Gray-Box Conformance Testing for Symbolic Reactive State Machines

Masoumeh Taromirad and Mohammad Reza Mousavi

Centre for Research on Embedded Systems (CERES)
Halmstad University, Sweden
[m.taromirad,m.r.mousavi]@hh.se

Abstract. Model-based testing (MBT) is typically a black-box testing technique. Therefore, generated test suites may leave some untested gaps in a given implementation under test (IUT). We propose an approach to use the structural and behavioural information exploited from the implementation domain to generate effective and efficient test suites. Our approach considers both specification models and implementation models, and generates an enriched test model which is used to automatically generate test suites. We show that the proposed approach is sound and exhaustive and cover both the specification and the implementation. We examine the applicability and the effectiveness of our approach by applying it to a well-known example from the railway domain.

1 Introduction

Model-based testing (MBT) has received significant attention in testing complex software systems. The benefit of model-based testing is primarily in automated test case generation and automated analysis of the test results. In an MBT process, test cases are automatically derived from a (preferably formal) model of the specification and are executed on the implementation under test (IUT). MBT is typically a black-box testing technique, in which the implementation is only accessible through its interfaces and thus, test data is generally selected based on the specification. Therefore, generated test suites may leave some untested gaps in a given IUT and/or redundantly cover the same logical path several times.

To address this issue test models and test case generation processes can be enriched with structural or behavioural information extracted from the implementation. This is a promising approach considering the existing techniques for extracting models from implementations, in particular, recent learning-based approaches inferring models from software (e.g., [1, 2]). Such models provide an *abstraction* of the implementation based on its observable behaviour. Using these models in testing improves the coverage of the IUT, up to the accuracy of the extracted model.

This paper proposes a gray-box testing strategy in that test suites are generated considering both the specification and an *abstraction* of the IUT. With such a test suite the coverage of the specification model and the implementation would be complementary to each other and hence, more faults could be uncovered. Moreover, such test suites are tailored to a given IUT and thus, a fewer number of test cases are generated –to satisfy a certain testing goal– in comparison to universal test suites that are supposed to detect faults in any possible implementation. The main contribution of this work is considering the partitioning of the input domain which can be obtained from (black-box) implementations (e.g., by model learning techniques) in generating test suites. We show that although such information may be generated for different purposes, it can be used in test generation and does improve the coverage of the generated test cases.

In this work, specifications and implementations are modelled with a specific type of transition systems, called *Symbolic Reactive State Machines (SRSMs)*. Given the SRSMs of the specification and the IUT, a *complete* test suite is generated based on the, so-called, *transition composition* of these models. In generating test cases, the justification of the proposed data selection is demonstrated by a special case of the *uniformity hypothesis* [3] –the theoretical foundation for testing with a finite subset of values.

The rest of the paper is structured as follows: Section 2 provides an overview of the related work. Section 3 introduces the formalism used in this paper and Section 4 defines our notion of conformance. The proposed testing strategy is outlined in Section 5. In Section 6, we provide the experimental results of examining the effectiveness of our approach. Section 7 discusses the future work and concludes the paper.

2 Background and Related Work

Several black-box test case generation methods are proposed in the literature for various formalisms (e.g., finite state machines [4, 5] and labeled transition systems [6]). The completeness of these methods (i.e., specifying all possible behaviour of a system) is typically explained with respect to a specified subset of possible implementations which is refered to as a *fault model* [7]. This is because in many practical cases, it is not possible to have a complete test suite as such a test suite would be infinitely large.

Gray-box model-based testing strategies provide a combination of black-box model-based testing with white-box testing to tune fault detection with respect to a given implementation. For example, in [8], the structure of the tests is generated using MBT (from the specification model) and then a white-box testing technique is used to find a set of concrete values for parameters that maximise code coverage. The approach presented in this paper, in a similar way, considers the IUT in generating test cases. However, it differs from [8] in that both the structure and the parameters of test cases are influenced by a combination of a test model and information from the implementation.

Our proposed approach has been largely established considering the promising results from existing learning-based techniques for inferring and extracting models from implementations. Some of the techniques have focused on sequential models typically in the form of FSMs (e.g., [9,10]) and some on data-dependant behaviour in the form of pre- and post-conditions (e.g., [11]). More recently, EFSMs are considered to infer more complete models (combining control and data). For example, Cassel et al. [2] introduce an active learning algorithm to infer a class of EFSMs. Walkinshaw et al. [1] provide a model inference technique (called MINT) which infers EFSMs from software executions. We believe that the model inference techniques which, in particular, infer EFSMs can provide the required abstract model of implementations in the context of our work (i.e., an inferred model can be translated into our formalism).

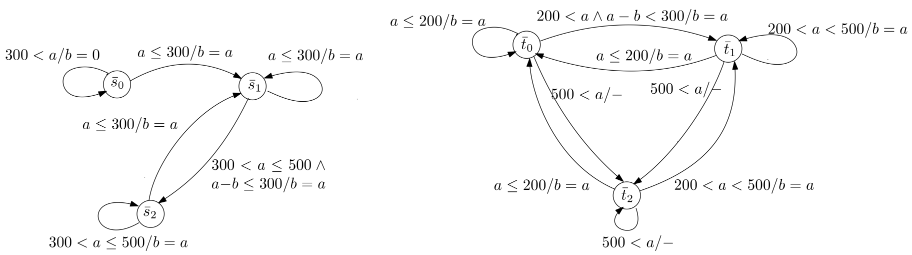
There are also a number of similar models, to our formalism, in the literature of MBT such as action machines (AM) [12], symbolic transition systems (STS) [13], FSMs with symbolic inputs [14], and symbolic input output FSMs (SIOFSM) [15]. SIOFSMs particularly support inputs with infinite domain. We expect that each of these underlying models (and their associated test case generation algorithms) can be adopted in our approach.

Another closely related line of work is equivalence-class-based testing. The theoretical foundation for this approach has been presented in [3] by the *uniformity hypothesis*, which states that it suffices to check the representatives of sub-domains in which the behaviour is the same among all elements. We discuss the justification of our strategy based on this hypothesis. Huang et al. [16] propose a complete model-based equivalence testing strategy applicable to reactive systems with large, possibly infinite input data types but finite internal and output data. Our approach is inspired by [16] and extends it by replacing the heuristics for refinement with the information extracted from the IUT. It also differs from [16] in that it allows for infinite output domains.

