Property-oriented Model-Based Testing With Fuzzing – Technical Report 09/2020 –

Wen-ling Huang, Niklas Krafczyk*, Hoang M. Le**, and Jan Peleska

Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany
huang@uni-bremen.de
niklas@uni-bremen.de
hle@uni-bremen.de
peleska@uni-bremen.de

Abstract. Fuzzing is a popular method for automated test generation in the context of coverage-driven software testing. In this paper, we show its applicability to a completely different testing domain: property-oriented (also called requirements-driven) model-based testing (MBT). In MBT, a reference model of the required behaviour of the system under test (SUT) is elaborated. Based on this model, relevant test cases are identified, either to verify some conformance relation between reference model and SUT, or to check whether the SUT satisfies requirements reflected by the model. By transforming SysML models into simulation code and test cases specified in Linear Temporal Logic (LTL) into observer code, a property-based testing task is mapped to a coverage-driven testing task. This can again be handled by fuzzing. The approach is illustrated by an example of an automotive control system. We present two main results: first, it is shown how combined MBT and property-oriented testing can be performed with *automated* property generation, so that users do not have to specify requirements or test cases using some temporal logic. Second, and somewhat surprising, we demonstrate that for models of medium complexity, automated test case generation for property-oriented MBT can be performed using only a fuzzer, without the help of SMT solvers, as used, for example, in concolic testing.

Keywords: Property-oriented testing, requirements-driven testing, fuzzing, SysML

1 Introduction

1.1 Motivation

Classical *model-based testing (MBT)* is based on a reference model specifying the expected behaviour of the system under test (SUT). Test cases are derived from

* Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 407708394.
** Funded by CRDF, University of Bremen.

the model with the objective to create test suites with high test strength for detecting conformance violations of the SUT behaviour in comparison to the model behaviour. Conformance is usually specified by I/O-language equivalence [7,27], reduction (I/O-language containment) [8], or by some other sort of refinement property [24]. Testing for model conformance is quite popular and effective in certain areas, for example, in the domain of protocol testing [16].

In other domains, such as the avionic, automotive, and railway domains, the applicable standards [30,9,6] do not have a notion of model conformance. Instead, both development artefacts (design documents, code, ...) and verification artefacts (reviews, analyses, test, ...) need to be traced back to requirements. This is denoted by the *requirements-driven approach* to system development and verification. Certification credit for test suites is not given for establishing some sort of model conformance, but for showing that every requirement is adequately covered by test cases, and that the test suite execution with the SUT shows that no uncovered code remains after having tested all requirements. Requirements can be formally regarded as *properties*, that is, sets of (usually infinite) computations. In classical *property-oriented testing*, test engineers abstain from creating models, but they create property specifications instead. Formal approaches to property-oriented testing specify both requirements and test cases by means of some temporal logic, typically a variant of LTL. In [13], an overview of this research field is given. As stated there, the main motivation for property-oriented testing is the possibility to reduce test suite size in comparison to a complete model-based conformance test suite: while the latter requires test cases to verify *all* possible behaviours in relation to a reference model, the former only requires to investigate the subset of possible behaviours which are relevant to verify a certain set or properties (i.e. requirements).

In this paper, a combined approach of MBT and property-oriented testing is advocated, with the objective to obtain certification credit for requirements-driven test suites with an automation degree as high as possible. For automated calculation of concrete test data, *fuzzing* is used.

1.2 Main Contributions

This paper has two main contributions.

1. Fully automated test case generation. We show that the availability of a model allows for automated generation of temporal logic formulas representing symbolic test cases[1] for a given requirement. As a consequence, the combined MBT and property-oriented approach described here reduces the manual effort for creating requirements-based test suites to creating the SysML model and tracing the requirements to structural or behavioural model elements. We consider this to be an essential achievement for property-oriented testing, since tem-

[1] These test cases are called symbolic because they are specified by formulas still possessing free variables, to be "instantiated" with concrete test data in order to obtain a concrete test case.

poral logic has not become an accepted specification technique in industry until today, whereas model development is considered to be a state-of-the-art technique. As proof of concept, we present a SysML model of a controller from the automotive domain, and exploit the SysML-specific approach to requirements tracing. We illustrate how modelling and requirements tracing in the SysML model is enough to produce test suites and test executions in a fully automated way.

2. Concrete test generation by fuzzing. Creating concrete test cases from symbolic ones requires some variant of constraint solving. In Concolic testing [10] and test generation techniques based on behavioural model semantics [22], concrete test data is usually derived from code or from a model and a test case specification in temporal logic. To obtain these solutions SMT solvers are frequently used.

In this paper, we explore a novel application of fuzzing to generate test cases in MBT. We leverage fuzzing as a coverage-maximising test generation engine to generate valid test cases demonstrating required behaviour. This is in stark contrast to previous applications that employ fuzzing as a form of negative testing by mostly generating syntactically or semantically malformed input data. To enable the application of fuzzing techniques, we transform the reference model into a semantically valid model simulation coded in C^{++}. The temporal logic formulas representing test cases are transformed into observer methods, such that covering certain code portions of the observer indicates fulfilment of the original formula with the concrete inputs and time stamps selected by the fuzzer.

It should be emphasised that the objective of this contribution is not to present new testing methods, but to show how combinations of existing methods and tools can be used to solve requirements-driven testing tasks in a more effective way, by increasing automation.

1.3 Overview

In Section 2, an overview over the complete tool chain for property-oriented MBT with fuzzing is presented. In Section 3, a controller from the automtive domain is presented as a SysML model. In Section 4, we present an LTL specification of the required controller behaviour. These specifications will not be used in the remainder of the paper: they have been elaborated to illustrate that the "pure" property-based approach to testing is too hard to be applied by practitioners from industry. In Section 5, the functionality of the automated test case generator is described, using the example from Section 3. In Section 6, an evaluation of the fuzzer performance is presented for different symbolic test cases created for the automotive controller. Section 7 contains the conclusion. A definition of LTL semantics is provided in Appendix A.

We refer to related work throughout this paper, whenever appropriate.

2 Testing Tool Chain and Underlying Methods

Modelling Tool. The tool chain used for property-oriented MBT is shown in Fig. 1. A SysML modelling tool is used for specifying the design and the expected behaviour of the SUT (an example is shown in Section 3). For the results elaborated in this paper, the Papyrus SysML 1.6 tool [29] has been used[2]. The tool stores models in XMI-format (XML Meta Data Interchange[3], a specialisation of XML), as defined by the Object Management Group OMG responsible for UML/SysML standardisation.

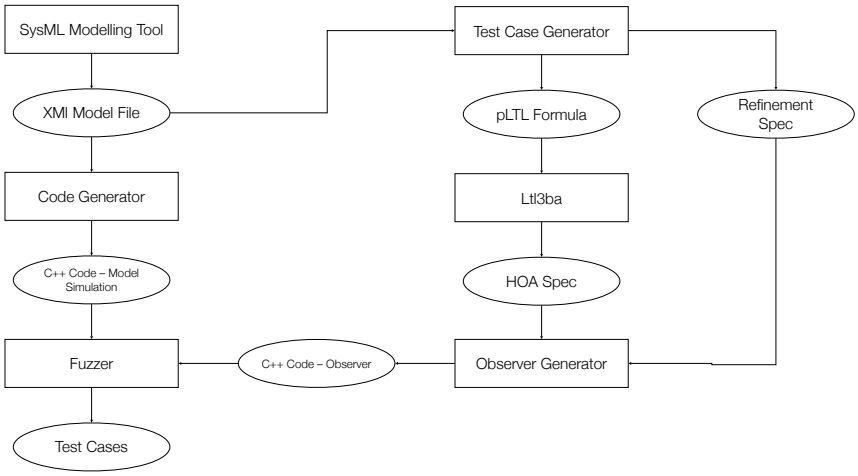

Fig. 1. Tool chain.

Code Generator. For parsing XMI files, we use the well-established library Libxml2[4]. Based on this parser, real-time capable C^{++} simulation code can be generated for SysML models. The code generator is outside the scope of this paper, but a detailed description can be found in the lecture notes [21].

Test Case Generator. The XMI file encoding the model in textual form is also parsed by the test case generator. The generator explores the SysML satisfy-relationship linking requirements to structural and behavioural model elements,

[2] `https://marketplace.eclipse.org/content/papyrus-sysml-16`
[3] `https://www.omg.org/spec/XMI/About-XMI/`
[4] `http://xmlsoft.org`

in particular, to state machine states and transitions. As a result of this exploration, symbolic test cases are created as temporal logic formulas written in LTL (see Appendix A). The term 'symbolic' means that the test cases are represented by formulas still containing free variables from V_I, V_O, V_M, so that the concrete test data is not yet completely determined. The LTL formula is delivered in two components: the first is a *propositional* LTL (pLTL) formula, containing names of atomic propositions only. The second is the *refinement specification* which associates concrete unquantified first-order expressions with each atomic proposition.

The functions of the test case generator are described in more detail in Section 5.

ltl3ba. Given the pLTL formula part of a symbolic test case, the Büchi automaton accepting the LTL formula is then generated using `ltl3ba` [3] which can present its results in the machine-readable *Hanoi Omega Automata (HOA)* format [2].

Observer Generator. The observer generator produces C^{++} code which checks whether the Büchi automaton associated with a pLTL formula has entered an accepting state. To this end, observer code is generated which abstracts concrete interface data and model variable valuations to the atomic propositions used in the pLTL formula in every evaluation cycle. The test case formulas we use (see examples in Section 5) are of the kind where an accepting state just has to be reached once, so that every infinite continuation of the trace performed so far results in an infinite trace fulfilling the test case formula.

To produce the observer code for identifying accepting states of the Büchi automaton associated with the test case formula, the HOA-format is parsed using the `cpphoafparser` [2] which can call methods of a class for the elements of an omega automaton in the fashion of the producer consumer pattern. We leverage this by implementing a consumer generating C^{++} code describing the aforementioned observer as follows: Whenever the parser accepts an expression as a state or edge of the omega automaton to be parsed, our consumer is notified, including the acceptance sets they are part of and, in the case of automata edges, the label of the edge. We then create a boolean state vector which has one element per automaton state and initialise it such that the initial state is marked. The observer is then generated, containing this state vector, a set of transitions to be covered for each accepting set. Additionally, the observer contains a function evaluating the state vector and current program execution state, determining the enabled state transitions and updating the accepting sets and the state vector. Afterwards, the evaluation of the acceptance condition is returned, allowing to detect formula satisfaction.

