Automated Generation of Requirements-Based Test Cases for an Automotive Function using the SCADE Toolchain

Adina Aniculaesei*, Andreas Vorwald* and Andreas Rausch*
* Institute for Software and Systems Engineering
Technische Universität Clausthal, Clausthal-Zellerfeld, Germany
Email: adina.aniculaesei@tu-clausthal.de, andreas.vorwald@tu-clausthal.de, andreas.rausch@tu-clausthal.de

Abstract—**Results of acceptance tests trigger various adaptations in the architecture and design of a complex software system. Several adaptation iterations are needed until all acceptance tests are successfully passed. Checking whether the adapted software system complies with an extensive catalogue of requirements is an elaborate task, which cannot be managed only via manual testing anymore. Over the years, model checking has established itself as an efficient method for the generation of requirements-based test cases. At the same time, the traction gained by model-based development tools, such as SCADE Suite, especially in the automotive and the avionics domains, facilitates the use of formal methods for the analysis and verification of complex software systems developed in these industries. This paper describes an approach which supports the generation of test cases from formalized requirements using the SCADE toolchain. In order to evaluate the applicability of our approach, we apply our concept on a simple system from the automotive domain and discuss outcomming results.**

Keywords–architecture adaptation; model-based development; requirements-based testing; model checking; automotive function; SCADE toolchain

I. Introduction

Control systems are installed in cars with the purpose to improve the driving experience and increase the safety of the vehicles and their passengers. Tasks which were previously carried out by the driver are now performed by complex software systems. An immediate consequence of this software complexity is that extensive catalogues of system requirements have become more common in the automotive industry [1].

Throughout their life cycle, automotive software systems are subjected to various modifications, which originate in different sources, e.g., change requests caused by defect removal or system enhancements triggered by end user demands. Once the changes have been implemented, the software system must pass the acceptance tests again in order to get into series production. This means that the software system must satisfy every requirement in the catalogue. Test results may expose further defects in the system design or in the system implementation. Thus, the architecture of the software system and its implementation may go through a series of adaptation iterations until the system has successfully passed all acceptance tests. These adaptations may further increase the complexity of the software system.

Software testing is a process which requires a lot of expert knowledge. Testing complex software systems against large requirements catalogue is a task which cannot be managed manually anymore. In the automotive industry, requirements specifications are often maintained as informal documents. In the best case scenario, they are broken down into lists of individual requirements, which are then maintained using a dedicated tool, e.g., IBM's Doors. The large number of requirements for every product version makes it impossible to guarantee consistency and to test the requirements in a rigorous manner. In the automotive domain, model-based development is used by software engineers to create formal models of the desired software system early in the development lifecycle and to test the software against these models, e.g., through back-to-back testing.

We have established the formal connection between system models and system requirements for the automotive domain in a previous work [2]. In [2], we used the approach presented in [3] and generated requirements-based test cases via model checking for a prototype of an adaptive cruise control system. Since model-based development has gained so much traction in the automotive industry, we are interested in finding out whether the approach developed in [2] is also applicable with software toolchains used in model-based development. In this paper, we focus on the SCADE toolchain [4] and we investigate the following research question:

RQ: *How can the SCADE toolchain be used to generate requirements-based test cases for an automotive function?*

As research methodology, we build an academic case study of a control system in the automotive domain. We apply requirements-based test case generation to a simple prototype of a door locking system using the SCADE Design Verifier as model checker. For the sake of simplicity, we formulate only one requirement for the door locking system. For this purpose, we use a controlled natural language with specific sentence patterns, showing which is the subject and which is the object targeted by the requirement [5]. Our work is meant to offer support to the test engineers, who need to develop meaningful and cost-efficient tests, but also to the software developers and software architects, who must decide on the basis of the test results whether any system adaptations are necessary.

Related Work. Using model checking for test-case generation is by now a well-known approach. The paper in [6] uses model checking to generate MC/DC model-based test sequences from the mode logic of a flight-guidance system. For the same type of system, Whalen et al. [3] use model checking to generate requirements-based test cases on the basis of three specific criteria: requirements coverage, antecedent coverage and unique first cause. In [7], the approach presented in [3] is evaluated on four industrial examples from the avionics domain. All four systems were modeled in the Simulink notation from Mathworks and then translated to the Lustre synchronous programming language for the purpose of test case generation.

Generating test cases from natural language requirements is addressed among others in [8]. 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 [9] [10].

In [11], a translator framework is introduced which allows the translation of commercial modeling languages, e.g., SCADE, in the input languages of verification tools, e.g., Prover or PVS. This allows the integration of commercial model-based development tools with verification tools. The approach is demonstrated on several case studies from the avionics domain.