2.1 Motivating Example

To motivate this work, we use one of the benchmarks provided in [2], namely the prepaid card, in which the card's balance is limited to 500 SEK, and no more than 300 SEK can be topped up in a single transaction. Fig. 1a illustrates the behaviour of this card for the update balance operation. Variable a is the amount to update the balance of the card, and variable b is the current balance of the card. Labels of the form 'C/O' on transitions state that the transition is triggered by inputs satisfying C and the outputs are updated according to O.

Assume that there is an implementation of this card and we have an abstract model of it which is generated by RaLib [2]. Fig. 1b shows the learned model. As it is observed in Fig. 1b, the learned model introduces a different partitioning of the inputs comparing to the specification's. This difference is typically observable between a learned model and the already existing (reference) models. In this work, we suggest to consider such information and we show that it will improve the coverage of the specification and the IUT in a testing experiment. Note that the abstract models extracted from implementations may not contain the exact input-output relation. They largely provide useful information about the partitioning of the input domain. Accordingly, we mainly consider and use the complementary information about the partitioning of inputs in generating tests.



(a) Specification ($\mathcal{S}^*_{\mathcal{PPC}}$) (b) Learned model ($\mathcal{T}^*_{\mathcal{PPC}}$)

Fig. 1. The behaviour of the example prepaid card.

3 Preliminaries

For formal reasoning, we need a model of a specification, and also assume that the behaviour of the IUT can be captured by some (unknown) formal model in a given formalism. In the following, we introduce the formalism used in this work to model specifications and abstractions of implementations, and then define conformance in its context.

3.1 Symbolic Reactive State Machines

A Symbolic Reactive State Machine (SRSM) is a symbolic representation of the state-based behaviour of a system, with a set of input/output variables. It is symbolic as it explicitly uses the notion of variables, rather than concrete values, in specifying transitions (e.g., data-dependent transitions) and outputs (e.g., output as a function of input variables).

Definition 1 (Symbolic Reactive State Machine (SRSM)). *An SRSM $\mathcal{S}^*$ is a 6-tuple $(\bar{S}, \bar{s_0}, \bar{\delta}, \bar{\lambda}, V, D)$, where*

- *$\bar{S}$ is the non-empty and finite set of symbolic states,*
- *$\bar{s}_0 \in \bar{S}$ is the initial symbolic state,*
- *V is the set of variables such that $V = I \cup O$, i.e., V is partitioned into disjoint sets I and O of input and output variables, respectively,*
- *D is the range of all variable valuations,*
  - *D_I: domain of input variables*
  - *D_O: domain of output variables*
- *$\bar{\delta} : \bar{S} \times \mathcal{P}(D_I) \to \bar{S}$ is the transition function, and*
- *$\bar{\lambda} : \bar{S} \times \mathcal{P}(D_I) \to \bar{\mathcal{E}}(I)$ is the output function.*
  - *$\mathcal{E}(I)$ is the set of expressions over input variables (I).*
  - *$\bar{\mathcal{E}}(I) \in \underbrace{\mathcal{E}(I) \times \ldots \times \mathcal{E}(I)}_{|O|}$, i.e., each expression gives the value of one output variable.*

Notations. Input variables are enumerated as $I = \{x_1, \ldots, x_k\}$ and $D_I = D_{x_1} \times \ldots \times D_{x_k}$ is the domain of inputs. $\mathcal{P}(D_I)$ is the powerset (the set of all subsets) of D_I, and $\boldsymbol{x} = (x_1, \ldots, x_k)$ is the input variable vector. We use small letters (e.g., c) to represent a single valuation of the input vector ($\boldsymbol{x} = c \in D_I$) and capital letters (e.g., C) to show a set of valuations of the input vector ($C \in \mathcal{P}(D_I)$). Symbolic states are labelled with overscored letters (e.g., $\bar{s}$, $\bar{S}$). The Greek letter φ is used to represent output functions and it is a vector of expressions (i.e., $\varphi \in \bar{\mathcal{E}}(I)$). Given a vector of expressions φ and an input $c \in D_I$, $\varphi[c]$ denotes the output vector with the valuation of each expression for input c: $\varphi[c] = o \in D_O$.

Example. Fig. 1a shows the behaviour of our example prepaid card as SRSM $\mathcal{S}^*_{\mathcal{PPC}} = (\bar{S}, \bar{s_0}, \bar{\delta}_s, \bar{\lambda}_s, I \cup O, D_I \cup D_O)$, where $\bar{S} = \{\bar{s}_0, \bar{s}_1, \bar{s}_2\}$, $\bar{s}_0$ is the initial state, $I = \{a\}$, $D_I = D_a = \mathcal{N}$, $O = \{b\}$, $D_O = D_b = [0, 500]$, $\bar{\delta}_s$, and $\bar{\lambda}_s$ are defined based on the given transitions. (Note that the machine remains in a same state and the outputs will remain unchanged for any input not satisfying the conditions in the labels.)

3.2 Concrete and Symbolic Paths

The behaviour of an SRSM is described in terms of the outputs produced for given inputs, which is formally represented by a set of paths (i.e., sequences of transitions) in the model. In an SRSM model, there are two types of paths, namely *concrete paths* and *symbolic paths*, which are defined below.

Definition 2 (Concrete Path). *In an SRSM $\mathcal{S}^* = (\bar{S}, \bar{s}_0, \bar{\delta}, \bar{\lambda}, V, D)$, a concrete path cp is a finite sequence $\bar{s}_0(c_1, \bar{s}_1)(c_2, \bar{s}_2) \ldots (c_k, \bar{s}_k)$ such that $\exists C \in \mathcal{P}(D_I) \bullet \bar{\delta}(\bar{s}_i, C) = \bar{s}_{i+1} \wedge c_{i+1} \in C$, for $1 \leq i \leq k$. $State(cp) = \bar{s}_0 \ldots \bar{s}_k$, $In(cp) = c_1 c_2 \ldots c_k$, and $Out(cp) = o_1 o_2 \ldots o_k$ where $\exists C \in \mathcal{P}(D_I) \bullet \bar{\lambda}(\bar{s}_i, C) = \varphi_{i+1} \wedge c_{i+1} \in C \wedge o_{i+1} = \varphi_{i+1}[c_{i+1}]$, for $0 \leq i < k$. The set of all concrete paths in $\mathcal{S}^*$ is denoted by $Path(\mathcal{S}^*)$ and for a set of concrete paths CP, $In(CP) = \{In(cp) \mid cp \in CP\}$.*

