# Specification-based Verification of Embedded Systems by Automated Test Case Generation

Christoph M. Kirchsteiger, Christoph Trummer, Christian Steger, Reinhold Weiss, and Markus Pistauer

**Abstract** It is time and resource intensive to derive test cases manually from the requirements specification to fully verify that the embedded system design fulfills its specification. However, automatic parsing to generate test cases is often not possible due to the informal, non-machine readable structure of the specification document. Formal specification languages would ease the parsing process, however they are difficult to use and rarely accepted. A promising trade-off are semi-formal specification languages, which are both easy-to-parse and easy-to-use.
This paper presents a novel approach developed in the SIMBA[1] project to tightly integrate a semi-formal requirements specification document into the design flow of embedded system designs. It considers the specification as a series of semi-formal textual use cases and automatically generates specification-based SystemC test cases. During a simulation with the System-under-Verification (SuV) the test cases are executed to determine whether the SuV fulfills the specification. A demonstration is given by a case study of an RFID controller. It shows that errors in the specification and discrepancies between the design and its specification are detected.

Christoph M. Kirchsteiger · Christoph Trummer · Christian Steger · Reinhold Weiss
Institute for Technical Informatics, Graz University of Technology, Inffeldgasse 16/1, 8010 Graz, Austria
e-mail: (c.kirchsteiger, trummer, steger, rweiss)@tugraz.at

Markus Pistauer
CISC Semiconductor Design & Consulting GmbH, Lakeside B07, 9020 Klagenfurt, Austria
e-mail: m.pistauer@cisc.at

[1] This project has been funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FFG contract FFG 812424

## 1 Introduction

In today's design of embedded systems, 70% of the entire design effort is spent on functional verification. Functional verification is mainly driven by finding adequate test cases to verify that the modeled system behaves according to its specification [21]. Clearly, deriving test cases manually by reading the large system specification document is very time and resource intensive and error-prone. On the other hand, it is infeasible to perform this task automatically due to the informal non-machine readable structure of the specification.
The approach presented here focuses on semi-formal description formats to specify requirements. A very promising and well-known semi-formal specification style are textual use cases [4]. Although they are similar to graphical UML use cases enhanced by UML sequence diagrams, they can be extended much more easily to cover additional domain-specific information (e.g. by inserting additional fields for non-functional requirements). Textual use cases are both widely accepted to communicate with a customer and suitable for automatic post-processing. They define the interaction and the behavior of a system under certain conditions (pre-/postconditions, trigger) as a sequence of interaction steps with the environment (=actors). Their structure is formal, table-based and composed of several fields for the name, the pre-/postconditions and the interaction scenarios. However, within each field the description is entirely informal. Thus, textual use cases are similar to natural language but used in a structured way, which makes them easy-to-learn for stakeholders from various domains.
A common textual use case description contains the following fields:

- Actor (communicates with the specified system)
- Pre-/postcondition and trigger
- Main success scenario (i.e. main interaction sequence)
- Extensions (i.e. alternative flows to the main scenario)

In this paper, we propose a novel design methodology (see Fig. 1) for the specification-based functional verification of embedded system models by simulation. We use simulation for verification without being concerned with the state-space explosion problem as in static verification techniques. The main steps of our approach as shown in Fig. 1 are highly automated and encompass both the error-correction of the original specification document and the functional verification of the system model. The generated test cases are based upon the SystemC Verification Library (SCV) [20] and can be used to verify both transaction-level models and RTL hardware designs [18].
The remainder of this paper is organized as follows: We start with an overview of related work in section 2. In section 3 we present our methodology and describe its implementation in section 4. Section 5 provides a case study of an Radio Frequency Identification controller (=RFID tag) to present the applicability of our methodology and its results. Finally we give a conclusion and list further work in section 6.

