Softw Syst Model
DOI 10.1007/s10270-017-0595-8

SPECIAL SECTION PAPER

Experimental evaluation of a novel equivalence class partition testing strategy

Felix Hübner[1] · **Wen-ling Huang**[1] · **Jan Peleska**[1]

Received: 8 July 2016 / Revised: 17 January 2017 / Accepted: 16 March 2017

Abstract In this paper, a complete model-based equivalence class testing strategy recently developed by the authors is experimentally evaluated. This black-box strategy applies to deterministic systems with infinite input domains and finite internal state and output domains. It is complete with respect to a given fault model. This means that conforming behaviours will never be rejected, and all non-conforming behaviours inside a given fault domain will be uncovered. We investigate the question how this strategy performs for systems under test whose behaviours lie *outside* the fault domain. Furthermore, a strategy extension is presented, that is based on randomised data selection from input equivalence classes. While this extension is still complete with respect to the given fault domain, it also promises a higher test strength when applied against members outside this domain. This is confirmed by an experimental evaluation that compares mutation coverage achieved by the original and the extended strategy with the coverage obtained by random testing. For mutation generation, not only typical software errors, but also critical HW/SW integration errors are considered. The latter can be caused by mismatches between hardware and software design, even in the presence of totally correct software.

Communicated by Prof. Alfonso Pierantonio, Jasmin Blanchette, Francis Bordeleau, Nikolai Kosmatov, Gabi Taentzer, Manuel Wimmer.

✉ Jan Peleska
jp@cs.uni-bremen.de
http://www.cs.uni-bremen.de/agbs

Felix Hübner
felixh@cs.uni-bremen.de

Wen-ling Huang
huang@cs.uni-bremen.de

[1] Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany

Keywords Model-based testing · Equivalence class partition testing · Adaptive random testing · SysML · SystemC · State transition systems

1 Introduction

1.1 Motivation

In model-based testing (MBT), *complete* testing strategies have guaranteed fault coverage with respect to a given fault model $\mathcal{F} = (\mathcal{S}, \leq, \mathcal{D})$. The latter are defined by a reference model $\mathcal{S}$, a conformance relation $\leq$, and a fault domain $\mathcal{D}$ containing models $\mathcal{S}'$ that may conform to the reference model ($\mathcal{S}' \leq \mathcal{S}$) or not ($\mathcal{S}' \not\leq \mathcal{S}$). Complete test suites are characterised by the fact that a system under test (SUT) whose true behaviour can be expressed by a member of the fault domain passes the test suite if and only if it conforms to the reference model.

Due to their guaranteed fault detection properties, complete testing strategies are not only interesting from a theoretical perspective, but also for practical application: in the field of safety-critical systems verification, the test suites performed have to be justified with respect to their strength in order to obtain certification credit. Complete test suites come with guaranteed strength, provided that it is safe to assume that the SUT is a member of the fault domain. This, however, can only be justified in rare cases.

This fact motivates the experimental evaluation of a complete input equivalence class partition (IECP) testing strategy recently developed by the authors [21]. We show that its test strength is still significant when applied to an SUT whose true behaviour is captured by models *outside* the fault domain.

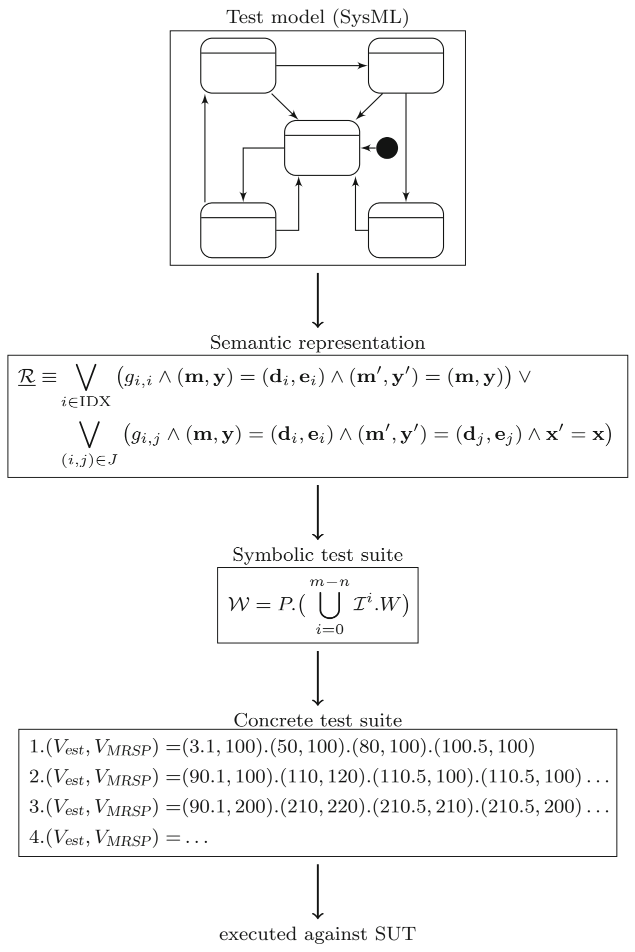

Test model (SysML)
Semantic representation
$\mathcal{R} \equiv \bigvee_{i \in \mathrm{IDX}} (g_{i,i} \wedge (\mathbf{m},\mathbf{y}) = (\mathbf{d}_i,\mathbf{e}_i) \wedge (\mathbf{m}',\mathbf{y}') = (\mathbf{m},\mathbf{y})) \vee \bigvee_{(i,j) \in J} (g_{i,j} \wedge (\mathbf{m},\mathbf{y}) = (\mathbf{d}_i,\mathbf{e}_i) \wedge (\mathbf{m}',\mathbf{y}') = (\mathbf{d}_j,\mathbf{e}_j) \wedge \mathbf{x}' = \mathbf{x})$
Symbolic test suite
$\mathcal{W} = P.(\bigcup_{i=0}^{m-n} \mathcal{I}^i.W)$
Concrete test suite
$1.(V_{est}, V_{MRSP}) = (3.1, 100).(50, 100).(80, 100).(100.5, 100)$
$2.(V_{est}, V_{MRSP}) = (90.1, 100).(110, 120).(110.5, 100).(110.5, 100) \ldots$
$3.(V_{est}, V_{MRSP}) = (90.1, 200).(210, 220).(210.5, 210).(210.5, 200) \ldots$
$4.(V_{est}, V_{MRSP}) = \ldots$
executed against SUT

Fig. 1 Tool-supported workflow

1.2 Workflow and tool support

In Fig. 1, the workflow associated with our testing approach is shown. Test models are represented in a concrete modelling formalism; for the models presented in this paper, SysML [36] has been used. As explained in Sect. 2, the test model is translated into a state transition system whose behavioural semantics is expressed by means of an initial condition and a transition relation, both represented as first-order formulas as described in [12, 2.1.1]. From the transition relation, equivalence classes are calculated. These give rise to a model abstraction as a deterministic finite-state machine (DFSM). Applying well-known complete testing strategies for DFSMs, a symbolic test suite is derived. Each test case of this suite is represented as a sequence of input equivalence classes. Selecting concrete input data from each of these classes by means of an SMT solver, a complete test suite for the original test model is generated. The whole process is automated and has been integrated in the model-based test automation tool RT-Tester [37].

1.3 Objectives and main contributions

As explained above, the main objective of this article is to investigate how complete test suites perform for an SUT whose behaviour is represented by models outside the fault domain. To this end, four test strategies are evaluated with respect to their strength.

STRAT-RND	Conventional random testing—this serves as a lower bound of test strength, to be surpassed by any more sophisticated strategy.
STRAT-1	The original complete IECP strategy from [21] which uses fixed representatives from each input equivalence class (IEC) for testing.
STRAT-2	An extension of IECP strategy STRAT-1 by random selections from each IEC, whenever the class is referenced in a test case.
STRAT-3	A refinement of STRAT-2 by equally partitioned random selections from each IEC, so that 50% of the selections come from the IEC's interior, and 50% from the boundary of the IEC.

An experimental evaluation is performed which is based on two test models: a speed monitor from the European Train Control System and an airbag controller for vehicles. Applying the four strategies against a collection of mutants, the experimental evaluation confirms significant test strength improvements in strategy STRAT-1, -2, -3 over STRAT-RND, and the highest test strength is achieved by STRAT-3.

When performing experimental evaluations of test strength, inadequate mutant selections represent one of the main threats to validity. As a first step, this threat has been mitigated by using different mutant generators. As a second step, we have considered two classes of mutations.

1. Typical software mutations created by fault injections in Boolean or arithmetic expressions, assignment targets, and jump targets.
2. Typical errors related to incompatibilities between software and hardware, such as byte order mismatches, insufficient arithmetic precision due to inadequate register width, insufficient sensitivity at input interfaces, stuck-at-faults, and port switches.

We emphasise that the work presented here targets *functional* software and HW/SW integration testing of embedded control systems. Therefore, our mutants do not produce typical *runtime errors* (e.g. violation of variable boundaries during memory copies, dereferencing of illegal pointer addresses, or violation of stack boundaries). This is justified by the well-known fact that runtime errors should rather be uncovered by means of abstract interpretation techniques during software development [31]. Functional testing is not well suited for this task.

1.4 Extensions of previous work

This article extends previous work published in [23]. These extensions comprise

– comprehensive revision of the test suite design used in the experimental evaluation, based on the experiences gained in [23,38],
– the design, application, and evaluation of the refined strategy STRAT-3 introduced above, and
– the development of a mutant generation technique that also covers HW/SW integration errors.

The new experimental evaluation presented here follows the objective to limit the size of the test suites involved, while still achieving effective fault coverage (we consider a fault coverage greater than 90% as effective). It is shown that this can be achieved with a far lower number of test cases than the one used in [23], when the new strategy STRAT-3 is applied.

While mutants used in [23] were based on Java reference implementations, they are now based on SystemC [24] which is a modelling language based on C++, allowing to combine software code expressing behaviour with constructs modelling the underlying hardware design. An analysis is performed which identifies typical error situations whose root causes are mismatches between hardware and software design. We have developed a new mutant generator which traverses the SystemC syntax tree and injects both software errors into the functional modules of the SystemC model and HW/SW integration specific errors as listed above. Since SystemC models are executable, they can be directly applied as (correct or mutated) SUTs running against the test suites created according to strategies STRAT-RND and STRAT-1, -2, -3.

1.5 Contributions to the state of the art in the field of test automation

To our best knowledge, the main advancements in the field of test automation provided by this article in relation to existing work can be summarised as follows.

1. The underlying approach to equivalence class testing is the first to handle infinite input domains for all modelling formalisms whose behavioural semantics can be expressed by state transition systems (this point is comprehensively justified in [21,22], where the completeness properties of the equivalence class testing strategy have been proven for deterministic and non-deterministic systems).
2. Our work seems to be the first to experimentally investigate the effectiveness of complete testing strategies under "real-world" conditions, where the true SUT behaviour cannot always be guaranteed to be captured by fault domain members.
3. Our selection of mutants for test strength evaluation not only covers software mutations, but also HW/SW incompatibilities simulated by systematic mutations of SystemC models. This approach is new for evaluating the strength of testing strategies typically applied to HW/SW integration testing of embedded control systems.

Further comparisons of our work to existing results are described in Sect. 5.

1.6 Overview

In Sect. 2, the original IECP testing strategy from [21] is summarised, and the strategy extension by means of randomisation is explained and justified. In Sect. 3, the two reference models used in the strategy evaluation are presented. The experimental setup, the evaluation approach, and the evaluation results, as well as potential threats to validity, are described in Sect. 4. Section 5 refers to related work, and Sect. 6 presents the conclusions and plans for future work.

2 Model-based random testing and equivalence class partition testing

2.1 Random testing

In *model-based random testing*, test cases are created by generating random values as SUT inputs. To this end, the input interface signatures of the SUT are extracted from the model, so that the random values are created in the appropriate data ranges. Apart from this, the input data creation is not guided further by the model. Additionally, the model is used as a test oracle, so that the observed SUT behaviour can be compared to the expected behaviour specified by the model.

When performing black-box tests of SUTs with internal states, the SUT behaviour depends on the *sequence* of inputs provided since the last SUT reset. As a consequence, test cases are specified by sequences of random inputs. Models serving as test oracles need to simulate the internal state changes to be performed by the SUT on each input, in order to predict the SUT reactions in a correct way. While random testing is quite easy to mechanise, its test strength is usually rather weak, because the test case selection does not take into account the required SUT behaviour. On the other hand, random testing is an obvious candidate for assessing the test strength of more refined model-based testing strategies: any successful refined strategy should have a test strength that is significantly higher than the random testing approach.

The simple random testing strategy described above will be denoted as STRAT-RND in the experimental evaluations described in Sect. 4.