Definition 3 (Symbolic Path). *In an SRSM $\mathcal{S}^* = (\bar{S}, \bar{s}_0, \bar{\delta}, \bar{\lambda}, V, D)$, a symbolic path sp is a finite sequence $\bar{s}_0(C_1, \bar{s}_1)(C_2, \bar{s}_2) \ldots (C_k, \bar{s}_k)$ such that $\bar{\delta}(\bar{s}_i, C_{i+1}) = \bar{s}_{i+1}$, for $1 \leq i \leq k$. $State(sp) = \bar{s}_0 \ldots \bar{s}_k$, $In(sp) = C_1 C_2 \ldots C_k$, and $Out(sp) = \varphi_1 \varphi_2 \ldots \varphi_k$ is the associated sequence of (output) expressions where $\bar{\lambda}(\bar{s}_i, C_{i+1}) = \varphi_{i+1}$, for $0 \leq i < k$. Also, a subpath of sp is a finite sequence $\bar{s}_0(C'_1, \bar{s}_1)(C'_2, \bar{s}_2) \ldots (C'_k, \bar{s}_k)$ such that $C'_i \subseteq C_i$, for $1 \leq i \leq k$. The set of all symbolic paths in $\mathcal{S}^*$ is denoted by $SymPath(\mathcal{S}^*)$ and for a set of symbolic paths SP, $In(SP)$ is defined as $\{In(sp) \mid sp \in SP\}$.*

Each transition represents a set of concrete transitions and thus, a symbolic path *sp* specifies a set of concrete paths, called its interpretation.

Definition 4 (Symbolic Path Interpretation). *In an SRSM $\mathcal{S}^*$, the interpretation of a symbolic path $sp = \bar{s}_0(C_1, \bar{s}_1) \ldots (C_n, \bar{s}_n)$, denoted by $[\![sp]\!]$, is the set of concrete paths defined as $\{cp_1, cp_2, \ldots\}$ such that for each cp_i $(i = 1, 2, \ldots)$*

- $State(cp_i) = State(sp)$,
- $In(cp_i) = c_{i,1}c_{i,2} \dots c_{i,n}$ *such that* $c_{i,j} \in C_j$, *for* $j = 1, 2, \dots, n$
- $Out(cp_i) = \varphi_1[c_{i,1}]\varphi_2[c_{i,2}] \dots \varphi_n[c_{i,n}]$, *where* $Out(sp) = \varphi_1\varphi_2 \dots \varphi_n$

A symbolic path can be partitioned into a set of *subpaths* such that these paths do not have any concrete path in common and altogether, they cover all the concrete paths in the main symbolic path.

Definition 5 (Symbolic Path Partitioning). *In an SRSM $\mathcal{S}^*$, a partitioning of a symbolic path $sp = \bar{s}_0(C_1, \bar{s}_1) \dots (C_n, \bar{s}_n)$, denoted by $Part(sp)$, is a set of subpaths defined as $Part(sp) = \{sp_1, sp_2, \dots, sp_k\}$ such that*

1. $\forall sp_i, sp_j \in Part(sp) \bullet i \neq j \implies \exists 0 < m \leq n \bullet C_{i,m} \cap C_{j,m} = \emptyset$ *($In(sp_l) = C_{l,1} \dots C_{l,n}$), and*
2. $[\![sp]\!] = \bigcup\limits_{p \in Part(sp)} [\![p]\!]$.

3.3 SRSM Models and Conformance

This section defines our notion of behavioural conformance between two SRSMs.

Definition 6 (Conformance). *Assume that $\mathcal{S}^*$ and $\mathcal{T}^*$ are two SRSMs defined over the same I/O variables. Then, $\mathcal{T}^*$ conforms to $\mathcal{S}^*$, denoted by $\mathcal{T}^*$ **conf** $\mathcal{S}^*$, if and only if the following two statements hold.*

1. $\forall seq_{in} \in In(Path(\mathcal{S}^*))\ \exists cp \in Path(\mathcal{T}^*) \bullet In(cp) = seq_{in}$, *and*
2. $\forall cp \in Path(\mathcal{T}^*)\ (\exists cp' \in Path(\mathcal{S}^*) \bullet In(cp) = In(cp')) \implies \exists cp'' \in Path(\mathcal{S}^*) \bullet In(cp) = In(cp'') \wedge Out(cp) = Out(cp'')$.

The first statement indicates that all the input sequences defined in the specification should be defined in the IUT. In particular, for non-deterministic behaviour, it indicates that the IUT should at least have one concrete path with the same inputs. Then, the second statement says that for those concrete paths whose inputs are defined in the specification, the IUT should satisfy the specification. The statement also implies that the IUT may have additional behaviour (i.e., sequences of inputs which are not defined in the specification).

The above definition of conformance implies that we need to examine each and every path in $Path(\mathcal{S}^*)$ with all paths in $Path(\mathcal{T}^*)$ and vice versa in order to detect a non-conformant IUT. However, this is not feasible in most practical contexts (e.g., infinite input domain or a large number of concrete paths). We address this problem by defining conformance in terms of symbolic paths. To do so, we first define two relationships, namely *compatibility* and *containment*, for comparing two symbolic paths with each other. These relations allow determining conformance by comparing symbolic paths rather than concrete paths. Subsequently, we show how checking conformance at the symbolic level can be reduced to checking conformance of a finite number of concrete paths in their interpretation.

Definition 7 (Symbolic Path Compatibility). *A symbolic path* sp *is compatible with a symbolic path* sp'*, denoted by* $sp \prec sp'$*, if and only if* $In(sp) \sqsubseteq In(sp')$*, where for* $In(sp) = C_1C_2 \ldots C_n$ *and* $In(sp') = C'_1C'_2 \ldots C'_n$*,* $In(sp) \sqsubseteq In(sp')$ *holds if and only if* $C_i \subseteq C'_i$ *for* $1 \leq i \leq n$*.*

Definition 8. *Two expressions* φ *and* φ' *are equivalent over a set of inputs* $X \in \mathcal{P}(D_I)$*, denoted by* $\varphi \overset{X}{\equiv} \varphi'$*, if and only if* $\forall x \in X \bullet \varphi[x] = \varphi'[x]$*. If* $X = D_I$*, then* φ *and* φ' *are equivalent which is denoted by* $\varphi \equiv \varphi'$*.*

Example. Consider symbolic paths $sp_1 \in SymPath(\mathcal{S}^*_{\mathcal{PPC}})$ and $sp'_1 \in SymPath(\mathcal{T}^*_{\mathcal{PPC}})$, defined as follows. sp'_1 is not compatible with sp_1 as $In(sp'_1) \not\sqsubseteq In(sp_1)$ and therefore $sp'_1 \not\prec sp_1$.