In this work, we provide a concept for a requirements-based test case generation, which is fully integrated with the SCADE toolchain. We apply it on an example system from the automotive domain. Some of the foundations of our approach are provided by the work of Whalen et al. in [3], [12], and [11].

Paper Outline. In Section II we describe preliminary notions, which are necessary to understand the approach presented in this work. In Section III, we give an overview of our concept. Section IV introduces our case study, while Section V describes the experiment carried out on the example system. In Section VI, we present the results of our experiment and discusses the lessons learned from this work. Section VII concludes the paper with a summary of our contribution and an overview of future work.

II. Preliminaries

In this section, we present the basic process of model checking and explain how this method can be used to generate test cases from system requirements. Furthermore, an overview of model-based development and formal verification with SCADE is given.

Basic Process of Model Checking. Given a system model and a system property to verify, a model checker builds a formal representation of the system model in the form of a finite state machine and explores its state space in search for states which falsify the system property. If the system property is falsified, the model checker returns a counter-example trace, showing how the state which falsifies the system property can be reached from the initial state of the system model.

Generation of Test Cases with Model Checking. Throughout the years, model checking has established itself as an efficient method to generate test cases from system requirements. One possibility to generate test cases using model checking is to build trap properties [3]. Trap properties are basically negations of the system properties, which are satisfied by the system model if the latter is correctly built. While verifying the system model against a trap property, the model checker searches for a counter-example to disprove the trap property. By the law of double negation in propositional logic, the counter-example which disproves the trap property is in fact an example showing how the original system property is satisfied. The counter-example is then used as a basis for building a requirements-based test case, which checks if the *System under Test (SuT)* satisfies the respective system requirement.

Model-based Development with SCADE. SCADE Suite is a development environment used for the model-based design and development of software system components.Software components are encapsulated in SCADE operators, which, in turn, are organised in SCADE projects. Each SCADE operator has inputs and outputs, which form the interface of the respective software component. The formal basis of the SCADE language is given by the declarative language Lustre and is defined in [4]. The systems of equations specific to the language Lustre are used to model the dataflow inside SCADE operators, connecting the input flows to the output flows of the operator. Hierarchical state machines are used to describe the control flow of SCADE operators.

Formal Verification with SCADE Design Verifier. In SCADE Suite, formal verification is performed using SCADE Design Verifier, a model checker based on a SAT-solver [13] [14]. The SCADE Design Verifier works on the basis of the SCADE observer principle. System properties which must be verified with the SCADE Design Verifier are first encapsulated as observers. An observer is a SCADE operator which takes as input both the input flows and the output flows of the system model. The observer produces a Boolean output flag. The system property is satisfied if the observer's output evaluates to *true* in every computation cycle. Should the flag evaluate to *false* in one computation cycle, the SCADE Design Verifier returns a counter-example which shows why this answer has been reached.

III. Test Case Generation from Requirements using the SCADE Toolchain

An overview of our approach is given in Figure 1. The concept illustrates the necessary steps for the generation and execution of requirements-based test cases.

System Model Construction and System Requirement Formalization. The system requirement is manually formalized as an obligation in *Linear Temporal Logic (LTL)*. The system model is designed with SCADE Suite on the basis of the system requirement, and therefore satisfies the LTL obligation. Both the system model and the system requirement are presented in Section IV.

Trap Property Generation. A trap property is the negation of an LTL obligation which is satisfied by the system model. Since the test case generation process using the SCADE toolchain is the focus of this paper, we limit ourselves to using only the *Requirements Coverage (RC)* criterium to build the trap property corresponding to the LTL obligation.

Test Case Generation using Model Checking. The system model and the trap property are given as input to the SCADE Design Verifier in order to generate traces. In the classical model checking process, the model checker explores the entire state space of the system model consisting of all the combinations of inputs and states in order to find violations of the LTL obligation. If found, then the model checker produces a conterexample trace which shows how the LTL obligation can be falsified. The trace is in turn transformed into a test case with which the SuT can be later executed.

Test Case Execution. The SuT, in our case the SCADE system model, is loaded in the SCADE Test Environment and then executed with the test input data specified in the generated test case.

