Automated Generation of Requirements-based Test Cases for an Adaptive Cruise Control System

Adina Aniculaesei
Institute for Applied Software Systems Engineering
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
adina.aniculaesei@tu-clausthal.de

Falk Howar
Department of Computer Science
Dortmund University of Technology
Dortmund, Germany
falk.howar@tu-dortmund.de

Peer Denecke
Institute for Applied Software Systems Engineering
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
peer.denecke@tu-clausthal.de

Andreas Rausch
Institute for Applied Software Systems Engineering
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

***Abstract*—Checking that a complex software system conforms to an extensive catalogue of requirements is an elaborate and costly task which cannot be managed only through manual testing anymore. In this paper, we construct an academic case study in which we apply automated requirements-based test case generation to the protoype of an adaptive cruise control system. We focus on two main research goals with respect to our method: (1) how much code coverage can be obtained and (2) how many faults can be found using the generated test cases. We report on our results as well as on the lessons learned.**

***Index Terms*—requierements-based testing, model-checking, adaptive cruise control system, automated testing**

I. Introduction

Control systems are added in automobiles in order to enhance the experience of the driver and increase the safety of the vehicle. Many tasks that were performed by the driver in the past are now executed by complex software systems. As an immediate consequence of this software complexity, extensive catalogues of system requirements become more common in the automotive industry [3]. In the last phase of development, a software system must pass the acceptance tests in order to be allowed into series production. This means that the software system must satisfy every requirement in the requirements catalogue. Testing of software is a process which requires a lot of expert knowledge. The task of testing a complex system against a large catalogue of requirements can only be managed by complementing manual testing with automated testing techniques.

In the automotive domain, requirements specifications are often maintained as informal textual documents. In the best case, they are broken down into lists of individual requirements and are maintained using a dedicated tool (e.g., IBM's DOORS). However, the sheer amount of requirements (in the tens of thousands) for every version of a product, makes it impossible to guarantee consistency, and test requirements in a meaningful and rigorous fashion. Model-based development is used in many projects, which means that formal models of a software exist and the software can be tested against these models, e.g., through back-to-back testing. Missing today is a (formal) connection between models and requirements that would allow for verification of models and software against requirements.

One approach, highlighted in previous research in the aeronautic domain, is to automatize the construction of tests from requirements through model checking [14]. A prerequisite of the automated generation of tests from requirements is that requirements are formalized. Formalizing requirements, however, is far from trivial and may even be impossible in some instances. System requirements expressed in natural language are characterized by a high degree of imprecision [3]. In order to evaluate the applicability of such an approach in the automotive domain, our investigation focuses on two research questions:

RQ1. How much code coverage can be obtained using requirements-based test cases on an automotive control system?

RQ2. How many faults can be found with requirements-based test cases in an automotive control system?

As research methodology, we use an academic case study on a control system in the automotive domain. We apply requirements-based test case generation via model checking to a prototype of an adaptive cruise control (ACC) system and report on the results and lessons learned. We use a controlled natural language in order to eliminate the imprecisions in the original formulation of the requirements of example system. We, e.g., impose the following restrictions: one sentence per requirement and the usage of specific patterns [12] that show which is the subject and which is the object targeted by the respective requirement.

Related Work. Using model checking for test-case generation is by now a well-known approach. The paper in [11] uses model checking to generate MC/DC model-based test sequences from the mode logic of a flight-guidance system. In [14] model checking is used to generate requirements-based test cases on the basis of three specific criteria: requirements coverage (RC), antecedent coverage (AC), and unique first cause coverage (UFC). The generated test suites are compared with each other with respect to four model coverages: state, transition, decision and MC/DC. Results show that the RC test suite has the lowest value for all model coverages. At the same time, the UFC test suite scores better results than the other two test suites with respect to all but one criterium, the decision coverage. Regarding the decision coverage, the UFC test suite is almost twice as good as the RC test suite but scores 3,73% less than the AC test suite.

In [13], the approach presented in [14] is evaluated on four industrial examples from the avionics domain. Test suites generated for each criteria are compared with each other and with randomly generated test suites in order to assess their fault finding ability. Results show that the UFC test suite outperforms the AC test suite as well as the RC test suite. However, no concluding evidence was found that the AC test suite is better regarding fault detection than the RC test suite.

