Integration of Requirements Engineering and Test-Case Generation via OSLC

Bernhard K. Aichernig*, Klaus Hörmaier†, Florian Lorber*, Dejan Ničković‡, Rupert Schlick‡, Didier Simoneau§, and Stefan Tiran*‡

* Institute for Software Technology, Graz University of Technology, Austria
{aichernig, florber, stiran}@ist.tugraz.at

† Infineon Technologies, Austria AG. Villach, Austria
klaus.hoermaier-ee@infineon.com

‡ AIT Austrian Institute of Technology, Department of Safety and Security, Vienna, Austria
{dejan.nickovic, rupert.schlick, stefan.tiran.fl}@ait.ac.at

§Dassault Systèmes, Plouzané, France
didier.simoneau@3ds.com

***Abstract*—We present a requirement-centered analysis and testing framework that integrates methods and tools for capturing and formalizing textual customer requirements, analyzing requirements consistency, generating test cases from formalized requirements and executing them on the implementation model. The framework preserves a fine grained traceability of informal and formal requirements, test cases and implementation models throughout every step of the workflow.**

We instantiate the framework with concrete tools that we integrate via a file repository and Open Services for Lifecycle Collaboration (OSLC). The standardized integration ensures that the framework remains generic – any specific tool used in our instantiation can be replaced by another one with compatible functionality.

We apply our framework on an industrial airbag control chip case study that we use to illustrate step-by-step our requirements-driven analysis and test methodology.

***Index Terms*—Requirements engineering, test-case generation, requirement consistency, interoperability, traceability**

I. Introduction

Requirements engineering constitutes the first step in the system development process. As a consequence, the formulation, documentation and management of system requirements has a decisive effect on the quality of the designed system. Failure to identify and document important aspects of the system and defining ambiguous or conflicting requirements can result in the development of poor or even incorrect designs. Requirements engineering develops methodologies based on systematic usage of techniques and tools that help the establishment of complete, consistent and relevant system requirements.

For safety-critical applications, correctness evidence for designed systems must be presented to the regulatory bodies (see for example the automotive standard ISO 26262 [ISO09]). It follows that verification techniques represent another key ingredient in the design of complex systems and must be used to provide evidence that the system meets its requirements. Roughly speaking, one can distinguish between *static* and *dynamic* verification methods.

Static verification usually refers to model checking, static analysis and theorem proving and aims at proving system correctness by analyzing the system description without its actual execution. Even though they are powerful, static verification techniques have their limitations. Despite tremendous improvements in the past decades, model checking still suffers from scalability problems, static analysis often returns many false positives resulting from excessive system over-approximations and theorem proving requires considerable expert user knowledge and manual effort.

In contrast, dynamic verification refers to techniques such as testing and monitoring, that require the effective execution, or at least simulation, of the system. Testing remains the most common practice in industry for gaining confidence in the design correctness. Due to the finite number of experiments, testing cannot prove the absence of errors in the system. However, it is a pragmatic and effective technique for catching bugs.

It follows that requirements engineering, static verification and testing all play a crucial role in the system design and its verification. The appropriate integration of the methods and tools coming from these complementary fields would greatly improve the quality of the design process. However, requirements engineering, static verification and testing are studied by different communities that insufficiently exchange their ideas and experiences. In addition to cultural differences between these communities, isolated development of methods and tools prevents their effective integration into a unified framework. The European ARTEMIS project MBAT[1] addresses this problem and aims at integrating different requirements engineering, analysis and testing tools into the so-called MBAT Reference Technology Platform (MBAT RTP).

[1]Combined Model-based Analysis and Testing for Embedded Systems.

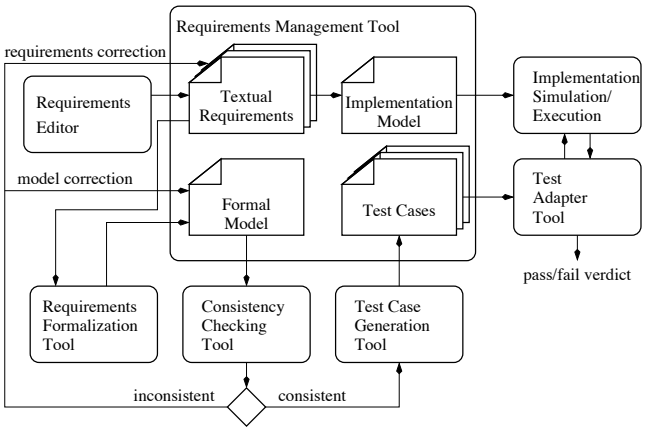

Fig. 1. Overview of the generic framework.

Following the MBAT vision, we develop a novel requirements-driven analysis and testing framework. The framework, illustrated in Figure 1, unifies and integrates methods and tools from requirements engineering, model checking, and model-based testing. In our framework, the starting point is the document containing textual customer requirements. The role of the requirements document in our framework is twofold – it is used to derive both the implementation of the system and its formal abstract model (specification). These tasks are usually separately done by two different teams. Requirement formalization tools are used to support the engineer in developing a formal model of the system from its informal requirements. The consistency checker tool, based on model checking techniques, is used to analyze whether the formal model admits at least one correct implementation. The failing result of this check indicates inconsistency in the requirements from which the formal model is derived. In that case, the requirements must be corrected and the process repeated. Alternatively, the inconsistency may also result from poor requirement formalization. In that case, it is the formal model that needs correction. The test-case generation tool creates a test suite from the consistent formal model. The test cases from the suite are executed on the system implementation via the test execution tool. All the artifacts created during the workflow, that is the (informal) textual requirements, formal models, test cases and implementation models, are handled by an integrated tool providing requirements management as well as an artifact repository.