Example 1 Consider the SysML state machine shown in Fig. 2 which operates on three simple states ℓ_1, ℓ_2, ℓ_3, model input $x \in D_x = \mathbb{Z}$ and model output $y \in \{0, 1\}$. For a simple random test according to STRAT-RND, test case generation evaluates the information $x \in D_x = \mathbb{Z}$ and creates sequences of uniformly distributed integral numbers as inputs for each test case. The state machine is then used to calculate the expected results. For a randomly generated input sequence $\overline{x} = 1.0.7.88$, for example, the expected output sequence would be $\overline{y} = 1.0.1.1$: the fourth input is processed by the state machine in state ℓ_3, so value 88 does not trigger any transition, and the output y keeps its value 1, which is again recorded in the last value of $\overline{y}$.

2.2 Equivalence class partition testing

2.2.1 Semantic domain

The novel equivalence class partition testing strategy presented in [21] is applicable to deterministic, livelock-free systems with conceptually infinite input domains and finite internal state and output domains. "Conceptually infinite" means that the domains are too large to be explicitly enumerated for test purposes. This includes physical models with real-valued inputs, but can also apply to finite but very large data types such as 64-bit integers or doubles as used in typical programming languages or modelling formalisms. As pointed out in [7,8,21], this class of systems is quite significant in the embedded systems domain: typical candidates are controllers processing analogue inputs and deriving discrete control decisions from these inputs, such as thrust reversal controllers in aircrafts, or the speed monitors and airbag controllers described in this paper.

The strategy has been proven to be complete on the semantic domain of *Reactive Input Output State Transition Systems (RIOSTSs)* $\mathcal{S} = (S, s_0, R, V, D)$. These systems have state spaces S, initial states $s_0 \in S$, and transition relations $R \subseteq S \times S$. Their state spaces consist of valuation functions $s : V \to D$, where V is a set of variable symbols and D is the union of all variable domains. The variable symbols can be partitioned into $V = I \cup M \cup O$, where I comprises input variables, M (internal) model variables, and O output variables. RIOSTSs distinguish between *quiescent* states $s \in S_Q$ and *transient* states $s' \in S_T$, such that $S_Q \cup S_T$ partitions the state space S. Transitions from quiescent states only change input valuations, while internal model variables and output variables remain unchanged. The resulting post-states may be quiescent or transient. Transitions from transient states always have uniquely determined quiescent post-states (so we only allow deterministic RIOSTSs here), and the associated transitions leave the inputs unchanged. This concept represents a natural abstraction of timed formalisms, where delay transitions allow for time to pass and inputs to be changed, while discrete transitions produce output and change internal state, but are executed in zero time [4, p. 687].

By associating atomic propositions AP with free variables in V, any RIOSTS can be extended to a Kripke Structure [12] $K(\mathcal{S}) = (S, s_0, R, V, D, L, AP)$. The labelling function $L : S \to 2^{AP}$ maps $s \in S$ to the set of all atomic propositions $p \in AP$ that evaluate to `true`, when replacing every free variable v of p by its valuation $s(v)$ in state s.

Notation In the exposition below, variable symbols are enumerated with the naming conventions $I = \{x_1, \ldots, x_k\}$, $M = \{u_1, \ldots, u_p\}$, $O = \{y_1, \ldots, y_q\}$. We use notation $\mathbf{x} = (x_1, \ldots, x_k)$ for input variable vectors, and their valuation in state s is written as $s(\mathbf{x}) = (s(x_1), \ldots, s(x_k))$. $D_I = D_{x_1} \times \cdots \times D_{x_k}$ denotes the Cartesian product of the input variable domains. Tuples $\mathbf{u}$, $\mathbf{y}$ and D_M and D_O are defined over model variables and outputs in an analogous way. By $s \oplus \{\mathbf{x} \mapsto \mathbf{c}\}$, $\mathbf{c} \in D_I$, we denote the state s' which coincides with s on all variables from $M \cup O$, but maps the input vector to valuation $s'(\mathbf{x}) = \mathbf{c}$. For $(s_1, s_2) \in R$, we also use the shorter expression $R(s_1, s_2)$. Restricting a state s to variable symbols from a set $U \subseteq V$ is denoted by $s|_U$. This function has domain U and coincides with s on this domain.

2.2.2 Application to concrete modelling formalisms

The test strategy described below is elaborated on the semantic domain of RIOSTSs. Every concrete modelling formalism whose behavioural semantics can be represented by RIOSTSs is automatically equipped with such a test strategy: the concrete model $\mathcal{M}$ is translated into its corresponding RIOSTS $\mathcal{S}$. Then, the test strategy is applied to $\mathcal{S}$, and this results in a set of test cases, each case represented by a finite sequence of inputs to the SUT. When executing the test cases, the transition relation of $\mathcal{S}$ is used to determine whether the SUT's reactions to these input sequences are adequate. In this article, concrete models are expressed by SysML state machines, and these can be associated with RIOSTS semantics which is consistent with the semi-formal specification of state machine behaviour in the UML/SysML standards [35,36].

Example 2 Consider again the SysML state machine from Fig. 2. Its behavioural semantics can be represented by RIOSTS $\mathcal{S} = (S, s_0, R, V, D)$ with $V = I \cup M \cup O$ and $I = \{x\}$, $M = \{\ell\}$, and $O = \{y\}$. Internal model variable ℓ represents the simple states of the state machine

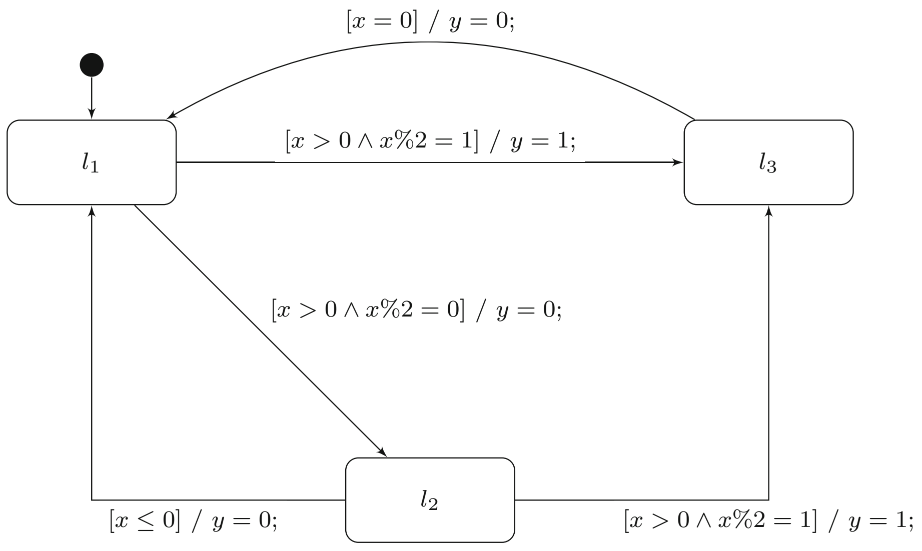

$[x = 0]$ / $y = 0$;
l_1
$[x > 0 \wedge x\%2 = 1]$ / $y = 1$;
l_3
$[x > 0 \wedge x\%2 = 0]$ / $y = 0$;
l_2
$[x \leq 0]$ / $y = 0$;
$[x > 0 \wedge x\%2 = 1]$ / $y = 1$;

Fig. 2 Sample state machine for Example 1

and has range $D_\ell = \{1, 2, 3\}$. The variable domains are $D = D_x \cup D_\ell \cup D_y$ with D_x, D_y defined in Example 1. Following [12], the initial state can be specified by the first-order predicate $\mathcal{I} \equiv x = 0 \wedge y = 0 \wedge \ell = 1$ over variable symbols from V: $\mathcal{I}$ specifies the uniquely determined initial state $s_0 : V \to D$ satisfying $s_0(x) = 0, s_0(y) = 0, s_0(\ell) = 1$.

The transition relation R can be expressed by means of a first-order predicate $\mathcal{R}$ with free variables in V and $V' = \{v' \mid v \in V\}$, such that the unprimed symbols $v \in V$ refer to pre-states and the primed variables $v' \in V'$ to post-states. For the state machine from Fig. 2, this predicate is

$$\begin{aligned}
\mathcal{R} &\equiv \varphi_1 \vee \varphi_2 \vee \varphi_3 \\
\varphi_1 &\equiv (x \leq 0 \wedge \ell = 1 \wedge \ell' = \ell \wedge y' = y) \vee \\
&\quad (x > 0 \wedge x\,\%2 = 0 \wedge \ell = 1 \wedge x' \\
&= x \wedge \ell' = 2 \wedge y' = 0) \vee \\
&\quad (x > 0 \wedge x\,\%2 = 1 \wedge \ell = 1 \wedge x' \\
&= x \wedge \ell' = 3 \wedge y' = 1) \\
\varphi_2 &\equiv (x > 0 \wedge x\,\%2 = 0 \wedge \ell = 2 \wedge \ell' = \ell \wedge y' = y) \vee \\
&\quad (x \leq 0 \wedge \ell = 2 \wedge x' = x \wedge \ell' = 1 \wedge y' = 0) \vee \\
&\quad (x > 0 \wedge x\,\%2 = 1 \wedge \ell = 2 \wedge x' \\
&= x \wedge \ell' = 3 \wedge y' = 1) \\
\varphi_3 &\equiv (x \neq 0 \wedge \ell = 3 \wedge \ell' = \ell \wedge y' = y) \vee \\
&\quad (x = 0 \wedge \ell = 3 \wedge x' = x \wedge \ell' = 1 \wedge y' = 0)
\end{aligned}$$

Predicate $\mathcal{R}$ is a disjunction of φ_i, $i = 1, 2, 3$, where each φ_i applies to simple state ℓ_i. Predicate φ_1, for example, specifies three disjunctions, each applicable in simple state ℓ_1; this is reflected by the condition $\ell = 1$ which is contained in each of these disjuncts. The first disjunct specifies the stability condition: if the system is in state ℓ_1 and the input is less or equal zero, the system will remain in ℓ_1 (this is expressed by condition $\ell' = \ell$), the output remains unchanged ($y' = y$), but the inputs may change in an arbitrary way (no condition about x' in this disjunct). The second disjunct in φ_1 specifies the transition to simple state ℓ_2: if $x > 0$ and x is even, a transition to ℓ_2 is performed ($\ell' = 2$) and y is set to zero ($y' = 0$). The input remains unchanged during this transition ($x' = x$). The third conjunct specifies the condition and effect of the transition from ℓ_1 to ℓ_3.

2.2.3 Equivalence classes

We use the term *trace* to denote finite sequences of states, input vectors, or output vectors. Applying a trace $\iota = \mathbf{c}_1 \ldots \mathbf{c}_k$ of input vectors $\mathbf{c}_i \in D_I$ to an RIOSTS $\mathcal{S} = (S, s_0, R, V, D)$ residing in some quiescent state $s \in S$ stimulates a sequence of state transitions, each pair of consecutive states connected by the transition relation R, and with associated output changes as triggered by these inputs. Restricting this sequence to quiescent states, this results in a trace of states $\tau = s_1.s_2 \ldots s_k$ such that $s_i(\mathbf{x}) = \mathbf{c}_i, i = 1, \ldots, k$, and $s_i(\mathbf{y})$ is the last RIOSTS output resulting from application of $\mathbf{c}_1 \ldots \mathbf{c}_i$ to state s.[1] This trace τ is denoted by s/ι. The restriction of s/ι to output variables is denoted by the trace $(s/\iota)|_O$. Since transient states have unique quiescent post-states, $(s/\iota)|_O$ is a uniquely determined output trace. Two quiescent states s, s' are *I/O-equivalent*, written $s \sim s'$, if every non-empty input trace ι, when applied to s and s', results in the same outputs, that is, $(s/\iota)|_O = (s'/\iota)|_O$. Two RIOSTSs $\mathcal{S}, \mathcal{S}'$ with the same input domain are I/O-equivalent, if their initial states are I/O-equivalent.

[1] Observe that the restriction to quiescent states does not result in a loss of information. Every transient state has the internal and output variable valuations coinciding with its quiescent pre-state, and its input valuation is identical to that of its quiescent post-state.

Since I/O-equivalence $\sim$ is an equivalence relation on quiescent states, we can factorise S_Q with respect to $\sim$. The *initial input equivalence class partitioning (IECP)* $\mathcal{I}_0 \subseteq \mathbb{P}(D_I)$ associated with $S_Q/\sim$ is the coarsest partitioning of D_I such that for all $\mathbf{q} \in S_Q/\sim, X \in \mathcal{I}_0$, there exists a uniquely determined I/O-equivalence class $\delta(\mathbf{q}, X) \in S_Q/\sim$, such that

$$\forall s \in \mathbf{q}, \mathbf{c} \in X : s/\mathbf{c} \in \delta(\mathbf{q}, X), \tag{1}$$

and there exists a well-defined output $\omega(\mathbf{q}, X) \in D_O$, such that

$$\forall s \in \mathbf{q}, \mathbf{c} \in X : (s/\mathbf{c})|_O = \omega(\mathbf{q}, X). \tag{2}$$

Equations (1) and (2) capture the intuition behind equivalence classes in a formal way: selecting two states s, s' from the same *state* equivalence class $\mathbf{q}$ and two inputs $\mathbf{c}, \mathbf{c}'$ from the same *input* equivalence class X will carry us to the same target state equivalence class $\delta(\mathbf{q}, X)$, regardless of whether we calculate this class by applying $\mathbf{c}$ to s or $\mathbf{c}'$ to s'. Moreover, these transition steps will always have the same effect $\omega(\mathbf{q}, X)$ on the outputs.