In [10], test suites providing requirements UFC coverage and test suites providing MC/DC coverage over the model are compared with respect to fault finding in the context of conformance testing. The evaluation is performed on four industrial examples in the avionics domain. Results show that a combination of the MC/DC and UFC test suites is more effective than any of the test suites used separately.

Generating test cases from natural language requirements is addressed among others in [7]. The approach uses NLP in order to generate knowledge graphs. Different graph traversing methods are used to construct the test cases. Other approaches use UML diagrams such as use-case or sequence diagrams for test generation [6], [9].

Outline. In Section II we give an overview of our concept. Section III describes our case study, while Section IV presents the experiment perfomed on the example system. In Section V we present the results of our experiment and discuss threats to their validity. Finally, in Section VI we give a summary of the paper and an overview of future work.

II. Automated Generation of Test Cases from Requirements

An overview of the evaluated approach is shown in Fig.1. The concept depicts the steps are taken to generate and execute the requirements-based test cases:

a) System Model and LTL Obligations Construction: Each requirement is manually formalized as an LTL obligation. The system model as well is built on the basis of the system requirements, and therefore it satisfies the LTL obligations. The requirements catalogue for the example system are given in Section III.

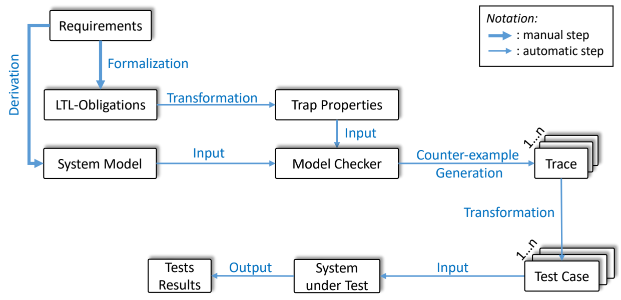


Fig. 1. Generating Test Cases From Requirements: Schematic Overview.

b) Trap Property Generation: A trap property is the negation of an LTL obligation which is satisfied by the system model. Three different criteria are used to built trap properties from LTL obligations: *requirements coverage* (RC), *antecedent coverage* (AC), and *unique first cause coverage* (UFC). Depending on the used criterion, several trap properties may be generated from each requirement. All trap properties constructed for each criterion are generated with a software tool which we developed for this purpose.

c) Test Case Generation with Model Checking: The system model and the trap properties are given as input to a model checker in order to generate traces. The model checker builds a finite state system of a system model and explores the state space of the model in order to find violations of LTL obligations. If found, the model checker returns a counter-example trace. By verifying a system model against a trap property, the model checker searches for a counter-example trace which disproves the trap property, and in turn, shows how the LTL obligation can be satisfied. Each trace forms the basis for a test case. We built a tool which takes a trace as input and constructs the corresponding test case. The transformation of a trace in a test case is described in Section IV. We generate a test suite for each of the three criteria.

d) Test Suite Execution: Given the fact that the system under test (SuT) is built so that it already satisfies the original system requirements, we applied our test suites to a set of system mutants. A detailed description of the mutant generation process is depicted in Section V. After the test execution, all generated test suites are compared with respect to their fault finding ability. In addition, the branch coverage for the test suites is measured on the original source code of SuT. This is motivated by the fact that our SuT is an automotive control system which is required to fullfill safety integrity levels higher than ASIL B, for which the automotive standard ISO 26262 [1] recommends branch coverage as a measurement criterion for the quality of tests.

III. Adaptive Cruise Control (ACC)

We construct our case study around a prototype of an ACC system [1]. General properties of ACC systems are specified in the ISO 15622 norm [5], which we have used as a guideline