We instantiate the framework with specific tools, shown in Figure 2. We structure the informal requirements into boilerplates with the support of the DODT tool from Infineon Austria [FMK+11], resulting in reduction of spelling mistakes, poor grammar and ambiguity. The resulting semi-formal requirements are transformed into formal *requirement interface* models that specify the intended behavior of the system. The MoMuT::REQs tool [ALNT14] developed by Austrian Institute of Technology and Graz University of Technology is used to (1) check the consistency of the formalized requirements; and (2) automatically generate test cases from requirement interfaces. We use MathWorks Simulink [Ong98] to develop the system implementation model, which we derive directly from the textual customer requirements. The test cases generated by MoMuT::REQs are executed on the Simulink implementation model via a test adaptor developed by Infineon Austria. The artifacts created in this workflow, including textual requirements, formal test and implementation models and test cases are managed in the SystemCockpit tool developed by Dassault Systèmes [Das]. This requirements management tool facilitates tracing formalized requirements, implementation model elements and test cases to the original textual customer requirements.

The integration of MoMuT::REQs and SystemCockpit is done via the MBAT Interoperability Specification (IOS) and the Open Services for Lifecycle Collaboration (OSLC) [Ope]. OSLC is a standard for linking systems and software engineering tools via web services. The OSLC integration preserves the generic nature of our framework. In particular, it facilitates replacement of a specific tool used in the framework instantiation with another one with the compatible functionality, as long as both comply with the OSLC standards. For instance, the requirements management tool SystemCockpit could be replaced by DOORS [IBM] without additional effort. The integration of the other tools is currently done via a file repository within SystemCockpit, where the work products are already stored and linked as OSLC resources.

We evaluate our requirement-centered analysis and test framework on a real-life industrial airbag case study provided by Infineon Technologies. In addition, we use the case study to illustrate every step in the framework workflow.

Organization. The rest of the paper is structured as follows: in Section II we introduce the case study, which serves as an illustrative example throughout the paper. In Section III, we present the stepwise formalization process, leading from textual customer requirements to formal models. In Section IV we provide technical details about the analysis phase of our workflow, i.e. the consistency check of the formal model. In Section V we describe the test case generation from formal requirements, followed by Section VI that illustrates how the tests are executed on the actual SIMULINK implementation. Then, in Section VII, we describe how we store our work products in the requirements management tool, facilitating traceability and also illustrate our tool integration via OSLC. In Section VIII, we discuss some insights we gained during the case study. Finally, in Section IX we set ourself in context to related work and in Section X we conclude our work and provide an outlook for future work.

II. Case Study Description

The case study of this paper revolves around an airbag control chip, which is the core component of a use case within the ARTEMIS project MBAT. The chip provides both the power electronics for actually deploying the airbag and the control logic interacting with the control CPU, supported by a so called safing engine. The safing engine double checks the validity of the commands sent from the CPU and can block deployment if the situation does not have the characteristics of

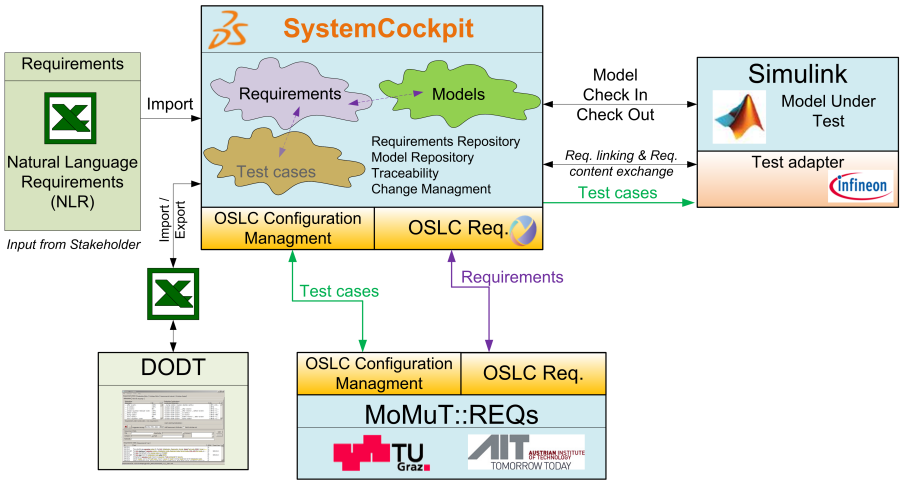

Fig. 2. Overview of the framework instantiation.

a crash. This is obviously safety-critical, as both, deployment of the airbag without a crash and no deployment of the airbag in case of a crash, may lead to hazardous events.

To prevent any malfunction of the safing engine, the basic functionality of the airbag is controlled by an internal state machine, consisting of seven different states. The airbag can only be triggered within a certain state and while a combination of several input signals is applied. The initial state of the airbag system is the RESET state (1). Apart from the initial condition, this state can also be reached whenever the airbag is reset by an external signal. Within the RESET state all functionality of the chip is deactivated. As soon as the external reset signal is turned off, the chip traverses to the INITIAL state (2). While in this state, self-diagnostic functions are active which test the integrity of the basic hardware functions. If an error is detected, a reset is enforced leading to a transition to the RESET state. After receiving a certain sequence of commands from the controller CPU via the Serial Peripheral Interface Bus (SPI), the controller chip moves to the DIAGNOSTIC state (3). Within the DIAGNOSTIC state, several additional tests are executed. Detecting an error in this state leads the system to the SAFE state (4), whereas a successful diagnosis leads the system to NORMAL state (5). While the system is in the SAFE state, it may not react to any inputs or trigger any outputs, until the system has been reset by an external signal. In the NORMAL state, the system is fully functional. It is reacting to all inputs and if they match certain conditions, the airbag can be triggered. The TEST state (6) can be reached from the NORMAL state via certain SPI commands. In this state, different output signals can be triggered directly via SPI commands to test functionality. The last state is the DESTRUCTION state (7), which allows the manual deployment of the airbags at the end of life of the airbag system.

III. From Textual to Formal Requirements

This section covers the second step in our methodology: requirement formalization. The first step, storing the requirements from the Excel sheet into SystemCockpit is skipped due to simplicity. The formalization is implemented via the tools DODT and MoMuT::REQs. We now present a subset of the airbag system's textual customer requirements, and then describe how these informal requirements are structured and finally formalized.

A. Textual Customer Requirements

The specification of the state machine consists of 39 textual requirements, that were provided by the customer. We illustrate our framework with three sample requirements from this specification.

