A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain

Jan Peleska[1], Artur Honisch[3], Florian Lapschies[1], Helge Löding[2], Hermann Schmid[3], Peer Smuda[3], Elena Vorobev[1], and Cornelia Zahlten[2]

[1] Department of Mathematics and Computer Science
University of Bremen, Germany
{jp,florian,elenav}@informatik.uni-bremen.de
[2] Verified Systems International GmbH, Bremen, Germany
{hloeding,cmz}@verified.de
[3] Daimler AG, Stuttgart, Germany
{artur.honisch,hermann.s.schmid,peer.smuda}@daimler.com

Abstract. In this paper we present a model for automotive system tests of functionality related to turn indicator lights. The model covers the complete functionality available in Mercedes Benz vehicles, comprising turn indication, varieties of emergency flashing, crash flashing, theft flashing and open/close flashing, as well as configuration-dependent variants. It is represented in UML2 and associated with a synchronous real-time systems semantics conforming to Harel's original Statecharts interpretation. We describe the underlying methodological concepts of the tool used for automated model-based test generation, which was developed by Verified Systems International GmbH in cooperation with Daimler and the University of Bremen. A test suite is described as initial reference for future competing solutions. The model is made available in several file formats, so that it can be loaded into existing CASE tools or test generators. It has been originally developed and applied by Daimler for automatically deriving test cases, concrete test data and test procedures executing these test cases in Daimler's hardware-in-the-loop system testing environment. In 2011 Daimler decided to allow publication of this model with the objective to serve as a "real-world" benchmark supporting research of model based testing.

1 Introduction

Model-based testing. Automated *model-based testing (MBT)* has received much attention in recent years, both in academia and in industry. This interest has been stimulated by the success of model-driven development in general, by the improved understanding of testing and formal verification as complementary activities, and by the availability of efficient tool support. Indeed, when compared to conventional testing approaches, MBT has proven to increase both quality and efficiency of test campaigns; we name [15] as one example where quantitative evaluation results have been given. In this paper the term model-based testing is used in the following, most comprehensive, sense: the behavior of the *system under test (SUT)* is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT. A *symbolic test case generator* analyzes the model and specifies *symbolic test cases* as logical formulae identifying model computations suitable for a certain *test purpose* (also called *test objective*). Symbolic test cases may be represented as LTL formulas of the form $\mathbf{F}\phi$, expressing that finally the test execution should produce a computation fragment where the test purpose specified by ϕ is fulfilled.

Constrained by the transition relations of SUT and environment model, a *solver* computes concrete model computations which are *witnesses* of symbolic test cases $\mathbf{F}\phi$. More formally, solvers elaborate solutions of so-called *bounded model checking instances*

$$tc(c, G) \equiv_{\text{def}} \bigwedge_{i=0}^{c-1} \Phi(\sigma_i, \sigma_{i+1}) \wedge G(\sigma_0, \ldots, \sigma_c) \tag{1}$$

In this formula σ_0 represents the current model state and Φ the transition relation associated with the given model, so any solution of (1) is a valid model computation fragment of length c. Intuitively speaking, $tc(c, G)$ tries to solve LTL formula $\mathbf{F}\phi$ within c computation steps, starting in model pre-state σ_0, so that each step is a valid model transition, and test purpose ϕ is encoded in $G(\sigma_0, \ldots, \sigma_c)$.

The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behavior observed during the test execution is compared against the *expected* SUT behavior specified in the original model. Both stimulation sequences and *test oracles*, i. e., checkers of SUT behavior, are automatically transformed into *test procedures* executing the concrete test cases in a software-in-the-loop or hardware-in-the-loop configuration.

Observe that this notion of MBT differs from “weaker” ones where MBT is just associated with some technique of graphical test case descriptions. According to the MBT paradigm described here, the focus of test engineers is shifted from test data elaboration and test procedure programming to modeling. The effort invested into specifying the SUT model results in a return of investment, because test procedures are generated automatically and debugging deviations of observed against expected behavior is considerably facilitated because the observed test executions can be “replayed” against the model.

Objectives and Main Contributions. The main objective of this article is to present a “real-world” example of a model used in the automotive industry for system test purposes. The authors have experienced the “validation powers” of such models with respect to realistic assessment of efforts in model development, and with respect to the tool capabilities required to construct concrete model computations – i. e., test data – for given symbolic test cases. We hope to stimulate a competition of alternative methods and techniques which can be applied to the same benchmark model in order to enable objective comparison of different approaches. For starting such a competition we also give an overview of the methods and algorithms applied in our tool and present performance values, as well as test generation results to be compared with the results obtained using other methods and tools.

To our best knowledge, comparable models of similar size, describing concurrent real-time behavior of automotive applications and directly derived from industrial applications are currently not available to the public, at least not for application in the MBT domain. As a consequence, no systematic approach to benchmark definitions has been made so far. We therefore suggest a simple classification schema for such benchmarks, together with a structuring approach for the test suites to be generated. While this article can only give an overview of the model, detailed information are publicly available on the website [19] (`www.mbt-benchmarks.org`).

Overview. In Section 2 we present an introductory overview over the benchmark model. In Section 3 the methods applied in our test generator are sketched, with the objective to stimulate discussions about the suitability of competing methods. The representation of symbolic test cases as *constraint solving problems (CSP)* is described, and we sketch how these CSPs are solved by the *test generation engine* in order to obtain concrete test stimulations to be passed from the test environment to the SUT.

In Section 4 we propose a classification of benchmarks which are significant for assessing the effectiveness and performance of model-based testing tools. This classification induces a structure for reference test suites. In Section 5 a test generation example is presented.

