a vehicle is identified by the ACC a safety distance shall be kept by actuating over the throttle or applying the brakes if necessary. The ACC system shall operate under a limited speed range, between 20 and 125 km/h. The distance for detecting vehicles shall be limited to 150 meters.	*AdaptiveCruiseController is entity.*
	Radar is entity.
	AdaptiveCruiseController getsDataFrom Radar.
	AdaptiveCruiseController has currentDrivenVehicleSpeed {currentDrivenVehicleSpeed is between 20 and 125 km/h}.
	AdaptiveCruiseController has currentDistance {currentDistance is between 1 and 150 meters to SubjectVehicles}.

An unstructured textual requirement specification typically includes information about the logical description of the SUT and the environment. It identifies operational constraints and conditions but also logical components, functions, and attributes which include important information to implement test environments and test cases. The structured transformations of that information are an important step to support the traceability of requirements to their corresponding testbench components in order to guarantee the compliance of the testbench to the requirements for an advanced quality assurance.

The target of the first transformation phase is the Textual Normal Form (TNF). TNF is a machine readable presentation composed of three word sentences (plus constraints) which are later automatically transformed into UML class diagrams as an intermediate format. During the different manual transformation steps, redundancies, incompleteness, and contradictions can be much better identified by visual inspections than in the unstructured sentences. In the first step, we remove filler words and empty phrases, like *'basically'* and '*most likely*'. Thereafter, we transform long sentences into short sentences without disambiguities and incomplete information as far as possible. For instance, we split long sentences and transform subordinate clauses into main clauses and replace pronouns by proper nouns. Then, subjects and objects are transformed into identifiers, articles removed, and sentences translated into present tense. If necessary, this also means to combine or extend subjects/objects with attributes like '*Adaptive cruise controller system*' to '*AdaptiveCruiseController*'. After this, each identifier has to refer to exactly one entity, i.e., two different identifiers are not allowed to refer to the same entity and the same identifier shall not refer to different entities.

Finally, for each identifier *X*, we add an explicit sentence '*X is entity*'. This helps for later automatic translations and completeness checks by visual inspection. After the identification of entities, we have to further proceed with attributes, functions, and relationships. In details, we identify the attributes of each entity and separate it into a individual sentence of form '*<entity id> has <attribute id>'*, e.g., '*AdaptiveCruise Controller has currentDistance'* (cf. Table 1). We also associate attribute sentences with the corresponding constraints and append them enclosed in curly brackets. Thereafter, the identification of functions with constraints is similarly and results in sentences like '*Driver does applyBrakePedal'*. It is important to note here that we combine the verb with the object id for the final name of the operation in the later class diagram. Finally, all relationships between entities are identified and sentences like '*AdaptiveCruiseController getsDataFrom Radar'* are separated.

We finally arrive at a forest structured transformation relationship between original sentences at the root and TNF sentences at the leaves. When applying a simple tool like MS Excel, we can easily sort the final sentences by the first identifier (i.e., the subject), which helps to easily check for duplicates or subjects with similar meaning and even for incomplete specifications which can hardly be detected in the unstructured original text. The final TNF is nothing else than the textual representation of Class Diagrams which can thus be automatically derived along early works of Bailin [1]. For this consider the following TNF sentence examples:

- *AdaptiveCruiseController is entity..*
- *AdaptiveCruiseController has currentDistance*
- *AdaptiveCruiseController does controlCurrentDistance.*
- *AdaptiveCruiseController getsDataFrom Radar.*

We can easily see their direct correspondence to the UML Class Diagram in Fig. 2. For more details, the reader is referred to [1].

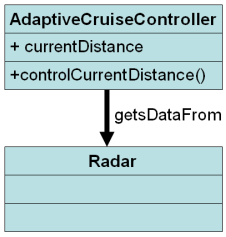

Figure 2. UML Class Diagram.

3.2 Transformation of Class Diagrams

In the second phase, Class Diagrams are structured and annotated before they are transformed into a classification tree, which is an intermediate representation for the automatic generation of the executable testbench.

As such, we first divide all classes into categories *<<environment>>* for the test environment and *<<system>>* for the SUT by assigning UML stereotypes to them. Thereafter, we analyze all attributes of all classes and divide them into: *in*, *out*, and