$sp_1 = \bar{s}_0\ (\{a \leq 300\}, \bar{s}_1)(\{a \leq 300\}, \bar{s}_1)$; $In(sp_1) = (\{a \leq 300\})(\{a \leq 300\})$

$sp'_1 = \bar{t}_0\ (\{a \leq 200\}, \bar{t}_0)(\{200 < a \wedge a - b < 300\}, \bar{t}_1)$; $In(sp'_1) = (\{a \leq 200\})(\{200 < a \wedge a - b < 300\})$ □

Definition 9 (Symbolic Path Containment). *A symbolic path* sp *is contained in a symbolic path* sp'*, denoted by* $sp \preceq sp'$*, if and only if* $sp \prec sp' \wedge Out(sp) \equiv Out(sp')$*, where for* $Out(sp) = \varphi_1\varphi_2 \ldots \varphi_n$ *and* $Out(sp') = \varphi'_1\varphi'_2 \ldots \varphi'_n$*,* $Out(sp) \equiv Out(sp)$ *holds if and only if* $\varphi_i \overset{C_i}{\equiv} \varphi'_i$ *for* $1 \leq i \leq n$*, where* $In(sp) = C_1C_2 \ldots C_n$*.*

Herein, the main issue is to find out whether two expressions are equivalent. It is not always possible to evaluate and compare two expressions for all the input values, for example when inputs are infinite. To overcome this issue, we introduce and define *n-uniformity* between two functions (expressions), which is defined w.r.t. the set of inputs on which they are both defined.

Definition 10 (n-Uniformity). *Let* $f : D_f \rightarrow D_O$ *and* $g : D_g \rightarrow D_O$ *be two functions where* $D_f, D_g \in \mathcal{P}(D_I)$*. Then,* f *and* g *are n-uniform over* $D_f \cap D_g$*, denoted by* $f \approx_n g$*, if and only if* n *is the smallest number for which the following statement holds.*

$(\forall 0 \leq i \leq n\ \exists x_i \in D_f \cap D_g \bullet (\forall 0 \leq j \leq n \bullet i \neq j \implies x_i \neq x_j) \wedge f(x_i) = g(x_i)) \implies f \overset{D_f \cap D_g}{\equiv} g$.

The degree of uniformity between f *and* g *is* n*, if* $f \approx_n g$*.*

Corollary 1. *Let* $f : D_f \rightarrow D_O$ *and* $g : D_g \rightarrow D_O$ *be two functions where* $D_f, D_g \in \mathcal{P}(D_I)$ *and* $f \approx_n g$*. Then* $n < |D_f \cap D_g|$*.*

Accordingly, if the degree of uniformity between output functions in two symbolic paths is determined, it is possible to find out if those paths are compatible or not and this could be done with a finite number of values. This is explained by the following lemma.

Lemma 1. *Let* $\mathcal{S}^* = (\bar{S}, \bar{s_0}, \bar{\delta}_s, \bar{\lambda}_s, V, D)$*,* $\mathcal{T}^* = (\bar{T}, \bar{t_0}, \bar{\delta}_t, \bar{\lambda}_t, V, D)$*,* $sp \in SymPath(\mathcal{S}^*)$*,* $sp' \in SymPath(\mathcal{T}^*)$*. Then,* $sp \preceq sp'$ *if and only if*

1. *$sp \prec sp'$*
2. *φ_i and φ'_i, $1 \leq i \leq n$, produce the same output for $d_i{+}1$ distinct input values, where $Out(sp) = \varphi_1\varphi_2 \ldots \varphi_n$ and $Out(sp') = \varphi'_1\varphi'_2 \ldots \varphi'_n$ and $\varphi_i \approx_{d_i} \varphi'_i$.*

Using the above lemma, for any pair of symbolic paths sp and sp', we can find the minimum number of distinct sequences of inputs required to determine if $sp \preceq sp'$ or not. This number, denoted by $DistDeg(sp, sp')$, can be calculated regarding the n-uniformity between the output functions associated to these paths.

Example. Consider symbolic paths $sp \in SymPath(\mathcal{S}^*_{\mathcal{PPC}})$ and $sp' \in SymPath(\mathcal{T}^*_{\mathcal{PPC}})$. $In(sp') \sqsubseteq In(sp)$ and hence $sp' \prec sp$. The output functions in these models (φ and φ') are polynomials of degree one, therefore $\varphi \approx_1 \varphi'$ and $DistDeg(sp, sp') = 2$: we can determine if $sp' \preceq sp$ with two sequences of inputs.

$sp = \bar{s}_0\ (\{a \leq 300\}, \bar{s}_1)(\{a \leq 300\}, \bar{s}_1)$; $In(sp) = (\{a \leq 300\})(\{a \leq 300\})$, $Out(sp) = \varphi = (b = a)(b = a)$

$sp' = \bar{t}_0\ (\{a \leq 200\}, \bar{t}_0)(\{a \leq 200\}, \bar{t}_0)$; $In(sp') = (\{a \leq 200\})(\{a \leq 200\})$, $Out(sp') = \varphi' = (b = a)(b = a)$

□

Although *n-uniformity* is an abstract concept, as the above example suggests, in many practical cases, it can be determined by statically analysing the model/program expressions.

4 Conformance Testing for SRSMs

This section formalises conformance testing in the context of this work and the introduced formal model.

4.1 Test case and Test Suite

A test case, defined below, specifies a sequence of inputs and their corresponding expected set of outputs according to the specification.

Definition 11 (Test Case and Test Suite).