Since the complete model description and the detailed explanation of algorithms used for test case generation and CSP solving is clearly beyond the page restriction of this submission, interested readers are referred to [19], where the material is presented in more comprehensive form, and the model can be downloaded in XMI format and as a model file for the EnterpriseArchitect CASE tool [25] which was used to create the model and its different export formats. Additionally an archive may be downloaded whose files allow to browse through the model in HTML format. The symbolic test cases used in the performance results are also available, so that the sequence of CSP solutions created by our test case generator can be repeated by other tools. The current benchmark evaluation results, including test generations performed with our tool are also published there.

Related Work. Benchmarking has been addressed in several testing domains. For “classical” software testing the so-called *Siemens benchmarks* [11] provide a collection of C programs with associated mutations rated as representative for typical programming bugs. A more comprehensive discussion and review of available software testing benchmarks is given in [16]. In [17] an initiative for event-driven software testing benchmarks has been launched.

In the field of model-based embedded systems testing only very few benchmarks are currently available, and none of them appear to describe comprehensive industrial control applications. In [12] a Matlab/Simulink model for a flight control system has been published. According to our classification proposed in Section 4 it addresses the benchmark category *test strength benchmarks*: A set of test cases is published which have been generated using random generation techniques inspired by Taguchi methods. To analyze the strength of test cases, several mutants of the model have been provided which may either be executed in Simulink simulation mode or as C programs generated from the mutant models. Since random test generation techniques on the input interface to the SUT are used, the model coverage achieved is only analyzed after test suite execution. As a consequence, no *test generation benchmarks* suggested in Section 4 are discussed. All existing benchmarks we are aware of may be classified as test strength benchmarks. Our proposition of test generation benchmarks seems to be a novel concept.

While our test generation approach relies on constraint solvers to find test-input-data, *search-based testing* techniques use randomized methods guided by optimization goals. In [2] the use of random testing, adaptive random testing and genetic algorithms for use in model-based black-box testing of real-time systems is investigated. To this end, the test-environment is modeled in UML/MARTE while the design of the SUT is not modeled at all, since all test data are derived from the possible environment behavior. An environment simulator is derived from the model that interacts with the SUT and provides the inputs selected by one of the strategies. The environment model also serves as a test oracle that reports errors as soon as unexpected reactions from the SUT are observed. This and similar approaches are easier to implement than the methods described in this paper, because there is no need to encode the transition relation of the model and to provide a constraint solver, since concrete test data is found by randomized model simulations. We expect, however, that the methods described in [2] do not scale up to handle systems of the size presented here, where the concurrent nature of the SUT requires to consider the interaction between several components in real-time (the model would be too large to construct a single large product automaton from the many smaller ones describing the component behavior). To the best knowledge of the authors there is no work on using search based testing on synchronous parallel real-time systems in order to achieve a high degree of SUT coverage, let alone to find test input data to symbolic test-cases.

The solutions presented here have been implemented in the RT-Tester test automation tool which provides an alternative to TRON [18, 8] which supports timed automata test models and is also fit for industrial-strength application. TRON is complementary to RT-Tester, because it supports an interleaving semantics and focuses on event-based systems, while RT-Tester supports a synchronous semantics with shared variable interfaces. RT-Tester also competes with the Conformiq Tool Suite [6], but focuses stronger on embedded systems testing with hard real-time constraints.

2 Model Overview

General. Our MBT benchmark model specifies the *turn indicator functions* available in Mercedes Benz cars; this comprises left-/right turn indication, emergency flashing, crash flashing, theft flashing and open/close flashing. The level of detail given in the model corresponds to the observation level for system testing. To provide the full functionality, several automotive controllers cooperate using various communication busses (CAN and LIN). The signals exchanged between controllers can be observed by the testing environment; additionally the environment can stimulate and monitor discrete and analogue interfaces between SUT and peripherals, such as switches, buttons, indicator lights and various dashboard indications. Capturing this functionality in a formal way requires a concurrent real-time system semantics.

System Interface. In Fig. 1 the interface between system under test (SUT) and testing environment (TE) is shown. Due to the state-based nature of the hardware interfaces (discretes, periodic CAN or LIN bus messages repeatedly sending state information) the modeling formalism handles interfaces as shared variables written to by the TE and read from by the SUT or vice versa.

The TE can stimulate the SUT via all interfaces affecting the turn indication functionality in the operational environment: $\mathsf{in\_CentralLockingRM} \in \{0, 1, 2\}$ denotes the remote control for opening and closing (i. e. unlocking and locking) cars by means of the central locking system. Signal $\mathsf{in\_CrashEvent} \in \{0, 1\}$ activates a crash impact simulator which is part of the TE, and $\mathsf{in\_EmSwitch} \in \{0, 1\}$ simulates the "not pressed"/"pressed" status of the emergency flash switch on the dashboard. Signal $\mathsf{in\_IgnSwitch} \in \{0, \dots, 6\}$ denotes the current status of the ignition switch, and $\mathsf{in\_TurnIndLvr} \in \{0, 1, 2\}$ the status of the turn indicator lever (1 = left, 2 = right). In special-purpose vehicles (SPV), such as taxis or police cars, additional redundant interfaces for activation of emergency flashing and turn indicators exist (e. g., $\mathsf{in\_EmSwitchSPV} \in \{0, 1\}$). Observe that these redundant interfaces may be in conflicting states, so that the control software has to perform a priority-dependent resolution of conflicts. Inputs to the SUT marked by OPTION specify different variants of vehicle style and equipments, each affecting the behavior of the turn indication functions. In contrast to the other input interfaces to the SUT, options remain stable during execution of a test procedure, since their change requires a reset of the automotive controllers, accompanied by a procedure for loading new option parameters. If the TE component does not contain any behavioral specifications, the test generator will create arbitrary timed sequences of input vectors suitable to reach the test goals, only observing the range specifications associated with each input signal. This may lead to unrealistic tests. Therefore the TE may be decomposed into concurrent components (typically called *simulations*) whose behavior describe the admissible (potentially non-deterministic) interaction of the SUT environment on some or all interfaces. The test generator interprets these simulations as additional constraints, so that only sequences of input vectors are created, whose restrictions to the input signals controlled by TE components comply with the transition relations of these simulations.