IV. Case Study Exampple

Our case study is constructed around a simple prototype of a door locking system in an automobile. The door locking system is regarded as a safety feature for the vehicle, as its primary goal is to ensure that the doors do not open while the

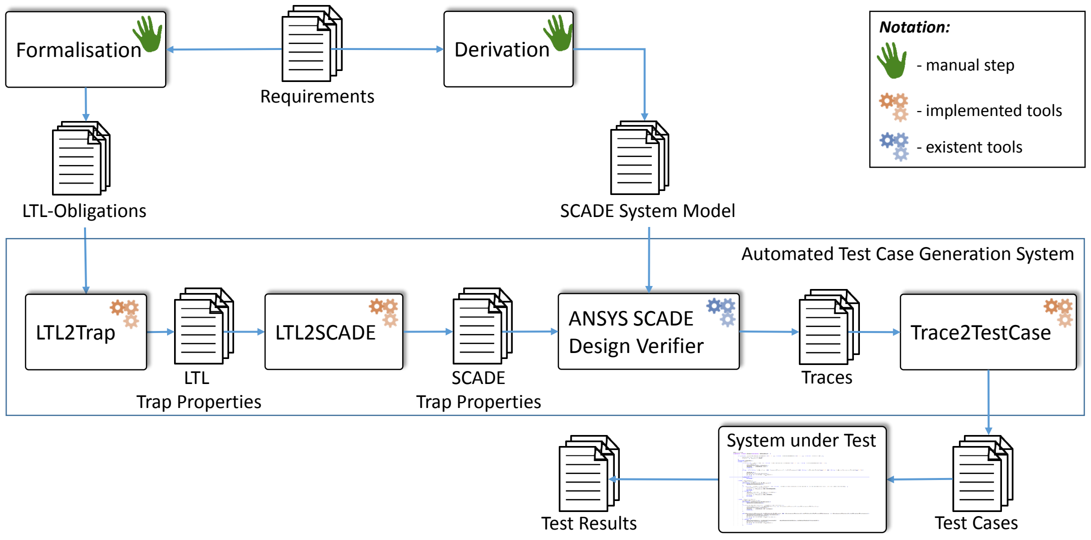


Figure 1. Test Case Generation from Requirements using the SCADE Toolchain.

vehicle is moving. Figure 2 depicts the implementation of the door locking system in the SCADE Suite.

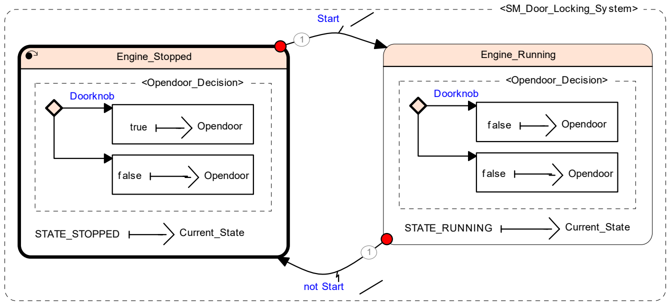


Figure 2. Case Study Example: A Simple Door Locking System.

A. System Model

We model the door locking system in strong correlation with the current vehicle status. Our example system is modeled as a state machine with two states which describe the vehicle status: state `Engine_Running` for the moving vehicle and state `Engine_Stopped` for the stillstanding vehicle. For the purpose of this case study, the functionality of the door locking system is kept rather simple. Thus, if a vehicle passenger operates the door handle while the engine is running, then the vehicle door stays closed. On the other side, if the engine is stopped and the vehicle passenger operates the door handle, then vehicle door opens.

The input interfaces of the door locking system consist of the boolean flags `Doorknob` and `Start`, which model the operation of the door handle by the vehicle passenger and respectively the start/stop of the vehicle engine. The output interfaces of the door locking system are represented by the boolean flag `Opendoor` and the enumeration variable `Current_State`. The former models the status of the door vehicle (opened/closed). The latter switches between the constants `STATE_STOPPED = 0` and `STATE_RUNNING = 1`, in order to keep track of the current state of the vehicle.

B. System Requirements

For the purpose of simplicity, we formulate one safety requirement for the door locking system. The system requirement, formulated in a controlled natural language [5], reads as follows:

R1. *If the motor is running and the doorknob is pushed, then the door shall not be opened.*

V. Formalization and Test Case Generation

Our process for the generation of test cases from requirements has already been published in a previous work [2]. However, the purpose of this paper is to automatize this process using the SCADE toolchain. For the purpose of completeness, we present the steps of the test case generation process, highlighting the details specific for the work with the SCADE toolchain:

A. From System Requirements to LTL Obligations

We build the LTL obligation for the door locking system from the system requirement *R1*, presented in Section IV. The corresponding LTL obligation is written in (1):

$$\begin{aligned}\phi : G(&Current\_State = STATE\_RUNNING \wedge \\ &Doorknob \rightarrow X(Opendoor = false))\end{aligned} \tag{1}$$