internal with corresponding stereotypes. Attributes of the first category are further qualified by the delivering class as it is shown in Fig. 3. As we are dealing with distributed systems, we have to compute the same attributes by different classes. In that figure, we can also see the *out* category is actually redundant as the information is already implicitly covered by the two other categories. However, this redundancy helps to better analyze the interaction between the classes and to detect further inconsistencies as all *in* and *out* attributes of the DUT give a complete definition the DUT interface. Thereafter, we have to formalize all <<*out*>> attributes of all <<*environment*>> classes. Let us consider *currentDrivenVehicleSpeed* of *DrivenVehicle* in Fig. 3 as an example. The original constraint defines that the ACC is only active between 20 and 125 km/h (see also the system class in Fig. 3). Considering a maximum vehicle speed of 250 km/h, we can formalize it by the definition of five intervals with 20 and 125 as corner values. In SystemVerilog syntax, this is defined as *{[0:19], 20, [21:124], 125, [126,260]}*. This example shows that several constraints can be retrieved from the original requirement specification. In practice, additional conventions and standards like IEC 61508 [13] have to be consulted to retrieve the complete set of constraints. Though our example defines closed intervals due to the limitations of SystemVerilog, without the loss of generality, we can also apply open intervals provided they are supported by the tools or verification language.

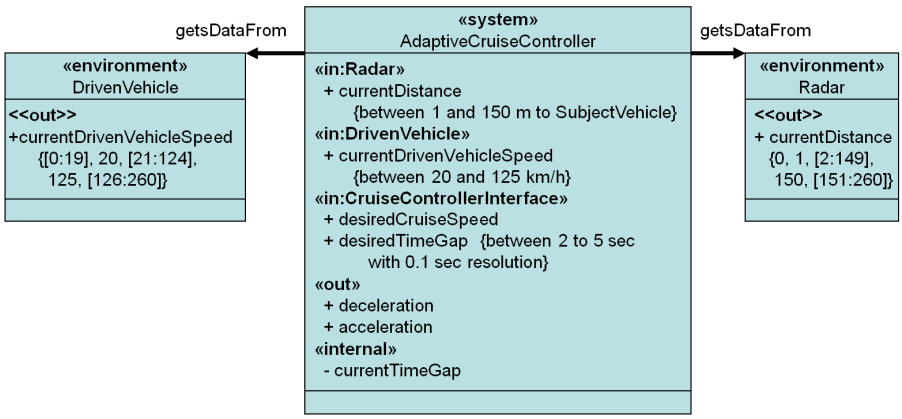

Figure 3. Modified UML Class Diagram.

The final version of the UML Class Diagram can now be directly translated into a classification tree (without a combination table) with the SUT at the root. The individual UML *environment* classes translate to the different compositions and the class attributes to classifications (cf. Fig. 4).

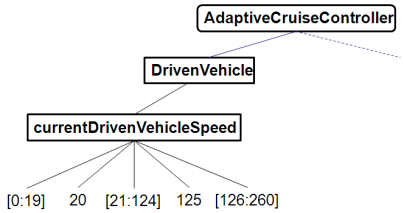

Figure 4. Fraction of a Classification Tree for Embedded Systems.

3.3 Definition of Test Scenarios

In the next step, we have to manually define test scenarios with test steps and test sequences for the completion of the classification tree. We start with the selection of one or more *environment* classes from the class diagram. The following example takes an interaction of the *Driver* and the (*Driven*) *Vehicle* with the ACC and defines a simplified scenario with five steps:

1. *Vehicle drives at a speed of 125 km/h.*
2. *Driver sets a new speed (desiredCruiseSpeed).*
3. *Driver sets distance to front vehicle (TimeGap) .*
4. *Vehicle reaches a medium speed.*
5. *Vehicle reaches a high speed.*

We now can link the entities in the description to the classifications in the classification tree and define a UML Sequence Diagram in order to formalize the five steps. The individual steps of the description have to be mapped to message interactions with intervals as parameters. The next step is the creation of several instances of this Sequence Diagram with respect to the timeline and variations of message parameters. Fig. 5 gives an example of a possible instantiation. In this example, we assign the timeline to the time points 0s, +2s, +3s, and +5s. The vehicle starts with a speed of 125 km/h at 0s. At 3s the speed changes to an interval between 21 and 124 km/h. Hereafter, the speed increases at 5s.

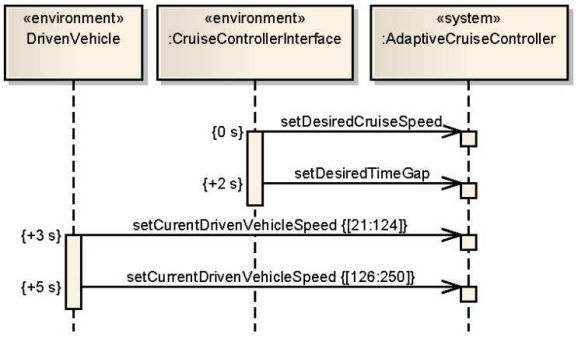

Figure 5. Test Sequence as an UML Sequence Diagram.