It is shown in [21] that $S_Q/\sim$ is finite if the RIOSTS $\mathcal{S}$ has finite internal state domains and finite output domains, while the input domains may be infinite. Moreover, the coarsest partitioning $\mathcal{I}_0$ exists, and it is finite and uniquely determined under these prerequisites. For these RIOSTSs, properties (1) and (2) induce an abstraction to DFSMs with state space $S_Q/\sim$, input alphabet $\mathcal{I}_0$, and output alphabet D_O: (1) specifies a well-defined total transition function $\delta : S_Q/\sim \times \mathcal{I}_0 \rightarrow S_Q/\sim$, and (2) a well-defined output function $\omega : S_Q/\sim \times \mathcal{I}_0 \rightarrow D_O$. When partitioning $\mathcal{I}_0$ further to a refined IECP $\mathcal{I}$, the characteristic properties (1), (2) are preserved.

A finite sequence $X_1 \dots X_k$ with $X_i \in \mathcal{I}$ is called a *symbolic test case*: concrete test input vectors $\mathbf{c}_i$ can be selected from each X_i, and when applied to the initial state s_0, this selection induces a trace $s_1 \dots s_k$ of quiescent states, such that

$$\exists \mathbf{q}_1, \dots, \mathbf{q}_k \in S_Q/\sim : \forall i \in \{1, \dots, k\} : s_i \in \mathbf{q}_i \wedge \mathbf{q}_i = \delta(\mathbf{q}_{i-1}, X_i).$$

Here, $\mathbf{q}_0 \in S_Q/\sim$ denotes the class containing the initial state $s_0 \in S_Q$. The IECP properties imply that the *expected results* associated with this test case are then specified by the output trace $\omega(\mathbf{q}_{i-1}, X_i), i = 1, \dots, k$.

Example 3 The practical calculation of IECPs is performed according to an algorithm described in [21]. It starts with an arbitrary first-order representation $\mathcal{R}$ of the RIOSTS transition relation, as has been illustrated for the state machine from Fig. 2. Next, we transform $\mathcal{R}$ into a disjunctive form enumerated over all reachable pairs of internal state valuation and output valuation vectors, such that stability conditions and transient conditions are separated into different disjuncts. The formula $\mathcal{R}$ calculated in Example 2 already has this representation: The first disjuncts in $\varphi_1, \varphi_2, \varphi_3$ specify the stability conditions for remaining in a quiescent state for a given pair of internal variable and output valuations. The other disjuncts have pre-conditions specifying transient states. Each pre-condition in a φ-disjunct is separated into conjuncts over input variables only and conjuncts involving only internal variables or outputs. This separation is always possible, since the domains for internal variables and outputs are finite. The conjuncts in φ_i involving input variables only are denoted by g_{ij} in the sequel: each g_{ij} specifies the input condition for transiting from I/O-equivalence class or sub-class i to (sub-)class j; g_{ii} specifies the stability condition for staying in i.

With the transformed transition relation predicate and the g_{ij} at hand, state equivalence sub-classes are derived, one for each stability condition. For our example, these sub-classes are A_1, A_2, A_3 which are specified as follows.

$$A_1 = \{s \in S \mid s \models (g_{11} \wedge (\ell, y) = (1, 0))\} \quad g_{11} \equiv x \leq 0$$
$$A_2 = \{s \in S \mid s \models (g_{22} \wedge (\ell, y) = (2, 0))\} \quad g_{22} \equiv x > 0 \wedge x \,\%2 = 0$$
$$A_3 = \{s \in S \mid s \models (g_{33} \wedge (\ell, y) = (3, 1))\} \quad g_{33} \equiv x \neq 0$$

We will see below that A_1, A_2, A_3 are really *sub*-classes of $S_Q/\sim$. Recall that for a formula ψ with free variables in V, $s \models \psi$ means that ψ evaluates to true when replacing v occurring free in ψ by their values $s(v)$.

For this example, the input conditions associated with transient states are given by

$$g_{12} \equiv g_{22} \equiv x > 0 \wedge x \,\%2 = 0 \quad g_{13} \equiv g_{23} \equiv x > 0 \wedge x \,\%2 = 1$$
$$g_{21} \equiv g_{11} \equiv x \leq 0 \quad g_{31} \equiv x = 0$$

Next, the IECPs are calculated by identifying all feasible conjunctions of input conditions associated with quiescent or transient state classes, one conjunct for each source state sub-class. The feasible conjunctions in this example are

$$g_{11} \wedge g_{21} \wedge g_{31} \equiv x = 0 \qquad g_{11} \wedge g_{21} \wedge g_{33} \equiv x < 0$$
$$g_{12} \wedge g_{22} \wedge g_{33} \equiv x > 0 \wedge x \,\%2 = 0 \qquad g_{13} \wedge g_{23} \wedge g_{33} \equiv x > 0 \wedge x \,\%2 = 1$$

This results in an input equivalence class partition $\mathcal{I} = \{X_1, X_2, X_3, X_4\}$ with

$$X_1 = \{c \in D_x \mid c = 0\} \qquad X_2 = \{c \in D_x \mid c < 0\}$$
$$X_3 = \{c \in D_x \mid c > 0 \wedge c \,\%2 = 0\} \qquad X_4 = \{c \in D_x \mid c > 0 \wedge c \,\%2 = 1\}$$

To illustrate the intuition behind these IECP definitions, consider class X_3 induced by predicate $g_{12} \wedge g_{22} \wedge g_{33}$. The nature of the conjuncts involved guarantees that

– when in state class A_1 and choosing an input from X_3, a transition into a state from class A_2 will be performed,

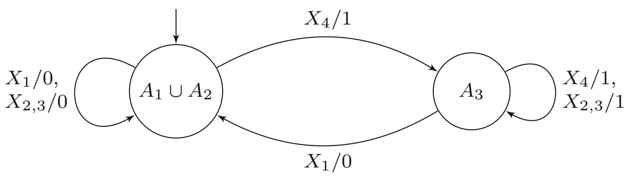


Fig. 3 Minimised DFSM resulting from abstraction of the SysML state machine from Fig. 2

– when in state class A_2 and choosing an input from X_3, the system remains in class A_2, and
– when in state class A_3 and choosing an input from X_3, the system remains in class A_3.

It is proven in [21] that the IECPs constructed as illustrated above imply the validity of a *uniformity hypothesis* as introduced in [18]: given a sequence of IECPs $Z_1 \dots Z_k \in \mathcal{I}^*$, it suffices to check the correctness of the SUT behaviour for a single input trace $\mathbf{c}_1 \dots \mathbf{c}_k$ with representatives $\mathbf{c}_i \in Z_i$, and this already implies the correct behaviour for *all* input traces $\mathbf{c}_1^{prime} \dots \mathbf{c}_k^{prime}$ with $\mathbf{c}_i^{prime} \in Z_i$. This holds provided that the true implementation behaviour is represented by a model contained in the fault domain specified in Sect. 2.2.4.

By minimising the state machine induced by A_1, A_2, A_3 and the classes X_i as input alphabet, it turns out that the resulting machine only has two states $A_1 \cup A_2$ and A_3, because the states in A_1 and A_2 are I/O-equivalent, and therefore, $S_Q/_\sim = \{A_1 \cup A_2, A_3\}$. Next, the input classes X_2, X_3 can be aggregated to a single input class $X_{2,3} = X_2 \cup X_3$, because in every state of the DFSM, X_2 and X_3 produce the same output and transit into the same target state. As a result, $\mathcal{I}_0 = \{X_1, X_{2,3}, X_4\}$ is the coarsest input partitioning fulfilling the characteristic properties (1) and (2) for the SysML state machine from Fig. 2. The resulting DFSM is shown in Fig. 3.

2.2.4 Fault models

The concept of fault models has originally been introduced in [45], specialised on the domain of finite state machines. For the semantic domain of RIOSTSs, the fault models $\mathcal{F} = (\mathcal{S}, \sim, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$ are specified as follows (more details are described in [21,22], where the foundations of the testing approach evaluated in this article are elaborated). For each $\mathcal{F}$, the reference model $\mathcal{S}$ is an RIOSTS expressing the expected input/output behaviour of the SUT (we use I/O-equivalence as conformance relation $\sim$).

Assume that the number of states in the minimised DFSM constructed for $\mathcal{S}$ as illustrated in Fig. 3 is n. Then the fault domain $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$ is defined for natural numbers $m \geq n$ and IECPs $\mathcal{I}$ refining the initial coarsest IECP $\mathcal{I}_0$ associated with $\mathcal{S}$.

$\mathcal{D}(\mathcal{S}, m, \mathcal{I})$ contains all RIOSTSs $\mathcal{S}'$ which may or may not conform to $\mathcal{S}$, but fulfil the following conditions.

1. The states of $\mathcal{S}'$ are defined over the same I/O variable space $I \cup O$ as defined for the model $\mathcal{S}$, and with the same domain D.
2. Initial state s_0' of $\mathcal{S}'$ coincides with initial state s_0 of $\mathcal{S}$ on $I \cup O$.
3. $\mathcal{S}'$ generates only finitely many different output values.
4. $\mathcal{S}'$ has a well-defined reset operation allowing to re-start the system from its initial state.
5. $\mathcal{I}$ is also an IECP for $\mathcal{S}'$.
6. The number of states in the minimised DFSM associated with $\mathcal{S}'$ and with input alphabet $\mathcal{I}$ is less or equal to m.

The number of RIOSTSs contained in $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$ is increased by increasing m or by refining $\mathcal{I}$. The complete finite test suites introduced below guarantee to uncover every faulty SUT, as long as its true input/output behaviour is reflected by some member $\mathcal{S}'$ of the fault domain.

It should be emphasised that RIOSTSs are *semantic* representations of systems expressed by concrete formalisms, such as the SysML state machines we use to express expected SUT behaviour or the SystemC programs expressing implementation behaviour. The concrete formalisms and the RIOSTS representations use the same input/output variables; therefore they can be compared with respect to I/O-equivalence. As a consequence, statements like "the SysML reference model has the input/output behaviour of RIOSTS $\mathcal{S}$" or "the observable SUT behaviour is that of the RIOSTS $\mathcal{S}'$ which is a member of the fault domain" are well-defined.

2.2.5 Complete finite test suites

Consider again the minimised DFSM abstraction $\mathcal{M}$ of $\mathcal{S}$ with states $S_Q/_\sim$, input alphabet $\mathcal{I}$, transition function and output function as characterised in (1), (2). $\mathcal{M}$ has $n = |S_Q/_\sim|$ states. This DFSM allows for application of finite complete DFSM testing strategies, such as the *W-Method* introduced in [11,50]. The general form of a W-Method test suite is

$$\mathcal{W} = P.\left(\bigcup_{i=0}^{m-n} \mathcal{I}^i.W\right) \tag{3}$$

where P is the state transition cover, $\mathcal{I}^i$ denotes the input trace segments of length i, and W is the characterisation set. Parameter n is the number of states of the minimised reference DFSM, and every DFSM in the fault domain is equivalent to a minimised DFSM with at most $m \geq n$ states. Every test of $\mathcal{W}$ consists of a (possibly empty) input trace from P, concatenated with an arbitrary input trace of length zero up to $m - n$, and terminated by an input trace from the characterisation set. P is the union of a *state cover* C and a *transition cover* $C.\mathcal{I}$: C contains the empty trace ε, and for any state $\mathbf{q}$ of $\mathcal{M}$, there exists an input trace in C which, when applied to the initial state, ends at $\mathbf{q}$. The transition cover is defined by $C.\mathcal{I} = \{\iota.X \mid \iota \in C, X \in \mathcal{I}\}$. Summarising, the input sequences of a state transition cover ensure that (1) every state of the reference DFSM $\mathcal{M}$ associated with the reference model $\mathcal{S}$ is visited, and (2) every transition from every state is exercised. A characterisation set is a set of input traces distinguishing each pair of states in a minimal DFSM. Using minimisation algorithms such as the one specified in [19], characterisation sets can be constructed as a by-product of the minimisation process.