SUT outputs are captured in the SignalsOut interface (Fig. 1 shows only a subset of them). The indicator lights are powered by the SUT via interfaces pwmRatio_FL, pwmRatio_FR, $\ldots \in \{0, \ldots, 120\}$ where, for example, FL stands for "forward left" and RR for "rear right". The TE measures the percentage of the observed power output generated by the lamp controllers, 100% denoting identity with the nominal value. System integration testing is performed in grey box style: apart from the SUT outputs observable by end users, the TE also monitors bus messages produced by the cooperating controllers performing the turn indication service. Message tim_EFS $\in \{0, 1\}$, for example, denotes a single bit in the CAN message sent from a central controller to the peripheral controllers in order to indicate whether the emergency flash switch indicator on the dashboard should be activated, and tim_FL $\in \{0, 1\}$ is the on/off command to the controller managing the forward-left indicator light.

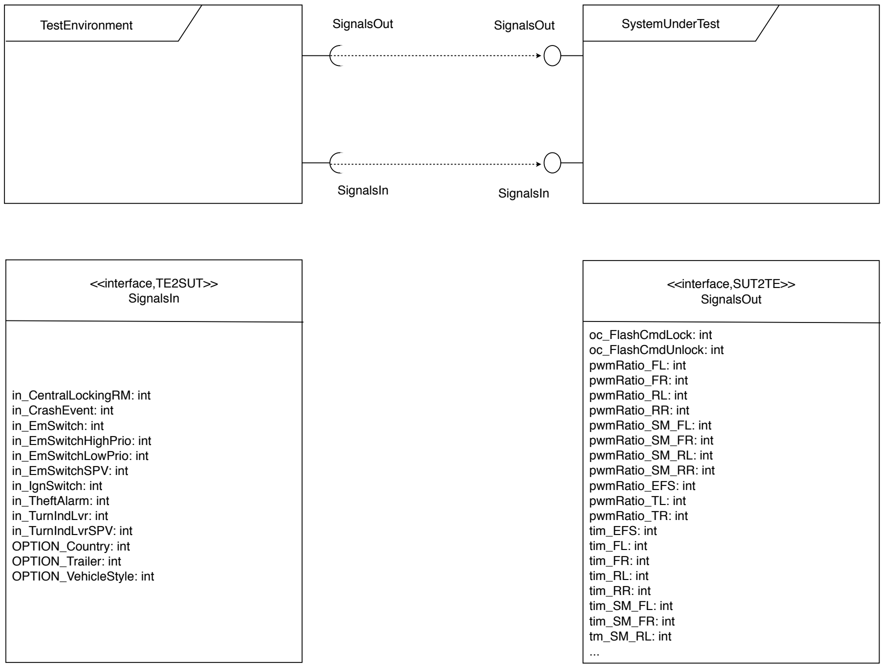


Fig. 1: Interface between test environment and system under test.

First-Level SUT Decomposition. Fig. 2 shows the functional decomposition of the SUT functionality. Component NormalAndEmerFlashing controls left/right turn indication, emergency flashing and the dependencies between both functions (see below). Component OpenCloseFlashing models the indicator-related reactions to the locking and unlocking of vehicles with the central locking system. CrashFlashing models indications triggered by the crash impact controller. TheftFlashing controls reactions triggered by the theft alarm system. These functions interact with each other, as shown in the interface dependencies depicted in Fig. 2: the occurrence of a crash, for example, affects the emergency flash function, and opening a car de-activates a theft alarm. The local decisions of the above components are fed into the PriorityHandling component where conflicts between indication-related commands are resolved: if, for example, the central locking system is activated while emergency flashing is active, the open/close flashing patterns (one time for open, 3 times for close) are not generated; instead, emergency flashing continues. Similarly, switching off the emergency switch has no effect if the high-priority emergency interface (in_EmSwitchHighPrio $\in \{0, 1\}$) is still active. Priority handling identifies the function to be performed and relays the left-hand/right-hand/both sides flashing information to the components OnOffDuration and AffectedLamps. The former determines the durations for switching lights on and off, respectively, during one flashing period. These durations depend both on the status of the ignition switch and the function to be performed. The latter specifies which lamps and dashboard indications have to participate in the flashing cycles. This depends on the OPTION_VehicleStyle which determines, for example, the existence of side marker lamps (interfaces pwmRatio_SM_FL, FR, RL, RR), and on the OPTION_Trailer which indicates the existence of a trailer coupling, so that the trailer turn indication lamps (pwmRatio_TL, TR) have to be activated. Moreover, the affected lamps and indications depend on the function to be performed: open-close flashing, for example, affects indication lamps on both sides, but the emergency flash switch indicator (pwmRatio_EFS) is not activated, while this indicator is affected by emergency, crash and theft flashing. The MessageHandling component transmits duration and identification of affected lamps and indicators on a bus and synchronizes the flash cycles by re-transmission of this message at the beginning of each flashing cycle. Finally, component LampControl comprises all output control functions, each function controlling the flashing cycles of a single lamp or dashboard indicator.