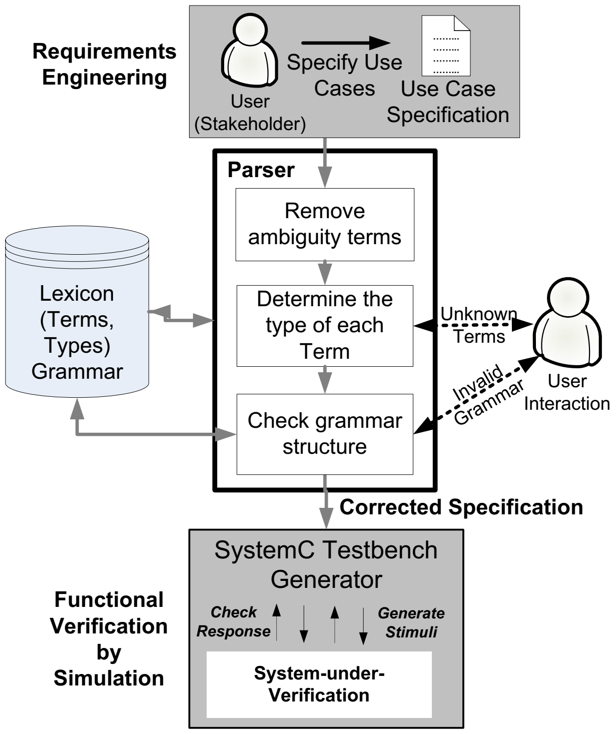

**Fig. 1** Novel highly automated approach from textual use cases to a SystemC testbench for functional verification.

## 2 Related Work

Test case generation from the specification has been widely studied in the research community. Most of them, like [24], [13] and [1] favor formal specification languages as UML or SDL. However, in the hardware domain, which constitutes an important portion of embedded systems, most of the designs are specified in a document-based way. UML and other formal specification languages are hardly used and are considered as a large burden, which confronts time-to-market. Although these specification formats are unambiguous, precise and consistent, it is very difficult for stakeholders from various domains, who specify requirements, to get familiar with these formats. In contrast, our approach is based on semi-formal textual use case-based descriptions as defined in [4]. They are both widely accepted and easy-to-use by stakeholders and suitable for automatic post-processing.
There are a number of approaches, which are dealing with textual use case-based descriptions for test case generation. Most of them, like [9] and [10] focus on the formal transformation of use cases to UML state, message sequence or activity charts, which are then used to generate the test cases. Whereas, the automatic test case generation from UML charts is widely studied in the research community [16], [15],

[11] the formal transformation of the use cases to UML charts, apart from the approaches stated below, is usually done by hand. However, this requires a lot of interaction effort since the number of use cases can be very large. Our work resolves this issue by generating the test cases directly from the use case specification with a high degree of automation and without the need for a transformation to UML charts. Significant approaches related to our work are [22], [5], [3] and [7]. In [22] and [7] the use case specification of a computer system is used to automatically generate test cases. However, only the first step consisting of transforming the use cases to UML activity diagrams is described. Nevertheless, as described in [3], the generated activity diagrams lack relevant information on the used message types and the connection to the SuV's interface, which is required to derive the test cases automatically from the diagrams. The same is true for [5], which requires the test designer to specify the test purpose of each test he wants to execute. In contrast, our approach automatically generates a verification environment consisting of stimuli generation and response checking and randomly selects and executes the test cases.

## 3 Novel Approach

We propose a novel specification-based functional verification by simulation methodology that aims for:

- Check the specification to remove ambiguities and incorrect grammar.
- Automate the functional test case generation from textual use case specifications.
- Provide a functional verification by executing the test cases to determine the discrepancies between the embedded system model and its specification.