The test suite generated according to (3) is called a *symbolic test suite*, because its elements are symbolic test cases as defined above: the inputs to be used in each test case are not yet represented by concrete input vectors $\mathbf{c}$, but by input equivalence classes $X \in \mathcal{I}$. For creating an executable test suite, inputs $\mathbf{c} \in X$ have to be selected for every $X \in \mathcal{I}$.

The W-Method is complete for the fault domain of all DFSMs over the same input and output alphabet, whose minimised equivalents have at most m states. It is shown in [21] that the associated test suites with concrete inputs $\mathbf{c} \in X$ are also complete for $\mathcal{F} = (\mathcal{S}, \sim, \mathcal{D}(\mathcal{S}, m, \mathcal{I}))$. This establishes the validity of a *regularity hypothesis* [18] for all SUTs whose behaviour is captured by the fault domain as explained in the previous section: it suffices to check correct SUT behaviour for the finite input sequences contained in the test suites obtained from application of the W-Method. This already implies the correct behaviour of the SUT an all infinite extensions of these finite traces.

This completeness result is independent on the choice of concrete input data selected from each input equivalence class $X \in \mathcal{I}$.

The fault domain $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$ introduced above can be extended by increasing m or by refining $\mathcal{I}$. Increasing m increases the maximal length of input sequences in test cases in a linear way. This affects the size of the test suite exponentially, but allows for fault domain members with higher *recurrence diameters* r [6]: this is the length of the longest loop-free path in a Kripke structure. Erroneous SUT behaviour that only occurs at the end of such a longest loop-free path may only be detected if the test cases use input sequences that are long enough to traverse the SUT state up to the length of the recurrence diameter.

Refining $\mathcal{I}$ increases the size of the IECP, and this size increases the number of test cases in a polynomial way. It has to be noted, however, that uniformly refining all members of $\mathcal{I}$—for example, by using a sub-paving strategy as it is well known from interval analysis [25]—increases the size of the IECP exponentially with each new refinement step. The resulting fault domain contains members $\mathcal{S}'$ possessing narrower *trapdoors*: these are refined input guard conditions $g \wedge g'$ applicable in certain $\mathcal{S}'$-states, where $\mathcal{S}'$ should behave uniformly for all inputs satisfying g. The true behaviour of $\mathcal{S}'$, however, conforms to the expected behaviour modelled by $\mathcal{S}$ only for inputs fulfilling $g \wedge \neg g'$, while erroneous behaviour is revealed for inputs satisfying $g \wedge g'$.

The complete IECP strategy described above is denoted by STRAT-1 in the experimental evaluations described in Sect. 4.

2.3 Randomisation of equivalence partition tests

As we have seen above, enlarging the fault domain $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$ via m or $\mathcal{I}$ seriously affects the size of the resulting complete test suite $\mathcal{W}$. We therefore investigate an alternative approach in this article that aims at increasing the test strength of $\mathcal{W}$ for SUTs $\mathcal{S}''$ whose true behaviours are reflected by RIOSTSs *outside* $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$. For obvious reasons it is assumed that these SUTs still fulfil the RIOSTS compatibility requirements 1–4 of the fault domain definition. This means that $\mathcal{S}''$ may have more than m I/O-equivalence classes and may need an IECP that is more fine-grained than $\mathcal{I}$, but it is still assumed that $\mathcal{S}''$ is an RIOSTS using the same I/O variables and possessing the same visible initial state and fulfilling a reset condition.

To this end, we observe that the completeness property of the test suites introduced above does not depend on the concrete values selected from each input equivalence class $X \in \mathcal{I}$ (a formal proof for this fact is given in [22]). For members $\mathcal{S}' \in \mathcal{D}(\mathcal{S}, m, \mathcal{I})$ it would suffice to fix one input vector $\mathbf{c}_X$ for every $X \in \mathcal{I}$. Alternatively, we could also choose different members at random, each time an input from some class X is required according to the symbolic test suite definition. While this alternative would not affect the suite's completeness property when applied against members of $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$, it favourably affects the test strength against RIOSTSs outside $\mathcal{D}(\mathcal{S}, m, \mathcal{I})$: the chances for uncovering trapdoors are obviously increased. This approach results in an *adaptive random testing strategy*, where the selection of input data is no longer performed uniformly over the complete input domain, but selectively for each input equivalence class $X \in \mathcal{I}$. Moreover, the random values from such an X are only applied when an X-input is required according to the symbolic test suite constructed from Equation (3).

Technically, the randomisation is implemented by running an SMT solver repeatedly to find concrete values of every input equivalence class $X \in \mathcal{I}$. The symbolic test suite constructed from Equation (3) is a sequence on input equivalence classes. According to our equivalence class construction [21], an input equivalence class $X \in \mathcal{I}$ is defined by a predicate[2]

[2] The predicate is guaranteed to have a solution, since it describes an input equivalence class, which has at least one member and thus at least one assignment that fulfils the predicate.

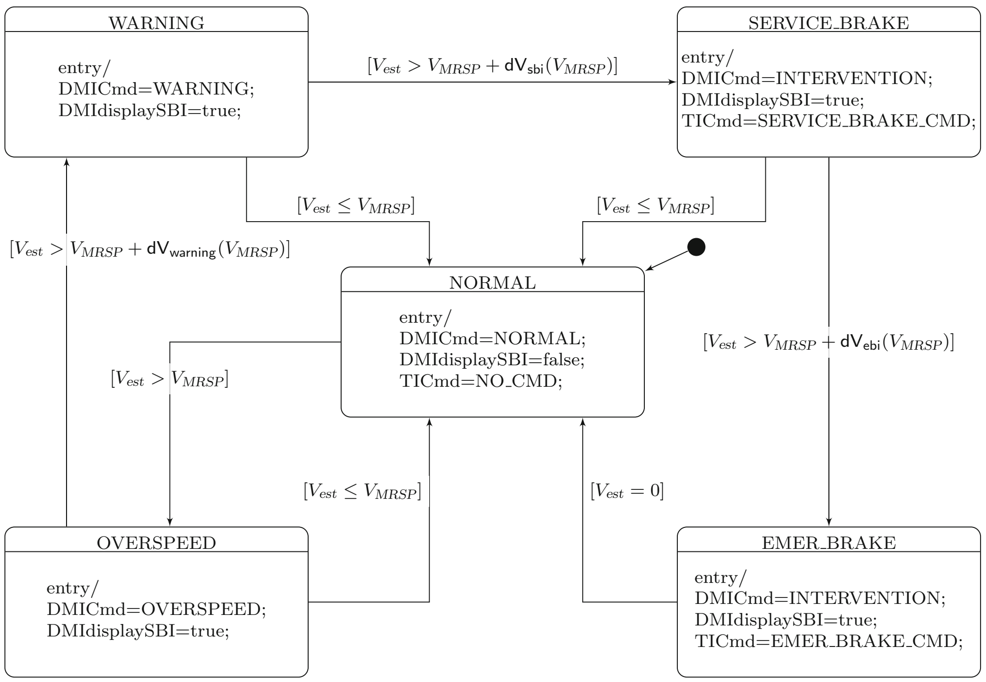


Fig. 4 State machine of the Ceiling Speed Monitor

g_X, containing solely variables in I. Using an SMT solver to solve g_X results in a concrete input vector $\mathbf{c} \in X$. Re-running the solver for the same X and prohibiting existing solutions $\mathbf{c}_1, \ldots, \mathbf{c}_{k-1}$ with a refined constraint $g_X \wedge \bigwedge_{i=1}^{k-1} \neg(\mathbf{x} = c_i)$ will result in a new solution $\mathbf{c}_k$, i.e. a new concrete input $\mathbf{c}_k \in X$ of the input equivalence class. The negation of existing solutions yields an exponential growth of the runtime of the SMT solver in the worst case. Therefore two other heuristics were implemented: (a) the internal heuristics of the SAT solver have been randomised to get a "random" solution of g_X. (b) Interval analysis can be used to find a sub-paving, that is an inner approximation of g_X. From this sub-paving, random elements can be selected using a random number generator. As another runtime optimisation, the input selection can be parallelised. Once the input equivalence class partitioning $\mathcal{I}$ is available, candidates from every input equivalence class X can be calculated separately, and in parallel, to find as many different concrete values as needed.

The randomised strategy described so far is denoted by STRAT-2 in the experimental evaluations described in Sect. 4. A more refined randomisation denoted by STRAT-3 is also applied. This strategy, when dealing with an input value selection from IEC X, decides first whether to select an internal value or a boundary value from X. This decision is made with probability 0.5 for each choice. Since the IECs are unions of convex sets, boundary values can be simply specified by transforming comparison operators in the defining predicates g_X, and by changing expressions $x < y$ to $x = y - 1$ in the case of integers, and to $x = y - ulp(y)$ in the case of floating-point numbers. Here, $ulp(y)$ denotes the *unit in the last place of* y, that is, $y - ulp(y)$ is the largest representable floating-point value which is still smaller than y. Since we use an SMT solver that is capable of handling bit-precise floating-point operations, the solver can again be used to calculate these boundary values.

3 Reference models

We use two test models as the basis for the experimental evaluation of the IECP strategy discussed in this paper, one from the railway domain and the other from the automotive domain. Their functional properties are described in this section. Both models were used in previous experiments [23].

3.1 Ceiling Speed Monitor

The main on-board controller of trains that are part of the *European Train Control System (ETCS)* executes a variety of automated train protection functions. One of these functional modules is the *Ceiling Speed Monitor (CSM)*, whose core behaviour is specified by the SysML state machine shown in Fig. 4. This state machine has been modelled from the ETCS standard [49]. The CSM inputs the current estimated train speed V_{est} and the current admissible maximal speed V_{MRSP} and reacts to overspeeding situations. The reactions are visible on the driver machine interface (DMI) (outputs

DMICmd, DMIdisplaySBI), and the CSM may interact with the service and emergency brakes (output TICmd).

As soon as the train starts overspeeding ($V_{est} > V_{MRSP}$), the CSM performs a transition from NORMAL to OVERSPEED, and an overspeed indication is displayed on the DMI. If the actual speed exceeds the V_{MRSP}-dependent threshold $V_{MRSP} + \mathsf{dV}_{\mathsf{warning}}(V_{MRSP})$, the DMI indication changes to WARNING. If the higher threshold $V_{MRSP} + \mathsf{dV}_{\mathsf{sbi}}(V_{MRSP})$ is violated, the CSM automatically triggers the service brakes. When in one of the control modes OVERSPEED, WARNING, or SERVICE_BRAKE, DMI indications and braking interventions are automatically reset as soon as the speed is back in the admissible range $V_{est} \leq V_{MRSP}$. If, however, the train continues overspeeding until the highest threshold $V_{MRSP} + \mathsf{dV}_{\mathsf{ebi}}(V_{MRSP})$ is violated, the CSM triggers the emergency brakes. From the associated state EMER_BRAKE, the transition to NORMAL is only performed when the train has come to a standstill ($V_{est} = 0$).

While internal state and output domains of the CSM are finite, the inputs V_{est}, V_{MRSP} represent speed values ranging from zero to the maximal train speed. This domain is too large to enumerate all possible value combinations during test campaigns. Therefore, an IECP strategy has to be applied. A more detailed description of the CSM model can be found in [7,8].

3.2 Airbag controller

The second test model describes an airbag controller. This system has two analogue inputs s1$\in$ [0, 10] and s2$\in$ [0, 10]. These inputs represent acceleration sensors that are used by the controller to detect a crash situation and decide whether the airbag shall be fired or not (output fire). While the airbag may ensure passenger safety in crash situations, its accidental activation is harmful in situations, when no crash is present but indicated by erroneous sensor data. Therefore, certain safety mechanisms have to be applied to guarantee (up to a certain degree of confidence) that the airbag is only fired, if a real crash situation is present. Additionally, defect sensors should be recognised and notified (output defect). The state machine in Fig. 5 models the functionality ensuring the safe operation of the airbag controller.

The system reads the sensor values s1 and s2 cyclically on every rising and falling edge of input t$\in$ [0, 1]. Both sensor values are checked for plausibility. The sensor values are considered plausible, if the value of sensor one (s1) does not exceed or drop below the value of sensor two (s2) by more than 5%, i.e. $\mathsf{s1} \in [0.95 \cdot \mathsf{s2}, 1.05 \cdot \mathsf{s2}]$. If the sensor values are plausible and an acceleration greater than 3 is measured in 3 consecutive cycles, the airbag is fired. This is done by setting output variable fire to 1. If instead the sensor values are implausible, internal variable error_ctr is incremented. This variable holds the number of implausible measurements, and if it reaches a value equal to 3, the output variable defect is set to 1, causing a shutdown of the complete airbag system and activating the service lamp to indicate a sensor defect of the airbag. After at least 3 consecutive cycles with plausible sensor values, the internal variable error_ctr is reset.

4 Experimental results

4.1 Experimental setup

For deriving test cases and concrete test data from the test models described in Sect. 3, the following strategies described in detail in Sect. 2 were applied.