1. *A test case tc is a tuple $(in_{seq},\ out_{seq})$, where*
   - *in_{seq} is a finite sequence of inputs $c_1c_2 \ldots c_k$ such that $c_i \in D_I$ for $1 \leq i \leq k$, and*
   - *out_{seq} is a set of finite sequences of outputs $\{O_1,\ O_2,\ \ldots,\ O_n\}$ where $O_i = o_{i,1} \ldots o_{i,k}$ such that $o_{i,j} \in D_O$, for $1 \leq i \leq n$ and $1 \leq j \leq k$.*

   By definition, $In(tc) = c_1c_2 \ldots c_k$ and $Out(tc) = \{O_1,\ O_2,\ \ldots,\ O_n\}$.
2. *A test suite is a finite set of test cases.*

In the context of this work, test cases are executed to a system, one by one: the inputs are given to the system and the outputs are observed. The comparison of the observed behaviour with the expected behaviour determines the test verdict (pass/fail).

Definition 12 (Test Case Execution). *Execution of a test case tc on an SRSM $\mathcal{S}^*$, denoted by $Exec(tc,\ \mathcal{S}^*)$, gives the sequence of outputs specified by the concrete path $cp \in\ Path(\mathcal{S}^*)$ such that $In(cp) = In(tc)$ and then, $Exec(tc,\ \mathcal{S}^*) = Out(cp)$. If there is no such concrete path the test case is not applicable on the model which is denoted by $Exec(tc,\ \mathcal{S}^*) = \perp$.*

Definition 13 (Test Verdict).

1. *An SRSM $\mathcal{S}^*$ passes a test case tc, denoted by $Pass(\mathcal{S}^*, tc)$, if and only if it is applicable on $\mathcal{S}^*$ and $Exec(tc,\ \mathcal{S}^*) \in Out(tc)$.*
   If $\mathcal{S}^*$ does not pass a test case tc, it fails, denoted by $Fail(\mathcal{S}^*, tc)$.
2. *An SRSM $\mathcal{S}^*$ passes a test suite TS, denoted by $Pass(\mathcal{S}^*, TS)$, if and only if $\forall tc \in TS\ \bullet\ Pass(\mathcal{S}^*, tc)$.*
   If $\mathcal{S}^*$ does not pass a test suite TS, it fails, denoted by $Fail(\mathcal{S}^*, TS)$.

4.2 Complete Test Suite

An ideal test suite should specify all possible behaviours of a system and its specification. Such a test suite is called *complete*. However, this is not possible in most practical cases. A common and typical approach to address this issue is to restrict the power of a test suite to only detecting conformance or only detecting non-conformance (i.e., soundness and exhaustiveness in [6]).

We define *completeness* in the context of our proposal in that we generate a test suite specifically enriched for testing a particular implementation such that

1. there would be no uncovered symbolic behaviour in any of the models (*coverage*),
2. none of the test cases fails, if the implementation conforms to the specification (*soundness*), and
3. for any non-conformant behaviour in the implementation, there is a specific test case which discovers that behaviour (*relative exhaustiveness*).

Accordingly, a *complete* test suite is the one that satisfies *test coverage*, *soundness*, and *relative exhaustiveness.*

Definition 14 (Test Coverage). *A test suite TS covers an SRSM $\mathcal{S}^*$ if and only if $\forall sp \in SymPath(\mathcal{S}^*)\ \exists tc \in TS \bullet In(tc) \in\ In(\llbracket sp \rrbracket)$.*

Definition 15 (Soundness). *A test suite TS is sound w.r.t. an SRSM $\mathcal{S}^*$ if and only if $\forall \mathcal{T}^*\ \bullet\ \mathcal{T}^*$ **conf** $\mathcal{S}^* \implies \forall tc \in TS\ \bullet\ Pass(\mathcal{T}^*, tc)$.*

Definition 16 (Relative Exhaustiveness). *A test suite TS is exhaustive relative to SRSMs $\mathcal{S}^*$, the reference model, and $\mathcal{T}^*$, the model to be tested, if and only if the following statements hold.*

1. *$\forall sp \in\ SymPath(\mathcal{S}^*)\ \forall sp' \in\ SymPath(\mathcal{T}^*) \bullet In(\llbracket sp \rrbracket) \cap In(\llbracket sp' \rrbracket) \neq \emptyset \implies \exists tc \in TS\ \bullet\ In(tc) \in In(\llbracket sp \rrbracket) \cap In(\llbracket sp' \rrbracket)$.*
2. *$\forall sp \in\ SymPath(\mathcal{S}^*)\ \exists Part(sp)\ \bullet\ \exists p \in\ Part(sp)\ \bullet\ In(\llbracket p \rrbracket) \cap In(Path(\mathcal{T}^*)) = \emptyset \implies\ \exists tc \in TS\ \bullet\ In(tc) \in In(\llbracket p \rrbracket) \wedge\ Fail(\mathcal{T}^*, tc)$.*

3. $\forall sp \in \mathit{SymPath}(\mathcal{T}^*)\ \exists \mathit{Part}(sp) \bullet \exists p \in \mathit{Part}(sp) \bullet \mathit{In}(\llbracket p \rrbracket) \cap \mathit{In}(\mathit{Path}(\mathcal{S}^*)) = \emptyset \implies \exists tc \in TS \bullet \mathit{In}(tc) \in \mathit{In}(\llbracket p \rrbracket)$.

In the next section, our proposed testing strategy to generate a complete test suite is presented.

5 Gray-Box Conformance Testing

In this section, we define the *transition composition* of two SRSM models which provides an integrated view of the transitions of both models in one model, regardless of their outputs. We then use this model to generate the target test suite.

5.1 Transition Composition

Intuitively, the transition composition is a (sub-)product of the models in that the transition function is defined based on the intersection of transitions.

Definition 17 (Transition Composition). *Let* $\mathcal{S}^* = (\bar{S}, \bar{s_0}, \bar{\delta_s}, \bar{\lambda_s}, V, D)$ *and* $\mathcal{T}^* = (\bar{T}, \bar{t_0}, \bar{\delta_t}, \bar{\lambda_t}, V, D)$ *be two SRSMs with the same I/O variables.* $\mathcal{M}^* = (\bar{M}, \bar{m_0}, \bar{\delta}, \emptyset, V, D)$ *is the transition composition of* $\mathcal{S}^*$ *and* $\mathcal{T}^*$*, denoted by* $\mathcal{M}^* = \mathit{trComp}(\mathcal{S}^*, \mathcal{T}^*)$*, where*