Fuzzing with LibFuzzer Fuzzing, or fuzz testing, is an automated software testing technique that repeatedly executes a program with random inputs until

an error condition can be triggered. Since its inception in the early 1990s [15], fuzzing has evolved to one of the most widely used techniques in security testing. The main focus of fuzzing is to detect security-related bugs that can lead to vulnerabilities. Its sweet spot is memory-unsafe languages such as C/C++, where low-level bugs such as buffer overflows, use-after-free situations, etc. can easily creep into programs.

In the last few years, *coverage-guided greybox fuzzing* (CGF) has risen to prominence. Many CGF tools, also referred to as *fuzzers* in this paper, such as AFL [31] or LibFuzzer [1], have been used to find thousands of serious bugs in widely-used critical open-source projects [26]. The key efficiency of CGF lies in its ability to discover different execution paths at nearly native speed. The basic idea behind this efficiency is surprisingly simple. CGF adds lightweight instrumentation to the *software unter test* to measure code coverage of each executed input. This coverage information is then used to guide the process of creating new inputs by (randomly) mutating existing ones (e.g. by favouring mutation of inputs whose execution discovered new coverage). Since its first academic treatment [5], CGF has attracted significant research attention and been improved in many ways, we refer to the excellent survey [14] for more details.

The fuzzer acts as a concrete test data generator for the symbolic test cases specified by LTL formulas. To this end, it executes the model simulation code, where the observer is called at the end of every main loop cycle, and the fuzzer's input data generator is called at the beginning of each cycle, to assign new (randomised) data to the SUT input interfaces, and to advance the model execution time by an adequate amount. The fuzzing goal is to create a sequence of input vectors and associated model execution time stamps, such that the observer covers the code indicating acceptance of the formula. Since the first-order LTL formula representing the test case typically refers to internal model variables and output variables, as well as input variables, the model simulation needs to be executed together with the observer: otherwise, the changes triggered by input changes and by advancement of the time would not lead to the expected changes in model outputs and internal model variables. In contrast to that, an LTL formula for a test case referring to input variables from V_I only, would not need a model simulation. It would suffice to execute the observer alone until the code indicating that the accepting state has been reached. This, however, would require that the users specify *every* input explicitly, so that practically all automation benefits are lost.

If the fuzzer covers the observer code indicating that an accepting state of the Büchi automaton has been reached, the test data generation is stopped, and the *concrete test case* consists of the sequence of input vectors with associated time stamps from simulation start until the accepting state has been reached.

The witness traces of these formulas are suitable for exploring whether the associated requirement has been adequately implemented. This is illustrated below in Section 5 using the sample model introduced in Section 3.

We use *in-process* coverage-guided fuzzing offered by LLVM LibFuzzer [1]. This form of fuzzing is much faster than traditional out-of-process fuzzing, which forks a new process for each execution of the *main* function, but requires the global state of the SUT to remain largely unchanged or to be reset between executions. Since the SUT is automatically generated and does not involve external calls, capturing and restoring the initial global state can also be fully automated. Essentially, we follow the transformation steps outlined in [12].

```
1 int LLVMFuzzerTestOneInput(uint8_t *data, size_t size) {
2     FuzzerRestoreGlobalState();
3     MakeGlobalCopy(data, size);
4     used = 0;
5     ExecuteModelAndObserver();
6 }
```

Fig. 2. Conceptual implementation of a fuzz target

```
1 int receiveModelInput(LeverPosition& lvr, Boolean& emer, Boolean& ign,
    BatteryVoltage& batvol, int& timeToNextInput) {
2     if (used + 2 > size) // not enough bytes, end of input sequence
3         return 0;
4     if (data[used] & 2)      lvr = NEUTRAL;
5     else if (data[used] & 1) lvr = LEFT;
6     else                     lvr = RIGHT;
7     emer                         = data[used] & 4;
8     ign                          = data[used] & 8;
9     batvol                       = data[used + 1] % 21;
10    timeToNextInput              = data[used + 2];
11    timeToNextInput             += 1;
12    used                        += 3; // consumed 3 bytes
13    return 1;
14 }
```

Fig. 3. Translation of random bytes into input data for the model

Furthermore, LibFuzzer requires the definition of a fuzzing target, i.e. an implementation of the *LLVMFuzzerTestOneInput* function. The *main* function provided by LibFuzzer will repeatedly call this function with fuzz inputs in a loop to perform fuzzing. Each fuzz input consists of an array of random bytes and its size. Figure 2 shows a conceptual implementation of *LLVMFuzzerTestOneInput*. First, the initial global state is restored. Then, the given fuzz bytes are copied into a global array and the number of bytes already consumed for fuzzing is set to zero. The translation of the fuzz bytes into a sequence of input vectors (with corresponding timestamps) is then carried out on-the-fly together with the execution of the SUT and the observer. Figure 3 shows the implementation for our example. Please note that input constraints must be considered in this step

to avoid generating invalid input. Since the input constraints are known, this step also can be fully automated.

An optimisation worth mentioning is the deeper integration of the observer semantics into the fuzzing loop. In many cases, the whole boolean state vector of the observer becomes unmarked during an execution. Without at least one marked state, it is impossible to eventually reach an acceptance state. Thus, this execution can be terminated early before all fuzz bytes are consumed. This simple optimisation boosts the fuzzing performance significantly as shown in the evaluation later.

Test Execution Environment. The test execution environment is not shown in Fig. 1, but is also available: a variant of the code generator creates *test oracles* fom the SysML model, as described in [19]. The oracle runs in back-to-back fashion with the SUT during the test execution and checks whether the concrete outputs produced by the SUT conform to the expected output calculated by the oracle. To this end, the concrete test sequence sent step by step to the SUT is provided simultaneously to the oracle. The oracle, however, never produces outputs but reads the SUT outputs and compares them to the internally calculated expected values.

3 Example: Turn Indication Controller

3.1 Function Description

As a running example, we study a turn indication and emergency flashing system as used in modern cars. We have simplified the example for the purposes of this paper; a real-world system description and associated model derived from the original specifications of a car manufacturer has been presented in [23], but would be too large for the context of this paper.

The turn indication function processes inputs from the turn indication lever which has three positions `NEUTRAL`, `LEFT`, and `RIGHT`. In the left and right position of the lever, the turn indication lights on the left-hand side (LHS) or right-hand side (RHS), respectively, are flashing with a frequency of approx. 1.5 Hz. More precisely, flashing has a period of 660ms. In each period, the signs stay on for 340ms, followed by an off-phase of 320ms. Turn indication flashing is only operable if the ignition is switched on. Moreover, turn indication provides a tip flashing function: if the turn indication lever is switched from NEUTRAL to LEFT or RIGHT, but then back to NEUTRAL before 440ms has passed, flashing is continued until the end of a tip flashing phase consisting of three flashing periods (1980ms).

The emergency flashing function activates the LHS and RHS indication lights in synchrony. A dashboard button can be pressed to start this function. When the button is released, emergency flashing is switched off immediately. Just as turn indication, emergency flashing has a flashing period of 660ms with on and off phases of 340ms and 320ms, respectively, if the ignition is switched on. In

contrast to the turn indication function, however, emergency flashing is also operable when the ignition is switched off: in this case, the duration of the on/off phases is reversed, in order to save battery power.

Turn indication and emergency flashing provide an override functionality: if left/right turn indication flashing is active when the emergency flashing button is pressed, emergency flashing starts immediately with a new ON flashing period on all indication lights. If then the emergency button is released, turn indication flashing is resumed if the turn indication lever is still in the previous position. If the turn indication lever is put into NEUTRAL while emergency flashing is active, all lights are switched off when emergency flashing is switched off. Conversely, if the turn indication lever position changes to a new non-neutral position while emergency flashing is active, emergency flashing is suspended in favour of turn indication flashing according to the new lever position. After the turn indication lever has been put back into neutral position, emergency flashing is resumed if the emergency button is still pressed.

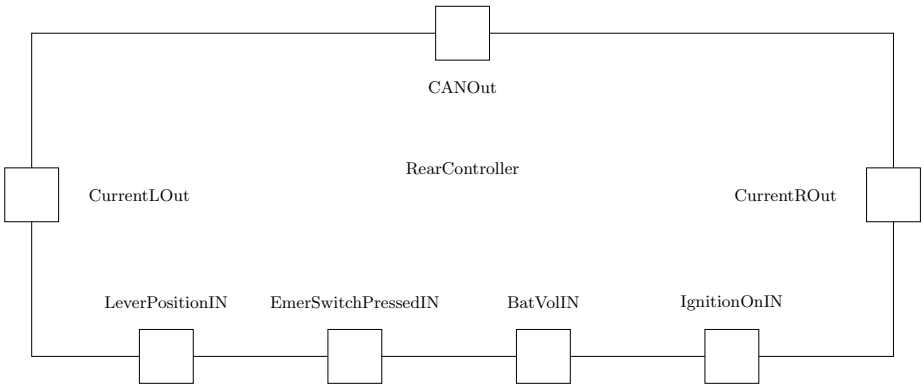

Fig. 4. Interfaces of the rear controller.

Regardless of the indication function, the lamp current is switched off immediately if the battery voltage is outside its admissible range $[10V, 15V]$. The control logic keeps its state (emergency flashing or turn indication flashing left-/right, or off), but no lamp current is provided until the battery voltage is back in range (if ever).

In the remainder of this section, a SysML model is presented, describing the structure and the behaviour of the rear controller.

3.2 Interface

In modern vehicles, the whole turn indication control function is distributed onto several controllers connected by an automotive bus system, such as CAN.

For the purpose of this paper, we focus on the rear controller with interfaces as depicted in the SysML [18] block diagram in Fig. 4. This controller inputs the turn indication lever status LeverPositionIN, the emergency switch status EmerSwitchPressedIN, the actual battery voltage BatVolIN, and the ignition status IgnitionOnIN. The control logic activates the turn indication lights left and right by providing a suitable positive current on interfaces CurrentLout and CurrentRout, respectively, for switching the rear indication lights on, and cuts of the current for switching them off. The on/off-phases are controlled by the computer. Finally, the rear controller distributes the on/off commands on CAN bus interface CANout to sub-ordinate controllers switching the front indication lights and mirror indication lights. The on message contains the information, whether flashing on the left-hand side or/and the right-hand side is required, and it provides the on-duration of the next period. Each controller switches its lights off on its own account, when the on-duration has expired. Each new flashing period, however, is activated by a new message from the rear controller, so that all controllers are kept in synchrony.