STRAT-RND	Conventional random testing. A sequence of input vectors is generated at random. A STRAT-RND test suite is generated in such a way, that it contains the same number of test cases (i.e. input sequences) as the STRAT-1, -2, -3 test suite it is compared to. Moreover, the number of test cases of length $k, k = 1, 2, 3, \dots$ is always the same in the STRAT-RND suite as in the related STRAT-1,-2,-3 test suites.
STRAT-1	The original complete IECP strategy from [21] which uses fixed representatives from each input equivalence class (IEC) for testing.
STRAT-2	An extension of IECP strategy STRAT-1 by random selections from each IEC, whenever the class is referenced in a test case.
STRAT-3	A refinement of STRAT-2 by equally partitioned random selections from each IEC, so that 50% of the selections come from the IEC's interior, and 50% from the boundary of the IEC.

Strategies STRAT-RND, STRAT-1, and STRAT-2 have already been applied in the previous work [23], while STRAT-3 was first investigated in [40] for an example from the field of railway interlocking systems.

For the experimental evaluation, correct SystemC implementations were generated from each model. The implementation was performed by hand in a straightforward way. Next, mutants were automatically generated from each implementation using a SystemC mutation tool as described below. Note that usually mutation tools are unaware of any conformance relation. Therefore, the generated mutants have been manually investigated and all I/O-equivalent mutants were discarded for the experiments. Afterwards, the derived test cases were executed against the remaining mutants in order to measure the mutation score of the test suite. The mutation score is the ratio of mutants that were "killed" by a test

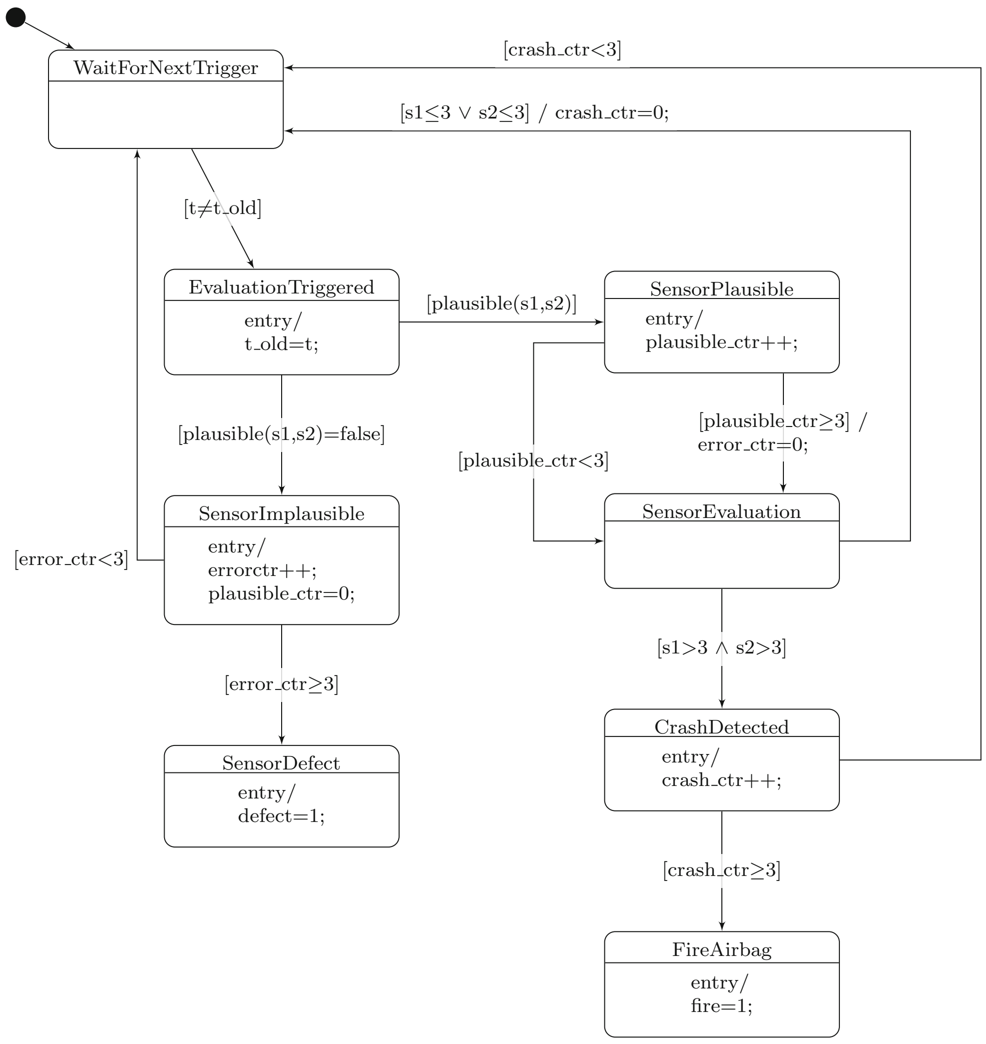


Fig. 5 State machine of the airbag controller

suite[3], to the total number of non-I/O-equivalent mutants. The mutation score is used as an indicator of each test suite's strength.

Since the strategies STRAT-2, -3 and STRAT-RND depend on the utilisation of random values for the concrete value selection, each of their test suites has been generated and executed against the mutants using 10 different random seeds. The experimental results show the mean number of killed mutants together with the standard deviation.

4.2 SystemC implementations

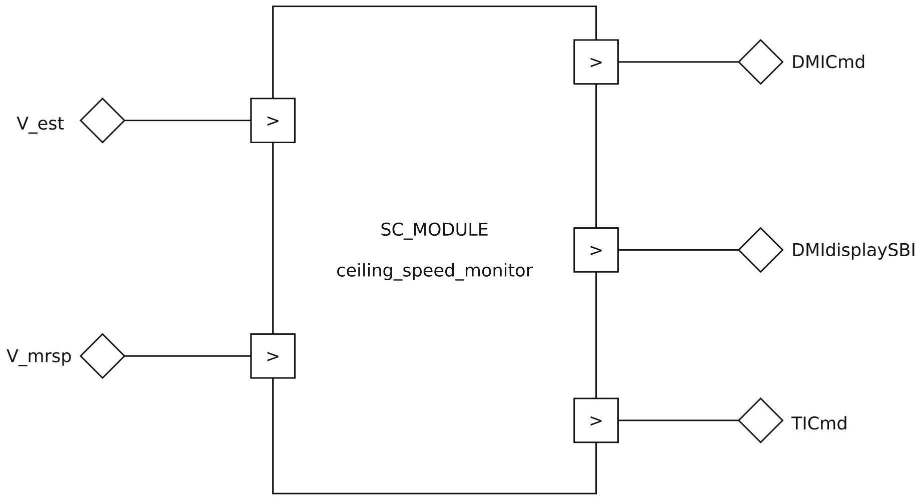
SystemC is a high-level system modelling and design language. It is integrated into C++ by means of class libraries. SystemC allows to design a system by defining modules. A module is a unit that implements some functionality. A module is connected to its environment by ports (in and out ports). A port itself can be connected to signals and to other ports via signals (also called channels). A module can have a method that implements its functionality. This method usually is called when the modules inputs change. To this end, the sensitivity list of a module defines signals, and every value change of such a signal in turn triggers the invocation of the module's method.

The SystemC implementations of both case studies as well as the SysML models are published at `mbt-benchmarks.org`.

Ceiling Speed Monitor SystemC implementation Figure 6 gives an overview of the SystemC implementation of the Ceiling Speed Monitor. The system has two input and three output ports, respectively. When running the test cases, these ports are connected to the signals as shown in Fig. 6. Signals `V_est` and `V_mrsp` are written by the test environment and the signals `DMICmd`, `DMIdisplaySBI` and `TICmd`

[3] A mutant is killed, if at least one test case of the test suite did not pass.

Fig. 6 Overview of SystemC implementation for the Ceiling Speed Monitor



are then compared to the expected results by the test oracle. In SystemC, the main module can be specified as follows:

```
SC_MODULE(ceiling_speed_monitor) {
   //inputs
   sc_in<float> V_est;
   sc_in<float> V_mrsp;
   //outputs
   sc_out<int> DMICmd;
   sc_out<bool> DMIdisplaySBI;
   sc_out<int> TICmd;
   //internals
   sc_signal<int> location;

   output_ctrl output_controller;

   SC_CTOR(ceiling_speed_monitor) {
      SC_METHOD(update);
      sensitive<<V_est;
      sensitive<<V_mrsp;
   }

   void update() { ... }
};
```

The Ceiling Speed Monitor is implemented by one SystemC module whose implementation file consists of approximately 160 lines of code. The implementation process was performed by hand by one of the authors and took approximately four hours. This includes several iterations of implementation, test execution, and debugging to get a correct implementation. The automatically generated test bed that executes the generated tests of one test strategy (stimulates the inputs of the SUT and checks for the expected outputs) consists of nearly 2500 lines of C++ code.

Airbag Controller SystemC implementation For the implementation of the airbag controller, different functionalities were implemented by sub-modules. Figure 7 gives an overview. The first sub-module `plausibility_check` checks for plausibility of the sensor values. The result of this check is written to signal `plausible`, and the number of successive cycles with plausible sensor values is written to signal `plausible_ctr`. Sub-modules `error_detection` and `crash_detection` update the values of signals `error_ctr` and `crash_ctr`, respectively. If a certain threshold is exceeded by these counters, an error is indicated by sub-module `error_indication`, or the airbag is fired by sub-module `crash_indication`. The signals in this implementation are named according to the variables used in the state machine shown in Fig. 5.

The implementation of the airbag controller consists of one SystemC module with five sub-modules. The implementation file has approximately 180 lines of code. The implementation process took approximately 6 hours. Again, several iterations of implementation, test execution, and debugging were needed to get a correct version. The utilisation of equivalence class tests was beneficial to find bugs that had been overseen in the first step. The automatically generated test bed for the execution of one test strategy contains more the 44 thousand lines of code.

4.3 Generation of SystemC mutants

The generation of SystemC mutants is performed in a systematic and automated manner. To this end, a tool based on the clang LibTooling library[4] has been implemented. This tool is a back-end to the well-known clang compiler. The clang compiler is used to parse C/C++ code and generates an abstract syntax tree (AST). Based on this AST, mutation operators are implemented: syntactic patterns are matched and replaced to generate single-fault mutations. The next paragraphs list and group all mutation operators that were used for our experiments. Recall from the discussion in Sect. 1, that the objective of these mutations is to introduce functional errors, but not

[4] see http://clang.llvm.org/docs/LibTooling.html.

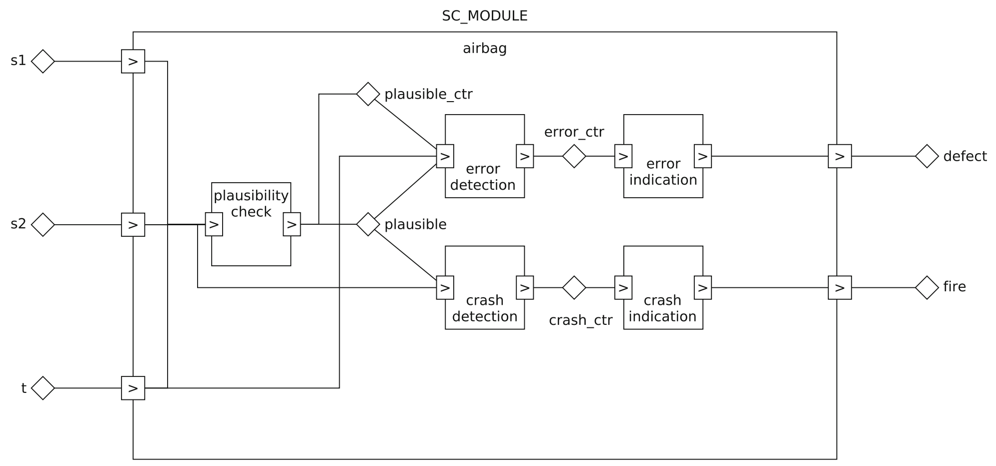


Fig. 7 Overview of SystemC implementation for the airbag controller

runtime errors, since it is assumed that the latter have already been detected and removed at an earlier stage.

Ordinary mutation operators: A group of traditional mutation operators has been implemented. The mutation operators are commonly used in other mutation tools [26,33]. These mutation operators include the replacement of logical, arithmetical, and relational operators as well as the replacement of conditions in while and if statements with constants true and false.[5] In the experiments performed in [23], theses kind of mutation operators were used for a Java implementation. In this work, we expect the results of these mutation operators to be similar to the results presented in [23,40].

Since the focus of this work is to demonstrate the applicability of our test method to HW/SW integration tests, we augmented the ordinary mutation operators with mutations that we would expect in typical HW/SW integration scenarios.