As shown in Fig. 1 our approach starts with a semi-formal use case specification of the System-under-Verification (SuV). The common textual use case descriptions [4] are extended by additional fields to cover constants, like message types or time delay constants. During the parsing of the use case specification we deal with typical natural language issues [8]. Therefore, we define a grammar and a lexical subset of the natural language to be used for specifying the use cases. This is done in collaboration with our industry partner, who has strong experience with common grammar structures and terms used for the specification. A list of guidelines is provided to keep the stakeholder to the given grammar structure and focus on terms from the lexical subset. It is not mandatory for the stakeholder to stick to these guidelines, although it decreases the required user interaction significantly. The interactions are also decreasing with the number of processed requirements as in the case of a missing term, which requires the user to specify the type of this unknown term. This decision is remembered the next time this term is analyzed without the need for an interaction by the user. After the parsing, the specification document is corrected and it is used as the input for the SystemC testbench generator, which generates the specification-based test cases. During a SystemC simulation these test cases are applied to the SuV to check if it corresponds to the specification. Output messages

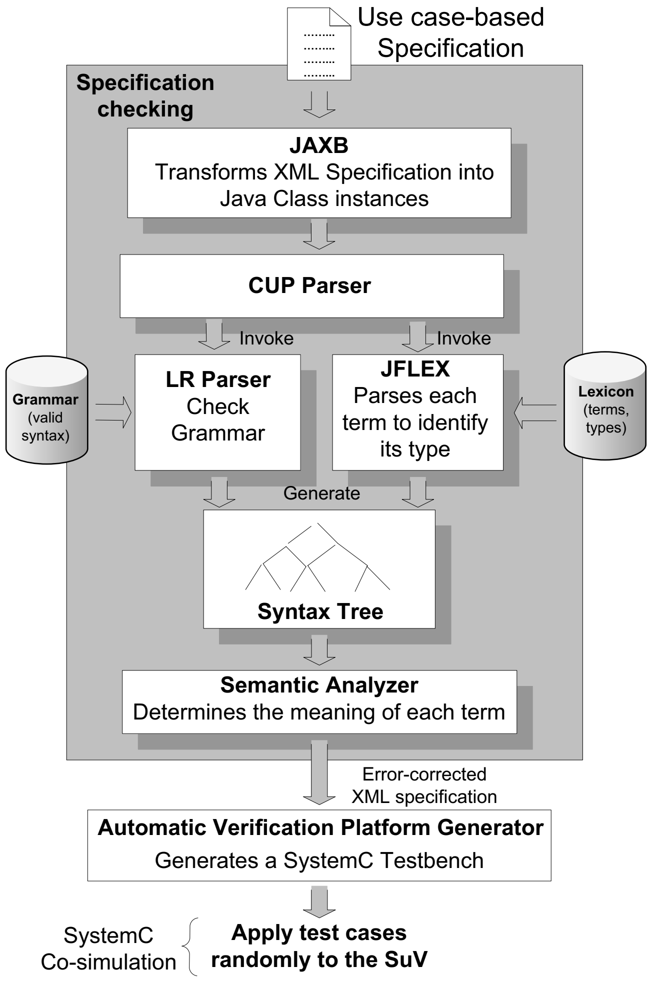
Use case-based Specification
Specification checking
JAXB
Transforms XML Specification into Java Class instances
CUP Parser
Invoke
Invoke
LR Parser
Check Grammar
JFLEX
Parses each term to identify its type
Grammar (valid syntax)
Lexicon (terms, types)
Generate
Syntax Tree
Semantic Analyzer
Determines the meaning of each term
Error-corrected XML specification
Automatic Verification Platform Generator
Generates a SystemC Testbench
SystemC Co-simulation
Apply test cases randomly to the SuV

**Fig. 2** Our implementation uses JAXB to extract data from the specification. The CUP Parser invokes the LR Parser and JFLEX to generate a syntax tree [2]. The semantic analyzer uses the syntax tree to provide the input for the test case generator.

convey information on the test progress, the test coverage as well as the test results to inform the verification engineer on-line about the current status of the simulation-based verification.