- *R1*: There shall be seven operating states for the safing engine: RESET state, INITIAL state, DIAGNOSTIC state, NORMAL state, TEST mode, SAFE state and DESTRUCTION state.
- *R2*: The safing engine shall change per default from RESET state to INIT mode.
- *R3*: On a reset signal, the SE shall enter RESET state and stay while the reset signal is active.

B. Structuring Textual Requirements

We use the DODT tool to structure the informal requirements and semi-formalize them, in order to achieve a textual representation that relies on the same vocabulary and uses code words that provide additional semantics. This is done with a six step process that transforms natural language (NL) requirements into boilerplates (see Figure 3), that were developed within the ARTEMIS CESAR project [RW13]. Boilerplates allow the reduction of spelling mistakes, poor grammar or ambiguity. Additionally they help to capture the underlying system-related semantics.

In the first step the relevant sentences are selected, as NL requirements can consist of several requirements. Thereafter, the typographic errors are corrected by using the domain ontology concepts as a dictionary. The tool highlights variable names that only occur once, or are very close to other used names. In the above example, it highlights *INIT mode* because it is called *INITIAL state* in the rest of the requirements document and *SE*, that should be called safing engine. Now requirements

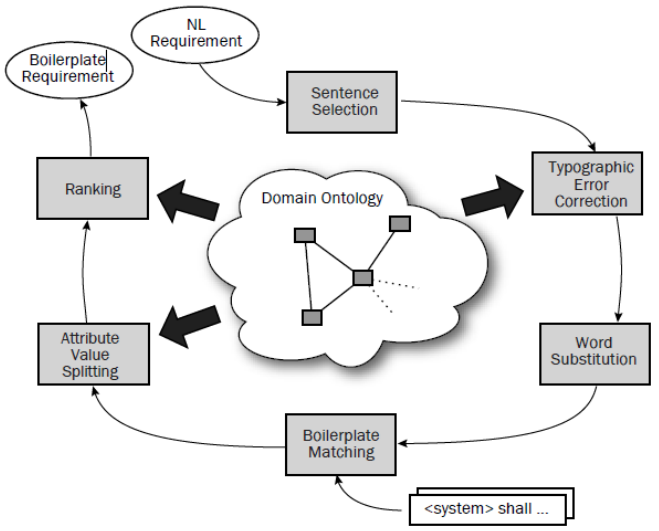

Fig. 3. Six-step Semi-Automated Conversion Process.

are transformed to find matching boilerplates. User-defined substitution rules are applied to replace expressions (like will or must) with synonymous fixed syntax elements (like shall). In the next step, the text between fixed syntax elements is assigned to boilerplate attributes and several adjacent attributes are split. Through these steps *R1* to *R3* were transformed into semi-formal requirements as depicted in Figure 4.

C. Formalizing Requirements

The semi-formal requirements are stored back to System-Cockpit and from there accessed by MoMuT::REQs, to be formalized in the *requirement interfaces* [ALNT14] modeling language. Requirement interfaces allow specification of open synchronous reactive systems that communicate over shared variables. The variables in requirement interfaces are partitioned into *input*, *output* and *hidden* variables. This variable partition enables modeling both the system behaviors and the interactions with its environment. A requirement interface consists of a set of *contracts*. A contract is a rule that defines an *assumption* about the external environment of the interface, and a *guarantee* that the system must satisfy if the assumption is met. In addition, every contract is associated to the set of customer requirements it was generated from. Note that we allow for an n:n relationship between customer requirements and formal contracts. Assumption and guarantee rules are expressed as relations between next (primed) and previous (unprimed) interface variables, and define the desired dynamic behavior of the system.

The 39 safing engine requirements from the case study were refined to 32 formal requirements. These formal requirements consist of 11 input variables, 5 output variables and 11 hidden state variables. We now illustrate the formalization of textual customer requirements *R1*, *R2* and *R3* into a requirement interface. The interface has a single Boolean input variable *reset* and a single output variable *state*. It consists of three contracts *FR1*, *FR2* and *FR3*, that are associated to textual requirements *R1*, *R2* and *R3*, respectively. The resulting contracts in the requirement interface are shown in Listing 1.

Listing 1. Requirement interface contracts formalizing the textual customer requirements *R1-R3*.

```
FR1: assume true
     guarantee state'=RESET or state'=INIT
               or state'=DIAG or state'=NORM
               or state'=TEST or state'=SAFE
               or state'=DESTR

FR2: assume state=RESET
     guarantee state'=INIT

FR3: assume reset'
     guarantee state'=RESET
```

Modeling with requirement interfaces encourages the *multiple viewpoint* compositional approach to the system specification. In this approach, each specific view of the system consisting of a subset of requirements is formalized by a separate requirement interface. Different views of the system are combined using the *conjunction* operation. Intuitively, a conjunction of two requirement interfaces is another requirement interface that subsumes all the behaviors of both interfaces. It follows that the overall system specification is given as the conjunction of requirement interfaces modeling its different views.

For the technical definition of requirement interfaces and the conjunction operation, we refer the reader to our technical report [ALNT14].

IV. Consistency Checking of Formal Requirements

Requirements validation is an important step in the requirements engineering process. It consists in studying potential conflicts in requirement documents and taking possible corrective actions. In this section, we describe the third step of our workflow, the automated consistency check on our requirements interfaces, realized by the MoMuT::REQs tool.

MoMuT::REQs takes as input a requirement interface and explores the state space of the model searching for a set of conflicting contracts. The tool implementation is based on bounded model checking [BCC+03] techniques and enables finding potential inconsistencies in the model up to a bound k pre-defined by the user. It expresses the bounded consistency check problem as an Satisfiability Modulo Theories (SMT) formula. This formula encodes the question whether for an arbitrary sequence of input variable valuations of size k, there exists a sequence of output and hidden variable valuations of the same size that satisfies all the contracts of the requirement interface. In the failing case an inconsistency is found. MoMuT::REQs uses the SMT solver Z3 [dMB08] from Microsoft to solve the above problem.