Byte order mutation A potential error in distributed systems communicating over network protocols results from the wrong interpretation of data. Typically data is sent in big endian (network) byte order, while the data is then interpreted in little endian (host) byte order on the local machine. To capture potential flaws resulting from this mismatch, we introduce a mutation operator that replaces every assignment of integer variables with an assignment using a right-hand side expression with an inverted byte order.

```
int32_t x;//little endian, width
   32 bit
x=13;
//assignment will be replaced by
x=htobe32(13);
// htobe32(n) transforms n in host
    byte order
// to the same integral number
   represented
// in 32-bit big endian byte order
```

Precision mutation Every floating-point operation is performed with a fixed precision. A mismatch in precision between the specification and the actual implementation might result in unexpected behaviour.[6] To mimic errors resulting from wrong precisions, we use a mutation operator that replaces constant floating-point literals with another constant floating-point literal of different precision. In C and C++, this subtle error can easily been introduced.

Example 4 For example, the following expression used in a condition of an if-statement is a candidate for the proposed mutation operator.

```
float x,y;
...
if(x*0.1f<y) { //literal will be
  replaced: if(x*0.1<y) {
   ...
}
```

The above example demonstrates that a subtle change in the floating-point literal can introduce non-conforming behaviour. In C/C++, the operations are automatically performed with the least required precision. Therefore, the expression `x*0.1f<y` is evaluated with single precision (float type), while the expression `x*0.1<y` is evaluated with

[5] Besides these, "traditional" mutation operators [33] also support object-oriented (OO) mutation operators. These operators have been neglected in this work, since these operators account for typical OO errors, which are not possible in the SystemC implementations of our two case studies. In our implementations, inheritance is only used to declare modules, ports, and signals. The application of OO mutation operators would not account for typical HW/SW integration errors but only for SystemC specific errors.

[6] We consider a mismatch in precision a real error, while in real-world examples a mismatch only resulting from different precisions might be negligible.

double precision (double type). Because there is no exact binary representation of the real number 0.1, the evaluation of both conditions will differ for some values of x and y.

This type of mutation might as well be applied to integer variables. This can be done by replacing fixed-width integer data types by data types with insufficient width. This would finally lead to integer overflows. However, since no complex integer calculations are performed inside our two test models, this type of mutation has not been applied in our experiments.

Sensitivity mutation Another source of errors is related to concurrency and timing. In SystemC, modules have a sensitivity list, i.e. a list of signals. Whenever the value of a signal in the module's sensitivity list changes, the method of the module is executed. An easy way to introduce timing and concurrency errors is to manipulate the sensitivity list of modules. Therefore, a simple mutation operator removes signals from a module's sensitivity list. This results in update methods to be called in wrong order or not being called at all.

Example 5 Consider, for example, the SystemC implementation of the airbag controller. In this case, the following mutation would prevent the firing of the airbag, because the update method of the crash_indication sub-module would not be called.

```
SC_MODULE(crash_indication) {
   sc_in<int> crash_ctr;
   sc_in<int> defect;
   sc_out<int> fire;

   SC_CTOR(crash_indication) {
      SC_METHOD(update);
      sensitive<<crash_ctr;
      //the previous line will be
        removed by the
      //sensitivity mutation operator
   }

   void update() { ... }

};
```

Stuck-at fault mutation Typically, hardware-related faults are modelled using fault models on bit level. Stuck-at faults model errors, where one signal is constantly stuck at the logical value one or zero. These faults can be caused by hardware defects introduced in the manufacturing process but also by design errors. These design errors are in the scope of HW/SW integration tests. Stuck-at faults are assumed to be permanent, whereas transient fault models introduce faults that happen at one point in time. It is assumed that these faults only appear temporarily and disappear in a consecutive clock cycle. Transient faults usually model faults that are triggered by external events (e.g. radiation). The focus of this fault model is mainly to evaluate the fault tolerance of a system. Fault tolerance is out of the scope of this work, and therefore, we only consider permanent errors using the stuck-at fault model.[7]

The stuck-at mutation is implemented by overriding the *read* method of SystemC signals. For each stuck-at mutation, a single bit of a single signal is constantly overridden with 1 or 0 whenever the signal's value is read.

Switch-ports mutation The last mutation operator we consider is an operator that interchanges the connection of signals to ports. In SystemC, a module can have ports for modelling the modules inputs and outputs. Finally, signals are connected to the ports, supplying the value that can be read or written by a module through its port.

The switch-ports mutation swaps the connection of two signals to ports of the same type and thus effectively swaps the variables that a module reads from or writes to. This type of fault mimics a possibly wrong configuration of the integrated system, where a module reads from a wrong memory location. The mutation operator captures a wrong wiring of input ports as well as two modules that communicate with inconsistent network protocol definitions.

Example 6 As an example, for the Ceiling Speed Monitor the two system inputs V_est and V_mrsp will be switched by the mutation operator leading to wrong control decisions in the mutated system under test.

```
//the system under test
ceiling_speed_monitor sut("ceiling_
   speed_monitor");

//signals for the system's inputs
fi_signal<float> V_est;
fi_signal<float> V_mrsp;

//signals for the system's outputs
fi_signal<signed int> DMICmd;
fi_signal<bool> DMIdisplaySBI;
fi_signal<signed int> TICmd;

int sc_main(int argc, char* argv[]) {

   //wrongly connected input signals
   sut.V_est(V_mrsp);
   sut.V_mrsp(V_est);

   //connect output signals to output
     ports
   sut.DMICmd(DMICmd);
   sut.DMIdisplaySBI(DMIdisplaySBI);
   sut.TICmd(TICmd);
   //run the test suite
   int result = Catch::Session().run
   (argc,argv);
   return result;
}
```

[7] We assume that the fault tolerance of the system under test is evaluated and validated using other means than our model-based testing approach.

Table 1 Results for the Ceiling Speed Monitor (SystemC mutations)

Mutation type	Strategy	Mutation score	
		IECP-tests (%)	STRAT-RND (%)
ByteOrder	STRAT-1	10.0 (0.0)/10 = 100.0	7.1 (2.1)/10 = 71.0
	STRAT-2	10.0 (0.0)/10 = 100.0	7.1 (2.1)/10 = 71.0
	STRAT-3	10.0 (0.0)/10 = 100.0	7.1 (2.1)/10 = 71.0
Ordinary	STRAT-1	69.0 (0.0)/91 = 75.8	48.1 (7.7)/91 = 52.9
	STRAT-2	83.2 (1.3)/91 = 91.4	48.1 (7.7)/91 = 52.9
	STRAT-3	86.7 (0.8)/91 = 95.3	48.1 (7.7)/91 = 52.9
Precision	STRAT-1	0.0 (0.0)/5 = 0.0	0.0 (0.0)/5 = 0.0
	STRAT-2	0.0 (0.0)/5 = 0.0	0.0 (0.0)/5 = 0.0
	STRAT-3	0.0 (0.0)/5 = 0.0	0.0 (0.0)/5 = 0.0
Sensitivity	STRAT-1	1.0 (0.0)/3 = 33.3	1.0 (0.0)/3 = 33.3
	STRAT-2	1.0 (0.0)/3 = 33.3	1.0 (0.0)/3 = 33.3
	STRAT-3	1.0 (0.0)/3 = 33.3	1.0 (0.0)/3 = 33.3
Stuckat	STRAT-1	46.0 (0.0)/50 = 92.0	30.8 (1.5)/50 = 61.6
	STRAT-2	46.1 (1.3)/50 = 92.2	30.8 (1.5)/50 = 61.6
	STRAT-3	46.7 (1.3)/50 = 93.4	30.8 (1.5)/50 = 61.6
Switchports	STRAT-1	2.0 (0.0)/2 = 100.0	2.0 (0.0)/2 = 100.0
	STRAT-2	2.0 (0.0)/2 = 100.0	2.0 (0.0)/2 = 100.0
	STRAT-3	2.0 (0.0)/2 = 100.0	2.0 (0.0)/2 = 100.0
All	STRAT-1	128.0 (0.0)/161 = 79.5	89.0 (10.4)/161 = 55.3
	STRAT-2	142.3 (1.8)/161 = 88.4	89.0 (10.4)/161 = 55.3
	STRAT-3	146.4 (1.4)/161 = 90.9	89.0 (10.4)/161 = 55.3

For each strategy, 93 test cases have been generated

4.4 Experimental results

Ceiling speed monitor For the Ceiling Speed Monitor model, 93 test cases were generated using the test strategies STRAT-1, -2, -3.

Mutants of the implementation were generated automatically as described above. The process of determining the number of "real" (i.e. non-I/O-equivalent) mutants took less than half an hour. This process had been accelerated by first running all test strategies against the mutants. Thirty-eight mutants, which were not killed by any of the test strategies, remained after this first test run. Only these mutants had to be considered for manual investigation. For the manual investigation, each remaining mutant was compared with the original implementation and the user (one of the authors) checked whether the mutant was I/O-equivalent or not. In most cases, this decision was trivial. For example, many similar mutations were I/O-equivalent because the range of valid input/output values restricts the number of possible stuck-at mutations. A stuck-at zero mutation of the most significant bit of a variable that never gets a value that is high enough for this bit to be set is trivially I/O-equivalent. As a result, 29 I/O-equivalent mutants were discarded from the experiments, resulting in an overall number of 161 non-I/O-equivalent mutants for the Ceiling Speed Monitor implementation.

Table 1 shows the mutation score for all test strategies, compared to a random test suite of the same size. The mutation score is shown in column three and four, in the form $k(\sigma)/t = p\%$ where k indicates the mean number of mutants that were killed by applying the test strategy 10 times with different random seeds. t indicates the total number of non-I/O-equivalent mutants and p indicates the ratio of k/t in per cent. σ denotes the standard deviation of k. The results are grouped for the different types of mutation operators and the last row shows the overall result consisting of all mutations.

It is obvious that the equivalence class testing approach outperforms random testing for most mutation operator types and never exhibits lesser test strength. From all compared strategies, STRAT-3 gets the best overall mutation score. The superiority of STRAT-3 over STRAT-2 (and STRAT-1) is mainly caused by the high mutation score for the class of ordinary mutations. In particular, the replacements of relational operators, namely the replacement of $>$ by $\geq$ and $<$ by $\leq$ as well as the reverse, can only be revealed by boundary value tests.

The results for the class of "ordinary" mutation operators are comparable to the results shown in [23] for a Java implementation of the Ceiling Speed Monitor: the strategy denoted by $(C, \mathcal{D}_2, 1)$ in [23, Table 2] corresponds to STRAT-2 in this article and achieves a mutation score of 87%. Our

Table 2 Results for the airbag controller (SystemC mutations)

Mutation type	Strategy	Mutation score	
		IECP-tests (%)	STRAT-RND (%)
ByteOrder	STRAT-1	3.0 (0.0)/3 = 100.0	2.0 (0.0)/3 = 66.7
	STRAT-2	3.0 (0.0)/3 = 100.0	2.0 (0.0)/3 = 66.7
	STRAT-3	3.0 (0.0)/3 = 100.0	2.0 (0.0)/3 = 66.7
Ordinary	STRAT-1	60.0 (0.0)/67 = 89.6	35.4 (2.9)/67 = 52.8
	STRAT-2	64.0 (0.0)/67 = 95.5	35.4 (2.9)/67 = 52.8
	STRAT-3	66.5 (0.5)/67 = 99.3	35.4 (2.9)/67 = 52.8
Precision	STRAT-1	0.0 (0.0)/2 = 0.0	0.0 (0.0)/2 = 0.0
	STRAT-2	0.0 (0.0)/2 = 0.0	0.0 (0.0)/2 = 0.0
	STRAT-3	1.0 (0.0)/2 = 50.0	0.0 (0.0)/2 = 0.0
Sensitivity	STRAT-1	10.0 (0.0)/13 = 76.9	6.8 (0.4)/13 = 52.3
	STRAT-2	10.0 (0.0)/13 = 76.9	6.8 (0.4)/13 = 52.3
	STRAT-3	10.0 (0.0)/13 = 76.9	6.8 (0.4)/13 = 52.3
Stuckat	STRAT-1	36.0 (0.0)/59 = 61.0	44.0 (1.1)/59 = 74.6
	STRAT-2	56.0 (0.0)/59 = 94.9	44.0 (1.1)/59 = 74.6
	STRAT-3	58.1 (0.5)/59 = 98.5	44.0 (1.1)/59 = 74.6
Switchports	STRAT-1	4.0 (0.0)/4 = 100.0	4.0 (0.0)/4 = 100.0
	STRAT-2	4.0 (0.0)/4 = 100.0	4.0 (0.0)/4 = 100.0
	STRAT-3	4.0 (0.0)/4 = 100.0	4.0 (0.0)/4 = 100.0
All	STRAT-1	113.0 (0.0)/148 = 76.4	92.2 (3.2)/148 = 62.3
	STRAT-2	137.0 (0.0)/148 = 92.6	92.2 (3.2)/148 = 62.3
	STRAT-3	142.6 (0.8)/148 = 96.4	92.2 (3.2)/148 = 62.3

For each strategy, 1768 test cases have been generated

STRAT-3 achieves a score of 95.3% in this category, which we consider as effective, so that a further increase in test cases (for example, by refining the input equivalence partitioning) is not necessary. It should be noted, however, that [23, Table 2] shows that 100% fault coverage can be achieved for this case study when refining the input partitioning and applying STRAT-2 with 610 test cases.