- $\bar{M} \subseteq (\bar{S} \cup \{err_s\}) \times (\bar{T} \cup \{err_t\})$,
- $\bar{m_0} = (\bar{s_0}, \bar{t_0})$,
- $\forall \bar{m} = (\bar{s}, \bar{t}) \in \bar{M}, C \in \mathcal{P}(D_I) \bullet \bar{s} \in \bar{S} \wedge \bar{t} \in \bar{T} \implies$
$$\bar{\delta}(\bar{m}, C) = \begin{cases} (\bar{s}', \bar{t}') & : \bar{s}' \in \bar{S} \wedge \bar{t}' \in \bar{T} \wedge \exists C', C'' \in \mathcal{P}(D_I) \bullet \bar{\delta_s}(\bar{s}, C') = \bar{s}' \\ & \wedge\ \bar{\delta_t}(\bar{t}, C'') = \bar{t}' \wedge\ C' \cap C'' \neq \emptyset \wedge\ C = C' \cap C'' \\ (\bar{s}', err_t) & : \bar{s}' \in \bar{S} \wedge\ \exists C' \in \mathcal{P}(D_I) \bullet \bar{\delta_s}(\bar{s}, C') = \bar{s}' \\ & \wedge(\exists C_e \subseteq C' \bullet \forall \bar{t}' \in \bar{T}, C'' \in \mathcal{P}(D_I) \bullet \\ & \bar{\delta_t}(\bar{t}, C'') = \bar{t}' \wedge C_e \cap C'' = \emptyset) \wedge\ C = C_e \\ (err_s, \bar{t}') & : \bar{t}' \in \bar{T} \wedge\ \exists C' \in \mathcal{P}(D_I) \bullet \bar{\delta_t}(\bar{t}, C') = \bar{t}' \\ & \wedge(\exists C_e \subseteq C' \bullet \forall \bar{s}' \in \bar{S}, C'' \in \mathcal{P}(D_I) \bullet \\ & \bar{\delta_s}(\bar{s}, C'') = \bar{s}' \wedge C_e \cap C'' = \emptyset) \wedge\ C = C_e \end{cases}$$
- $\forall \bar{m} = (\bar{s}, err_t) \in \bar{M}, C \in \mathcal{P}(D_I) \bullet \bar{s} \in \bar{S} \implies$
$\bar{\delta}(\bar{m}, C) = (\bar{s}', err_t)$ *if* $\bar{s}' \in \bar{S} \wedge\ \exists C' \in \mathcal{P}(D_I) \bullet \bar{\delta_s}(\bar{s}, C') = \bar{s}' \wedge C = C'$*, and*
- $\forall \bar{m} = (err_s, \bar{t}) \in \bar{M}, C \in \mathcal{P}(D_I) \bullet \bar{t} \in \bar{T} \implies$
$\bar{\delta}(\bar{m}, C) = (err_s, \bar{t}')$ *if* $\bar{t}' \in \bar{T} \wedge\ \exists C' \in \mathcal{P}(D_I) \bullet \bar{\delta_t}(\bar{t}, C') = \bar{t}' \wedge C = C'$.

In a transition composition, the outgoing transitions on each state are defined based on the intersection of the valid input domains of the transitions of the components. The specific symbols err_s and err_t identify situations in which there is a set of inputs defined in one model but not in the other. Note that we keep tracking states involving err_s and err_t as we do not want to lose any possible transition in any of the models.

Corollary 2. *Let* $\mathcal{S}^* = (\bar{S}, \bar{s_0}, \bar{\delta_s}, \bar{\lambda_s}, V, D)$*,* $\mathcal{T}^* = (\bar{T}, \bar{t_0}, \bar{\delta_t}, \bar{\lambda_t}, V, D)$*,* $\mathcal{M}^* = (\bar{M}, \bar{m_0}, \bar{\delta}, \emptyset, V, D)$*, and* $\mathcal{M}^* = \ crComp(\mathcal{S}^*, \mathcal{T}^*)$*. Then for all* $\bar{m}, \bar{m}' \in \bar{M}$ *and* $C \in \mathcal{P}(D_I)$ *such that* $\bar{\delta}(\bar{m}, C) = \bar{m}'$ *the following two statements hold*

- $\exists \bar{s}, \bar{s}' \in \bar{S}, C' \in \mathcal{P}(D_I) \bullet \bar{m} \in \{\bar{s}\} \times (\bar{T} \cup \{err_t\}) \wedge \bar{m}' \in \{\bar{s}'\} \times (\bar{T} \cup \{err_t\}) \wedge \bar{\delta}_s(\bar{s}, C') = \bar{s}' \implies C \subseteq C'$
- $\exists \bar{t}, \bar{t}' \in \bar{T}, C' \in \mathcal{P}(D_I) \bullet \bar{m} \in (\bar{S} \cup \{err_s\}) \times \{\bar{t}\} \wedge \bar{m}' \in (\bar{S} \cup \{err_s\}) \times \{\bar{t}'\} \wedge \bar{\delta}_t(\bar{t}, C') = \bar{t}' \implies C \subseteq C'$

Example. Fig. 2 shows a part of the transition composition of the models in Fig. 1a and Fig. 1b.

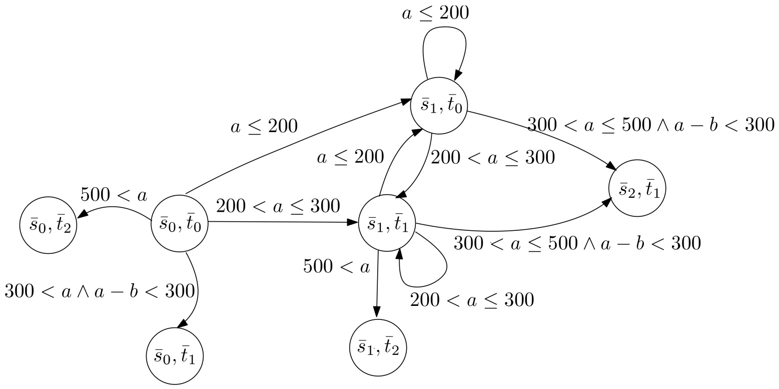


Fig. 2. An excerpt of the transition composition of $\mathcal{S}^*_{\mathcal{PPC}}$ and $\mathcal{T}^*_{\mathcal{PPC}}$.

The transition composition of two SRSM models has two main properties which allow generating a complete test suite. First, according to Definition 18, it covers both of its underlying models (Theorem 1). Second, all the symbolic paths in the transition composition is at least compatible with a symbolic path in one of the underlying models indicating that the transition composition does not have any extra behaviour (Theorem 2).