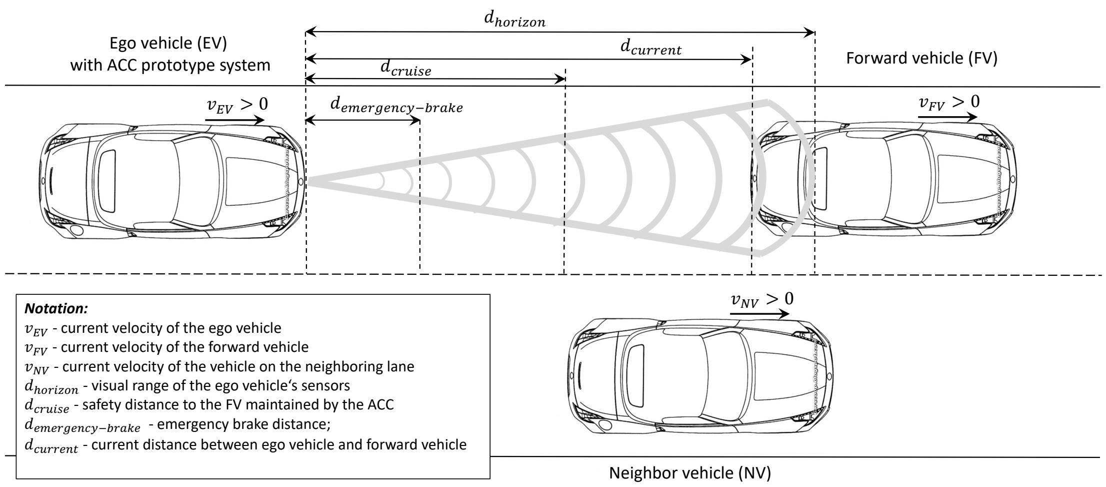

$d_{horizon}$
$d_{current}$
d_{cruise}
Ego vehicle (EV)
with ACC prototype system
Forward vehicle (FV)
$v_{EV} > 0$
$d_{emergency-brake}$
$v_{FV} > 0$
$v_{NV} > 0$
Notation:
v_{EV} - current velocity of the ego vehicle
v_{FV} - current velocity of the forward vehicle
v_{NV} - current velocity of the vehicle on the neighboring lane
$d_{horizon}$ - visual range of the ego vehicle's sensors
d_{cruise} - safety distance to the FV maintained by the ACC
$d_{emergency-brake}$ - emergency brake distance;
$d_{current}$ - current distance between ego vehicle and forward vehicle
Neighbor vehicle (NV)

Fig. 2. Scenario of the Case Example.

for the ACC developed in this case study. Our example scenario is shown in Fig.2: The ACC system has the function to adjust the current velocity of the ego vehicle (EV) towards a target *cruise velocity* defined by the EV's driver. In case the EV gets too close to the forward vehicle (FV), the ACC system must adjust the current distance between the two and maintain a certain safety distance. The safety distance is from hereon denoted as the EV's *cruise distance*. Additionally, the EV's driver has the several possibilities to intervene by: (1) activating the system via an ACC button, i.e. boolean flag *active* is set; (2) deactivating the system via the ACC button, i.e. boolean flag *active* is reset; (3) deactivating the system by braking or accelerating the car. Thus, the presented ACC is not a fully autonomous system.

A. Assumptions

We have made simplifying assumptions in our case study:

1) The EV has a sensor on its front end with the range $d_{horizon}$, and is able to measure the FV's velocity v_{FV} and the distance $d_{current}$ to the FV.
2) If the vehicle on the neighbour lane (NV) changes the lane in front of the EV, then it becomes the new FV and is subject to monitoring by the sensors of the EV.
3) Cars drive forwards: $v_{EV} > 0, v_{FV} > 0$, and $v_{NV} > 0$.
4) Only the EV's driver can decide if the emergency brake is necessary and trigger it accordingly.

When the ACC system is activated, it stores the current velocity v_{EV} as the cruise velocity and adjusts the cruise velocity if possible. As soon as the current distance to the FV falls below the cruise distance, the EV's ACC system starts to adjust the EV's current distance to the FV. The system gives out a warning if the current distance to the FV gets close to the emergency brake distance, i.e. in this situation the driver needs to decide upon the necessity of an emergency brake.

[1]Our ACC system is available online at: https://bitbucket.org/aaniculaesei/automated-test-case-generation

B. System Requirements

The requirements catalogue defined for the ACC prototype system contains ten requirements that are formulated in structured language and use modes. Du to lack of space, we only provide three representative examples here.

1) If the *active* flag is not set, then the system shall switch to the state *off*.
2) If the system is in the state *off* and the *active* flag is set, then the system shall switch to the state *speed control* and set the current velocity as the cruise velocity and the current distance as the cruise distance.
3) If the current state is *speed control* or *distance control*, the forward vehicle is visible and the current distance is less than the emergency brake distance, then the system shall display a warning.

IV. Formalization and Test Case Generation

We generate test cases from requirements in three steps: First, we formalize requirements as LTL properties and develop a formal system model (as would be done during model-based development). We then generate so-called trap properties from LTL requirements which become the basis for automated generation of test cases with a model checker.