MoMuT::REQs enables both *monolithic* and *incremental* consistency checking. The monolithic algorithm takes as input the overall specification of the system as a single requirement interface and checks its consistency. This is a very expensive approach, as the complexity of the consistency check exponentially increases with the number of checked contracts. In contrast, the incremental algorithm exploits the

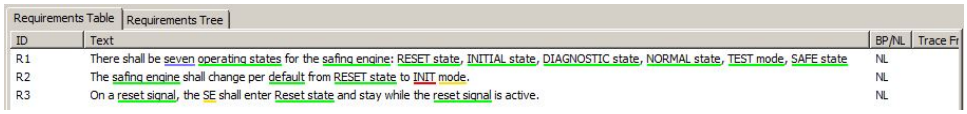

Fig. 4. Highlighting of variables and fixed syntax elements in DODT.

compositional structure of the formal system specifications in which different system views are expressed as separate requirement interfaces. Finding an inconsistency in a subset of system views implies that the entire system specification is inconsistent. The incremental algorithm enables much more efficient detection of conflicting requirements. Additionally, MoMuT::REQs supports the consistency check of subsets of the contracts within a requirements interface, allowing to use the incremental approach, even if the separation into different interfaces was not possible.

For the sake of scalability we therefore propose to incrementally check the consistency of various subsets of requirements, to identify and correct all inconsistencies that can be found in that way, and do the expensive monolithic check afterwards. Applied to the safing engine requirements during and after the formalization process, the incremental approach was able to detect two major inconsistencies (real contradictions) and several minor ones (under specifications or formalization faults).

When an inconsistency is detected, MoMuT::REQs identifies the minimal set of contracts that cause the inconsistency. The tool uses QuickXPlain, a standard conflict set detection algorithm [Jun04]. The minimal set of inconsistent contracts can be traced back to the associated textual customer requirements and the engineer can correct the conflicting requirements. Figure 5 depicts the MoMuT::REQs tool that displays the minimal set of conflicting contracts in a requirement interface.

The consistency checking algorithms for requirements and their properties are formalized and presented in details in our technical report [ALNT14].

We have applied the consistency check to the requirements formalized in Section III-C. The MoMuT::REQs tool found that the formal requirements are inconsistent with the minimal conflict set $\{FR2, FR3\}$. The minimal conflict set provides useful debugging information to the designer that enables easier identification of the source and causes of the conflict. A conflict can arise either from a poor specification of the textual customer requirements or from their incorrect formalization. In our example, the conflict is due to the following scenario. When the reset signal is triggered while the system is in the reset state, *FR2* requires the system to be in the initial, while *FR3* requires it to be in the reset state in the next step.

Listing 2. Repairing the inconsistency of *FR2*.

```
FR2.1: assume state=RESET and not reset'
       guarantee state'=INIT
FR2.2: assume state = RESET
       guarantee state' = RESET or state' = INIT
```

The conflict can be resolved in two ways, both illustrated in Listing 2. The first resolution consists in changing the textual customer requirement *R2* and adapting accordingly its formalization. In fact, *R2* is a potentially ambiguous requirement

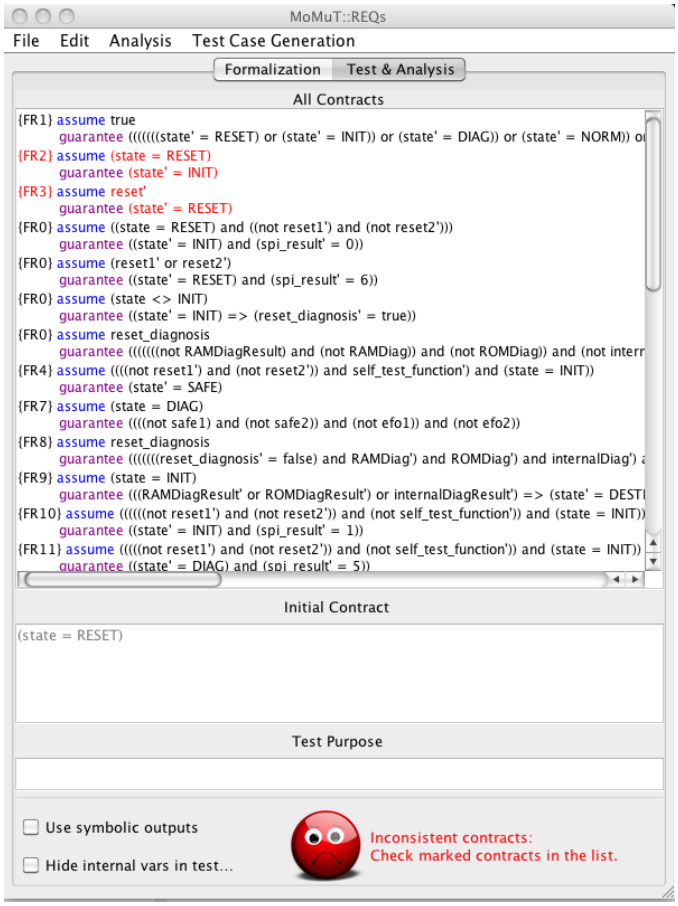

Fig. 5. MoMuT::REQs displaying the minimal inconsistent contract set.

because it describes the correct system behavior when it is in the reset state but does not refer to the reset input signal at all. It makes sense that a reset signal is never triggered when the system is already in the reset state, hence we make this assumption explicit in *R2* and in its formalization *FR2.1*. On the other hand *FR2.2* corrects *FR2* by making a more direct translation of the original textual customer requirement *R2*. In fact, *R2* does not explicitly define the conditions required for making a transition from the reset to the initial state. We correct *FR2* with *FR2.1* by making this observation explicit – the update of the next state is chosen non-deterministically. In this scenario, the designer needs to ensure that there exists another requirement in the specification that defines more precisely when the state change is allowed to occur.

This example shows that choosing the right conflict resolution of inconsistent requirements may not be a straight-forward task. The corrections may involve changes in either textual customer requirements, formalized requirements or both. This process requires interactions between the verification engineers

and the supplier that provides the textual customer requirements in order to identify the exact causes of conflicts and to find a solution that satisfies the intended meaning of the specification. After repairing the inconsistency, the tool reports that all formalized requirements are consistent, as illustrated in Figure 6.