Observe that time model of LTL differs from the time model of the SCADE language, i.e., LTL looks from the present time point into the future while SCADE looks from the present point into the past. Therefore, the LTL obligation in (1) must be transformed using the `last` operator as shown in (2), so that it conforms to the SCADE time model:

$$\begin{aligned}\phi : G(&last'Current\_State = STATE\_RUNNING \wedge \\ &Doorknob \rightarrow X(last'Opendoor = false))\end{aligned} \tag{2}$$

B. From LTL Obligations to Traces

In order to obtain a test case which satisfies the system requirement *R1*, the first step is to build a trap property by simply negating the LTL obligation shown in (2). Thus, the corresponding trap property for requirement *R1* is given in

(3):

$$\phi : \neg G(last' Current\_State = STATE\_RUNNING \wedge Doorknob \rightarrow X(last' Opendoor = false)) \quad (3)$$

The trap property is then transformed in SCADE code against which the SCADE system model can be verified using SCADE Design Verifier as model checker. Figure 3 gives an overview of the necessary steps for the transformation of the LTL trap property in SCADE code.

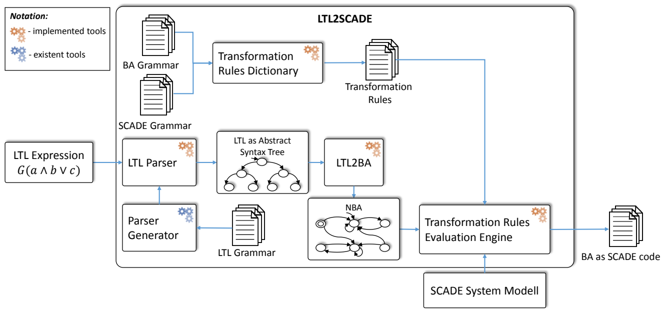


Figure 3. Transformation of LTL Obligations in SCADE Code.

LTL Parser Generation and AST Construction. We generate an LTL parser on the basis of the LTL grammar, in order ensure the correct parsing of the system requirements formalised in LTL with respect to operator priority rules. The operator precedence is encoded in the *Abstract Syntax Tree (AST)*. In our implementation of the LTL grammar, the logical and temporal operators of LTL, as well as the arithmetic operators are nonterminal symbols. As terminal symbols, our grammar allows the boolean constants, *true* and *false*, variable names and arithmetic constants in atomic propositions, e.g $a \leq 10$, but also the keyword `last`, which is specific to the SCADE language.

Büchi Automaton Construction. The common basis of LTL and SCADE is represented by automata. This is based on the fact that the concept of automata is integrated in the SCADE language [15] [4], and that nondeterministic Büchi automata (NBA) are an alternate representation of LTL formulae [16]. Figure 4 illustrates the steps needed to transform an LTL formula into an NBA. This transformation is based on the algorithm defined by Gerth et al. in [17]. The algorithm transforms an LTL formula expressed in positive normal form (PNF) into a generalized nondeterministic Büchi automaton (GNBA). Once this transformation is complete, only atomic propositions occur as transition guards. The atomic propositions are connected via the logical operator AND ($\wedge$), if there is more than one as transition guard on the same transition. Then, the GNBA is transformed into an NBA using the algorithm presented in [16]. The NBA constructed from the trap property of requirement *R1* is presented in Figure 5.

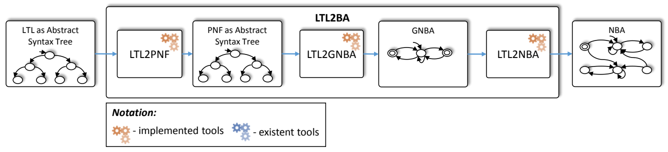


Figure 4. Transformation of LTL Obligations in Non-deterministic Büchi Automata.

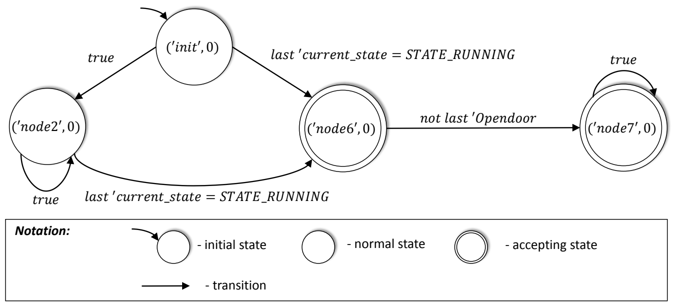
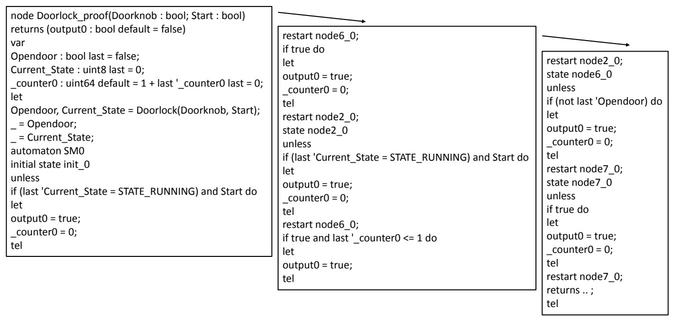