Behavioral Semantics. Model components behave and interact according to a concurrent synchronous real-time semantics, which is close to Harel's original micro-step semantics of Statecharts [10]. Each leaf component of the model is associated with a hierarchic state machine. At each step starting in some model pre-state σ_0, all components possessing enabled state machine transitions process them in a synchronous manner, using σ_0 as the pre-state. The writes of all state machine transitions affect the post-state σ_1 of the micro-step. Two concurrent components trying to write different values to the same variable in the same micro-step cause a *racing condition* which is reflected by deadlock of the transition relation and – in contrast to interleaving semantics – considered as a modeling error. Micro-steps are discrete transitions performed in zero time. Inputs to the SUT remain unchanged between discrete transitions. If the system is in a stable state, that is, all state machine transitions are disabled, time passes in a delay transition, while the system state remains stable. The delay must not exceed the next point in time when a discrete transition becomes enabled, due to a timeout condition. At the end of a delay transition, new inputs to the SUT may be placed on each interface. The distinction between discrete and delay transitions is quite common in concurrent real-time formalisms, and it is also applied to interleaving semantics, as, for example, in Timed Automata [23]. The detailed formal specification of the semantic interpretation of the model is also published on the website given above [21].

Deployment and Signal Mapping. In order to support re-use, the model introduced in this section only specifies a *functional* decomposition. Its deployment on a distributed system of automotive controllers, the network topology, and the concrete interfaces implementing the logical ones shown in the model, depend on the vehicle production series. Even the observability of signals may vary between series, due to different versions and increments in the test equipment. For this reason, model interfaces are mapped to concrete ones by means of a signal map: this map associates concrete signal names which may be written to or read from in the TE with the abstract signals occurring in the model and – in case of SUT outputs – specify their acceptable tolerances.

3 Benchmark Reference Tool

Tool Components and Basic Concepts. The reference data for the benchmarks have been created using our model-based testing tool RT-Tester. It consists of a parser front-end transforming textual model representations (XMI export provided by the CASE tool) into internal representations of the abstract model syntax.

A constraint generator derives all model coverage goals from the abstract syntax tree and optionally inputs user-defined symbolic test cases. Users may select which symbolic test cases should be discharged in the same test procedure. A transition relation generator traverses the model's abstract syntax tree and generates the model transition relation Φ needed for expressing computation goals according to Equation (1). During the test data and test procedure generation process, the constraints associated with these symbolic test cases are passed on to an abstract interpreter. The interpreter performs an abstract conservative approximation of the model states that are reachable from the current model state within a pre-defined number n of steps. The goals which may be covered within n steps according to the abstract interpretation are passed on in disjunctive form to an SMT solver. The solver unrolls the transition relation in a step-by-step manner and tries to solve at least one of the goals. If this succeeds, a timed sequence of input vectors to the SUT is extracted from the

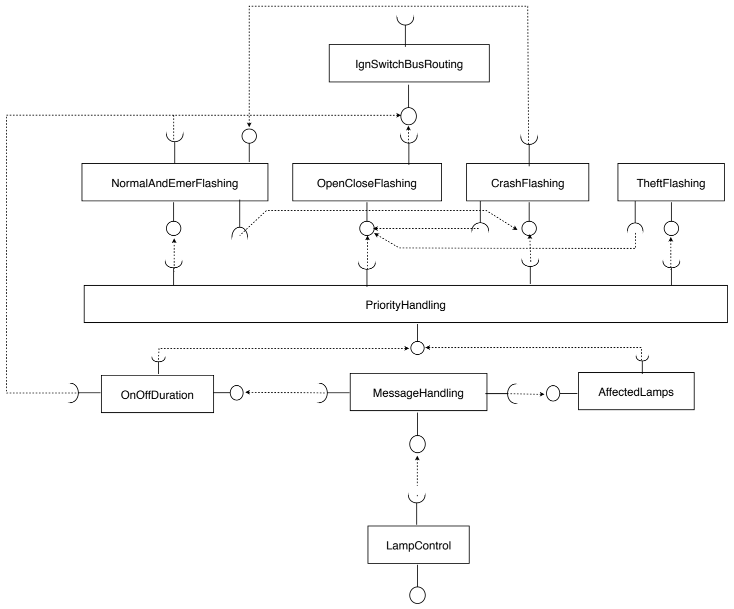


Fig. 2: First-level decomposition of system under test.

solution provided by the SMT solver. Starting from the current model state, a concrete interpreter executes this sequence until a new stable state is reached where further inputs may be generated to cover the remaining goals. If the solver cannot discharge any goal within n steps, random simulations and/or backtracking to model states already visited can be performed in order to identify other model states from where the next goal may be reached. Constraint generator, interpreters and solver represent the core of the tool, called the *test generation engine*. Its components only depend on the abstract syntax representation of the model and its transition relation, but not on the concrete modeling syntax and the syntax required by the test execution environment for the test procedures.

At the end of the generation process a multi-threaded test procedure is generated which stimulates the SUT according to the input sequences elaborated by the solver and simultaneously checks SUT reactions with respect to consistency with the model. In the sections below we highlight the most important features of the tool; a detailed description is given in [21].

SMT-Solver. The constraint solving problems of type (1) may contain linear and non-linear arithmetic expressions, bit-operations, array-references, comparison predicates and the usual Boolean connectives. Data types are Booleans, signed and unsigned integers, IEEE-754 floating-point numbers and arrays.