V. Generating Tests from Formal Requirements

In this section, we present the next step in our framework's workflow - test case generation from formalized requirements. In addition to consistency checking, MoMuT::REQs also generates test cases from requirement interfaces. A test case is an experiment executed on the implementation model by a test adapter. We represent a test case generated from a requirement interface as a total sequential function that takes as input the history of previous input and output observations and returns either the next input value or a **pass** or **fail** verdict.

Listing 3. Two steps of a testcase leading to to the Init state.

```
STEP 0
INPUT spi_command = NONE
INPUT reset1 = true
INPUT overvoltage = false
OUTPUT safe1 = false
OUTPUT efo2 = false
HIDDEN operating_state = RESET

STEP 1
INPUT spi_command = NONE
INPUT reset1 = false
INPUT overvoltage = false
OUTPUT safe1 = false
OUTPUT efo2 = false
HIDDEN operating_state = INIT
```

When generating a test case, MoMuT::REQs requires a *test purpose* in addition to the requirement interface. The test purpose is given in the form of a predicate that defines a set of target states that the implementation model is intended to reach during testing. Test case generation in MoMuT::REQs is based on bounded model checking techniques. Given a requirements interface, a test purpose and a predefined bound k, the tool first aims at finding a sequence of inputs that guides the system towards the state specified by the test purpose. This so-called reachability problem is encoded as a satisfiability problem, which is again solved using the Z3 SMT solver.

Similarly to the consistency checking, MoMuT::REQs implements both monolithic and incremental test-case generation algorithms, in which that latter exploits the compositional structure of the specification. In particular, the incremental algorithm enables generating a test case for a single view of the system. Once created, the test case can be extended to a test case for the overall system, by adding the constraints defined by the remaining requirement interfaces modeling other views

Fig. 6. MoMuT::REQs reporting that the overhauled requirements are consistent.

of the system. The incremental test-case generation procedure is more efficient than its monolithic counterpart.

Test cases generated by MoMuT::REQs encode a sequence of input vectors and constraints relating input, output and hidden valuations that the system must satisfy and that are defined by the requirement interface. In the case of deterministic specifications without hidden variables, a test case has a simpler representation in the form of a sequence of input and output valuations. MoMuT::REQs test cases are not adaptive – the tool fixes the input vector and uses the requirement interface specification as a *monitor* that observes the outputs of the implementation model to decide whether its execution satisfies or violates the specification. Such a test case guarantees that for an arbitrary input vector, the resulting **pass/fail** verdict is correct. On the other hand, non-adaptive test cases cannot guarantee that the test purpose is always reached, due to the potential implementation freedom allowed by the output constraints in the specification.

Details on the test-case generation technique for requirement interfaces can be found in our technical report [ALNT14].

We generate test cases from the requirement interfaces that formalize textual customer requirements of the safing engine case study. The safing engine specification does not contain non-determinism in output or hidden variables. It follows that MoMuT::REQs generates test cases in the form of input/output sequences.

Listing 3 shows the test case that specifies the correct transition from the reset to the initial state, as defined by the specification. We simplify the presentation of the test case by projecting away variables that do not play a role in this scenario in order to simplify presentation.

VI. Executing Tests on the Implementation Model

For the airbag system case study, we realized the implementation, simulation and execution of our implementation model via MATLAB Simulink. The test cases generated in the previous section are executed on this model.

An extract of the Simulink model is illustrated in Figure 7. It shows the internal state machine of the safety engine. To execute the abstract test cases on the implementation model, a test adapter was implemented. It translates the abstract input labels, such as *reset* or *spi_command*, into the actual input signal implemented in the system. The outputs produced by the implementation model are then transformed back to the abstract test results, which can then be compared to the test oracle. The test adapter does not only translate from one syntax (e.g. test case output) to another one (e.g. target language). It also translates test driven test cases into time dependent test cases. This is done via the workflow depicted in Figure 8.

For the refinement of the abstract test cases, each signal is considered an object with assigned properties and methods. The methods assigned to the signals include the procedure to translate the signal from one to another abstraction level. During the translation, timing information is added and abstract test cases are translated into sequences. Information which is

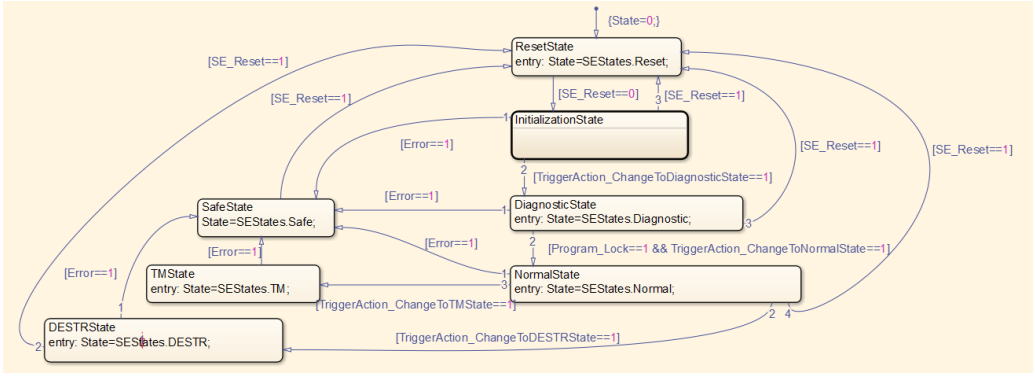

Fig. 7. The Simulink model of the state machine described in Section II.