Figure 5. Büchi Automaton for Requirements *R1*.

SCADE Code Generation. The core idea of this process step is to create for each LTL trap property a new SCADE operator which has the same input as the SCADE system model. This SCADE operator is then used later to generate test cases. The inputs of the new SCADE operator are redirected to the SCADE system model to perform a computation cycle. Then, a unique output as observer for the LTL trap property and a corresponding state machine implementing the NBA of the LTL trap property are created. Furthermore, the name of the SCADE system model is used in order to build the name of the newly created SCADE operator: `<SCADE-system-model>_proof`. Figure 6 shows the SCADE Code generated for the system requirement *R1*.



```
node Doorlock_proof(Doorknob : bool; Start : bool)
returns (output0 : bool default = false)
var
Opendoor : bool last = false;
Current_State : uint8 last = 0;
_counter0 : uint64 default = 1 + last '_counter0 last = 0;
let
Opendoor, Current_State = Doorlock(Doorknob, Start);
_ = Opendoor;
_ = Current_State;
automaton SM0
initial state init_0
unless
if (last 'Current_State = STATE_RUNNING) and Start do
let
output0 = true;
_counter0 = 0;
tel
```

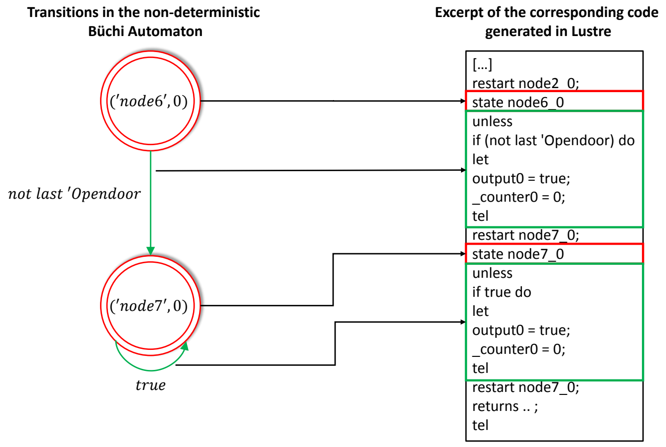
```
restart node6_0;
if true do
let
output0 = true;
_counter0 = 0;
tel
restart node2_0;
state node2_0
unless
if (last 'Current_State = STATE_RUNNING) and Start do
let
output0 = true;
_counter0 = 0;
tel
restart node6_0;
if true and last '_counter0 <= 1 do
let
output0 = true;
tel
```

```
restart node2_0;
state node6_0
unless
if (not last 'Opendoor) do
let
output0 = true;
_counter0 = 0;
tel
restart node7_0;
state node7_0
unless
if true do
let
output0 = true;
_counter0 = 0;
tel
restart node7_0;
returns .. ;
tel
```

Figure 6. SCADE code corresponding to the NBA generated from Requirement *R1*.

The SCADE Design Verifier is the model checker of the SCADE Suite and can be interfaced by using observers which are evaluated in each computation cycle during the state space exploration [15]. The model checker explores the state space in search of a trace which falsifies the trap property. It stops the state space exploration when it has exhausted the entire state space or when the observer of the trap property switches to *false*. When the latter occurs, a trace of input assignments is printed, which shows how the trap property can be falsified.

In SCADE, an NBA is represented by a statemachine. Automata are a feature of SCADE, and respectively of the Lustre language [15] [4]. Figure 7 illustrates the transformation of the NBA corresponding to the requirement *R1* in SCADE based on two transitions extracted from Figure 5. Every state of the NBA is represented in SCADE via the keyword `state`, while the initial state of the NBA is also marked by the keyword `initial`. A transition in the NBA is transformed into an *if*-statement within the state. In order to ensure optimal transition execution, the outgoing transitions of a state are sorted descending according to the number of transition guards. For example, if a state s_1 has two outgoing transitions, (s_1, a_0, s_2) and $(s_1, a_0 \wedge a_1, s_3)$, then the second one is prefered.