Definition 18 (Model Coverage). *An SRSM* $\mathcal{S}^*$ *covers an SRSM* $\mathcal{T}^*$ *if and only if* $\forall sp \in \ SymPath(\mathcal{T}^*) \ \exists sp' \in \ SymPath(\mathcal{S}^*) \ \bullet \ sp' \prec sp$*.*

Theorem 1. *Let* $\mathcal{S}^*$ *and* $\mathcal{T}^*$ *be two SRSMs and* $\mathcal{M}^* = \ trComp(\mathcal{S}^*, \mathcal{T}^*)$*. Then* $\mathcal{M}^*$ *covers* $\mathcal{S}^*$ *and* $\mathcal{T}^*$*.*

Theorem 2. *Let* $\mathcal{S}^*$ *and* $\mathcal{T}^*$ *be two SRSMs and* $\mathcal{M}^* = \ trComp(\mathcal{S}^*, \mathcal{T}^*)$*. Then*

- $\forall sp \in SymPath(\mathcal{M}^*) \ \bullet \ (\forall \bar{m} \in State(sp) \ \bullet \ \bar{m} \in \bar{S} \times (\bar{T} \cup \{err_t\})) \implies \exists sp' \in SymPath(\mathcal{S}^*) \ \bullet sp \ \prec \ sp'$.
- $\forall sp \in SymPath(\mathcal{M}^*) \ \bullet \ (\forall \bar{m} \in State(sp) \ \bullet \ \bar{m} \in (\bar{S} \cup \{err_s\}) \times \bar{T})) \implies \exists sp' \in SymPath(\mathcal{T}^*) \ \bullet sp \ \prec \ sp'$.

5.2 Test Suite Generation

Having defined the transition composition of two SRSMs, we next generate a complete test suite. First, we define the test cases for each symbolic path in the transition composition, which are then accumulated in the final and complete test suite.

Definition 19. *Let $\mathcal{S}^*$ be the specification model, $\mathcal{T}^*$ be the implementation model, and $\mathcal{M}^* = trComp(\mathcal{S}^*, \mathcal{T}^*)$ be the transition composition. For each $sp \in SymPath(\mathcal{M}^*)$, $TC(sp)$ is a set of test cases to examine the compatibility between the two symbolic paths in $\mathcal{T}^*$ and $\mathcal{S}^*$ in which sp is contained, and defined as follows.*

1. *If there exists $sp' \in SymPath(\mathcal{S}^*)$ and $sp'' \in SymPath(\mathcal{T}^*)$ such that $sp \prec sp'$ and $sp \prec sp''$, then $TC(sp)$ is a set of test cases $\{tc_1, \ldots, tc_k\}$, where $k = DistDeg(sp',\ sp'')$, such that $In(tc_i) \subseteq In(\llbracket sp \rrbracket)$ and $Out(tc_i)$ is determined the output(s) produced by $\mathcal{S}^*$ for $In(tc_i)$, $1 \leq i \leq k$.*
2. *If there exists $sp' \in SymPath(\mathcal{S}^*)$ such that $sp \prec sp'$ and there is no $sp'' \in SymPath(\mathcal{T}^*)$ such that $sp \prec sp''$, then $TC(sp)$ contains only one test case tc such that $In(tc) \subseteq In(\llbracket sp \rrbracket)$ and $Out(tc)$ is the output(s) produced by $\mathcal{S}^*$ for $In(tc)$.*
3. *If there exists $sp' \in SymPath(\mathcal{T}^*)$ such that $sp \prec sp'$ and there is no $sp'' \in SymPath(\mathcal{S}^*)$ such that $sp \prec sp''$, then $TC(sp)$ contains only one test case tc such that $In(tc) \subseteq In(\llbracket sp \rrbracket)$ and $Out(tc) = \bot$ (i.e., undefined). Note that such a test case observes the behaviours not specified in the specification.*

Definition 20 (Composition-based Test Suite). *Given the specification model $\mathcal{S}^*$, the implementation model $\mathcal{T}^*$, and their transition composition $\mathcal{M}^*$, a composition-based test suite, denoted by $C_{omp}TS(\mathcal{S}^*,\ \mathcal{T}^*)$, is defined as follows.*

$$C_{omp}TS(\mathcal{S}^*,\ \mathcal{T}^*) = \bigcup_{sp \in SymPath(\mathcal{M}^*)} TC(sp)$$

The following theorem demonstrates that a composition-based test suite satisfies test coverage, soundness and exhaustiveness properties.

Theorem 3. *Let $\mathcal{S}^*$ be the specification model, $\mathcal{T}^*$ be the implementation model, and $\mathcal{M}^* = trComp(\mathcal{S}^*, \mathcal{T}^*)$. Then, $C_{omp}TS(\mathcal{S}^*,\ \mathcal{T}^*)$ is a sound and exhaustive test suite and covers $\mathcal{S}^*$ and $\mathcal{T}^*$.*

6 Experimental Results

In order to check the effectiveness of our approach, we use our method in the context of a well-known example from the *European Train Control System (ETCS)*, namely the *Ceiling Speed Monitor (CSM)* module which monitors the speed of a train and triggers the required actions if the maximal speed is exceeded. A complete description of the system can be found in [19]. We applied our method in testing six different (faulty) implementations of the CSM module and compared the outcomes with random testing and the equivalence class testing introduced in [16]. Implementations are mutants of a correct implementation of the CSM module. In the first implementation (IUT_1) the faults are related to boundary values (e.g., $<$ replaced by $\leq$). In the next four implementations (IUT_2, IUT_3, IUT_4, and IUT_5), the faults are in the guard condition, but they are not related to boundary values. Moreover, in IUT_4 and IUT_5, the difference between the sets of inputs defined by the correct condition and the wrong condition is too narrow (i.e., for limited number of input values the difference could be discovered). The last implementation (IUT_6) contains a fault in an output function associated to one of the transitions.

In the experiment, we mainly investigated the question whether our method observed the faults or not. We also considered the number of test cases generated by each method. Additionally, in order to have an approximation of the overhead associated with our method, we considered the time required to generate the transition composition. This time is computed based on the number of basic computation steps in generating the composition (assuming that all steps consume a constant amount of time, this time is proportional to the number of steps).