Our SMT-solver SONOLAR uses the classical bit-blasting approach that transforms a formula to a propositional satisfiability problem and lets a SAT-solver try to find a solution [13, 3]. Variables in the formula are translated to vectors of propositional variables (i. e., *bit vectors*). The lengths of these bit vectors correspond to the bit width of the respective data types. Operations are encoded as propositional constraints relating input to output bit vectors. Since we reason on bit-level, this enables us to precisisely capture the actual semantics of all operations. Integer arithmetic takes potential overflows into account and each floating point operation is correctly rounded to the selected IEEE-754 rounding-mode [1].

To this end, the formula is first represented as an acyclic expression graph, where each variable and each operation of the formula is represented as a node. Using structural hashing on these nodes, identical terms are shared among expressions. This representation allows us to perform word-level simplifications, normalization and substitutions. The expression graph is then bit-blasted to an *And-Inverter Graph (AIG)*. AIGs are used by several SMT solvers to synthesize propositional formulas [13, 3, 14]. Each node of an AIG is either a propositional variable or an *and*-node with two incoming edges that may optionally be inverted, i.e. negated. The AIG is structually hashed and enables us to perform bit-level simplifications. Readers are referred to [9, 4] for more information on logic synthesis using AIGs. The AIG is then translated to CNF using the standard Tseitin encoding and submitted to a SAT solver.

In order to handle the extensional theory of arrays we adopted the approach described in [5]. Instead of bit-blasting all array expressions to SAT up-front, array expressions that return bit-vectors associated with array-reads or checks for array equality are replaced by fresh variables. This results in an over-abstraction of the actual formula since the array axioms are left out. If the SAT solver is able to find a solution to this formula the model is checked for possible array inconsistencies. In this case, additional constraints are added on-demand to rule out this inconsistency. This process is repeated until either the SAT solver finds the refined formula to be unsatisfiable or no more array inconsistencies can be found. While unrolling the transition relation constraints are incrementally added to the SMT solver.

Abstract Interpretation. Our abstract interpreter has been developed to compute over-approximations of possible model computations in a fast way. Its main application is to determine lower bounds of the parameter c in Formula (1) specifying the number of times the transition relation Φ must be unrolled before getting a chance to solve $tc(c, G)$. This considerably reduces generation time, because (a) the SMT solver can skip solution trials for $tc(c, G)$ with values of c making a solution of $tc(c, G)$ infeasible, and (b) the abstract interpretation technique provides the means for non-chronological backtracking in situations where it is tried to solve $tc(c, G)$ from a former model state already visited (see [20] for a more detailed description).

The abstract interpreter operates on abstract domains: interpretation of model behavior is performed using an abstract state space Σ_A instead of the concrete one. Σ_A is obtained by replacing each concrete data type D of the concrete state space Σ with an adequate abstract counterpart $L(D)$. Functions defined over concrete data types $D_0, \ldots, D_n$ are lifted to the associated abstract domains $L(D_0), \ldots, L(D_n)$. In order to be able to reason about concrete computations while computing only the abstract ones, concrete and abstract states are related to one another by Galois connections. A Galois connection is a tuple of mappings $(\triangleright : \mathbb{P}(\Sigma) \to \Sigma_A, \triangleleft : \Sigma_A \to \mathbb{P}(\Sigma))$ defining for any set of concrete states the associated abstract state and vice versa, see [20] for additional details. Finally, each abstract domain $L(D)$ is equipped with a join operator $\sqcup : L(D) \times L(D) \to L(D)$ which establishes the basis for the join operator over two abstract states $\sqcup : \Sigma_A \times \Sigma_A \to \Sigma_A$. This operator is essential as it allows to reduce the complexity usually arising when interpreting large models where the computation of all reachable states would otherwise be infeasible, due to the number of states and the number of control decisions.

The abstract interpreter uses the interval, Boolean and power set lattices as abstract domains for numerical data types, Booleans and state machine locations, respectively. The interpretation of a given model is parametrized by an initial abstract state $\sigma_A^0 \in \Sigma_A$, an integer c_{max} denoting the maximal number of transition steps to be interpreted and a test case goal G to be checked for satisfiability. Starting in the given initial state, the interpreter computes a sequence of up to c_{max} abstract states $\langle \sigma_A^1, \sigma_A^2, \ldots \rangle$ where each state σ_A^{i+1} is guaranteed to "include"[4] every concrete state σ^{i+1} reachable from any of the concrete states represented by σ_A^i. The interpretation stops as soon as either the maximal number of steps has been reached or the test case goal G evaluates to `true` or $\top$[5]. In the latter case the actual step number is returned.

Given any abstract pre-state σ_A, the interpreter computes the abstract post-state $\sigma_A^{'}$ by executing a delay or a discrete transition, depending on the transition type the interpreted model would engage in. The transition type is established by evaluating the trigger condition for discrete transitions in the pre-state σ_A which may yield one of three different cases to be distinguished by the interpreter: (1) if the trigger condition evaluates to `false`, a delay transition is performed, that is, all inputs are set to their associated value ranges and the model execution time is advanced by the time shift interval $[ts_{min}, ts_{max}]$. In this interval, ts_{min} represents the smallest time unit distinguishable in the test system and ts_{max} is the amount of time that has to elapse before at least one state machine is guaranteed to be able to perform a discrete transition. (2) In the case that the trigger condition evaluates to `true`, the interpreter performs a discrete transition. Because one single state machine is sufficient to yield this case, in general, there will also be state machines with disabled discrete transitions and such for which it cannot be determined whether a discrete transition is enabled or not. In order not to miss any possible computation, the interpreter creates a post-state which

[4] In the sense that $\sigma^{i+1} \in (\sigma_A^{i+1})^{\triangleleft}$

[5] $\top$ is the Boolean lattice value representing the concrete value set $\{\mathtt{false}, \mathtt{true}\}$.

includes every possible combination of state machine behaviors where at least one state machine performs a transition or executes some do-actions. (3) If the trigger condition evaluates to $\top$, both, a delay and a discrete transition have to be considered. In this case, the interpreter computes two possible post-states $\sigma_A^{'1}$ and $\sigma_A^{'2}$ assuming that the trigger condition evaluates to `true` or `false`, respectively. The post-state $\sigma_A^{'}$ is then obtained as the join of the individual results.