Transitions in the non-deterministic Büchi Automaton
Excerpt of the corresponding code generated in Lustre
('node6', 0)
not last 'Opendoor
('node7', 0)
true

```
[...]
restart node2_0;
state node6_0
unless
if (not last 'Opendoor) do
let
output0 = true;
_counter0 = 0;
tel
restart node7_0;
state node7_0
unless
if true do
let
output0 = true;
_counter0 = 0;
tel
restart node7_0;
returns .. ;
tel
```

Figure 7. Excerpt of Transformation from non-deterministic Büchi Automata to Lustre code.

C. From Traces to Test Cases

The trace generated by the SCADE Design Verifer contains only test input data. However, in order to get the full test case, output data are also needed. Figure 8 displays the workflow used to obtain the test output data and generate test cases.

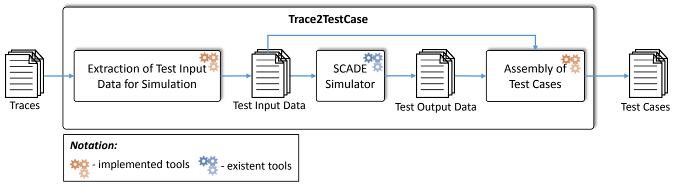


Figure 8. Transformation of Traces in Test Cases.

To begin with, the test input data is extracted from the trace. Then, the SCADE system model is run with the test input data in the SCADE Simulator. The test output data obtained from the simulation is assembled with the test input data extracted from the trace in a test case file (`.sss`-file), which can later be run in the SCADE Test Environment. A trace and the corresponding test case showing how requirement *R1* may be satisfied are shown in Figure 9.

VI. EVALUATION

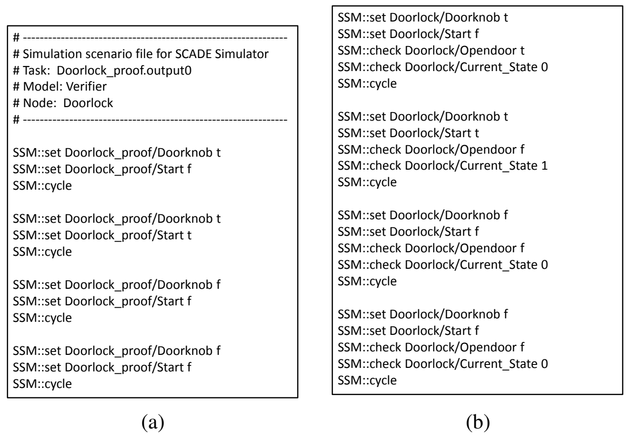
A. Setup

The system in Figure 2 was modeled in ANSYS SCADE 19.1 (build 20180327). To implement the workflow described in Figure 1, a Python 3.4 script using the Python-API of SCADE was created. The parser was then generated with the parser generator `ply` [18].

The SuT was then executed with the test case obtained from the system requirement *R1*. The results of the test case execution are displayed in Figure 10.

B. Lessons Learned

In order to successfully generate requirements-based test cases with the SCADE toolchain, test engineers must understand the innerworkings of SCADE state machines. According



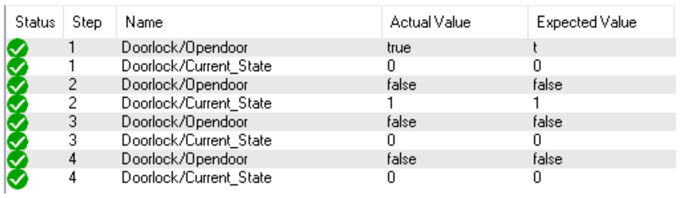
```
# ------------------------------------------------------------
# Simulation scenario file for SCADE Simulator
# Task:  Doorlock_proof.output0
# Model: Verifier
# Node:  Doorlock
# ------------------------------------------------------------

SSM::set Doorlock_proof/Doorknob t
SSM::set Doorlock_proof/Start f
SSM::cycle

SSM::set Doorlock_proof/Doorknob t
SSM::set Doorlock_proof/Start t
SSM::cycle

SSM::set Doorlock_proof/Doorknob f
SSM::set Doorlock_proof/Start f
SSM::cycle

SSM::set Doorlock_proof/Doorknob f
SSM::set Doorlock_proof/Start f
SSM::cycle
```