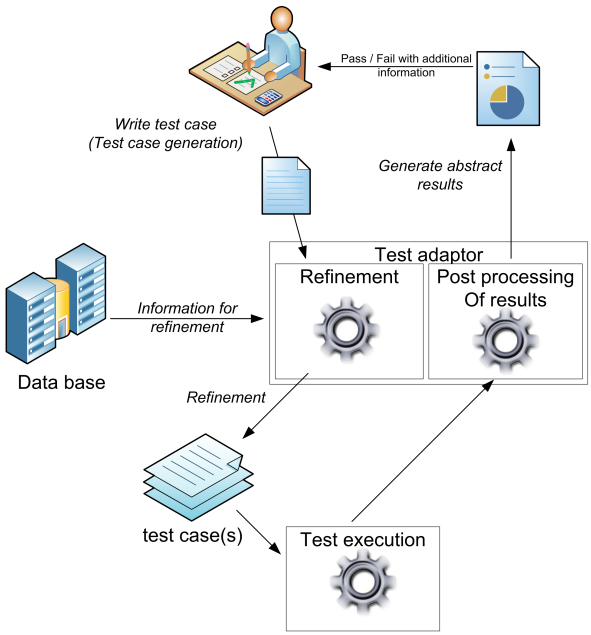

Fig. 8. Test-case execution process.

not included in the abstract test cases, but is important for the execution of the Simulink model (e.g. timing information) is retrieved from the product design specification.

The translated test cases are executed on the Simulink model by running the simulation using the newly configured signals. The simulation results cannot be directly compared to the test oracle on the higher abstraction level. Therefore the test adapter abstracts the results to match the high abstract level.

Currently the test driver also handles the timings, since the delay between inputs and outputs varies depending on the inputs. It searches for the input that needs the longest delay until it is processed, and waits this delay, before it reads the produced outputs.

VII. Managing and Tracing Requirements, Models and Tests

The core component of our approach is a requirements management system implementing an OSLC provider. It is used as central repository not only for the natural language requirements but also for any model involved in the workflow as well as the traceability links between those elements. In our instance of the toolchain we use a tool called SystemCockpit [Das] developed by Dassault Systèmes. It is an experimental platform initially developed within the ARTEMIS CESAR project and then continued in the ARTEMIS MBAT project. In the context of the MBAT project the concrete instance of OSLC RM is referred to as MBAT/IOS. This also includes extensions which enable services for navigation and creation of the traceability links using the modern web technologies HTML 5 and AJAX.

As depicted in Figure 1, the requirements tool is supposed to store and link the customer requirements, the semi formal requirements, the formal requirements, the test cases and the implementation model. SystemCockpit provides a way to build these IOS links via an internal web interface, as well as directly via the connected tools.

The navigation is done using tree and 2D graphs representations of the models, in which the user can easily follow traceability links. The user interface can also display a traceability graph between user selected models. Figure 9 shows an example of such a graph: this view shows a partially expanded model, customer requirements and formalized requirements, illustrating the traceability links between the currently exposed elements. Bold links are pseudo links used to indicate that actual links exist between some elements within the subtrees of the source or destination. As the user expands these sub-trees, these actual links are displayed. This provides a convenient visual help to perform coverage analysis.

OSLC Integration.

OSLC is an open community that provides specifications for tool integration and communication. These standards allow easy data exchange as well as the option to replace tools with others that comply to the same OSLC sub-specification(s). In OSLC, data is linked via the HTTP protocol, and can be accessed and modified through creating, retrieving, updating and deleting (CRUD) commands.

The OSLC standard defines two types of tools: OSLC providers and OSLC consumers. Providers are meant to store and provide data, allowing the consumers easy access to browse, create and retrieve the data. One of the main advantages of SystemCockpit is its compliance to the OSLC

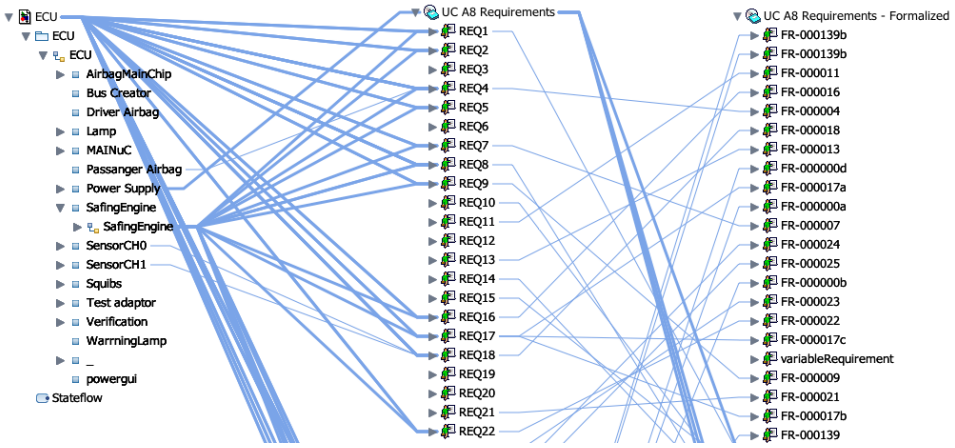

Fig. 9. Traceability view of SystemCockpit, illustrating the links between customer requirements, formal requirements and the implementation model.

standard, by operating as an OSLC provider. While it supports basic storage and access of work products via a file repository, it stores these work products as OSLC resources and allows the creating, retrieving, updating, deleting and linking of data via OSLC. All OSLC resources are equipped with a uniform resource identifier (URI). Upon creation of a formal requirement in SystemCockpit via the tool MoMuT::REQs, it is immediately linked to its associated customer requirements by the *satisfies*-link, using the URIs of the requirements. A second link, as displayed in Figure 9, is created between the customer requirements and the implementation model parts.

MoMuT::REQs already uses this advanced communication as an OSLC consumer, by directly importing customer requirements, exporting the formalized requirements and linking them, all via OSLC. It is part of the plans for the last year of the ongoing MBAT project, to also fully integrate the other tools into the OSLC workflow.

VIII. Lessons Learned

Separation of concerns applies to requirements engineering. Dividing the requirements document into separate sections, covering different views / aspects of the system, improves the quality and readability of the document, enhances the compositional properties of the requirements, and ultimately ensures scalability during the consistency checks and the test case generation. By generating a unique requirements interface for each view and individually applying the test and analysis methods on them, we considerably increase the performance of the algorithms. In particular, this allows the verification of systems that could not be processed at all by our tools, if modeled monolithic.