The tool chain that takes LTL obligations and builds test cases from them is shown in Figure:

a) From System Requirements to LTL Obligations: We build the system LTL obligations from the requirements catalogue in Section III-B. Looking at the first requirement, the corresponding LTL obligation is given in (1).

$$G(\neg active \rightarrow X(state = off)) \qquad (1)$$

Note that all LTL obligations have the form $G(A \rightarrow B)$. For the LTL obligation in (1), we use the RC, AC and UFC criteria defined in [14] to obtain the corresponding trap properties.

A test suite constructed on the basis of the RC criterion contains one test per requirement which illustrates one way in which this requirement is met. In order to obtain this

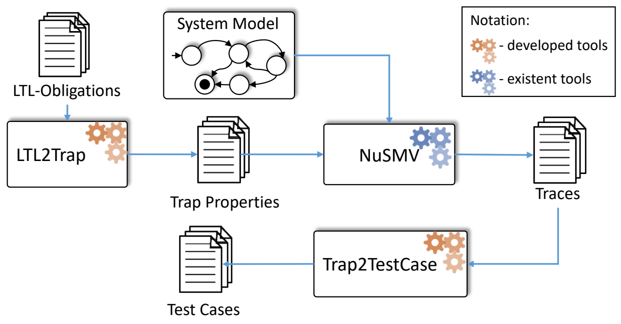


Fig. 3. Tool Chain for the Automated Construction of Requirements-based Test Cases.

test suite, we simply negate each requirement captured as an LTL obligation. The model checker then finds a counter-example trace which disproves the negated LTL obligation, and consequently shows how the requirement can be satisfied. The trap property for the RC criterion is given in (2).

$$\neg G(\neg active \rightarrow X(state = off)) \tag{2}$$

Consider an LTL obligation of the form $G(A \rightarrow B)$. In a test suite built with the AC criterion, a test case must ensure that the antecendent A becomes true at leat once before B is true along a path which satisfies this requirement. Thus, the test case must in fact satisfy the LTL property $G(A \rightarrow B) \wedge F(A)$. The trap property is then obtained by simple negation. The trap property for the AC criterion for the LTL obligation in (1) is shown in (3):

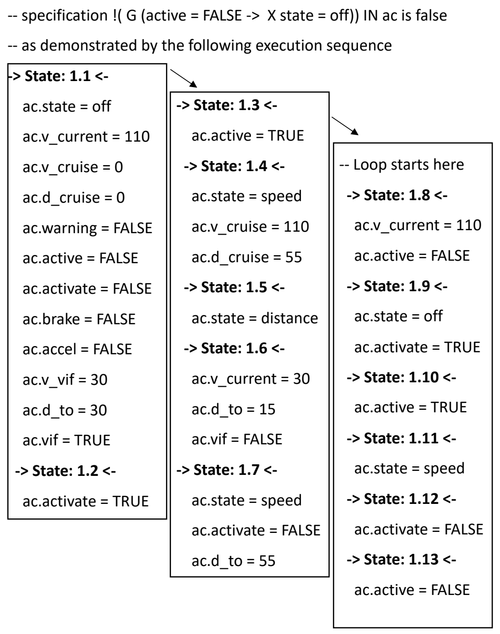
$$\neg(G(\neg active \rightarrow X(state = off)) \wedge F(\neg active)) \tag{3}$$

The UFC criterion is a requirements coverage criterion which is adapted from the structural coverage criterion MC/DC. While MC/DC is defined over the states of a system model, UFC is built from LTL specifications and is thus defined over the paths of the model. Similarly to MC/DC, UFC guarantees two things: (1) every basic condition in each formula has taken on all possible outcomes at least once and (2) every basic condition in each formula has been shown to independently affect the formula's outcome. Below is a list of the UFC rules which we used in this paper.