Independently from the considered case, the interpreter always computes one single post-state $\sigma_A^{'}$. Due to the including property of the join operator this necessarily yields post-states that are relatively coarse. Our recent work, therefore, is related to precision improving techniques like trace partitioning as described e.g in [7] for the domain of program analysis.

Test Oracles. Our tool automatically generates test oracles from the model which run in *back-to-back* fashion against the SUT, performing on-the-fly checks of SUT reactions against expected results. To this end, all state machines in the SUT-portion of the model are transformed into separate tasks by a model-to-text generator. Depending on the TE infrastructure, these tasks can be distributed on several CPU cores and TE computers. As a result, every observable SUT output is continuously checked against its expected value. Values which are not observable are calculated by the corresponding model components and passed on to other components consuming these values, so that they can proceed with their computations without the availability of the corresponding value produced by the SUT. We prefer this technique for creating test oracles to the more conventional one where expected SUT outputs are only checked at specific points in time during the test execution, for example, when changes at some SUT output interface are expected: the conventional technique has the draw back that it may fail to detect transient instabilities at the SUT output interface.

Observe, however, that our technique depends on the deterministic behavior of the SUT. Fortunately this is guaranteed for the automotive controllers under consideration, and for highly safety-critical control systems in avionics and in the railway domain.

4 MBT Benchmark Classification

We propose to classify MBT benchmarks according to the following characteristics, denoted by *test strength benchmarks* and *test generation benchmarks.*

Test strength benchmarks investigate the error detection capabilities of concrete test cases and test data generated by MBT tools: even if two MBT tools produce test suites of equivalent model coverage, they will usually possess different strength, due to different choices of symbolic test cases, representatives of equivalence classes, boundary values and timing of input vectors passed to the SUT, or due to different precision of the test oracles generated from the model.

Mutation testing is an accepted approach to assessing test suite strength; therefore we suggest to generate model mutants and run test suites generated from the unbiased model as model-in-the-loop tests against these mutants. The evaluation criterion is the percentage of uncovered mutations for a fixed set of mutant models.

Test generation benchmarks input symbolic test cases as introduced in Section 3 and measure the time needed to generate concrete test data. We advocate a standard procedure for providing these test objectives for a given model, structuring symbolic test cases into several sets. The first sets should be related to model coverage criteria [26], such as (1) control state coverage (every control state of the SUT model is visited by at least one test), (2) state machine transition coverage (every transition of every state machine is taken at least once) and (3) MC/DC coverage (conditions of the type $\phi_1 \wedge \phi_2$ are tested at least once for each of the valuations $(\phi_1, \phi_2) = (\texttt{false}, \texttt{true}), (\texttt{true}, \texttt{false}), (\texttt{true}, \texttt{true})$, and conditions of the type $\phi_1 \vee \phi_2$ are tested at least for $(\phi_1, \phi_2) = (\texttt{false}, \texttt{false})$, $(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true})$).

Conventional model coverage criteria as the ones listed in (1 — 3) do not possess sufficient strength for concurrent real-time systems, because the dependencies between state machines operating in parallel are not sufficiently addressed. Since models as the one under consideration are too large to consider coverage of all state vector combinations, a pragmatic compromise is to maximize the coverage of all basic control state pairs of interacting components C_1, C_2, combined with pairs of input equivalence classes of signals influencing C_1, C_2. As a consequence we suggest symbolic test cases consisting of (a subset of) these combinations as a forth set. As a fifth set of symbolic test cases it is proposed to define application-specific test cases of specific interest.

Given these classes for a specific model, this induces 5 test suites to realize a comprehensive test generation benchmark.

Evaluation criteria for test generation benchmarks. Apart from the time needed to generate concrete test data, the number of SUT resets involved in the resulting test procedures should be minimized as well: since SUT resets usually consume significant time when testing embedded systems, hardware-in-the-loop tests avoiding resets significantly reduce the test suite execution time. Moreover, test executions covering many test cases drive the SUT into more internal states than test executions resetting the SUT between two or only a small number of test cases. As a consequence, the error detection capabilities of test procedures are usually increased with the number of test cases they cover between resets. Avoiding resets is adverse to the reduction of generation time: if some goal $\mathbf{F}\phi$ is very time consuming to reach from a given model state σ_0, backtracking to a former model state from where computation fragments fulfilling $\mathbf{F}\phi$ can be reached more easily frequently helps to reduce the generation time in a significant way. Since the SUT is usually unable to roll back into a previous state, backtracking enforces a SUT reset, after which the new computation can be exercised. For comparing the performance of tools we suggest to calculate Pareto frontiers of pairs *(generation time,number of resets)* for each competing tool, and compare the tool-dependent frontiers.