3.3 Structure

The internal rear controller structure is shown in the block diagram in Fig. 5. Part controllogic monitors the inputs LeverPositionIN, EmerSwitchPressedIN, and IgnitionOnIN and decides whether LHS/RHS turn indication, or emergency flashing should be activated. The control decision is passed on port flashCmdOut to part candriver which produces cyclic messages to remote controllers and loopback messages to local parts lampCtrlSlaveLeft and lampCtrlSlaveRight. Each message signifies the beginning of a flashing period and contains the length of the ON-duration, depending on the ignition status. Parts lampCtrlSlaveLeft/Right supervise the length of the ON-phases and send switchIn commands to the LHS and RHS power control parts pwrLeftLamp and pwrRightLamp, respectively. After the end of an ON-phase, lampCtrlSlaveLeft/Right send ¬switchIn commands to the power control parts. These two parts switch the lamps on and off by providing and cutting off the required current to the lamps on port CurrentLOut and CurrentROut, respectively. They always cut off the current if the battery voltage is out of range.

3.4 Behaviour

The behaviour of the parts is modelled by means of state machines. Part controllogic has the state machine shown in Fig. 6 as classifier behaviour. This state machine uses three states, depending on whether emergency flashing, LHS/RHS turn indication flashing, or no flashing should be performed. Additional pseudo states are used to control the branching behaviour. On initialisation, state EmergencyFlashing is entered, if the emergency switch is pressed. This happens regardless of the state of the turn indication lever, since emergency flashing has priority over LHS/RHS-flashing.

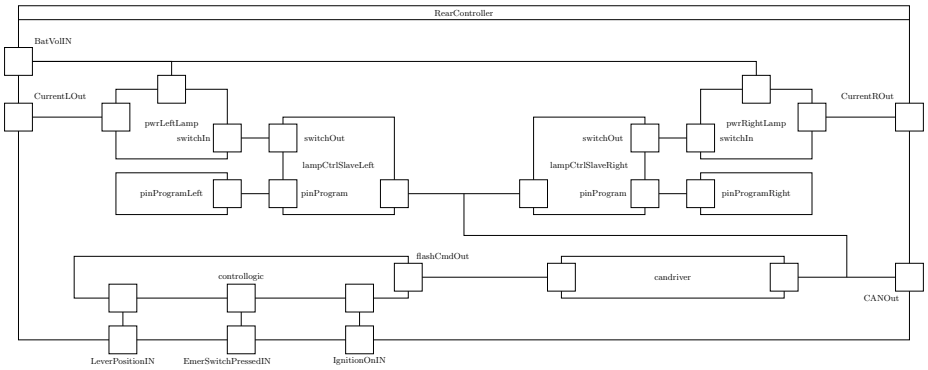

Fig. 5. Rear controller block diagram.

Turn indication is performed in composite state LRFlashing which can only be entered if the ignition is on. The sub-machine in Fig. 7 determines in state UNSTABLE, whether LHS or RHS flashing should be performed. Tip flashing is controlled by introducing a state transition into state STABLE, if the turn indication lever has not been switched back to NEUTRAL within 440ms. From then on, turn indication can be immediately switched off by putting the lever into position NEUTRAL. If the neutral position is taken while still in state UNSTABLE, the state machine remains in this state until three flashing periods have passed, whereafter turn indication is switched off. If the turn indication lever is put into another non-neutral position, a corresponding self loop is performed on composite state LRFlashing, so that a fresh tip flashing period is started.

The different override situations are controlled by direct transitions between states EmergencyFlashing and LRFlashing. Whenever the control logic changes the functional status, a StatusChange signal is sent which alerts the candriver that updated messages need to be distributed.

When describing the functionality of the test case generator, the state machine associated with parts pwrLeftLamp and pwrRightLamp will be discussed. It is shown in Fig. 8; state BATTERY_VOLTAGE_OK is composite with a sub-machine shown in Fig. 9.

4 Behavioural Specifications in LTL

In this section, formal requirements specifications of the rear controller behaviour are presented in quantified first-order LTL. This is meant as an "intellectual exercise", with the objective to demonstrate that model-free specifications in temporal logic are quite difficult to construct, whereas the model-based approach is closer to "imperative programming on a higher level", and therefore easier to introduce in an industrial context.

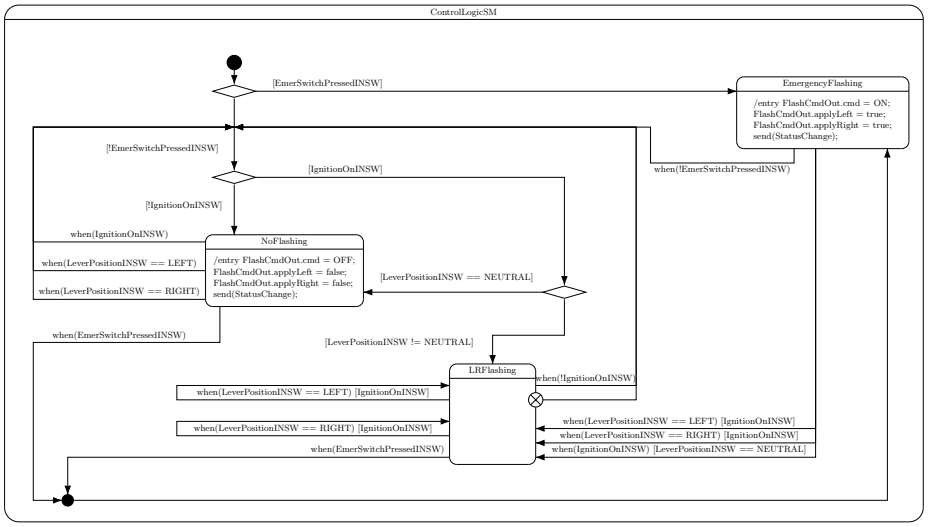

Fig. 6. Control Logic state machine.

For setting up the more complex formulas, it is useful to first define some abbreviations for first-order expressions in Table 1. The abbreviated formulas refer to the interface names introduced in Fig. 4.

Since we need to refer to concrete durations, variable $\hat{t}$ is used to denote the current execution time since system start. Moreover, several Boolean auxiliary variables are introduced to facilitate the expression of certain causal connections reflected in the temporal logic formulas to be specified below: i carries a copy of the ignition status observed in certain states along a path, o_e is true if emergency flashing has been overridden by turn indication left or right, and o_{turn} indicates whether turn indication left/right has been overridden by emergency flashing. Boolean variable tf is true as long as a tip flashing phase (up to 1980ms) applies. The utilisation of these auxiliary variables is explained below in the context of the requirements where they occur.

Requirement 1. Off conditions. The following formulas specify off conditions for both turn indication and emergency flashing. In these situations, the lamp current is cut off. It states that indications lights must be off as long as

- the turn indication lever is in neutral position and previous tip flashing phases have terminated (if any), or the ignition is switched off (so that turn indication is inoperative), and
- the emergency switch is in off position.

In this situation, all override conditions are cancelled, and no tip flashing phase can start.

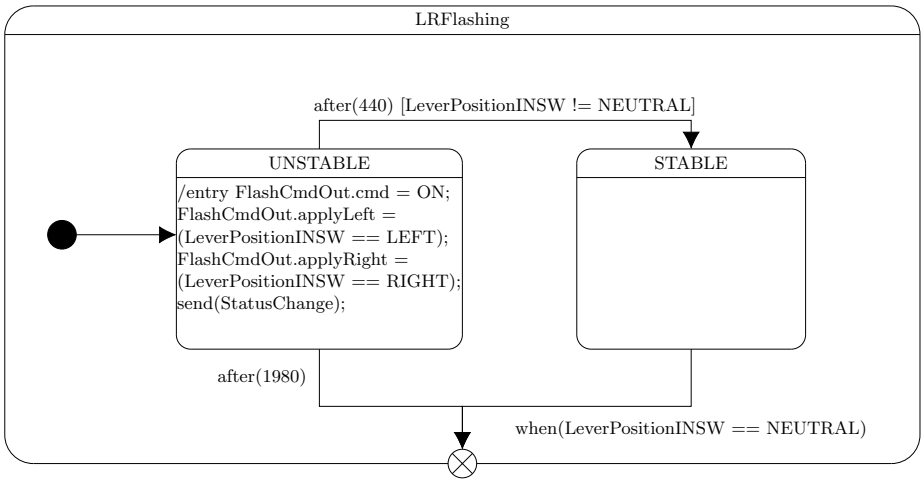

Fig. 7. Sub-machine LRFlashing.

$$\text{OFFCND} \equiv \mathbf{G}\big(((n \wedge \neg tf) \vee \neg ig) \wedge \neg e \Rightarrow \tag{1}$$

$$\mathbf{X}(\text{off}_{\ell r} \wedge \neg o_e \wedge \neg o_{\text{turn}} \wedge \neg tf)\big) \tag{2}$$

Requirement 2. Turn indication left. Formula TLEFT below specifies the trigger condition and the behaviour of turn indication flashing on the LHS. The trigger condition A becomes true when the turn indication lever is switched from any position other than LEFT (i.e. from NEUTRAL or RIGHT) to LEFT. This is expressed by $\neg\ell \wedge \mathbf{X}\ell$. Turn indication, however, is only activated if the ignition is on, so ig must also hold together with ℓ. Formula A also specifies that turn indication may be "implicitly" switched on by the ignition being switched on, if the turn indication lever has been put into a non-neutral position before. Emergency flashing has priority over turn indication flashing. Therefore, A also requires that emergency flashing must not be simultaneously activate. Otherwise turn indication is immediately overridden, as explained below. LHS turn indication is resumed as soon as the override condition no longer applies ($\neg o_{\text{turn}}$) and the turn indication lever is in position LEFT.