In random testing, test cases are created by generating random values in the appropriate data ranges. For equivalence class testing, we considered a refinement of the initial coarsest input equivalence class partitioning (IECP) that reflects all case distinctions visible in guard conditions of the CSM model, which implies the fault model for this testing method. Note that the number of test cases generated by IECP is the same for all the six cases. We used the test data provided in [20], for the number of generated test cases by IECP. For random testing, in each case, a random test suite of the same length as our method's, was selected and used for comparison.

Table 1 summarises the results of this experiment. Basically, the results show that our method performs better than random testing with the same number of test cases. They also show that in cases the behaviour of the IUT lies outside the fault domain of the IECP testing, in particular when the input equivalence classes are narrow, our approach performs better than IECP. This is because, in such cases, the desired input values have very low probabilities to be chosen. Therefore, in both random testing and IECP, an increase in the number of test cases has limited effect on their testing strength. The IECP testing could not kill IUT_4 and IUT_5 which are outside its fault domain and have narrow equivalence classes. IUT_2 and IUT_3 are both out of the fault domain and the set of inputs to discover their faults is not narrow (i.e., a proper input values could be chosen by random input selection). However, only IUT_3 was killed by IECP. Finally, the time required to generate the transition composition and the number of test cases could be an indication of the efficiency of our method.

Nevertheless, this experiment provides a preliminary result. In particular, having treated only one type of case study is a threat to the validity of our results. To remedy this, we plan to carry out more testing experiments considering different kinds of cases. To address the efficiency and scalability question more thoroughly, in addition to more case studies, we need to collect additional information from other methods to have a valid comparison between methods, such as the time required to transform the original test model into the desired formalism.

Table 1. Experimental results

IUT	**Random Testing**		**IECP**		**Our Method**		
	Killed	No. TCs	Killed	No. TCs	Killed	No. TCs	No. steps
1	×	24	✓	186	✓	24	18
2	×	30	×	186	✓	30	19
3	×	25	✓	186	✓	25	19
4	×	37	×	186	✓	37	22
5	×	24	×	186	✓	24	21
6	✓	21	✓	186	✓	21	16

7 Conclusions and Future Work

In this paper, we presented a gray-box model-based testing strategy in that test suites are generated considering both the specification and an abstraction of the IUT. Specifications and implementations abstraction are modelled as *Symbolic Reactive State Machines (SRSMs)*, which are finite state machines with symbolic input and output. Given the SRSMs of a specification and an IUT, test cases are generated based on the *transition composition* of these models. We considered models with infinite input domain and then introduced the notion of *n-uniformity* which allows us confining the number of test cases for each symbolic path. We studied and proved coverage, soundness, and relative exhaustiveness of the proposed approach.

As for future work, we plan to roll out more testing experiments to investigate the applicability of the proposed strategy (in particular, the notion of *n-uniformity*) in different situations and discover its limitations. Moreover, we plan to study models with infinite set of symbolic paths and, then, how to select a finite subset of paths sufficient to generate a complete test suite, according to the regularity hypothesis [3]. Finally, we would like to work on efficient algorithms for generating the *transition composition* (e.g., adapting bi-simulation algorithms) and also for determining *n-uniformity*.

References

1. Walkinshaw, N., Taylor, R., Derrick, J. Inferring extended finite state machine models from software executions. Empir. Software Eng. **21**: 811-853. (2016)

2. Cassel, S., Howar, F., Jonsson, B., and Steffen, B. Active learning for extended finite state machines. Formal Asp. Comput. **28**(2): 233-263. (2016)
3. Gaudel, M.C.: Testing can be formal, too. In Proc. of TAPSOFT'95. pp. 82-96. Springer-Verlag London. (1995)
4. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE TSE **4**(3): 178-187. (1978)
5. Petrenko, A., von Bochmann, G., Yao, M.Y.: On fault coverage of tests for finite state specifications. Computer Networks and ISDN Systems **29**(1): 81-106. (1996)
6. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers. Lecture Notes In Computer Science **4949**: 1-38. (2008)
7. Petrenko A., Yevtushenko N., Bochmann G.: Fault models for testing in context. In Formal description techniques IX–theory, application and tools. pp. 163-178. Springer US. (1996)
8. Kicillof, N., Grieskamp, W., Tillmann, N., Braberman, V.: Achieving both model and code coverage with automated gray-box testing. In Proc. of A-MOST'07. pp. 1-11. ACM. (2007)
9. Giantamidis, G., Tripakis, S.: Learning Moore Machines from Input-Output Traces. In Proc. of FM'16. pp. 291-309. Springer International Publishing. (2016)
10. Lee, C,. Chen, F., Rosu, G.: Mining Parametric Specifications. In Proc. of ICSE'11, pp. 591-600. ACM. (2011)
11. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE TSE 27(2): 99-123. (2001)
12. Grieskamp, W., Tillmann, N., Campbell, C., Schulte, W., Veanes, M.: Action machines – towards a framework for model composition, exploration and conformance testing based on symbolic computation. In Proc. of QSIC'05. pp. 72-29. IEEE. (2006).
13. Frantzen, L., Tretmans, J., Willemse, T.: A symbolic framework for model-based testing. In Proc. of FATES/RV'06. pp. 40-54. Springer-Verlag Berlin. (2006)
14. Petrenko, A., Simao, A.: Checking experiments for finite state machines with symbolic inputs. In Proc. of ICTSS'15. pp. 3-18. Springer-Verlag New York. (2015)
15. Petrenko, A.: Checking experiments for symbolic input/output finite state machines. In Proc. of IEEE ICSTW'16. pp. 229-237. (2016)
16. Huang, W.-L., Peleska, J.: Complete model-based equivalence class testing. Int. Journal on Software Tools for Technology Transfer **18**(3): 262-283. Springer-Verlag Berlin. (2016)
17. Farzan, A., Holzer, A., Veith, H.: Perspectives on white-box testing: Coverage, concurrency, and concolic execution. In Proc. of ICST'15. pp. 1-11. IEEE. (2015)
18. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In SIGPLAN Conference on Programming Language Design and Implementation. pp. 213-223. ACM. (2005)
19. Braunstein C., Peleska J., Schulze U., Hübner F., Huang W., Haxthausen A., Vu Hong L.: A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor: Technical report, Work Package 4. Technical University of Denmark. (2014)
20. Hübner F., Huang W., Peleska J.: Experimental Evaluation of a Novel Equivalence Class Partition Testing Strategy. Tests and Proofs Lecture Notes in Computer Science **9154**: 155-172. (2015)