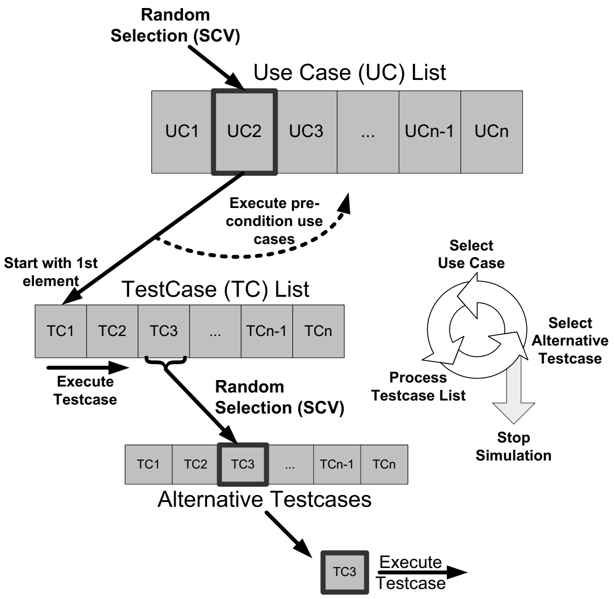

**Fig. 3** Proposed algorithm for the random selection of the generated test cases during a SystemC simulation.

## 4 Implementation

Figure 2 shows the implementation of our approach. JAXB [23] is used to generate Java classes and fills the instances of these classes with information from the XML-based use case specification. These instances are analyzed by the Java CUP parser [12], which invokes JFlex [17] to identify the type of each term. The CUP parser uses an LR-Parser [19] to check the grammar and generates a syntax tree [2] from each phrase stored in the use case instances. This is used by the semantic analyzer to determine the meaning of each term and to generate the error-corrected XML specification. Finally, the Automatic Verification Platform Generator uses this XML specification to generate the SystemC testbench.

The SystemC testbench selects and executes SystemC test cases during a simulation and consists of the two threads: random test case selection and test execution. The algorithm of the random test case selection thread is shown in Fig. 3. The entire process in Fig. 3 is reiterated until each use case has been selected by a user-specified number of times or the simulation is stopped by the verification engineer. In each iteration our algorithm uses SCV constructs to randomly select a use case from the list of use cases. This list is generated each time the Automatic Verification Platform Generator reads the error-corrected XML specification input file and generates the SystemC testbench module. Each use case may contain a list of predecessor use cases. These are defined in the use case's precondition statement and are executed before the current use case is processed. Each use case contains a list of test cases, which correspond to the steps in its scenarios. Each test case may also contain a list of alternative test cases specified in the extension scenarios. When a use case is processed our algorithm goes through its test cases sequentially starting at the first element in the list. For each test case it determines if the test case has a list of alternative test cases. If so, it uses the SCV constructs to randomly select a test case from the list for execution. Otherwise, the current test case is selected and executed by the test execution thread, which generates the stimuli, estimates and stores the SuV's internal state, checks the SuV response and prints the test case's name and status for verification reporting. The test execution thread is a verification state machine generated from the input XML use case specification as shown in Fig. 4. For each step in the use case scenario, which corresponds to a test case, the verification state machine contains a case block to execute this step. The case block *SET_UP_TAG_1_RECEIVES_ACTION* in Fig. 4 corresponds to the specified use case step

Tag receives Query Message with matching SL Flag

from Fig. 5 and generates the corresponding stimuli to apply this step to the SuV. The case block *SET_UP_TAG_10_TRANSMITS_ACTION* checks the system response at step

Tag transmits 16bit Random Number

from Fig. 5. The functions marked as grey-tone are the corresponding transactor functions. A transactor component is also automatically generated by our methodology and is inserted between the SystemC testbench module and the SuV to map the test cases to the SuV. Since the interface of the SuV can change the transactor is adapted by the verification engineer to connect it to the interface of the SuV. The mapping of the transaction-level test cases to the SuV's interface would go beyond the topic of this paper and is not explained here any further.