Specifying consistent requirements is hard. Even though for the scenario described in this article we used a mature version of a requirements document, which describes a system that has already been implemented, there are still ambiguities that can result in their inconsistent interpretation. In our case study, the definitions involving the usage of the reset signal are ambiguous. Requirements describing what should be done on the occurrence of the reset signal often conflict with requirements describing the normal behavior. For creating formal contracts for a test model, it is necessary to explicitly state in the assumption, that the reset signal must not occur in order to operate in normal behavior mode.

We have learned that the consistency check provided by MoMuT::REQs turned out to be a powerful tool in order to detect inconsistencies in the requirements.

Textual modeling languages are perspicuous. The main difference to our experiences in former projects, besides finally having an integrated tool chain, was caused by the fact that we switched from graphical specification models (UML state charts and timed automata) to a textual representation. Initially, we thought that we would gain a more formal and precise model, at the cost of comprehensibility. Yet it turned out that reading or writing one single requirement is a very straight forward task. Already 30 minutes after we first suggested a contract based requirements formalization during a project meeting, our industrial partner started to formalize the requirements on his own.

Keeping the traceability link requires direct modeling. When creating a formal test model from requirements, it is often tempting to include other implicit knowledge of the system that is not explicitly stated in a specific requirement or to assign slightly different names to the variables in the formal requirements. This diminishes the benefits from the traceability, since the natural link between formal and informal requirements is lost. Keeping this link as tight as possible makes traceability a lot easier in both directions, from the requirements down to the components and test cases, as well as from the verification results backwards to the requirements. An important aspect of the traceability and navigation capabilities is their support in keeping the different artifacts consistent.

Requirement interfaces are easy to maintain. Requirement interfaces were designed with compositional properties in mind. In particular, requirement interfaces are naturally composed by conjunction, following the structure of the standard requirement documents. We have learned from the case study that this property of requirement interfaces enables stepwise and compositional formalization of requirements. The compositional properties also supports in case of a change request: whenever a requirement is updated, only the asso-

ciated contracts need to be revised. Then all test cases that were constructed from these contracts need to be generated and executed again, to see whether the implementation still conforms to the updated requirements.

Tool integration reduces reiteration times. The lack of integration between tools often causes considerable overheads in the design process. The integration of the tools used in the case study with SystemCockpit enabled instant access to all artifacts used in the workflow and enabled traceability between them. This resulted in an increase of efficiency, in particular it allowed to quickly track inconsistent requirements.

IX. Related Work

As pointed out by Graham [Gra02], tighter integration between requirements engineering and verification is a major challenge in industry and thus considerable effort has been invested in this direction in the recent past.

For instance, Post et al. [PSM+09] use formalized requirements for implementation level verification, by integrating *assume* / *assert* statements in the source code, that are directly derived from the requirements. Contrary to our work, this is a white-box method, while we work on black-box testing.

Uusitalo et al. [UKKD08] define best practices for increase the integration between testing and requirements engineering. While the proposed practices advocate establishing a connection between requirements and generated test cases, the requirements are not explicitly used for the test case generation. This is contrary to our approach, that directly generates tests from the formalized requirements.

Sabaliauskaite et al. [SLE+10] provide a study about possible issues in aligning requirements engineering with verification. They conclude their article by naming four major issues: The lack of suitable tools for managing requirements and test cases (pointing out the poor interfaces between different tools), the lack of updates to requirements once they are stored, missing traceability between artifacts, and insufficient communication within the companies. Our methodology covers the first three of these issues, by providing an integrated tool chain and OSLC connections between the requirements and the other artefacts. These connections grant traceability and encourage to updated connected requirements at change requests.

The ARTEMIS project CESAR [RW13] defined a Reference Technology Platform (RTP), a framework facilitating tighter integration between different tools and enhanced traceability. The outcomes of MBAT, and therefore the tool integration described within this paper, partially rely on the results of CESAR. A description of the overall MBAT analysis and test methodology has been provided by Nielsen [Nie14]. Other instances of the MBAT methodology were presented by Kacimi et al. [KEOS14] showing another automotive case study and Dierkes [Die14] discussing an aerospace case study.

Heitmeyer et al. [HJL96] check consistency between requirements and designs. Contrary to our work, they define consistency between two types of models, that are consistent if they exhibit the same behavior, whereas we define consistency over formal requirements, that are consistent if they do not contain any contradictions.

Our work is also related to several results developed within the EU project SPEEDS. An overview of this project is given by Passerone et al. [PDBH+09]. One of the main focuses of SPEEDS was to improve interoperability of tools. Contrary to the IOS of MBAT that enables direct communication between all tools, SPEEDS introduced a central engineering bus that handles the communication among all tools and the model repository. The model-bus enforces a strict communication standard among all tools, while the direct communication between the tools enables a higher flexibility, but also poses the risk to loose interoperability. However, the MBAT IOS enforces a standard protocol, to reduce that risk.

Another development within SPEEDS was the concept of *heterogenous rich-components*, a concept that also uses contracts in order to specify behavior [BFMSV08], [BRR+10]. Yet they do not link requirements directly to contracts, but only to components while we preserve a more fine grained traceability, including links from requirements to contracts, from contracts to test cases and from requirements to components.

In the past, we also generated test cases from formalized requirements [ABJ+14], [AJK13], [ALN13] using graphical representations (UML state charts and timed automata) that only provided limited abilities to link them to the requirements.

X. Conclusion and Future Work

Conclusion. We presented a tool-supported methodology that effectively combines requirements engineering, analysis and test case generation and showed one successful application on an industrial case study. The main benefit of the presented workflow is the tight coupling of these three areas, that provides the ability to trace both test and analysis results back to the original requirements and to identify the area of impact from changing requirements. These capabilities significantly contribute to the "first-time-right" requirement from industry by improving requirements and thus the concept in an early phase. Time can be saved by early identification of bugs in the implementation model and ultimately V&V costs are reduced. Additionally, our workflow enables the easy exchange of work products between different tools, both research and commercial tools, and due to its generic structure and the compliance to OSLC it allows to change the concrete tools without much additional effort.
Another advantage of the OSLC integration is that file exchange is enabled from within the tools, reducing the time overhead arising from indirect file sharing.