1) **Atoms.** $x^+ = \{x\}$, $x^- = \{\neg x\}$ (for atomic cond. x)
2) **Negation.** $(\neg A)^+ = A^-$
3) **Disjunction.**
$(A \vee B)^+ = \{a \wedge \neg B \mid a \in A^+\} \cup \{\neg A \wedge b \mid b \in B^+\}$
4) **Globally.** $G(A)^+ = \{A\ U\ (a \wedge G(A)) \mid a \in A^+\}$
5) **Next.** $X(A)^+ = \{X(a) \mid a \in A^+\}$

To construct the trap property for the UFC criterion, we first build the UFC obligations, as shown in (4) and (5):

$$\begin{aligned}(\neg active) \rightarrow (X(state = off))\ U \\ ((active \wedge X(state \neq off)) \wedge \\ G((\neg active) \rightarrow (X(state = off))))\end{aligned} \tag{4}$$



```
-- specification !( G (active = FALSE -> X state = off)) IN ac is false
-- as demonstrated by the following execution sequence
-> State: 1.1 <-
  ac.state = off
  ac.v_current = 110
  ac.v_cruise = 0
  ac.d_cruise = 0
  ac.warning = FALSE
  ac.active = FALSE
  ac.activate = FALSE
  ac.brake = FALSE
  ac.accel = FALSE
  ac.v_vif = 30
  ac.d_to = 30
  ac.vif = TRUE
-> State: 1.2 <-
  ac.activate = TRUE
-> State: 1.3 <-
  ac.active = TRUE
-> State: 1.4 <-
  ac.state = speed
  ac.v_cruise = 110
  ac.d_cruise = 55
-> State: 1.5 <-
  ac.state = distance
-> State: 1.6 <-
  ac.v_current = 30
  ac.d_to = 15
  ac.vif = FALSE
-> State: 1.7 <-
  ac.state = speed
  ac.activate = FALSE
  ac.d_to = 55
-- Loop starts here
-> State: 1.8 <-
  ac.v_current = 110
  ac.active = FALSE
-> State: 1.9 <-
  ac.state = off
  ac.activate = TRUE
-> State: 1.10 <-
  ac.active = TRUE
-> State: 1.11 <-
  ac.state = speed
-> State: 1.12 <-
  ac.activate = FALSE
-> State: 1.13 <-
  ac.active = FALSE
```

Fig. 4. Example Trace for the Trap Property in (2) demonstrating how the first requirement is satisfied.

$$\begin{aligned}(\neg active) \rightarrow (X(state = off))\ U \\ ((\neg active \wedge X(state = off)) \wedge \\ G((\neg active) \rightarrow (X(state = off))))\end{aligned} \tag{5}$$

The construction of the trap properties for the three criteria is automatized with a software tool developed for this purpose which implements the AC and UFC rules described above and the negation of the LTL obligations.

b) From Trap Properties to Traces: The model checker takes the trap properties and the system model as input and generates traces. Each trace corresponds to a trap property and represents a counter-example provided by the model checker that disproves the trap property. Consequently, the system model is proven to satisfy the corresponding LTL obligation. Figure 4 shows the trace obtained through the verification of the trap property in (2).

c) From Traces to Test Cases: Each test cases for the SuT is based on a trace generated by the model checker. Since the system model satisfies the system requirements, the SuT can satisfy the requirements if and only if it displays the same behavior as the system model. There are two types of variables in a trace: (1) *input variables*, which contain the inputs for the SuT, provided by the driver model and the environment model; (2) *oracle variables*, which describe the desired behaviour of the SuT and are provided by the system model. These variables act as test oracles for the respective test cases.

V. Evaluation

A. Setup

Using our software tool, we generated 10 trap properties for the RC and AC criteria respectively and 38 properties for the UFC criterion from the LTL obligations of our system.

TABLE I
EXPERIMENT RESULT

	RC	***AC***	***UFC***
Number of Trap Properties	10	10	38
Number of Test Cases	6	7	18
Number of Mutants	524	524	524
Number of Killed Mutants	488	488	488
Number of Living Mutants	36	36	36
Killed Mutants (%)	93.12	93.12	93.12
Branch Coverage	78.3	78.3	86.7

In order to generate the traces, we verified the system model with the `NuSMV` model checker [2] against the trap properties. Each trap property was violated by the system model and resulted in a trace for a test case. For each criterion, we removed the duplicate traces, in order to reduce the number of test cases. We generated the mutants with the `muJava` tool [8]. We used the software tool `EclEmma` [4] to measure the branch coverage on the original SuT. Additionally, we used mutants to evaluate the error finding capabilities of the generated test cases. The mutants of the SuT are created with the help of the `muJava` tool [8]. The tool executes the following modifications on the SuT: *arithmetic operator insertion*, *conditional operator deletion*, *conditional operator insertion*, *conditional operator replacement*, *operator deletion*, *relational operator replacement*, and *statement deletion*. One mutant introduces only one defect in the SuT. Through these modifications to its source code, the `muJava` tool generates 524 mutants of our system under test.