The formula specifying the LHS-flashing behaviour is decomposed into a conjunction $B \wedge T$, where the second operand deals with the flag tf indicating a tip flashing phase. To explain the behaviour B triggered by condition A, quantified first-order LTL is needed, because we need to refer to the point in time LHS flashing was activated. Rigid variable t_0 stores the value of the model execution time when $\mathbf{X}(ig \wedge \ell)$ became true. The indication behaviour is encoded in the weak until sub-formula of B, where sub-formulas C, D specify the indication behaviour, while E specifies the condition when C, D no longer apply

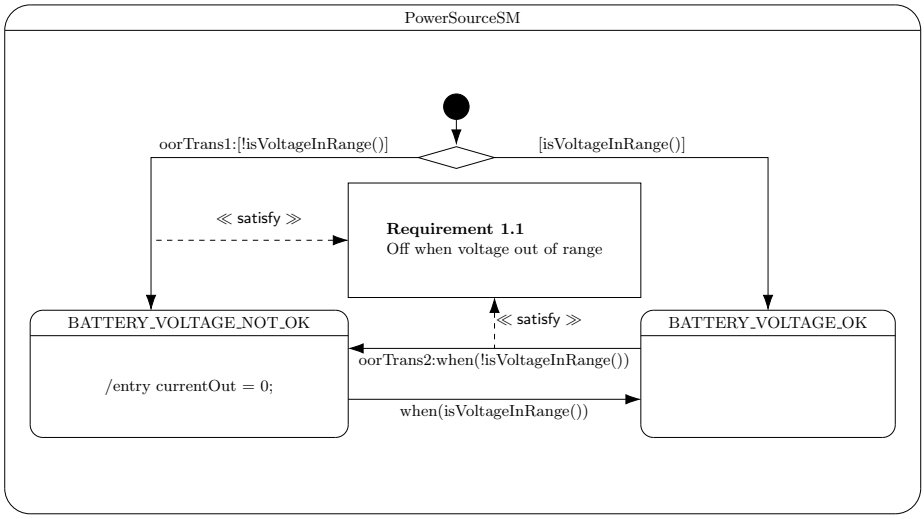

Fig. 8. PowerSource state machine, top-level.

(i.e. LHS flashing is terminated). The weak until operator $\mathbf{W}$ is used because it is not mandatory that the LHS flashing behaviour should terminate. Predicate C states that the LHS lamp is provided with current (on_ℓ) while the flashing period is in its on-phase (% denotes the modulo operator), provided that the battery voltage is in range (recall the definition of on_ℓ in Table 1). Formula D states that the LHS lamp current is cut off during the off phase $(340, 660)$. The RHS-lamp never gets current during LHS-flashing ($\neg c_r$-conjunct in on_ℓ and $\text{off}_{\ell r}$).

$$\text{TLEFT} \equiv \mathbf{G}\big(A \Rightarrow \mathbf{X}(B \wedge T)\big) \tag{3}$$
$$A \equiv (\neg ig \vee \neg \ell \vee o_{\text{turn}}) \wedge \mathbf{X}(ig \wedge \ell \wedge \neg o_{\text{turn}}) \tag{4}$$
$$B \equiv \exists t_0.\big(t_0 = \hat{t} \wedge ((C \wedge D)\ \mathbf{W}\ E)\big) \tag{5}$$
$$C \equiv (\hat{t} - t_0)\%660 \leq 340 \Rightarrow \text{on}_\ell) \tag{6}$$
$$D \equiv (\hat{t} - t_0)\%660 > 340 \Rightarrow \text{off}_{\ell r}) \tag{7}$$
$$E \equiv \neg ig \vee r \vee (n \wedge \neg tf) \vee o_{\text{turn}} \tag{8}$$
$$T \equiv \exists t_0.\big(t_0 = \hat{t} \wedge (T_1 \wedge T_2 \wedge T_3 \wedge T_4\ \mathbf{W}\ \neg ig \vee r \vee o_{\text{turn}})\big) \tag{9}$$
$$T_1 \equiv \hat{t} - t_0 < 440 \Rightarrow tf \tag{10}$$
$$T_2 \equiv \hat{t} - t_0 = 440 \wedge \neg n \Rightarrow (\neg tf\ \mathbf{W}\ \neg \ell) \tag{11}$$
$$T_3 \equiv \hat{t} - t_0 \leq 440 \wedge n \Rightarrow \tag{12}$$
$$(tf\ \mathbf{U}\ (\hat{t} - t_0 = 1980 \vee \neg ig \vee o_{\text{turn}} \vee \neg n)$$
$$T_4 \equiv \hat{t} - t_0 > 1980 \Rightarrow \neg tf \tag{13}$$

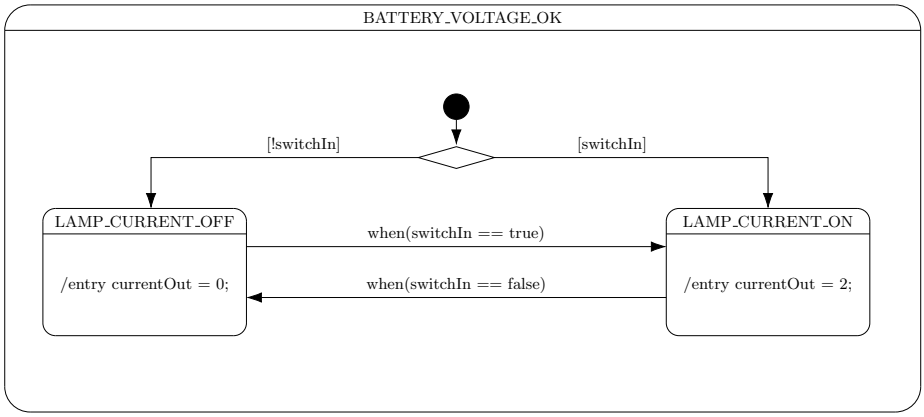

Fig. 9. PowerSourceSM sub-machine, BATTERY_VOLTAGE_OK.

The LHS flashing behavior $C \wedge D$ no longer applies as soon as the termination condition E becomes true: this happens as soon as one of the following conditions hold:

- The ignition is off (turn indication is only operable when the ignition is on).
- The turn indication lever has been switched into position RIGHT.
- The turn indication lever has been put back into neutral position and a potential tip flashing phase no longer applies ($\neg tf$).
- Turn LHS indication has been overridden by emergency flashing (o_{turn}).

The effect of E becoming true is specified in other formulas below; TLEFT only specifies that LHS-flashing is no longer assured when E applies.

The setting of the tip flashing phase flag tf in case of LHS flashing is controlled by formula T. Just as in B, the start time of LHS flashing is recorded in t_0. The sub-rules $T_1, \ldots, T_4$ state that

- tf is always true before 440ms have passed since the LHS-flashing activation (Formula T_1).
- tf switches to false if the turn indication lever is still in the non-neutral position at the end of the 440ms phase (Formula T_2).
- If the turn indication lever is put into neutral position *before* 440ms have passed (Formula T_3), the tf-flag remains true for 1980ms (i.e. three flashing periods) since start of LHS-flashing or until
  - the ignition has been switched off, or
  - turn indication has been overridden, or
  - the turn indication lever is no longer in neutral position.

  When at least one of these three conditions applies, the value of the tf-flag is specified by other formulas.
- The tf-flag is always reset to false after 1980ms have passed (Formula T_4).

Table 1. Abbreviations for first-order expressions

v	$\equiv$	$\mathsf{BatVolIN} \in [10, 15]$	battery voltage is in range
e	$\equiv$	$\mathsf{EmerSwitchPressedIN}$	emergency switch is pressed (=active)
n	$\equiv$	$\mathsf{LeverPositionIN} = \mathsf{NEUTRAL}$	turn indication lever in neutral position
ℓ	$\equiv$	$\mathsf{LeverPositionIN} = \mathsf{LEFT}$	turn indication lever in left position
r	$\equiv$	$\mathsf{LeverPositionIN} = \mathsf{RIGHT}$	turn indication lever in right position
c_l	$\equiv$	$\mathsf{CurrentLOut} > 0$	LHS lamp current is on
c_r	$\equiv$	$\mathsf{CurrentROut} > 0$	RHS lamp current is on
on_ℓ	$\equiv$	$(c_\ell \Leftrightarrow v) \wedge \neg c_r$	LHS lamp current is on iff voltage ok
on_r	$\equiv$	$\neg c_\ell \wedge (c_r \Leftrightarrow v)$	RHS lamp current is on iff voltage ok
$\mathrm{on}_{\ell r}$	$\equiv$	$(c_\ell \Leftrightarrow v) \wedge (c_r \Leftrightarrow v)$	both sides on iff voltage ok
$\mathrm{off}_{\ell r}$	$\equiv$	$\neg c_\ell \wedge \neg c_r$	lamp current off, both sides
ig	$\equiv$	$\mathsf{IgnitionOnIn}$	ignition is on

The rules for setting tf in case of LHS tip flashing no longer apply as soon as the ignition is switched off, RHS-flashing is activated, or turn indication is overridden by emergency flashing. This is specified in the right operand of the weak until operator in T.

Requirement 3. Start and continue turn indication right. RHS-flashing is specified in analogy to LHS-flashing.

$$\mathrm{TRIGHT} \equiv \mathbf{G}\big(A_r \Rightarrow \mathbf{X}(B_r \wedge T_r)\big) \tag{14}$$

$$A_r \equiv (\neg ig \vee \neg r \vee o_{\mathrm{turn}}) \wedge \mathbf{X}(ig \wedge r \wedge \neg o_{\mathrm{turn}}) \tag{15}$$

$$B_r \equiv \exists t_0.\big(t_0 = \hat{t} \wedge ((C_r \wedge D)\ \mathbf{W}\ E_r)\big) \tag{16}$$

$$C_r \equiv (\hat{t} - t_0)\%660 \leq 340 \Rightarrow \mathrm{on}_r) \tag{17}$$

$$D \equiv (\hat{t} - t_0)\%660 > 340 \Rightarrow \mathrm{off}_{\ell r}) \tag{18}$$

$$E_r \equiv \neg ig \vee \ell \vee (n \wedge \neg tf) \vee o_{\mathrm{turn}} \tag{19}$$

$$T_r \equiv \exists t_0.\big(t_0 = \hat{t} \wedge (T_1 \wedge T_2^r \wedge T_3 \wedge T_4\ \mathbf{W}\ \neg ig \vee \ell \vee o_{\mathrm{turn}})\big) \tag{20}$$

$$T_1 \equiv \hat{t} - t_0 < 440 \Rightarrow tf \tag{21}$$

$$T_2^r \equiv \hat{t} - t_0 = 440 \wedge \neg n \Rightarrow (\neg tf\ \mathbf{W}\ \neg r) \tag{22}$$

$$T_3 \equiv \hat{t} - t_0 \leq 440 \wedge n \Rightarrow \tag{23}$$
$$(tf\ \mathbf{U}\ (\hat{t} - t_0 = 1980 \vee \neg ig \vee o_{\mathrm{turn}} \vee \neg n)$$

$$T_4 \equiv \hat{t} - t_0 > 1980 \Rightarrow \neg tf \tag{24}$$

Requirement 4. Emergency flashing. Formula EMR specifies emergency flashing. The condition K enabling emergency flashing requires a state change

of the switch from 'not pressed' to 'pressed'. Alternatively, emergency flashing is re-activated after having been overridden. This is specified by the second conjunct of K. There are no dependencies on the ignition switch.