Significance of test generation benchmarks. According to standards applicable to safety-critical systems verification [24, 22] the error detection strength of a test suite is just one aspect to be addressed when justifying the adequateness of test cases. Complementary to that the standards require to account for sufficient coverage on the levels of requirements, design and code. As a consequence, the capability of test automation tools to generate sufficient test cases to achieve such coverage has significant impact on verification and certification efforts.

5 Test Generation Example

The benchmark website [19] contains two classes of symbolic test cases: (1) *user-defined test cases* reflect specific test purposes identified by test engineers. They serve either to test more complex requirements which cannot be traced in the model to simple sets of basic control states or transitions to be covered, or they are used to explore SUT reactions in specific situations where a failure is suspected. (2) *Model-defined test cases* aim at covering certain parts of the model according to pre-defined strategies, such as basic control state coverage, state machine transition coverage and MC/DC coverage. They are automatically derived by our tool from the abstract syntax representation of the model.

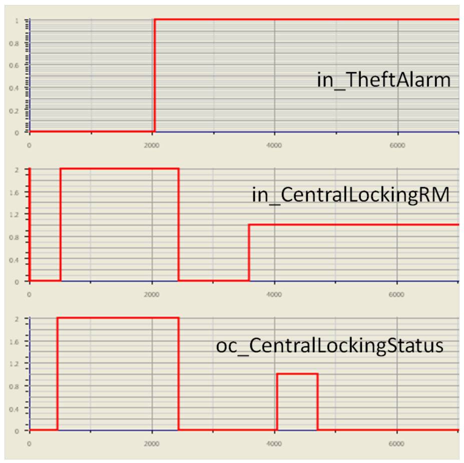
In this section a test generation based on user-defined test cases from [19, Test UD 003] is presented. The underlying test purpose is to investigate the interaction between theft alarm and open/close flashing: theft alarm flashing is only enabled when the doors are locked. As a reaction to alarm-sensor activation the turn indicator lights shall start flashing on both sides. Pressing the remote control key to unlock the doors automatically shuts off the alarm flashing.

Instead of explicitly determining the sequence of timed input vectors to the SUT which is suitable for covering the test purpose described, we specify simpler and shorter symbolic test cases that may also refer to internal model states and leave it up to the tool's generation engine to calculate the concrete test input data and its timing. Symbolic test case[6]

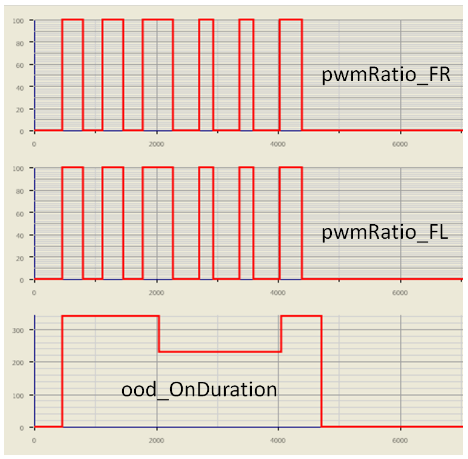
```
TC-turn_indication-THEFT_ALARM-0001;
   [ SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_ACTIVE.ALARM_OFF
   && ! in_TheftAlarm ]
   Until
   [ _timeTick >= 2000 && in_TheftAlarm ]
```

refers to SUT inputs (theft alarm-sensor `in_TheftAlarm`) the model execution time (`_timeTick`) and basic control states of state machines which are part of the SUT model (`SystemUnderTest.TheftFlashing...ALARM_OFF`). The LTL

[6] The symbols used in the test cases below are taken from the turn indicator model published in [19]. A detailed description of inputs, outputs and internal model variables can be found there.



(a) Generated inputs and internal model state `oc_CentralLockingStatus`.



(b) Expected outputs and internal model state `ooo_OnDuration`.

Fig. 3: Generation results of theft alarm test procedure.

formula is a directive to the test generation engine to find a computation which finally reaches a model state where theft alarm flashing is enabled but not yet active (this is the case when the basic control state `...ALARM_OFF` is reached), and the theft alarm-sensor should remain passive until 2000ms have passed since start of test (the leading finally operator is always omitted in our symbolic test case specifications). The inputs derived by the generation engine to cover this test case drive the SUT into a state where an alarm is signaled and the SUT has to react by activating theft alarm flashing. The next symbolic test case to be processed is

```
TC-turn_indication-THEFT_ALARM-0002;
  [ SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_ACTIVE.ALARM_ON
    && in_TheftAlarm ]
  Until
  [ _timeTick >= 4000 && IMR.in_TheftAlarm &&
    SystemUnderTest.oc_CentralLockingStatus == 1 ]
```

This formula is a directive to stay in the theft alarm state for at least another 2 seconds after which a model state is to be reached where the internal model variable `oc_CentralLockingStatus` has value 1 (= *"unlocked"*), indicating that an *"unlock doors"* command has been given via remote key control. Again, the associated inputs and timing is calculated by the test generation engine. The final symbolic test case to be processed by the generator is

```
TC-turn_indication-THEFT_ALARM-0003;
  [ SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_OFF ]
  Until
  [ _timeTick >= 6000 &&
  SystemUnderTest.TheftFlashing.TheftFlashing.THEFT_ALARM_OFF ]
```