B. Discussion of Results

The results of our experiment are shown in Table I. According to Table I, the test suite generated for the UFC criterion is as good as the test suites generated for the RC and the AC criteria with respect to fault detection. We consider 93,12% of killed mutants a promising result. On a more complex system with a larger set of requirements the UFC test suite may perform better than the RC and AC test suites. At the same time, the UFC test cases are better with respect to branch coverage than RC test cases and than the AC test cases. UFC guarantees that every basic condition takes all possible outcomes and is shown to independently affect the outcome of a given LTL formula. As a consequence, the branch coverage is higher in the case of UFC.

C. Lessons Learned

One lesson we learned rather early in the project is that the way the system requirements are formulated plays an important role in the understanding of the system. We invested an important amount of work in building an abstract system model and formalizing the system requirements. For the construction of the trap properties, we decided to build a software tool to automatize this step. This decision paid off in the end because it allowed us to eliminate any errors which might have appeared during the manual construction of the trap properties. Moreover, we were able to shorten the project time by several weeks. In order to ensure that no fault is counted twice, we have analyzed our system with respect to the number of mutants killed for the fault detection measurements. However, for larger industrial system we recommend to employ statistical analysis in order to ensure that any bias is removed from the results.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we applied model checking to generate requirements-based test cases to a prototype of an ACC system. To this end, we developed a protoype of such an ACC system and formalized a set of requirements. We obtained three test suites each built according to one of the RC, AC or UFC criteria. We measured these test suites with respect to fault finding and branch coverage and reported on the results. We found out, that for our case example the RC and AC test suites were as good as the UFC test suite with regard to fault detection. With respect to branch coverage, our findings show that the UFC test suite has a 8.4 % better branch coverage than the RC and AC test suites respectively. In future, we plan to apply our approach on an industrial case example in the automotive domain and together with automotive software engineers, which is bound to have a higher complexity, and thus, a larger set of system requirements. We plan to investigate how well structured language and formalization work for more realistic projects in the automotive domain.

REFERENCES

[1] Iso 26262 road vehicles—functional safety—part 6: Product development at the software level, 2011.

[2] FBK. Nusmv. http://nusmv.fbk.eu/, 2015. Accessed: 2017-07-10.

[3] M. Fockel, J. Holtmann, and M. Meyer. Mit Satzmustern hochwertige Anforderungsdokumente effizient erstellen. OBJEKTspektrum, 2014.

[4] M. R. Hoffmann, B. Janiczak, and E. Mandrikov. Eclemma. http://www.eclemma.org, 2016. Accessed: 2017-07-10.

[5] International Organization for Standardization. ISO 15622:2010. Intelligent transport systems – Adaptive Cruise Control systems – Performance requirements and test procedures. https://www.iso.org/standard/50024.html, May 2017. [Online; accessed 25-May-2017].

[6] N. Kosindrdecha and J. Daengdej. A test case generation process and technique. *Journal of Software Engineering*, 4(4):265–287, November 2014.

[7] P. Kulkarni and Y. Joglekar. Generating and analyzing test cases from software requirements using nlp and hadoop. *International Journal of Current Engineering and Technology*, 4(6):3934–3937, December 2014.

[8] Y.S. Ma, J. Offutt, and Y.R. Kwon. Mujava: An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[9] C. Nebut, S. Pickin, Y. Le Traon, and J.-M. Jézéquel. Automated requirements-based generation of test cases for product families. In *ASE '03'*, pages 263–266, Montreal, Quebec, Canada, 2003.

[10] A. Rajan, M.W. Whalen, M. Staats, and M.P.E. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *FMSE '08*, pages 86–104, Kitakyushu, Japan, 2008.

[11] S. Rayadurgam and M.P.E. Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In *SEW '03*, pages 91–96, Greenbelt, Maryland, USA, 2003.

[12] Chris Rupp and SOPHISTen. Schablonen für alle fälle, 2016. Accessed: 2017-07-10.

[13] M. Staats, M.W. Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness. In *NFM '10'*, pages 161–170, Washington D.C., USA, 2010.

[14] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *ISSTA '06*, pages 25–36, Portland, ME, USA, 2006.