The emergency flashing behaviour is specified in formula L with its subformulas. As in turn indication flashing, we store the activation time in t_0. Since the duration of the on/off phases now depend on the ignition status, separate specifications are made: M_3, M_4 specify the on/off phases for the ignition-on case, and M_5, M_6 for the ignition-off case. A change of the ignition status does not affect the current on/off flashing period. This is taken into account by means for formulas M_1, M_2: the ignition status at the beginning of a period (i.e. when $(\hat{t}-t_0)\%660 = 0$) is recorded in auxiliary variable i and kept constant throughout the following period.

The emergency flashing rules cease to apply when the emergency switch is no longer pressed, or emergency flashing is overridden by turn indication. This is specified in formula N.

$$\mathrm{EMR} \equiv \mathbf{G}\big(K \Rightarrow \mathbf{X}L\big) \tag{25}$$
$$K \equiv (\neg e \vee o_e) \wedge \mathbf{X}(e \wedge \neg o_e) \tag{26}$$
$$L \equiv \exists t_0.(t_0 = \hat{t} \wedge (M_1 \wedge M_2 \wedge M_3 \wedge M_4 \wedge M_5 \wedge M_6 \ \mathbf{W} N)) \tag{27}$$
$$M_1 \equiv (\hat{t} - t_0)\%660 = 0 \wedge \neg ig \Rightarrow \big(\exists u.(u = \hat{t} \wedge (\neg i \ \mathbf{U} \ \hat{t} - t_0 = 660))\big) \tag{28}$$
$$M_2 \equiv (\hat{t} - t_0)\%660 = 0 \wedge ig \Rightarrow \big(\exists u.(u = \hat{t} \wedge (i \ \mathbf{U} \ \hat{t} - t_0 = 660))\big) \tag{29}$$
$$M_3 \equiv (\hat{t} - t_0)\%660 \leq 340 \wedge i \Rightarrow \mathrm{on}_{\ell r} \tag{30}$$
$$M_4 \equiv (\hat{t} - t_0)\%660 > 340 \wedge i \Rightarrow \mathrm{off}_{\ell r} \tag{31}$$
$$M_5 \equiv (\hat{t} - t_0)\%660 \leq 320 \wedge \neg i \Rightarrow \mathrm{on}_{\ell r} \tag{32}$$
$$M_6 \equiv (\hat{t} - t_0)\%660 > 320 \wedge \neg i \Rightarrow \mathrm{off}_{\ell r} \tag{33}$$
$$N \equiv \neg e \vee o_e \tag{34}$$

Requirement 5. Override conditions. The emergency flashing override condition becomes true or stays true according to the specifications EOVR_i, $i = 1, \ldots, 5$. In each of these situations, the emergency flashing switch stays pressed with the ignition being turned on, and

- the turn indiction lever is moved into a new non-neutral position, or
- the turn indiction lever is moved into the neutral position during the tip flashing period, or
- the turn indiction lever stays in a non-neutral position, and the ignition changes from off to on.

$$\mathrm{EOVR}_1 \equiv \mathbf{G}\big(e \wedge n \wedge \mathbf{X}(ig \wedge e \wedge \neg n) \Rightarrow \mathbf{X}o_e\big) \tag{35}$$
$$\mathrm{EOVR}_2 \equiv \mathbf{G}\big(e \wedge \ell \wedge \mathbf{X}(ig \wedge e \wedge r) \Rightarrow \mathbf{X}o_e\big) \tag{36}$$
$$\mathrm{EOVR}_3 \equiv \mathbf{G}\big(e \wedge r \wedge \mathbf{X}(ig \wedge e \wedge \ell) \Rightarrow \mathbf{X}o_e\big) \tag{37}$$
$$\mathrm{EOVR}_4 \equiv \mathbf{G}\big(e \wedge \neg n \wedge tf \wedge \mathbf{X}(ig \wedge e \wedge n \wedge tf) \Rightarrow \mathbf{X}o_e\big) \tag{38}$$
$$\mathrm{EOVR}_5 \equiv \mathbf{G}\big(\neg ig \wedge \neg n \wedge e \wedge \mathbf{X}(ig \wedge \neg n \wedge e) \Rightarrow \mathbf{X}o_e\big) \tag{39}$$

The emergency flashing override condition is switched off and stays off, as long as the emergency switch is not pressed or turn indication is inactive or has been overridden.

$$\text{EOVROFF} \equiv \mathbf{G}\big(\neg e \vee \neg ig \vee (n \wedge \neg tf) \vee o_{\text{turn}} \Rightarrow \neg o_e\big) \tag{40}$$

The turn indication override condition is activated as soon as the emergency switch is pressed while the ignition is already on or simultaneously activated[5], so that the turn indication lever is in a non-neutral position.

$$\text{TOVR} \equiv \mathbf{G}\big(\neg e \wedge \mathbf{X}(ig \wedge \neg n \wedge e) \Rightarrow \mathbf{X}o_{\text{turn}}\big) \tag{41}$$

The turn indication override condition remains off, as long as turn indication is inactive, the emergency switch is not pressed, or an emergency override condition is true.

$$\text{TOVROFF} \equiv \mathbf{G}\big(\neg ig \vee n \vee \neg e \vee o_e \Rightarrow \neg o_{\text{turn}}\big) \tag{42}$$

5 Requirements Tracing and Atomated Test Case Generation

In the previous section, it has been shown how the required behaviour of the target system can be specified using quantified first-order LTL *alone* – without referring to a model. This, however, is hardly ever used in industrial practice[6], due to the fact that developing these formulas is quite hard and requires more than just programming skills.

Therefore, we advocate to specify the required target system structure and its expected behaviour by means of a SysML model. Then the requirements, which now don't have to be formalised in temporal logic, can be traced back to structural and behavioural model elements. To this end, SysML provides the so-called *satisfy relationship* which takes a structural or behavioural model element as client (the source end of the directed relationship) and a requirement as a supplier (the target end of the relationship) [18, 16.3.2.7]. The interpretation of this relation is that the client element contributes to the realisation of the requirement. The requirements themselves are usually not formalised. The formalisation is encoded in the structural and behavioural parts of the model which have a well-defined semantics, and in the satisfy relationship: every model computation[7] covering a model element linked via satisfy-relationship to the requirement is a witness for the requirement.

[5] This override variant specifies that emergency flashing has priority over turn indication, when both are simultaneously activated.

[6] We collaborate with Verified Systems International, Airbus, and Siemens Mobility in the field of HW/SW integration testing.

[7] A computation is a sequence of state valuations for all inputs, outputs, internal variables, and state machine states that can be performed according to the behavioural model semantics. See also [19,22,20].

Just creating links between requirements and model elements may seem like an over-simplification at first glance. Since the model, however, encodes all requirements, only simple links are needed to "mark" which model elements have been created for each requirement. The following example, where symbolic test cases are created for a natural-language requirement, illustrates this insight. It is explained how the test case generator introduced in Section 2 can perform test case identification as well as the elaboration of symbolic test cases in an automated way, using static model analysis techniques. The approach described here is based on previous work published in [19,22,20].

Requirement 1.1 Off when voltage out of range. When the battery voltage is out of range, all lights are switched off immediately.

This requirement is a natural-language specialisation of the formal Requirement 1 specified in (1), but we will see in the subsequent paragraphs that this fact ist not relevant for elaborating test cases: no formal requirements specification is needed. When analysing the rear controller design (Fig. 5), we see that switching lamps off due to inadequate battery voltage is performed in parts pwrLeftLamp and pwrRightLamp. The switch-off is realised by the transitions labelled with oorTrans1 and oorTrans2 ("out-of-range transition 1,2) in state machine PowerSourceSM shown in Fig 8. Transition oorTrans1 applies to the situation where battery voltage is out of range at system initialisation, while transition oorTrans2 is taken when battery voltage fails after a normal operation period, which is represented by state BATTERY_VOLTAGE_OK. State BATTERY_VOLTAGE_NOT_OK applies to the voltage-out-of-range situation and, therefore, sets the lamp current to zero, independent of any switchIn commands the parts receive from parts lampCtrlSlaveL and lampCtrlSlaveR, respectively (see Fig. 5). As a consequence, transitions oorTrans1 and oorTrans2 are linked to the requirement by the satisfy-relationship, as shown in Fig. 8.[8]

Using this requirement as an example, the functionality of the test case generator (see Fig.1) is now explained. Observe that every step described here can be performed by means of static model analysis, performed on the XMI-model representation.

Step 1. Test Case Identification. One requirement – even if specified by just one "atomic" statement in natural language – usually needs more than one test case to perform thorough tests. The different test cases needed are identified by the test case generator by static analysis of the model. The identification begins by associating separate test case groups with each satisfy relationship pointing to the same supplier requirement. In our example, it is obvious that we need

[8] The graphical notation for the satisfy-relationship is one of several options provided by the SysML syntax. For large models implementing many requirements, a tabular notation is usually preferred, where requirements in one column are associated with a list of model elements in another column.

at least one test case exercising transition oorTrans1 and at least one exercising oorTrans2.

For transition oorTrans1, the guard condition shows that it only depends on the input BatVolIN. The block PowerSource which has this state machine as classifier behaviour, has yet another Boolean interface specified by port switchIn. To detect unwanted hidden dependencies in the SUT, two test cases would therefore be produced for oorTrans1, one with switchIn = `true` and one with switchIn = `false`.

For transition oorTrans2, the source state contains a sub-machine implementing a do-action. This sub-machine is shown in Fig. 9. It is evident that the correct effect of transition oorTrans2 has to be tested for each of the sub-machine states: battery voltage out-of-range should occur in one test case when the sub-machine is in state LAMP_CURRENT_OFF, and in another test case when the sub-machine is in state LAMP_CURRENT_ON.