Each of the Sequence Diagrams can be easily transformed to a test sequence of the classification tree. Fig. 5 shows part of the final classification tree, which is translated from the diagram of Fig. 5. Fig. 6 also shows some interpolation functions between synchronization points, which have to be defined before the generation of the testbench.

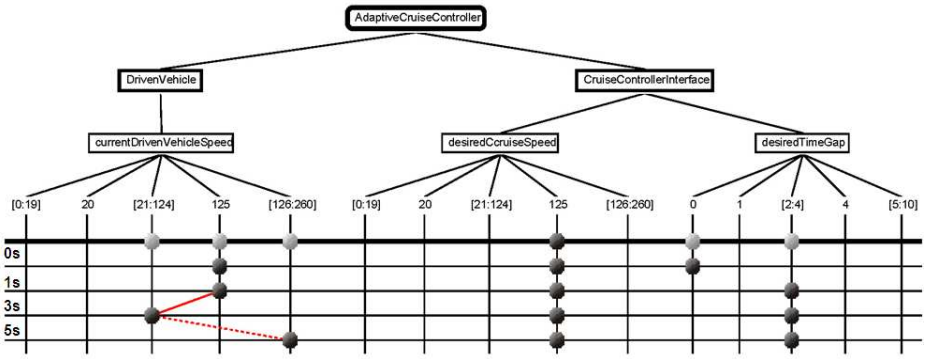

Figure 6. Extended Classification Tree.

3.4 Testbench Generation

The final phase generates an executable testbench from the classification tree which includes the test sequences. More details of this phase can be found in [12]. Though we apply SystemVerilog here, other verification languages which support random test pattern generation and functional coverage can be taken as well. As an example, we focus on the application of SystemVerilog constraints for random test pattern generation in the following outlines.

After randomization, the input vectors with interpolation functions are applied to the specified SUT interface of the <<*system*>>. Due to the current tool support, the general execution is controlled from SystemVerilog where the SUT can be implemented in other languages like SystemC or C code generated from Matlab/Simulink. For the translation of classification tree test sequences, each test sequence is translated to a SystemVerilog class with array variables which correspond to the classifications of the classification tree, i.e., an input signal of the SUT:

```
class AdaptiveCruiseController_Sequence1;
   rand Int_class_sp
   currentDrivenVehicleSpeed[];
      …
```

Furthermore, each array element corresponds to a test step of a classification tree test sequence for which randomization (*rand*) is applied. The array element has a data structure which includes an attribute for the time point of the test step, the value of the signal, and the individual interpolation function, like *ramp* or *sinus*. For each SystemVerilog class, we also generate a constraint block, which implements the

constraints specified in the classification tree. The constraints implement the timing behavior and the selection of the equivalence class as follows:

```
constraint ctmemb{
   currentDrivenVehicleSpeed[0].t==0*SEC;
   currentDrivenVehicleSpeed[0].v==125;
   currentDrivenVehicleSpeed[1].t==
                  currentDrivenVehicleSpeed[0].t+2*SEC;
   currentDrivenVehicleSpeed[1].v==125;
   desiredTimeGap[0].t==currentDrivenVehicleSpeed[0].t;
   desiredTimeGap[0].v==0;
   desiredTimeGap[1].t==currentDrivenVehicleSpeed[1].t;
   desiredTimeGap[1].v inside {[2:4]}; …
}
```

This example implements the constraints of the first two test steps (with index *1* and *0*) of test *Sequence1* for the two signals *currentDrivenVehicleSpeed* and *desiredTimeGap*. For each signal at each step, the time and the value is assigned. Here, *SEC* stands for the adjustment to the time unit of the simulation time. Along the classification tree specification, the second synchronization point is 2 seconds after the first one. The last line takes the interval *[2,4]* for the *desiredTimeGap* directly from the classification tree to SystemVerilog. Additionally, we define a method *pre_randomize()* for the SystemVerilog class, which instantiates array data in preparation for randomization and initializes values that are not randomized like the interpolation function.

```
function void pre_randomize
    foreach(currentDrivenVehicleSpeed[i])
          currentDrivenVehicleSpeed[i]=new();
    …
    currentDrivenVehicleSpeed[1].ipol = ramp;
    …
```

4 Experimental Results

We have applied the introduced the textual requirement specification of an Adaptive Cruise Controllers (ACC) in [3]. The original key requirements were composed of 23 long sentences. Those sentences were transformed by our methodology in to their Textual Normal Form (TNF) with finally 79 sentences. They were translated to a UML class diagram with 10 classes. Table 2 gives an overview of the details of the generated class diagram. The adapted class diagram had 6 classes, which were translated into a classification tree. Details of that classification tree can be found in Table 3. For our first evaluation we defined a limited set of 2 test scenarios which were related to 2 test sequences with 14 test steps each. The final test environment which was automatically generated from the classification tree was composed of 526 lines of SystemVerilog code.