(a)

```
SSM::set Doorlock/Doorknob t
SSM::set Doorlock/Start f
SSM::check Doorlock/Opendoor t
SSM::check Doorlock/Current_State 0
SSM::cycle

SSM::set Doorlock/Doorknob t
SSM::set Doorlock/Start t
SSM::check Doorlock/Opendoor f
SSM::check Doorlock/Current_State 1
SSM::cycle

SSM::set Doorlock/Doorknob f
SSM::set Doorlock/Start f
SSM::check Doorlock/Opendoor f
SSM::check Doorlock/Current_State 0
SSM::cycle

SSM::set Doorlock/Doorknob f
SSM::set Doorlock/Start f
SSM::check Doorlock/Opendoor f
SSM::check Doorlock/Current_State 0
SSM::cycle
```

(b)

Figure 9. a) Trace generated with the SCADE Design Verifier out of the System Requirement *R1*; b) Test Case generated from the Trace in a) with the SCADE Simulator.



Status	Step	Name	Actual Value	Expected Value
✓	1	Doorlock/Opendoor	true	t
✓	1	Doorlock/Current_State	0	0
✓	2	Doorlock/Opendoor	false	false
✓	2	Doorlock/Current_State	1	1
✓	3	Doorlock/Opendoor	false	false
✓	3	Doorlock/Current_State	0	0
✓	4	Doorlock/Opendoor	false	false
✓	4	Doorlock/Current_State	0	0

Figure 10. Execution of the System under Test with the Generated Test Case.

to their semantics, the data flow which connects the inputs to the outputs of a state in a SCADE state machine is executed synchronously in a single cycle, before any active transitions in the system model can be fired.

Often, it is necessary to know the state in which the computation of state variables used within a cycle originated. This is why it is often indispensable to make use of the `last` operator, especially for system requirements formulated over state variables. Take for example LTL formulae of the form $G(\mu \rightarrow X\psi)$ in which the logical formulae μ and ψ are expressed over state variables. In this case, the SCADE operator `last` is needed for the state variables in μ as well as for those in ψ, in order to get the state from which the calculation within a cycle starts. The use of the SCADE operator `last` can vary, depending on the system requirement and on the implementation of the system model. Since we found no sound way to normalize it, we decided to enable the use of this operator. Indeed, syntactically the `last` operator is considered as a terminal symbol, yet it has effect on LTL operators. To be more specific, the LTL formula $G(a \leq 10 \rightarrow X(\textbf{last}'\,b = 5))$ is semantically equivalent to the LTL formula $G(a \leq 10 \rightarrow b = 5)$, where b is a state variable in the current state. Here, the effect of the `last` operator on the LTL operator *X (neXt)* is similar to that of the absorbtion law in propositional logic. It is then up to the test engineers to decide whether the usage of the `last` operator is appropriate, depending on the system requirements and on the SCADE system model.

Furthermore, one has to consider that the link between SCADE and LTL is not perfect. Nondeterministic Büchi automata are usually used in automata-based LTL model checking to construct the product transition system of the negated property and the system model [16]. The SCADE Design Verifier is built on top of a SAT-Solver [13], with the property observer as its interface. This allowed us to use nondeterministic Büchi automata as a uniform approach to create chronological state sequences from LTL formulae. However, for LTL (sub-)formulae of the type $\mu\ U\ \psi$, the generated NBA contains non-accepting states. These states can have recursive transitions, which always fire. This behavior leads to endless loops, which in turn can generate wrong results. We solved this problem by adding a counter which increments up to a predefined maximum value everytime a recursive transition of a non-accepting state fires. The counter is then reset, if the next state is an accepting state. The maximum value can be defined by the test engineer. The defined value should not be too high, as it may cause long or indeterminate system runs, and not too low, so that correctness can be ensured.

VII. Conclusion and Future Work

In this paper, we implemented the concept defined by Aniculaesei et al. [2] on the industrial toolchain ANSYS SCADE, in order to generate test cases straight from the system requirements formalized in LTL and the SCADE system model. We used the SCADE Design Verifier as model checker to deliver test inputs in form of traces, which were then simulated to calculate the corresponding test outputs. The test cases, containing the test inputs and outputs, were assembled in SCADE scenario files (`.sss`-files). The scenario files are given as input to the SCADE Testing Environment in order to automatically execute the test cases on the System under Test, in this case the SCADE system model itself.

In order to connect LTL with SCADE, we needed to express chronological sequences of states over infinite time. For this purpose, we generated nondeterministic Büchi automata from formalised requirements in LTL by applying the algorithm defined by Gerth et al. in [17]. Here, we found out that this connection is not all-embracing when non-accepting states with recursive transitions occur in the generated NBA (see Section VI). Since we found no uniform concept for mapping of the LTL time model onto the SCADE time model, we enabled the use of `last` operator from SCADE in terminal symbols and we gave the user the liberty to decide upon the usage of this operator within LTL formulas. Based on our case study, valid test cases were generated (see Figure 9) and executed (see Figure 10).