The performance of the IECP tests is very high for mutations of the stuck-at fault model. All faults that were caused by wrongly connected signals (switch-ports mutation) and byte order mutations were uncovered by IECP tests, while random tests were able to reveal all switch-ports mutations as well.

All strategies performed equally weak for precision and sensitivity mutations, though the low number of these mutations might weaken the validity of these results. Due to the CSM model structure, it is not possible to generate more non-equivalent mutants for these mutant operator types. The reason why two sensitivity mutations were not uncovered by the test strategies can be explained as follows: the sensitivity mutations removed one of the two input variables (V_{est} and V_{MRSP} respectively) from the modules sensitivity list. These mutants are non-I/O-equivalent, because the change of only one input variable will not trigger an update of the systems internal state. Nevertheless, the test strategies do not reveal these errors, because the change of input values between test steps is not restricted to one input variable. As a result, the values of all the input variables are changed simultaneously in each test step. A test strategy that only changes one input variable at a time will be superior in revealing these kinds of errors.

The precision mutants that were not killed by the test strategies have a structure similar to the following mutant:

```
float dV_sbi
  (float v) {
 if (v <= 110) {
  return 5.5f;
 } else if (v <=
   210) {
  return 0.045f
  * v
    + 0.55f;
 } else {
  return 10.0f;
 }
}
```

```
float dV_sbi
    (float v) {
 if (v <= 110) {
  return 5.5f;
 } else if (v <=
   210) {
  return 0.045 * v
    + 0.55f;
 } else {
  return 10.0f;
 }
}
```

The behaviour of the mutant (right-hand side) differs from the original implementation in the way that the original expression `0.045f * v + 0.55f` is calculated with single precision, while the expression in the mutant is calculated in double precision. The I/O behaviour is only affected in cases where the narrowed double-precision value of the overall expression differs from the original result completely calculated in single precision. Unfortunately, this condition is not necessarily fulfilled when calculating arbitrary or even boundary values. A more sophisticated approach is needed to reveal these kinds of subtle errors.

Airbag Controller The higher complexity of the airbag controller compared to the Ceiling Speed Monitor increases the required test effort. Total of 1768 test cases were generated for each of the strategies STRAT-1, -2, -3 and for the naive random testing strategy. Applying the mutation generation tool results in 148 non-I/O-equivalent mutants. Again, I/O-equivalent mutants were discarded by manual inspection. An overall number of 27 mutants had to be investigated. These 27 mutants were not killed by any of the test strategies. Twenty-three of these mutants were I/O-equivalent.

The experimental results for the airbag controller shown in Table 2 confirm the observations seen before. The IECP testing approach has a test strength that is significantly higher than the test strength of naive random testing. Again, test strategy STRAT-3 achieves the best mutation score caused by the superiority over STRAT-1, -2 for ordinary mutations but for stuck-at faults as well. The results for the airbag controller show that the mutation score for stuck-at faults highly depends on the number of different input vectors. STRAT-1 (61%) performs significantly worse than STRAT-2 (94.9%). The selection of boundary values is able to improve this mutation score to a total of (98.5%).

Compared to the Ceiling Speed Monitor, the results for the sensitivity mutation operator are remarkable. Due to a higher complexity, the airbag controller uses more ports and signals. This explains the higher number of mutations. In this case, the IECP tests were able to uncover almost all sensitivity mutations. Again, the sensitivity mutations that affect a single input variable were not uncovered because the test strategies change multiple input variables simultaneously. Apart from that, all sensitivity mutations that affect the sensitivity list of internal sub-modules of the airbag controller implementation are uncovered by the IECP strategies, because a missing sensitivity to internal signals results in sub-modules that do not update the internal state correctly or in time.

When comparing the results achieved for the "ordinary" mutant operators with [23, Table 3], the suite denoted there by $(C, \mathcal{D}_1, 1)$ compares to our STRAT-2. There a fault coverage of 96% is achieved with 368 test cases, while we use 1768 test cases to achieve 95.5% fault coverage. The lesser number of test cases applied in [23] is based on the fact that some of the 1768 test cases are equivalent when applying them to "ordinary" mutations only. Here, the full set of cases is needed, in order to cope with the mutations related to HW/SW integration.

4.5 Threats to validity

Threats caused by model selection The selection of models might have an impact on the experimental results. To reduce this threat, we used two models with opposing characteristics. The ceiling speed monitor has a very small recurrence diameter, a small number of internal states, and relatively wide input equivalence classes. The airbag controller on the other hand has many internal states, a high recurrence diameter, and narrow input equivalence classes. While both models have input classes that are unions of convex subsets of $\mathbb{R}^n$, the case study presented in [38] works with input classes which are subsets of (very large) discrete sets specified by complex predicates. The mutation score achieved there did not differ significantly from the scores presented in this article. The experiments described in [38] also deal with systems under test (railway interlocking systems) where the state space is significantly larger than the spaces of the SUTs investigated in this article. This shows that the approach scales up in an appropriate way. Therefore, we conclude that the residual threat related to model selection is very low.

Threats caused by the mutation generator Threats to validity might be caused by the mutant generator that we used in our experiments. In our previous work [23], we used three different Java mutation generation tools. The results from the μJava tool were presented. Apart from that, the PITest[8] and the Major mutation framework [26] were applied. All these tools used similar mutation operators. So far in all of our experiments, the observed impact of the choice of a mutation generation tool was quite low. Based on that, we implemented the "ordinary" mutation operators mentioned above. The results of all three Java mutation generators were comparable to the results of our SystemC mutation generator using the subset of ordinary mutation operators.

Threats caused by the application domains The threats to validity discussed above are further mitigated by experiments with SUTs from other application domains, confirming the results presented in this article. In [38], the IECP strategies STRAT-1, -2, -3 described here have been applied to testing route controllers of railway interlocking systems. The associated test models are distinguished from the models discussed in this article in a significant way: while ceiling speed monitors and airbag controllers have input equivalence classes that are—sometimes after adding boundaries—compact subsets of $\mathbb{R}^n$, the input classes of route controllers contain discrete elements representing configurations of trains and track elements in the railway network. For these classes, interior values and boundary values have to be defined in a different way (for more detailed explanations, see the technical report [39]). Despite these considerable differences, the

[8] See http://pitest.org/.

results obtained in [38] with test strategy STRAT-3 are just as good as those achieved for the case studies investigated in this article.

Dependencies on the DFSM test strategies applied The comparison of results also indicates that the test strength of STRAT-1, -2, -3 is robust with respect to the selection of complete DFSM testing strategies: while we have applied the W-Method in this article, the Wp-Method has been applied in addition to the W-Method in the evaluation [39]. The Wp-Method [17,32] usually results in fewer test cases than the W-Method, while guaranteeing the same fault coverage on state machine level. Fewer test cases, however, also imply that fewer random selections from input equivalence classes will be performed on the concrete system level, when applying strategies STRAT-2 and STRAT-3. The experiments from [38,39] indicate that this does not affect the test strength for SUT behaviours outside the fault domain. Therefore, we expect that similar results as the ones presented above will also hold if the Wp-Methods is applied for STRAT-1, -2, -3 when testing the ceiling speed monitor or the airbag controller.

Results are not applicable to runtime error detection In this work, runtime errors like stack overflow errors, memory access violations caused by invalid pointer addresses, or invalid array indexes have not been considered. These kinds of faults are hard to be uncovered by functional testing. Therefore a variety of tools exists to tackle this error type, including static analysers, abstract interpreters, and dynamic memory debugging tools [13,14]. In preliminary stages of our experiments we also tried out some types of runtime error mutations, like invalid writes, but the results confirm the well-known observation that runtime errors are hard to detect by means of functional testing [28].

5 Related work

The framework for constructing complete test suites in general, and for introducing equivalence class testing methods preserving completeness in particular, has been laid out in [18]. The novelty of our equivalence class testing approach in comparison to existing work is extensively discussed in [21,22].

Notable examples for complete test methods have been given for various formalisms (FSMs, Timed Automata, process algebras, and process algebras handling complex data structures) in [9,11,17,41,47,48,50], further references on the state of the art of automated model-based testing are given in [2,44]. Since these contributions focused on establishing the completeness aspects of their theories, none of them considered the question whether the testing strategies proposed might still be efficient for SUTs whose true behaviour does not fully comply with the underlying testing hypotheses, such as being contained the fault domain.

Adaptive random testing [10] focuses on techniques to evenly spread the test cases over the complete input domain. Most research concentrates on testing non-reactive software modules, where test cases are specified by single input vectors instead of the input sequences considered in our reactive systems setting. An example of the application of adaptive random testing and search-based testing to real-time embedded systems is given in [3].

In [20,27], the problem of test data generation in the presence of input/output variables and associated constraints resulting from guard conditions is analysed in the context of extended finite-state machines (EFSMs). EFSMs can be encoded as RIOSTSs in a straight-forward way. Our general approach to model-based test generation implemented in the RT-Tester tool differs from [20,27] in the way that test objectives are encoded as bounded model checking instances, and we use an SMT solver to calculate concrete test data [37,42]. The calculation of representatives of input equivalence classes not only works for integral data types, but for floats as well, because the SMT solver SONOLAR integrated in RT-Tester supports floating-point datatypes and operations [30].

The fact that boundary value testing yields higher fault coverage in comparison with equivalence partitioning only is not new. Evidence for this fact is given in [46]. A formal definition of boundary coverage criteria is presented in [29]. Previous work mainly focuses on the "stateless" definition and application of boundary tests. The boundary values in [29] are defined on domains that do not depend on a system state of a reactive system. Obviously, the definitions can easily be generalised for the application to reactive systems. A naive generalisation of the approach in [29] would require a very fine partitioning of the input domain. Then, this partitioning could be used as an input alphabet for an FSM abstraction of our RIOSTSs. Intentionally, we do not use such a rigorous approach that first selects boundary values—using a criterion like AB (all boundary values) or AE (all edges)—and uses these values as input alphabet for our FSM. This would result in an infeasible amount of test case generated by our approach. Instead, we use our own heuristic in STRAT-3 which neither increases the number of test cases nor compromises the completeness property of our approach. Still our heuristic yields a very high mutation score. Moreover, while no guarantees with respect to fault coverage are made in [29,46], our approach ensures that all conformance violations will be uncovered for SUTs whose true behaviour is captured by the fault domain. This holds regardless of the boundary value selection heuristic that is used in our approach.

Plenty of work exists in the area of mutation testing. Our previous experiments were all based on Java mutations

[26,33]. When evaluating test strategies for HW/SW integration tests, software-only mutations are not appropriate anymore. In this case, SystemC offers the advantage that it allows the modelling of HW and SW aspects in one formalism. However, to the best of our knowledge no automated mutation generation approach for SystemC models exists that focuses on the evaluation of HW/SW integration tests. The work presented in [34] comes close our needs. There the authors present an error and fault injection method on multiple levels. Hardware faults are injected at register transfer level (RTL). Additionally, high-level errors are injected at behavioural level by manipulating variable values directly. However, the main focus of their work is to evaluate fault tolerance mechanisms and to reduce the number of masked faults. The authors in [43] propose a similar approach using fault injection for the assessment of fault tolerance.

A variety of model-based test tools exists[9]. Many tools are based on the semantics of labelled transition systems (LTSs) and on a conformance relation called *ioco*. One example is JTorX [5]. In contrast to our approach, this tool is limited to systems with finite input domains. Other approaches are based on the semantics of EFSMs. The tool TestCast [16] uses a model checker to generate test cases. For this tool, the state space has to be finite. In our case, the state space, namely the input space, is not restricted to finite domains. Other tools use fault-based approaches to generate tests. MoMuT::UML [1] uses mutation operators on the UML test model to introduce faults to the model. Then, a test case is calculated that uncovers this mutation. While this approach guarantees that certain syntactic mutations are covered by the generated test suite, there is no formal fault domain and consequently no formal test strength guarantee comparable to the fault model we introduced in Sect. 2.2.4. It has to be noted that our fault domain is syntactically independent in contrast to approaches base on syntactic mutation operators.