Further static analysis for the oorTrans2-group shows that we have two concurrent parts, namely pwrLeftLamp and pwrRightLamp, exercising two instances of the state machine, and each instance is controlled by separate interfaces pwrLeftLamp.switchIn, pwrRightLamp.switchIn, written to by different suppliers (lampCtrlSlaveL, lampCtrlSlaveR). (Interfaces pwrLeftLamp.BatVolIN and pwrRightLamp.BatVolIN are not independent: they receive data from the same source RearController.BatVolIN.) The independent sources for switchIn suggest that test cases covering different pairs of concurrent states need to be exercised; these pairs are

- (lampCtrlSlaveL.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_OFF, lampCtrlSlaveR.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_OFF),
- (lampCtrlSlaveL.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_OFF, lampCtrlSlaveR.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_ON),
- (lampCtrlSlaveL.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_ON, lampCtrlSlaveR.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_OFF),
- (lampCtrlSlaveL.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_ON, lampCtrlSlaveR.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_ON).

Summarising, the test cases shown in Fig. 10 are needed, and all of them could be identified by means of static model analysis, evaluating the interfaces and data flows on block diagrams, state machine transitions and guard conditions.

At a first glance, test case identification step described here looks similar to the well-known *classification tree* technique applied with Statecharts [11]. There are, however, significat differences: first, the authors of [11] aim at synthesising state machines from classification trees. The tree has to be constructed in a manual way and augmented by additional constraints. Second, the need for requirements tracing is not considered in [11]. Our approach is requirements-driven and uses state machines as a fundamental means for specifying behaviour, so there is no intention to synthesise them from some other representation. We use trees like the one depicted in Fig. 10 only internally, for the purpose of test case identification performed automatically by the test case generator.

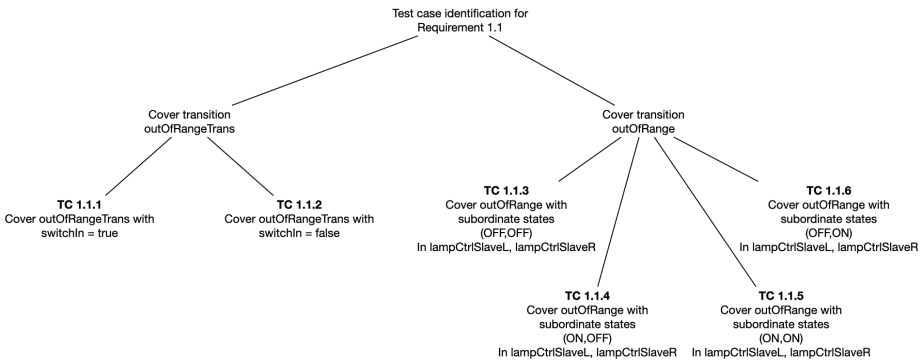

Fig. 10. Test case identification for Requirement 1.1.

Step 2. Test Case Formula Initialisation. For each identified test case, the test case generator initialises an unquantified first-order LTL formula, to be refined in the subsequent steps 3 and 4. We illustrate these steps using TC 1.1.4 from Fig. 10. For all test cases, the transition which is client to the supplied requirement needs to be covered. For TC 1.1.4, this is transition oorTrans2. The trigger is a change event[9] with condition ! isVoltageInRange() (see Fig 8). With the abbreviations introduced in Table 1, ! isVoltageInRange() is the same as $\neg v$. Therefore, our test case needs v to be true in one cycle and $\neg v$ in the following one, which is expressed by $v \wedge \mathbf{X}\neg v$.

For the transition to fire, the state machine needs to be in one of the subordinate states LAMP_CURRENT_ON or LAMP_CURRENT_OFF. For test case TC 1.1.4, however, this has already been refined in Step 1. Part lampCtrlSlaveL needs to be in the ON-state, and part lampCtrlSlaveR in the OFF-state, when the change event occurs. Further static analysis shows that the ON-state of lampCtrlSlaveL is active if and only if CurrentLout = 2. Furthermore, the OFF-state of lampCtrlSlaveR is active if and only if CurrentRout = 0. Using further abbreviations from Table 1, this can be written as $c_\ell \wedge \neg c_r$.

Summarising, we need a computation which finally fulfils $v \wedge \mathbf{X}\neg v$, and the initial condition v is strengthened by $c_\ell \wedge \neg c_r$. This results in a preliminary test case formula

$$\mathbf{F}\big((v \wedge c_\ell \wedge \neg c_r) \wedge \mathbf{X}\neg v\big) \tag{43}$$

Formula (43) has many witnesses that turn out to be inadequate test cases, since

1. input changes appear too quickly, so that they are difficult to observe in a HW/SW integration test, and/or
2. input changes trigger different behaviours concurrently, so that their interference makes it hard or even impossible to determine whether the functional aspect to be tested has really been exercised.

[9] A change event occurs when the condition changes from `false` to `true` [17, 13.3.3.3].

As an example of the first problem, consider the test input sequence shown in Table 2. At time $\hat{t} = 0$, the turn indication lever ist put into position LEFT, while the emergency switch is not pressed and the ignition is on. As a consequence, the LHS-turn indication lights are switched on. One millisecond later, the voltage is out of range, and the lights are switched off. This computation is certainly a witness for Formula (43), but, due to hardware interface latency, it will be very difficult to observe that the LHS-lights really received a non-zero current before the battery voltage dropped out of range.

Table 2. Test sequence for Formula (43) which is not adequately observable.

$\hat{t}$	**LeverPositionIN**	**EmerSwitchPressedIN**	**IgnitionOnIN**	**BatVolIN**
0	LEFT	`false`	`true`	12
1	LEFT	`false`	`true`	8

This example shows that states before and after a relevant test step need to remain stable for a sufficient time interval, so that the resulting outputs can be adequately observed. The necessary changes of Formula (43) to achieve this stability are described in Step 3.

As an example of the second problem, consider the test input sequence shown in Table 3. As in the previous example, it is easy to see that the computation stimulated by this short timed input sequence is a witness of Formula (43).

Table 3. Test sequence for Formula (43) where different functionalities interfere.

$\hat{t}$	**LeverPositionIN**	**EmerSwitchPressedIN**	**IgnitionOnIN**	**BatVolIN**
0	LEFT	`false`	`true`	12
300	LEFT	`false`	`false`	8

Also, the activation of the LHS-turn indication lights stays stable for 300ms which suffices for adequate observability. However, at $\hat{t} = 300$, two inputs change simultaneously: the voltage drops out of range, and the ignition is switched off. As a consequence, both requirements "ignition off switches turn indication lights off" and "voltage out of range switches all lights off" are applicable in this step. Therefore, it will remain unclear in a black-box test, whether the lights-off reaction was due to the change in the battery voltage or due to the state in the ignition status. This problem is removed by a further static analysis performed by the test case generator, described in Step 4 below.

Step 3. Test Case Refinement – Stability. Stability is achieved by forcing the test data generator (in this paper, the fuzzer) to set inputs and delays be-

tween input changes in such a way, that the model simulation "lingers" in certain states for an appropriate amount in time. This can be specified in a practical way using the auxiliary *timer variable* which is associated with every state and stores the point in time when the state has been entered. This variable is denoted by

```
1   <state-name>.t
```

and is used to determine the point in time when a relative time event after x (x a duration in milliseconds) should be triggered: using the actual model execution time $\hat{t}$, expression

$$\hat{t} - \text{<state-name>.t}$$

specifies how long the state machine resides in the state since it had last been entered. Transitions triggered by time events are shown in Fig. 7.

The test case generator implements two rules to ensure sufficient stability:

1. The simple state[10] the model resides in when the critical transition is fired must have been entered for an observable amount of time[11] before the transition is fired.
2. The simple target state of the critical transition must be kept for an observable amount of time.

For our critical transition oorTrans2, these rules are applied as follows. Part pwrLeftLamp needs to linger in state LAMP_CURRENT_ON for at least 100ms before the critical transition is triggered. This is expressed by[12]

$$\begin{aligned}\phi_0 \equiv\ & \text{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_OK.LAMP\_CURRENT\_ON} \wedge \\ & (\hat{t} - \text{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_OK.LAMP\_CURRENT\_ON.t}) >= 100\end{aligned}$$

After the target state has been reached by the critical transition, we have to stay there as well for an observable amount of time. This is expressed by formula

$$\begin{aligned}\phi_1 \equiv\ & \text{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK} \wedge \\ & (\text{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK}\ \mathbf{U} \\ & (\hat{t} - \text{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK.t}) >= 100)\end{aligned}$$

With these refining propositions, Formula (43) is strengthened to

$$\mathbf{F}\big((v \wedge c_\ell \wedge \neg c_r \wedge \phi_0) \wedge \mathbf{X}(\neg v \wedge \phi_1)\big) \tag{44}$$

[10] A simple state is one without internal vertices (i.e. sub-ordinate states) or transitions [17, 14.2.3.4.1].

[11] Depending on the hardware-in-the-loop testing environment and on the type of application, "observable" means some duration between $500\mu s$ and a multiple of $100ms$. In any case, the amount needs to be smaller than the shortest time event associated with any transition emanating from the state under consideration.

[12] State machines are used like Boolean expressions: state symbol pwrLeftLamp.PowerSourceSM.BATTERY_VOLTAGE_OK.LAMP_CURRENT_ON evaluates to `true` if and only if the model simulation resides in this state.

Step 4. Test Case Refinement – Non-Interference. For test case TC-1.1.4, all possible interference from other functional aspects are reflected by the only additional input switchIn changing its value. Avoiding interference with the battery voltage out-of-range reaction is ensured by avoiding a status change of switchIn for both parts pwrLeftLamp and pwrRightLamp, until the desired transition oorTrans2 has fired, and a bit longer to ensure stability. This can be specified by sub-formulas

$$
\begin{aligned}
\phi_2 \equiv\ & \mathsf{pwrLeftLamp.switchIn} \wedge \\
& (\mathsf{pwrLeftLamp.switchIn}\ \mathbf{U} \\
& (\mathsf{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK} \wedge \\
& (\hat{t} - \mathsf{pwrLeftLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK.t} \geq 100))))\\
\phi_3 \equiv\ & \neg\mathsf{pwrRightLamp.switchIn} \wedge \\
& (\neg\mathsf{pwrRightLamp.switchIn}\ \mathbf{U} \\
& (\mathsf{pwrRightLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK} \wedge \\
& (\hat{t} - \mathsf{pwrRightLamp.PowerSourceSM.BATTERY\_VOLTAGE\_NOT\_OK.t} \geq 100)))
\end{aligned}
$$

Summarising, the resulting formula for test case TC-1.1.4 refines Formula (44) to

$$\mathbf{F}\big((v \wedge c_\ell \wedge \neg c_r \wedge \phi_0) \wedge \mathbf{X}(\neg v \wedge \phi_1 \wedge \phi_2 \wedge \phi_3)\big) \tag{45}$$

Step 5. Boundary value tests. With the LTL formulas generated in steps 1 — 4 at hand, symbolic boundary value tests can be constructed by replacing comparison operators and associated constants in a suitable way.