In future work, we want to extend the concept for requirements-based test case generation developed for the SCADE toolchain with the approach in [2]. We plan to apply the extended concept on a more complex system in the automotive field. Here we plan to use construction methodologies for test case generation via model checking based on three different criteria, requirements coverage (RC), antecedent coverage (AC) and unique first cause coverage (UFC) as defined in [3] [7]. Furthermore, we plan to measure the quality of our test suites with respect to MC/DC Coverage [19] [20] by creating mutants on code level.

References

[1] M. Fockel, J. Holtmann, and M. Meyer, "Mit Satzmustern hochwertige Anforderungsdokumente effizient erstellen (*engl.*: Using Sentence Patterns to Efficiently Create High-quality Requirements Documents)," OBJEKTspektrum, no. RE/2014, jun 2014, pp. 1–4.

[2] A. Aniculaesei, F. Howar, P. Denecke, and A. Rausch, "Automated generation of Requirements-Based Test Cases for an Adaptive Cruise Control System," in 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST@SANER). IEEE, 2018, pp. 11–15.

[3] M. W. Whalen, A. Rajan, M. P. Heimdahl, and S. P. Miller, "Coverage metrics for requirements-based testing," in Proceedings of the 2006 international symposium on Software testing and analysis, L. Pollock and M. Pezzè, Eds. New York, NY: ACM, 2006, pp. 25–36.

[4] Esterel Technologies S.A.S., Scade Language Reference Manual. ANSYS, Inc., 2018.

[5] C. Rupp and SOPHISTen, "Schablonen für alle Fälle (*engl.*: Patterns for all Purposes)," SOPHIST GmbH, 2016. [Online]. Available: https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/MASTeR_Broschuere_3-Auflage_interaktiv.pdf

[6] S. Rayadurgam and M. P. E. Heimdahl, "Generating mc/dc adequate test sequences through model checking," in Proceedings of 28th Annual NASA Goddard Software Engineering Workshop. IEEE, 2003, pp. 91–96.

[7] M. Staats, M. W. Whalen, M. P. E. Heindahl, and A. Rajan, "Coverage metrics for requirements-based testing: Evaluation of effectiveness," in Proceedings of the Second NASA Formal Methods Symposium. NASA, april 2010, pp. 161–170.

[8] P. P. Kulkarni and Y. Joglekar, "Generating and Analyzing Test cases from Software Requirements using NLP and Hadoop," International Journal of Current Engineering and Technology, vol. 4, no. 6, 2014, pp. 3934–3937.

[9] N. Kosindrdecha and J. Daengdej, "A test case generation technique and process," Journal of Software Engineering, vol. 4, no. 4, 2010, pp. 265–287.

[10] C. Nebut, S. Pickin, Y. Le Traon, and J.-M. Jézéquel, "Automated Requirements-based Generation of Test Cases for Product Families," in Proceedings of 18th IEEE International Conference on Automated Software Engineering, 2003. Montreal, Quebec, Canada: IEEE, 2003, pp. 263–266.

[11] S. P. Miller, M. W. Whalen, and D. D. Cofer, "Software model checking takes off," Commununications of the ACM, vol. 53, no. 2, feb 2010, pp. 58–64. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646372

[12] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm, "Integration of Formal Analysis into a Model-based Software Development Process," in Proceedings of the 12th International Conference on Formal Methods for Industrial Critical Systems, ser. FMICS'07. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 68–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=1793603.1793612

[13] A. Bouali and B. Dion, "Formal verification for model-based development," SAE transactions, 2005, pp. 171–181.

[14] Esterel Technologies S.A.S., SCADE Suite User Manual. ANSYS, Inc., 2018, vol. SCS-UM-19 - DOC/rev/35771-03.

[15] ——, Scade Language Primer. ANSYS, Inc., 2018.

[16] C. Baier and J.-P. Katoen, Principles of model checking. Cambridge, Massachusetts, USA: MIT Press, 2008.

[17] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV. London, UK, UK: Chapman & Hall, Ltd., 1996, pp. 3–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=645837.670574

[18] "PLY (Python Lex-Yacc)," accessed: 21.03.2019. [Online]. Available: https://www.dabeaz.com/ply/

[19] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," Software Engineering Journal, vol. 9, no. 5, 1994, pp. 193–200.

[20] K. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage," National Aeronautics and Space Administration, Langley Research Center, Tech. Rep., 2001. [Online]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20010057789.pdf