The tool that comes closest to our approach is JSXM. In [15], the authors use Stream X-machines (SXMs) to model a system. An SXM is a finite-state machine that is enriched by a possibly infinite memory, and labels of the state machines are so-called processing functions that produce an output and update the memory. The JSXM tool generates test cases for these SXMs using an adjusted kind of the W-Method. The authors claim that their approach is complete under certain design-for-test conditions. The main condition for the completeness of their approach is the assumption that the processing functions themselves are correctly implemented. In our work, we do not need such a strict assumption. We assume that the IECP used for test generation is an IECP of the SUT (compare Sect. 2.2.4). This is a relaxed version of the design-for-test condition that is used in [15].

6 Conclusions

In this paper, a complete equivalence class testing strategy has been experimentally evaluated with respect to its test strength, when applied to SUTs whose behaviour is outside the fault domain for which the completeness assertion applies. The experiments show that this strategy has significantly greater strength in comparison with conventional random testing. Moreover, a randomisation of the equivalence class testing strategy has been proposed that increases the test strength even further by selecting different values from each input equivalence class, whenever a member of this class is required as input according to the original strategy. The best fault coverage is achieved if this randomisation generates values that are uniformly distributed over interior and boundary values of each input equivalence class from which concrete inputs are selected.

At the same time, it is clear that this “randomisation in the $\mathcal{I}$-dimension” does not increase the test strength, if the implementation $\mathcal{S}'$ has a larger recurrence diameter than $\mathcal{S}$, and if erroneous behaviour of $\mathcal{S}'$ is only revealed at the end of a longest loop-free path. Therefore, we suggest to add a “randomisation in the m-dimension” by attaching a random input sequence of a given fixed length at the end of each test case for in-depth exploration of the SUT behaviour. Observe that in most embedded system tests, the costs for resetting the SUT are higher than those for increasing the test suite length. Therefore, increasing the length of test cases is generally acceptable, while increasing the number of test cases is usually a costly decision. The effect of increasing the test case length is currently investigated by the authors. Note that this requires more complex mutations increasing the recurrence diameter and inserting erroneous behaviours at the end of maximal loop-free paths only.

The experimental evaluation used mutants modelled in SystemC, where both the “usual” software bugs and typical errors resulting from mismatches between software and hardware design were injected. For assessing the strength of testing strategies for embedded—potentially safety-critical—control systems, this is an important extension of the conventional mutant approach: subtle errors may occur even for completely correct software, if the underlying hardware cannot cope with performance or resource-related requirements to be fulfilled for correct execution of the integrated system.

In [22], it has been shown that the complete equivalence class testing strategy evaluated in this article can be extended to non-deterministic systems. We expect similar test strength to be achieved for non-deterministic SUTs outside the fault

[9] For an exemplary list, see http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html.

domain when applying this extended strategy. This is currently under investigation.

Acknowledgements The authors would like to thank the anonymous referees for the careful and constructive review of this article. The work presented in this paper has been elaborated within project *Implementable Testing Theory for Cyber-physical Systems (ITTCPS)* (see http://www.informatik.uni-bremen.de/agbs/projects/ittcps/index.html) which has been granted by the University of Bremen in the context of the German Universities Excellence Initiative (see http://en.wikipedia.org/wiki/German_Universities_Excellence_Initiative).

References

1. Aichernig, B., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: MoMuT::UML model-based mutation testing for UML. In: Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–8 (2015). doi:10.1109/ICST.2015.7102627
2. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
3. 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, ICTSS'10, pp. 95–110. Springer, Berlin (2010)
4. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Belinfante, A.: JTorX: A tool for on-line model-driven test derivation and execution. In: SpringerLink, pp. 266–270. Springer, Berlin (2010). doi:10.1007/978-3-642-12002-2_21
6. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Log. Methods Comput. Sci. (2006). doi:10.2168/LMCS-2(5:5)2006. arXiv:cs/0611029
7. Braunstein, C., Haxthausen, A.E., Huang, W.L., Hübner, F., Peleska, J., Schulze, U., Hong, L.V.: Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In: Merz, S., Pang, J. (eds.) Proceedings of the ICFEM 2014, No. 8829 in Lecture Notes in Computer Science, pp. 380–395. Springer, Berlin (2014)
8. Braunstein, C., Huang, W.L., Peleska, J., Schulze, U., Hübner, F., Haxthausen, A.E., Hong, L.V.: A SysML test model and test suite for the ETCS ceiling speed monitor. Technical Report, Embedded Systems Testing Benchmarks Site (2014-04-30). http://www.mbt-benchmarks.org
9. Cavalcanti, A., Gaudel, M.C.: Testing for refinement in circus. Acta Inform. **48**(2), 97–147 (2011)
10. Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.H.: Adaptive random testing: the art of test case diversity. J. Syst. Softw. **83**(1), 60–66 (2010)
11. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **SE–4**(3), 178–186 (1978)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Okada, M., Satoh, I. (eds.) Eleventh Annual Asian Computing Science Conference (ASIAN'06), pp. 1–24. Springer, Berlin, LNCS (2006) **(to appear)**
14. Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Rival, X.: Why does Astrée scale up? Form. Methods Syst. Des. (FMSD) **35**(3), 229–264 (2009)
15. Dranidis, D., Bratanis, K., Ipate, F.: JSXM: A tool for automated test generation. In: SpringerLink, pp. 352–366. Springer, Berlin (2012). doi:10.1007/978-3-642-33826-7_25
16. Ernits, J.P., Kull, A., Raiend, K., Vain, J.: Generating Tests from EFSM Models Using Guided Model Checking and Iterated Search Refinement. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification, No. 4262 in Lecture Notes in Computer Science, pp. 85–99. Springer, Berlin (2006). http://link.springer.com/chapter/10.1007/11940197_6
17. Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991). doi:10.1109/32.87284
18. Gaudel, M.C.: Testing can be formal, too. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) TAPSOFT, Lecture Notes in Computer Science, vol. 915, pp. 82–96. Springer, New York (1995)
19. Gill, A.: Introduction to the Theory of Finite-State Machines. McGraw-Hill, New York (1962)
20. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: Katoen, J.P., Stevens, P. (eds.) TACAS, Lecture Notes in Computer Science, vol. 2280, pp. 327–341. Springer, New York (2002)
21. Huang, W., Peleska, J.: Complete model-based equivalence class testing. STTT **18**(3), 265–283 (2016). doi:10.1007/s10009-014-0356-8
22. Huang, W., Peleska, J.: Complete model-based equivalence class testing for nondeterministic systems. Form. Asp. Comput. (2016). doi:10.1007/s00165-016-0402-2
23. Hübner, F., Huang, W., Peleska, J.: Experimental evaluation of a novel equivalence class partition testing strategy. In: Blanchette, J.C., Kosmatov, N. (eds.) Proceedings of the Tests and Proofs: 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22–24, 2015. Lecture Notes in Computer Science, vol. 9154, pp. 155–172. Springer (2015). doi:10.1007/978-3-319-21215-9_10
24. IEEE Std 1666–2005: IEEE standard SystemC language reference manual. IEEE Computer Society, New York, USA (2006)
25. Jaulin, L., Kieffer, M., Didrit, O., Walter, É.: Applied Interval Analysis. Springer, London (2001)
26. Just, R.: The Major mutation framework: Efficient and scalable mutation analysis for Java. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 433–436. San Jose (2014)
27. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: ICST, pp. 230–239. IEEE Computer Society (2009)
28. Kästner, D., Ferdinand, C.: Applying abstract interpretation to verify EN-50128 software safety requirements. In: Lecomte et al. [31], pp. 191–202. doi:10.1007/978-3-319-33951-1_14
29. Kosmatov, N., Legeard, B., Peureux, F., Utting, M.: Boundary coverage criteria for test generation from formal models. In: Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 139–150 (2004). doi:10.1109/ISSRE.2004.12
30. Lapschies, F.: SONOLAR homepage (2014). http://www.informatik.uni-bremen.de/agbs/florian/sonolar/
31. Lecomte, T., Pinger, R., Romanovsky, A. (eds.): Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification—First International Conference, RSSRail 2016, Paris, France, June 28–30, 2016, Proceedings, Lecture Notes in Computer Science, vol. 9707. Springer (2016). doi:10.1007/978-3-319-33951-1

32. Luo, G., von Bochmann, G., Petrenko, A.: Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. IEEE Trans. Softw. Eng. **20**(2), 149–162 (1994). doi:10.1109/32.265636
33. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: an automated class mutation system. Softw. Test. Verif. Reliab. **15**(2), 97–133 (2005). doi:10.1002/stvr.v15:2
34. Mueller-Gritschneder, D., Maier, P.R., Greim, M., Schlichtmann, U.: System C-based multi-level error injection for the evaluation of fault-tolerant systems. In: Proceedings of the 2014 International Symposium on Integrated Circuits (ISIC), pp. 460–463 (2014). doi:10.1109/ISICIR.2014.7029567
35. Object Management Group: OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. Technical Report, OMG (2011)
36. Object Management Group: OMG Systems Modeling Language (OMG SysML), Version 1.4. Technical Report, Object Management Group (2015). http://www.omg.org/spec/SysML/1.4
37. 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). doi:10.4204/EPTCS.111.1
38. Peleska, J., Huang, W., Hübner, F.: A novel approach to HW/SW integration testing of route-based interlocking system controllers. In: Lecomte et al. [31], pp. 32–49. doi:10.1007/978-3-319-33951-1_3
39. Peleska, J., Huang, W., Hübner, F.: A Novel Approach to HW/SW Integration Testing of Route-Based Interlocking System Controllers: Technical Report. Technical Report, University of Bremen (2016-03-10). Available under http://www.cs.uni-bremen.de/agbs/jp/jp_papers_e.html
40. Peleska, J., Huang, W.L., Hübner, F.: A novel approach to HW/SW integration testing of route-based interlocking system controllers. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) Reliability, Safety, and Security of Railway Systems Modelling, Analysis, Verification, and Certification, No 9707 in Lecture Notes in Computer Science, pp. 32–49. Springer, New York (2016). doi:10.1007/978-3-319-33951-1_3
41. Peleska, J., Siegel, M.: Test automation of safety-critical reactive systems. South Afr. Comput. J. **19**, 53–77 (1997)
42. 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 (2011)
43. Perez, J., Azkarate-askasua, M., Perez, A.: Codesign and simulated fault injection of safety-critical embedded systems using systemC. In: Dependable Computing Conference (EDCC), 2010 European, pp. 221–229 (2010). doi:10.1109/EDCC.2010.34
44. Petrenko, A., Simao, A., Maldonado, J.C.: Model-based testing of software and systems: recent advances and challenges. Int. J. Softw. Tools Technol. Transf. **14**(4), 383–386 (2012). doi:10.1007/s10009-012-0240-3
45. Petrenko, A., Yevtushenko, N., Bochmann, G.V.: Fault models for testing in context. In: Gotzhein, R., Bredereke, J. (eds.) Formal Description Techniques IX: Theory, Application and Tools, pp. 163–177. Chapman & Hall, Boca Raton (1996)
46. Reid, S.C.: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: Proceedings Fourth International Software Metrics Symposium, pp. 64–73 (1997). doi:10.1109/METRIC.1997.637166
47. Springintveld, J., Vaandrager, F., D'Argenio, P.: Testing timed automata. Theor. Comput. Sci. **254**(1–2), 225–257 (2001)
48. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing, Lecture Notes in Computer Science, vol. 4949, pp. 1–38. Springer, New York (2008)
49. UNISIG: ERTMS/ETCS System Requirements Specification, Chapter 3, Principles, vol. Subset-026-3, chap. 3 (2012). Issue 3.3.0
50. Vasilevskii, M.P.: Failure diagnosis of automata. Kibernetika (Transl.) **4**, 98–108 (1973)

Felix Hübner received his B. Sc. and M. Sc. degrees in computer science from the University of Bremen in 2011 and 2014, respectively. Currently he is completing his doctoral thesis in computer science at the University of Bremen, where he is member of the Research Group Operating Systems and Distributed Systems. His research focus is on model-based testing, especially on MBT applied to control systems in the railway domain.

Wen-ling Huang Since 2006, Dr. Wen-ling Huang is professor for mathematics at the University of Hamburg. Her research field focuses on various aspects of linear algebra and geometry, including differential geometry and its applications to Einstein's Theory of General Relativity. In 2012, Wen-ling Huang extended her research interests to formal methods in computer science, with special focus on model-based testing, bounded model checking, and the verification of infinite state systems. Applications of her research are in the domain of safety-critical cyber-physical systems, railway control systems, avionic systems, and automotive control.

Jan Peleska Since 1995, Dr. Peleska is professor for computer science (operating systems and distributed systems) at Bremen University in Germany. He is also co-founder of Verified Systems International GmbH, a company specialised on tools and services in the field of safety-critical system development, verification, validation, and test. His current research interests include formal methods for the development of dependable systems, test automation based on formal methods with applications to embedded real-time systems, and formal methods in combination with CASE methods. Current industrial applications of his research work focus on the development and verification of avionic software, space mission systems, and railway and automotive control systems.