Contribution. The main contribution of this work is the realization of a development process including requirements engineering, formal specification and analysis, as well as testing. The methodology supports fine grained traceability down to the level of a single natural language requirement, enabled via the OSLC technique. In contrast to developing a new monolithic tool, we integrated a number of existing tools via web services.

The main novelty is the detailled linking of all artifacts facilitating the navigation between informal requirements, contracts, test cases, and associated elements in a simulation model across multiple tools throughout all steps of our process. It should be emphasized that this is not an isolated case study. It is part of an ongoing standardization process in the MBAT project, investigating how software engineering tools should be combined into an industrial reference technology platform.

Future work. It is a crucial part of the plans for our last project year, to continue the conversion of the remaining tools towards a full compliance of the OSLC standard. In addition, we will extend the framework to support requirements with real-time constraints, since many safety critical applications have hard timing requirements.

Acknowledgement The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement N° 269335 and from the Austrian Research Promotion Agency (FFG) under grant agreement N° 829817 for the implementation of the project MBAT, Combined Model-based Analysis and Testing of Embedded Systems.

REFERENCES

[ABJ+14] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, 2014.

[AJK13] Bernhard K. Aichernig, Elisabeth Jöbstl, and Matthias Kegele. Incremental refinement checking for test case generation. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2013.

[ALN13] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer Berlin Heidelberg, 2013.

[ALNT14] Bernhard K. Aichernig, Florian Lorber, Dejan Ničković, and Stefan Tiran. Require, test and trace it. Technical Report IST-MBT-2014-03, TU Graz, 2014. Online. https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=637834&pCurrPk=77579, Visited: 2014-03-06.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[BFMSV08] Luca Benvenuti, Alberto Ferrari, Emanuele Mazzi, and Alberto L. Sangiovanni-Vincentelli. Contract-based design for computation and verification of a closed-loop hybrid system. In Magnus Egerstedt and Bud Mishra, editors, *Hybrid Systems: Computation and Control*, volume 4981 of *Lecture Notes in Computer Science*, pages 58–71. Springer Berlin Heidelberg, 2008.

[BRR+10] Andreas Baumgart, Philipp Reinkemeier, Achim Rettberg, Ingo Stierand, Eike Thaden, and Raphael Weber. A modelbased design methodology with contracts to enhance the development process of safetycritical systems. In SangLyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 59–70. Springer Berlin Heidelberg, 2010.

[Das] Dassault Systèmes. Systemcockpit. Online. http://www.3ds.com/about-3ds/3dexperience-platform, Visited: 2014-03-06.

[Die14] Michael Dierkes. Combining test and proof in MBAT - an aerospace case study:. In Joaquim Filipe Luis Ferreira Pires, Slimane Hammoudi and Rui César das Neves, editors, *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 636–644. SCITEPRESS - Science and and Technology Publications, 2014.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[FMK+11] Stefan Farfeleder, Thomas Moser, Andreas Krall, Tor Stålhane, Herbert Zojer, and Christian Panis. DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In *Proceedings of the 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2011, Cottbus, Germany, April 13-15, 2011*, pages 271–274, April 2011.

[Gra02] Dorothy Graham. Requirements and testing: seven missing-link myths. *Software, IEEE*, 19(5):15–17, Sep 2002.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, July 1996.

[IBM] IBM. Rational doors. Online. http://www-03.ibm.com/software/products/en/ratidoorfami, Visited: 2014-03-06.

[ISO09] ISO/DIS 26262-1. Road vehicles – Functional safety – Part 1 Glossary. Technical report, International Organization for Standardization / Technical Committee 22 (ISO/TC 22), Geneva, Switzerland, July 2009.

[Jun04] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004.

[KEOS14] Omar Kacimi, Christian Ellen, Markus Oertel, and Daniel Sojka. Creating a reference technology platform - performing model-based safety analysis in a heterogeneous development environment:. In Joaquim Filipe Luis Ferreira Pires, Slimane Hammoudi and Rui César das Neves, editors, *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 645–652. SCITEPRESS - Science and and Technology Publications, 2014.

[Nie14] Brian Nielsen. Towards a method for combined model-based testing and analysis. In Joaquim Filipe Luis Ferreira Pires, Slimane Hammoudi and Rui César das Neves, editors, *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pages 609–618. SCITEPRESS - Science and and Technology Publications, 2014.

[Ong98] Chee-Mun Ong. *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*, volume 5. Prentice Hall PTR Upper Saddle River, NJ, 1998.

[Ope] Open Services for Lifecycle Collaboration. What is OSLC? - OSLC primer. Online. http://open-services.net/resources/tutorials/oslc-primer/what-is-oslc/, Visited: 2014-03-06.

[PDBH+09] Roberto Passerone, Werner Damm, Imene Ben Hafaiedh, Susanne Graf, Alberto Ferrari, Leonardo Mangeruca, Albert Benveniste, Bernhard Josko, Thomas Peikenkamp, Daniela Cancila, Arnaud Cuccuru, Sebastien Gerard, Francois Terrier, and Alberto Sangiovanni-Vincentelli. Metamodels in europe: Languages, tools, and applications. *Design Test of Computers, IEEE*, 26(3):38–53, May 2009.

[PSM+09] Hendrik Post, Carsten Sinz, Florian Merz, Thomas Gorges, and Thomas Kropf. Linking functional requirements and software verification. In *Proceedings of the 17th IEEE International Requirements Engineering Conference, 2009. RE'09.*, pages 295–302, Aug 2009.

[RW13] Ajitha Rajan and Thomas Wahl. *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer Vienna, 2013.

[SLE+10] Giedre Sabaliauskaite, Annabella Loconsole, Emelie Engström, Michael Unterkalmsteiner, Björn Regnell, Per Runeson, Tony Gorschek, and Robert Feldt. Challenges in aligning requirements engineering and verification in a large-scale industrial context. In Roel Wieringa and Anne Persson, editors, *REFSQ*, volume 6182 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2010.

[UKKD08] Eero J. Uusitalo, Marko Komssi, Marjo Kauppinen, and Alan M. Davis. Linking requirements and testing in practice. In *Proceedings of the 16th IEEE International Requirements Engineering, 2008. RE'08.*, pages 265–270, Sept 2008.