## 5 A Case Study of an Radio-Frequency Identification Controller

To demonstrate our methodology we have implemented it in the HW/SW co-design tool *SyAD*® (System Architect Designer) [14]. *SyAD*® enables the development of system-level HW/SW co-designs and supports a multi-language and multi-level co-simulation framework of SystemC, VHDL, VHDL-AMS and MATLAB Simulink. As a case study we have considered a use case-based specification of an Radio-Frequency Identification controller (= RFID tag) state machine provided by our industrial cooperation partner. The specification document is derived from the controller state diagram specified in the EPCGlobal Class-1 Generation-2 UHF RFID protocol for communications [6]. The use case-based specification document covers the entire tag state diagram (see Fig. 6.19. in [6]) and encompasses 53 use case scenarios. Fig. 5 shows a small excerpt from the use case-based specification document. We applied our methodology implemented in *SyAD*® to the entire use case

```
int testbench::test_execution(){
 switch(state){
  case  SET_UP_TAG_1_RECEIVES_ACTION :
   transmit_Message->set_value(QUERY_MESSAGE, SL_FLAG);
   send_to_DUT(transmit_Message);
   break;

  case …

  case  SET_UP_TAG_10_TRANSMITS_ACTION :
   message *received_message;
   receive_from_DUT(received_message, PC_EPC_CRC_MSG);
   if(check_message(received_message))
    return TB_PASSED;
   else
    return TB_FAILED;
   break;
  }
}
```

**Fig. 4** Automatically generated source code of the test execution thread.

***Name:*** *Set up Tag*
***Description:*** *Use case accessed when tag enters the Reader field.*
***Scope:*** *UHF RFID Tag (=Tag).*
***Primary actor:*** *Interrogator (=Reader)*
***Precondition/Trigger:*** *Tag (re)-enters the Reader Field*
***Main Success Scenario:***
 *1. Tag receives Query command with matching SL Flag from Reader*
 *...*
 *10. Tag transmits 16bit Random number*
 *11. Tag exits use case and goes to "Reply Tag" Use Case*
***Alternate Flows:***
 *1a. Tag receives Select command from Reader*
 *...*
***LocNonfunctional Requirements:***
 ***Timing Constraints:*** *Step 1 until step 11 shall be done within t1.*

**Fig. 5** Use case derived from the protocol specification of an RFID controller state machine.

specification and discovered 6 syntax errors (due to invalid grammar and missing verbs and articles) and added 8 unknown terms to the lexicon during the parsing steps. In a next step our Automatic Verification Platform Generator generated the SystemC testbench module consisting of 131 test cases derived from the use case specification document. Fig. 6 demonstrates the results of the simulation of the SystemC testbench with the SuV for 5, 10, 15 and 20 iterations of the SystemC test case selection algorithm from Fig. 3. Use case 1 (UC 1) is executed most of all, since it is in the precondition list of all other use cases. In contrast, UC 5 is less often executed since it does not occur in the precondition list of any other use case. The ratio of executed use case scenarios to the total number of use case scenarios specifies the covered amount of the specification by the simulated test cases. The left diagram in Fig. 3 shows a comparison of the number of identified errors and the verified portion of the specification for 5, 10, 15 and 20 iterations. A 80% functional coverage de-

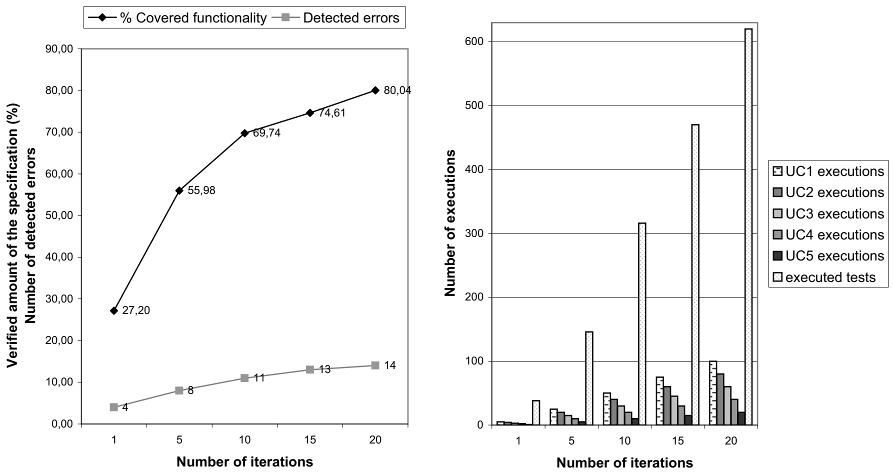

**Fig. 6** Simulation results for 5, 10, 15 and 20 iterations of the proposed test case selection algorithm applied to the RFID controller model.