6 Evaluation

The performance of the fuzzer for generating concrete test data from symbolic test cases specified in LTL has been evaluated using several test cases created according to the test case generation concept presented in Section 5. For each test case, we show the generated data which consists of input vectors to the SUT, together with time stamps specifying when the next input vector should be written.

The performance is summarised in Table 4. For each symbolic test case, 20 test data generations with the fuzzer were tried out, since the fuzzer relies on random input data creation, so that generations differ in time. A test data generation run is aborted as soon as the observer indicates that the formula can no longer be solved with data created so far. The table shows the number of succesful generation runs within these 20 tries (note that each try was successful), the average, minimal, and maximal time needed to find this witness.

TC-TIPFL – Tip flashing. The following formula specifies a symbolic test case related to the tip flashing function. The formula states that we wish to start with the turn indication lever in neutral position (n). Throughout the test execution, battery voltage should stay in range, the ignition should stay on, and no interference by emergency flashing should occur. The test generation should end after approx. 10 seconds. This is expressed by $\mathbf{X}((v \wedge ig \wedge \neg e)\ \mathbf{U}\ \hat{t} \geq$

Table 4. Performance summary. #Tests denotes the number of successful generation runs within the 20 runs used to find the witness trace. Avg, Min, Max denote average, minimal, and maximal generation time in seconds.

Test Case	#Tests	Avg	Min	Max
TC-TIPFL	20	3.26	1.42	6.86
TC-LRFL	20	127.88	9.7	394.06
TC-EMRFL	20	25.88	1.02	176.79
TC-IGOFF	20	49.97	14.44	196.58
TC-VOOR	20	7.64	3.48	26.55

10000). The generated trace should lead to state UNSTABLE (see state machine in Fig. 7), such that the model simulation stays there for at least 1970ms ($\mathbf{F}\big(\text{UNSTABLE} \wedge \hat{t} - \text{UNSTABLE}.t > 1970\big)$). Analysis of the state machine and the one specified in Fig. 6 shows that this exactly reflects a tip flashing situation. This slightly indirect specification of the test objective is due to the fact that the transition from UNSTABLE triggered by the after(1980) time event is linked to the tip flashing requirement by means of the satisfy relationship. Moreover, it is simpler to specify than explicitly stating when the turn indication lever should be set to a specific position.

The generated test data with associated time stamps (also determined by the fuzzer) is shown in Table 5. The turn indication lever is put into position LEFT and then back to NEUTRAL after 260ms. This triggers the tip flashing reaction, that three flashing cycles are performed. Recall that the expected results need not to be elaborated by the test data generator, since we are using a test oracle version of the model simulation, running back-to-back with the SUT during test executions.

$$\begin{aligned}\text{TC-TIPFL} \equiv n \wedge \mathbf{X}\big((v \wedge ig \wedge \neg e)\ \mathbf{U}\ \hat{t} \geq 10000\big) \wedge \\ \mathbf{F}\big(\text{UNSTABLE} \wedge \hat{t} - \text{UNSTABLE}.t > 1970\big)\end{aligned}$$

Table 5. Generated test sequence for symbolic test case TC-TIPFL.

$\hat{t}$	LeverPositionIN	EmerSwitchPressedIN	IgnitionOnIN	BatVolIN
0	NEUTRAL	`false`	`true`	5
10	LEFT	`false`	`true`	11
270	NEUTRAL	`false`	`true`	11

TC-LRFL – LHS/RHS flashing with emergency override. This symbolic test case specifies that emergency flashing should be active, but overridden by left

or right turn indication after 5000ms. The generated test data with associated time stamps is shown in Table 6.

$$\begin{aligned}\text{TC-LRFL} \equiv \mathbf{X}((v \wedge ig \wedge e)\ \mathbf{U}\ \hat{t} \geq 10000) \wedge \\ \mathbf{X}\big(\text{EMERGENCY\_FLASHING}\ \mathbf{U}\ (\hat{t} = 5000 \wedge \\ (\text{LRFlashing} \wedge (\text{LRFlashing}\ \mathbf{U}\ \hat{t} \geq 10000)))\big)\end{aligned}$$

Table 6. Generated test sequence for symbolic test case TC-LRFL.

$\hat{t}$	**LeverPositionIN**	**EmerSwitchPressedIN**	**IgnitionOnIN**	**BatVolIN**
0	NEUTRAL	`true`	`true`	10
2560	NEUTRAL	`true`	`true`	10
4980	NEUTRAL	`true`	`true`	10
4990	RIGHT	`true`	`true`	10
7360	RIGHT	`true`	`true`	12
9730	RIGHT	`true`	`true`	12

TC-EMRFL – Emergency flashing after turn indication. This test case specifies to first exercise LHS or RHS turn indication and then emergency flashing. The generated test data with associated time stamps is shown in Table 7.

$$\begin{aligned}\text{TC-EMRFL} \equiv \mathbf{X}(v\ \mathbf{U}\ \hat{t} \geq 10000) \wedge \\ \mathbf{X}\big(\text{LRFlashing}\ \mathbf{U}\ (\hat{t} = 5000 \wedge \\ (\text{EMERGENCY\_FLASHING} \wedge \\ (\text{EMERGENCY\_FLASHING}\ \mathbf{U}\ \hat{t} \geq 10000)))\big)\end{aligned}$$

Table 7. Generated test sequence for symbolic test case TC-EMRFL.

$\hat{t}$	**LeverPositionIN**	**EmerSwitchPressedIN**	**IgnitionOnIN**	**BatVolIN**
0	LEFT	`false`	`true`	15
2400	LEFT	`false`	`true`	15
4950	LEFT	`false`	`true`	15
4990	NEUTRAL	`true`	`true`	14
7550	NEUTRAL	`true`	`true`	10

TC-IGOFF – Ignition off while emergency flashing. This test case specifies to turn off the ignition during emergency flashing. The generated test data with associated time stamps is shown in Table 8.

$$
\begin{aligned}
\text{TC-IGOFF} \equiv\ & \mathbf{X}(v\ \mathbf{U}\ \hat{t} \geq 10000) \wedge \\
& \mathbf{X}\big(\text{EMERGENCY\_FLASHING}\ \mathbf{U} \\
& \quad \hat{t} - \text{EMERGENCY\_FLASHING}.t \geq 10000\big) \wedge \\
& \mathbf{X}\big(ig\ \mathbf{U}\ (\hat{t} \in [5400, 5530] \wedge \\
& \quad \text{LAMPS\_ON} \wedge \hat{t} - \text{LAMPS\_ON}.t \in [100, 250] \wedge \neg ig \wedge \\
& \quad (\neg ig\ \mathbf{U}\ \hat{t} \geq 10000))
\end{aligned}
$$

Table 8. Generated test sequence for symbolic test case TC-IGOFF.

$\hat{t}$	**LeverPositionIN**	**EmerSwitchPressedIN**	**IgnitionOnIN**	**BatVolIN**
0	NEUTRAL	`true`	`true`	13
2560	NEUTRAL	`true`	`true`	10
5120	NEUTRAL	`true`	`true`	10
5510	NEUTRAL	`true`	`false`	10
8070	NEUTRAL	`true`	`false`	10

TC-VOOR – Battery voltage out of range while turn indication flashing. This test case is intended to check the SUT reaction when batter voltage drops out of range while LHS turn indication performs an ON-phase. The generated test data with associated time stamps is shown in Table 9.

$$
\begin{aligned}
\text{TC-VOOR} \equiv\ & \mathbf{X}\big(ig\ \mathbf{U}\ \hat{t} \geq 10000\big) \wedge \\
& \mathbf{X}\big(v\ \mathbf{U}\ (\text{CurrentLout} > 0 \wedge \text{CurrentRout} = 0 \wedge \\
& \quad \hat{t} - \text{pwrLeftLamp.BATERRY\_VOLTAGE\_OK.LAMP\_CURRENT\_ON}.t = 150) \wedge \\
& \quad \mathbf{X}(\neg v \wedge (\neg v\ \mathbf{U}\ \hat{t} \geq 10000))\big)
\end{aligned}
$$

7 Conclusion

We have presented an approach to property-oriented model-based testing which is suitable for requirements-driven testing as needed for certifiable systems in the avionic, automotive, and railway domains. The utilisation of models represented in SysML allows for simple tracing mechanisms linking structural or behavioural

Table 9. Generated test sequence for symbolic test case TC-VOOR.

$\hat{t}$	LeverPositionIN	EmerSwitchPressedIN	IgnitionOnIN	BatVolIN
0	LEFT	`false`	`true`	11
2130	NEUTRAL	`false`	`true`	3
2750	NEUTRAL	`false`	`true`	5
5260	NEUTRAL	`false`	`true`	5
7770	NEUTRAL	`false`	`true`	5

model elements to requirements. These links can be exploited to generate symbolic test case formulas in LTL in a fully automated way. As a consequence, the whole process of property specification in temporal logic can be hidden from users, who only have to provide the model and its traceability information.

For generating concrete test cases from the symbolic ones represented in LTL, we use fuzzing as a coverage-maximising test generation engine. The fuzzer operates on a C^{++} model simulation extended by observers indicating when a witness for a symbolic test case has been found. Evaluation data shows that this is a very effective light-weight alternative to test data generation by SMT solvers.