Table 2. UML Class Diagram Numbers.

Class	#attributes	#methods	#assoc. in	#assoc. out
AdaptiveCruiseControler	13	11	1	7
CruiseControlerInterface	3	8	2	2
Radar	5	6	1	1
Driver	0	2	1	3
SubjectVehicle	1	1	1	0
DrivenVehicle	1	0	3	0
BrakePedal	1	1	2	1
Accelerator	1	1	2	1
Brake	0	0	2	1
Throttle	0	0	2	1

Table 3. Classification Tree Numbers.

Signal	# equivalence classes	component
acceleratorPosition	3	Accelerator
brakePedalPosition	3	BrakePedal
currentDistance	5	Radar
currentDrivenVehicleSpeed	5	DrivenVehicle
desiredCruiseSpeed	5	CruiseControlerInterface
desiredMode	2	CruiseControlerInterface
desiredTimeGap	5	CruiseControlerInterface
vehicleInSameLane	2	Radar

5 Conclusions and Outlook

This article presented a structured semi-automatic methodology for the generation of executable SystemVerilog testbenches from unstructured natural language requirement specification via UML and classification trees. After transformation into a Textual Normal Form (TNF), UML class diagrams are generated. After some annotations and simple adjustments they are further translated into a classification tree from which a SystemVerilog testbench is automatically generated. We successfully applied and evaluated our methodology to the requirements specification of an Adaptive Cruise Controller which was implemented in SystemC/C.

Our evaluation has shown that in the first phase until the final derivation of the TNF, several incomplete and redundant statements could be easily identified. This is a very important issue for industrial application as identification of inconsistencies and errors in very early design phases may result in a significant reduction of design respins. Due to our experience, with some extend, the detection of such errors based on manual transformations and visual inspection is currently still the most efficient and fastest method compared to a first time consuming transformation to a first formal model like finite state machines and logical formulae.

However, the main advantage of our methodology is definitely the complete traceability of each individual requirement to the corresponding objects or methods in the testbench. Though we just have used MS Excel to capture the requirements in our studies, it was easily possible to trace single requirements via subrequirements to SystemVerilog classes, methods and attributes. This greatly simplifies feedback with the customers in order to quickly resolve open design issues. Our studies have also indicated that it is very hard to achieve a complete automation of the first phase as transformations of natural language statements still require the dedicated expertise of a domain engineer. In contrast, transformations in later phases are subject of further possible automation. Our studies gave promising results and more evaluations have to follow.

Acknowledgments. The work described herein is partly funded by the BMBF through the SANITAS (01M3088) and the VERDE project (01S09012).

References

1. Bailin, S. C.: An Object-Oriented Requirements Specifications Method. In: Communication of the ACM, 32(5), May 1989.
2. Bergeron, J., Cerny, E., Nightingale, A., Hunter, A.: Verification Methodology Manual for SystemVerilog, Springer, 2006.
3. Bernin, F., Lundell, M., Lundkvist, O.: Automotive System Case Study, Deliverable D1.1 PUSSEE Project IST-2000-30103, 2002.
4. Conrad, M.: Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien. Deutscher Universitätsverlag, 2004.
5. Grossmann, J., Conrad, M., Fey, I., Krupp, A., Lamberg, K., Wewetzer, C.: TestML – A Test Exchange Language for Model-based Testing of Embedded Software. In: Automotive Software Workshop ’06, San Diego, March 2006.
6. Grochtmann, M., Grimm, K.: Software Testing, Verification and Reliability: Classification Trees for Partition Testing. John Wiley & Sons Verlag, 2006.
7. IEEE: Draft Specification for Component Standard of Automatic Test Markup Language (ATML) for Exchanging Test Results via XML. December 2004.
8. IEEE: IEEE Std.1800-2005 - Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language. November 2005.
9. IEEE: IEEE Std.1850-2005 - IEEE Standard for Property Specification Language (PSL). September 2005.
10. IEEE: IEEE Std.1647-2006 - The Functional Verification Language ’e’. March 2006.
11. IEC: Functional safety electrical/electronic/ programmable electronic safety-related systems - IEC 61508 Part 1-7, Geneva, Switzerland, 1998.
12. Krupp, A.: Verification Plan for Systematic Verification of Mechatronic Systems. Doctorial Thesis, Paderborn University, 2008.
13. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In Proc. of UML99, Fort Collins, USA, 1999.
14. Ostrand, T.J., Balcer, M.J.: The Category-Partition Method for Specifying and Generating Functional Tests. ACM, 1988
15. OVM Homepage. www.ovmworld.org.
16. Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A.: The UML 2.0 Testing Profile and its Relation to TTCN-3 (2003). In: Proc. of TestCom2003, Sophia Antipolis, 2003.