tects 14 errors and requires 20 iterations of the test case selection algorithm, which results in more than 600 executed tests as illustrated in Fig. 3

## 6 Conclusion and Further Work

In this paper we presented a novel functional verification methodology for embedded system designs. The methodology supports both the correction of errors in the specification document and the automated test case generation from the specification. The test cases are used to verify whether the system model fulfills its specification (=functional verification) and close the gap between the specification and the design.

Our approach focuses on textual case-based specifications, which are suitable for black-box test case generation. We used a case study based on the semi-formal specification of a higher class RFID controller to demonstrate and prove our methodology. We showed that our methodology can be used to correct the specification document and to automatically generate SystemC test cases, which are executed randomly during simulation to determine the discrepancies between the design and its specification. As a further step we plan to improve the verification reporting by introducing functional coverage monitors into our design flow. This provides on-line information on how much functionality has been verified.

## References

1. J.R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, August 1996.
2. S. Bird, E. Klein, and E. Loper. *Introduction to Natural Language Processing*. 2001.
3. L.C Briand and Y. Labiche. A uml-based approach to system testing. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 194–208, London, UK, 2001. Springer-Verlag.
4. A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2001.
5. A.L.L. de Figueiredo, W.L. Andrade, and P.D.L. Machado. Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006.
6. EPCGlobal. EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz 960 MHz, 1.0.9.
7. A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Application of Linguistic Techniques for Use Case Analysis. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 157–164, 9-13 Sept. 2002.
8. Centre for Language Technology. Controlled Natural Languages, 2007.
9. M. Friske and H. Schlingloff. Von use cases zu test cases: Eine systematische vorgehensweise. 2005.
10. P. Fröhlich and J. Link. Automated test case generation from dynamic models. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 472–492, London, UK, 2000. Springer-Verlag.
11. J. Hartmann, C. Imoberdorf, and M. Meisinger. Uml-based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70, New York, NY, USA, 2000. ACM.
12. S. Hudson. Cup - lalr parser generator for java, 2007.
13. Y. JinShan, L. Tun, and T. QingPing. The Use of UML Sequence Diagram for System-on-Chip System Level Transaction-based Functional Verification. In *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, volume 2, pages 6173–6177, 21-23 June 2006.
14. S. Kajtazovic, C. Steger, A. Schuhai, and M. Pistauer. Automatic generation of a verification platform for heterogeneous system designs. In *Advances in Design and Specification Languages for SoCs - Selected Contributions from FDL'05*, 2005.
15. S. Kansomkeat and W. Rivepiboon. Automated-generating test case using uml statechart diagrams. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 296–300, , Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.
16. Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test cases generation from uml state diagrams. *Software, IEE Proceedings -*, 146(4):187–192, Aug 1999.
17. G. Klein. Jflex - the fast scanner generator for java, 2007.
18. Cadence Labs. The Transaction-Based Verification Methodology. Technical report, 2000.
19. Naumann and B.G. Lang. *Parsing*. 1994.
20. C. Norris and S. Swan. A tutorial introduction on the new systemc verification standard. Technical report, 2003.
21. A. Piziali. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004.
22. M. Riebisch and M. Hubner. Traceability-driven Model Refinement for Test Case Generation. In *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, pages 113–120, 4-7 April 2005.
23. Sun. Java architecture for xml binding (jaxb), 2003.
24. Q. Zhu, R. Oishi, T. Hasenawa, and T. Nakata. System-On-Chip Validation using UML and CWL. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 92–97, 2004.