References

1. LibFuzzer - a library for coverage-guided fuzz testing, available at `https://llvm.org/docs/LibFuzzer.html`
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The hanoi omega-automata format. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_31, `https://doi.org/10.1007/978-3-319-21690-4_31`
3. Babiak, T., Kretínský, M., Rehák, V., Strejcek, J.: LTL to büchi automata translation: Fast and more deterministic. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7214, pp. 95–109. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_8, `https://doi.org/10.1007/978-3-642-28756-5_8`
4. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
5. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1032–1043. ACM (2016). https://doi.org/10.1145/2976749.2978428, `https://doi.org/10.1145/2976749.2978428`

6. CENELEC: EN 50128:2011 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems (2011)
7. Dorofeeva, R., El-Fakih, K., Yevtushenko, N.: An improved conformance testing method. In: Wang, F. (ed.) Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3731, pp. 204–218. Springer (2005). https://doi.org/10.1007/11562436_16, https://doi.org/10.1007/11562436_16
8. Hierons, R.M.: Testing from a nondeterministic finite state machine using adaptive state counting. IEEE Trans. Computers **53**(10), 1330–1342 (2004). https://doi.org/10.1109/TC.2004.85, http://doi.ieeecomputersociety.org/10.1109/TC.2004.85
9. ISO/DIS 26262-4: Road vehicles – functional safety – part 4: Product development: system level. Tech. rep., International Organization for Standardization (2009)
10. Kim, Y., Lee, D., Baek, J., Kim, M.: Concolic testing for high test coverage and reduced human effort in automotive industry. In: Sharp, H., Whalen, M. (eds.) Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019. pp. 151–160. IEEE / ACM (2019). https://doi.org/10.1109/ICSE-SEIP.2019.00024, https://doi.org/10.1109/ICSE-SEIP.2019.00024
11. Kruse, P.M., Wegener, J.: Test sequence generation from classification trees. In: Antoniol, G., Bertolino, A., Labiche, Y. (eds.) Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012. pp. 539–548. IEEE Computer Society (2012). https://doi.org/10.1109/ICST.2012.139, https://doi.org/10.1109/ICST.2012.139
12. Le, H.M.: Llvm-based hybrid fuzzing with libkluzzer (competition contribution). In: Wehrheim, H., Cabot, J. (eds.) Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12076, pp. 535–539. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_29, https://doi.org/10.1007/978-3-030-45234-6_29
13. Machado, P.D.L., Silva, D.A., Mota, A.C.: Towards Property Oriented Testing. Electronic Notes in Theoretical Computer Science **184**(Supplement C), 3–19 (Jul 2007). https://doi.org/10.1016/j.entcs.2007.06.001, http://www.sciencedirect.com/science/article/pii/S157106610700432X
14. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering (2019). https://doi.org/10.1109/TSE.2019.2946563
15. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279, https://doi.org/10.1145/96267.96279
16. Mizuno, T., Higashino, T., Shiratori, N. (eds.): Protocol Test Systems, 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems. IFIP Advances in Information and Communication Technology, Springer Science+Business Media Dordrecht (1995)
17. Object Management Group: OMG Unified Modeling Language (OMG UML), version 2.5.1 (2017)

18. Object Management Group: OMG Systems Modeling Language (OMG SysML), Version 1.6. Tech. rep., Object Management Group (2019), http://www.omg.org/spec/SysML/1.4
19. Peleska, J.: Industrial-strength model-based testing - state of the art and current challenges. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings Eighth Workshop on Model-Based Testing, Rome, Italy, 17th March 2013. Electronic Proceedings in Theoretical Computer Science, vol. 111, pp. 3–28. Open Publishing Association (2013). https://doi.org/10.4204/EPTCS.111.1
20. Peleska, J.: Model-based avionic systems testing for the airbus family. In: 23rd IEEE European Test Symposium, ETS 2018, Bremen, Germany, May 28 - June 1, 2018. pp. 1–10. IEEE (2018). https://doi.org/10.1109/ETS.2018.8400703, https://doi.org/10.1109/ETS.2018.8400703
21. Peleska, J.: Specification of Embedded Systems, Session 7 – Automated Model-based Code Generation. University of Bremen (August 2020), http://www.informatik.uni-bremen.de/agbs/jp/papers/ses/Session-7-Automated-Code-Generation.pdf, lecture notes
22. Peleska, J., Brauer, J., Huang, W.: Model-based testing for avionic systems proven benefits and further challenges. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 11247, pp. 82–103. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6_11, https://doi.org/10.1007/978-3-030-03427-6_11
23. Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., Zahlten, C.: A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In: Wolff, B., Zaidi, F. (eds.) Testing Software and Systems. Proceedings of the 23rd IFIP WG 6.1 International Conference, ICTSS 2011. LNCS, vol. 7019, pp. 146–161. IFIP WG 6.1, Springer, Heidelberg Dordrecht London New York (November 2011)
24. Peleska, J., ling Huang, W., Cavalcanti, A.: Finite complete suites for csp refinement testing. Science of Computer Programming **179**, 1 – 23 (2019). https://doi.org/https://doi.org/10.1016/j.scico.2019.04.004, http://www.sciencedirect.com/science/article/pii/S0167642319300620
25. van de Pol, J., Meijer, J.: Synchronous or Alternating? In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday, pp. 417–430. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-22348-9_24
26. Ruhstaller, M., Chang, O.: A new chapter for OSS-Fuzz, available at https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html
27. Simão, A., Petrenko, A., Yevtushenko, N.: On reducing test length for FSMs with extra states. Software Testing, Verification and Reliability **22**(6), 435–454 (Sep 2012). https://doi.org/10.1002/stvr.452, https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.452
28. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects of Computing **6**(5), 495–511 (Sep 1994). https://doi.org/10.1007/BF01211865, http://link.springer.com/article/10.1007/BF01211865
29. Theobald, M., Tatibouet, J.: Using fuml combined with a DSML: an implementation using papyrus uml/sysml modeler. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of the 7th International Conference on

Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019. pp. 248–255. SciTePress (2019). https://doi.org/10.5220/0007310702500257, https://doi.org/10.5220/0007310702500257
30. WG-71, R.S.E.: Software Considerations in Airborne Systems and Equipment Certification. Tech. Rep. RTCA/DO-178C, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036 (December 2011)
31. Zalewski, M.: American fuzzy lop (AFL) white paper, available at http://lcamtuf.coredump.cx/afl/technical_details.txt

A Semantics of LTL

The code generator used here for creating embedded systems code from SysML models operates according to the Mealy machine paradigm. The main program operates single-threaded and performs a non-terminating main loop, where first new inputs are received (if any), then the state machines are scheduled sequentially until a run-to-completion (RTC) has been finished. After that, the outputs are written. Conceptually, the state machines operate in zero-time until the RTC is completed. Time passes while inputs remain stable, until the next input change occurs or the next timer elapses.

To specify requirements and test cases for systems of this type, unquantified first-order Linear Temporal Logic (LTL) formulas are suitable. Their syntax is specified as follows.

- Every unquantified first-order formula f over symbols from $V = V_I \cup V_O \cup V_M$ and constants from D as specified below is an unquantified first-order LTL formula.
- If ϕ, ψ are unquantified first-order LTL formulae, then $\neg\phi$, $\phi \wedge \psi$, $\mathbf{X}\phi$ (*Next*), $\phi \mathbf{U} \psi$ (*Until*) are also unquantified first-order LTL formulae.

Operators $\mathbf{X}$, $\mathbf{U}$ are called *path operators*. Further Boolean and path operators are specified as syntactic abbreviations by $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$, $\mathbf{F}\phi \equiv (\mathtt{true}\ \mathbf{U}\ \phi)$ (*Finally*), $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$ (*Globally*), $\phi \mathbf{W} \psi \equiv (\mathbf{G}\phi) \vee (\phi \mathbf{U} \psi)$ (*Weak Until*) .

The Mealy execution paradigm suggests to use the synchronous interpretation of LTL formulas described in [25]. In this context, the models of LTL formulas are infinite traces

$$\pi = (x_0/y_0).(x_1/y_1) \cdots \in (I \times O)^\omega,$$

where I denotes the set of all valuation functions $x : V_I \longrightarrow D$ mapping input symbols of the (SysML) model interface to their current value in the union D over all variable domains. Set O contains all valuation functions $y : V_O \cup V_M \longrightarrow D$ over output symbols from V_O and internal model variable symbols from V_M. Note that though this paper is about black-box testing, the *reference model* specifying the expected SUT behaviour is interpreted in white-box fashion, so that LTL formulas may refer to inputs, outputs, *and* to internal model variables. Input valuations x_i are interpreted at the beginning of a processing cycle, after

the actual state of the input interface has been updated. The valuations y_i are interpreted at the end of the cycle, after all state machines have performed their runs to completion, when the outputs become visible to the environment.

As usual for a temporal logic, the model traces are infinite. In the context of testing, however, we are only interested in requirements expressed by *safety properties*, because their violation can be detected on a finite trace prefix [28] (we do not consider tests of infinite length). Furthermore, safety properties φ can be regarded as limit sets (so-called *closures*) of finite traces that do not violate φ [4, 3.3.2, Lemma 3.27].

For Mealy machines, input alphabet I and output alphabet O are expected to be finite. Indeed, though the sets of valuation functions used here may be very large, they are really finite, if we consider concrete data types like `int` or `float` represented in an embedded control systems. Even the model execution time $\hat{t}$ which is considered as an element of V_I has a finite domain, because the internal representation (usually by 64-bit integer variables) only allows for a finite range.

A first-order expression f is evaluated on a given valuation pair x/y by replacing every symbol $v \in V_I$ occurring in f by $x(v)$, and every symbol $z \in V_O \cup V_M$ by $y(z)$. This results in Boolean expression e containing arithmetic expressions and comparison operators and Boolean sub-expressions, where all variable symbols have been replaced by constant values. We write $x/y \models f$ if and only if e evaluates to `true`.

The semantics of LTL formulas is then specified in Table 10. Given a trace π and an index $i \geq 0$, expression π^i denotes the path suffix of π starting with the ith element $\pi(i)$. In Table 10, f denotes an unquantified first-order formula and ϕ, ψ arbitrary LTL formulas.

Table 10. Semantics of LTL formulas.

$\pi^i \models$ `true` for all $i \geq 0$
$\pi^i \not\models$ `false` for all $i \geq 0$
$\pi^i \models f$ iff $\pi(i) \models f$ as specified above for first-order formulas f
$\pi^i \models \neg\phi$ iff $\pi^i \not\models \phi$
$\pi^i \models \phi \wedge \psi$ iff $\pi^i \models \phi$ and $\pi^i \models \psi$
$\pi^i \models \mathbf{X}\phi$ iff $\pi^{i+1} \models \phi$
$\pi^i \models \phi \mathbf{U} \psi$ iff there exists $j \geq 0$ such that $\pi^{i+j} \models \psi$ and $\pi^{i+k} \models \phi$ for all $0 \leq k < j$