It is a directive to stay in the *"theft alarm disabled"* state `...THEFT_ALARM_OFF` for at least another two seconds, so that it can be observed that after one flash period signaling that the doors have been unlocked, no further alarm indications are made. The signal flow associated with this test (inputs and expected SUT outputs) is depicted in Fig. 3. The generator created a sequence of input vectors where first doors are closed by means of the remote key control input `in_CentralLockingRM` (2 = lock, 1 = unlock). This triggers three flashing periods for left and right indicator lamps (`pwmRatio_FR, pwmRatio_FL`, see Fig. 3b). For open/close flashing the on-duration of a flashing period is 340ms; this is captured in the internal model variable `ood_OnDuration` whose contents will be transmitted by the SUT via CAN bus and can therefore be observed and checked by the threads running on the test engine and acting as test oracles. After two seconds an alarm is raised by setting `in_TheftAlarm = 1`. This changes the on-duration of the flashing period to 220ms. Theft alarm flashing is switched off at model execution time stamp 4000, approx. 500ms after the unlock-doors signal has been given (`in_CentralLockingRM=1`); the change of the internal `oc_CentralLockingStatus` from 0 to 1 indicates that the *“doors unlocked”* status has now been realized (see Fig. 3a). One flashing period signals “doors unlocked” (again with on-duration 340ms), after which no further alarm indications occur.

6 Conclusion

We have presented a model of an automotive control application, covering the full functionality related to turn indication, emergency flashing, crash, theft and open/close flashing. The model is a 1-1-transcription of the one currently used by Daimler for system testing of automotive controllers. As the only adaptation we have presented the model in pure UML 2.0 style, while Daimler uses a specific UML profile optimized for their hardware-in-the-loop testing environment. The model is made available to the public in complete form through the website [19], together with benchmark test suites and performance values achieved with our reference tool. Additionally, a classification of benchmarks for embedded systems test tools has been suggested which takes into account both the test strength and the performance for automated test suite generation.

The underlying methods of the MBT tool used by the authors for performing embedded systems test have been described, in order to facilitate the comparison of competing techniques applied to the benchmarks in the future. The tool is currently applied in industrial projects in the automotive, railway and avionics domains.

References

1. IEEE Standard for Floating-Point Arithmetic. Tech. rep., Microprocessor Standards Committee of the IEEE Computer Society (2008)
2. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-box system testing of real-time embedded systems using random and search-based testing. In: Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems. pp. 95–110. ICTSS’10, Springer-Verlag, Berlin, Heidelberg (2010)

3. Brummayer, R.: Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays. Ph.D. thesis, Johannes Kepler University Linz, Austria (November 2009)
4. Brummayer, R., Biere, A.: Local two-level and-inverter graph minimization without blowup. In: In Proceedings of the 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS06 (2006)
5. Brummayer, R., Biere, A.: Lemmas on Demand for the Extensional Theory of Arrays. In: Proc. 6th Intl. Workshop on Satisfiability Modulo Theories (SMT'08). ACM, New York, NY, USA (2008)
6. Conformiq Tool Suite: (2010), http://www.conformiq.com
7. Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Rival, X.: Why does Astrée scale up? Formal Methods in System Design (FMSD) 35(3), 229–264 (December 2009)
8. David, A., Larsen, K.G., Li, S., Nielsen, B.: Timed testing under partial observability. In: Proc. 2nd International Conference on Software Testing, Verification and Validation (ICST'09). pp. 61–70. IEEE Computer Society (2009)
9. Eén, N., Mishchenko, A., Sörensson, N.: Applying logic synthesis for speeding up SAT. In: Marques-Silva, J.a., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing (SAT 2007), Lecture Notes in Computer Science, vol. 4501, chap. 26, pp. 272–286. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
10. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (October 1996)
11. Harrold, M.J., Rothermel, G.: Siemens programs, hr variants. http://ww.cc.gatech.edu/aristotle/Tools/subjects
12. Jeppu: A benchmark problem for model based control systems tests - 001. http://www.mathworks.com/matlabcentral/fileexchange/28952-a-benchmark-problem-for-model-based-control-system-tests-001 (2010)
13. Jha, S., Limaye, R., Seshia, S.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: Computer Aided Verification, pp. 668–674 (2009), http://dx.doi.org/10.1007/978-3-642-02658-4_53
14. Jung, J., Sülflow, A., Wille, R., Drechsler, R.: SWORD v1.0. Tech. rep. (2009), SMTCOMP 2009: System Description
15. Löding, H., Peleska, J.: Timed moore automata: test data generation and model checking. In: Proc. 3rd International Conference on Software Testing, Verification and Validation (ICST'10). IEEE Computer Society (2010)
16. Lu, S., Li, Z., Quin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the evaluation of software defect detection tools (2005)
17. Memon, A.M.: http://www.cs.umd.edu/ atif/newsite/benchmarks.htm
18. Nielsen, B., Skou, A.: Automated test generation from timed automata. International Journal on Software Tools for Technology Transfer (STTT) 5, 59–77 (2003)
19. Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E., Zahlten, C.: Embedded systems testing benchmark (2011), http://www.mbt-benchmarks.org
20. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) Nasa Formal Methods, Third International Symposium, NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Pasadena, CA, USA (April 2011)
21. Peleska, J., Vorobev, E., Lapschies, F., Zahlten, C.: Automated model-based testing with RT-Tester. Tech. rep. (2011), http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/tool/rtt-mbt.pdf

22. RTCA,SC-167: Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B. RTCA (1992)
23. Springintveld, J., Vaandrager, F., D'Argenio, P.: Testing timed automata. Theoretical Computer Science 254(1-2), 225–257 (March 2001)
24. European Committee for Electrotechnical Standardization: EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems. CENELEC, Brussels (2001)
25. Systems, S.: Enterprise architect 8.0. http://www.sparxsystems.de (2011)
26. Weißleder, S.: Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines. Doctoral thesis, Humboldt-University Berlin, Germany (2010)