Complete Agent-driven Model-based System Testing for Autonomous Systems*

Kerstin I. Eder
Department of Computer Science, University of Bristol, United Kingdom
Kerstin.Eder@bristol.ac.uk

Wen-ling Huang
Department of Mathematics & Computer Science, University of Bremen, Germany
huang@uni-bremen.de

Jan Peleska
Department of Mathematics & Computer Science, University of Bremen, Germany
peleska@uni-bremen.de

In this position paper, a novel approach to testing complex autonomous transportation systems (ATS) in the automotive, avionic, and railway domains is described. It is intended to mitigate some of the most critical problems regarding verification and validation (V&V) effort for ATS. V&V is known to become infeasible for complex ATS, when using conventional methods only. The approach advocated here uses complete testing methods on the module level, because these establish formal proofs for the logical correctness of the software. Having established logical correctness, system-level tests are performed in simulated cloud environments and on the target system. To give evidence that "sufficiently many" system tests have been performed with the target system, a formally justified coverage criterion is introduced. To optimise the execution of very large system test suites, we advocate an online testing approach where multiple tests are executed in parallel, and test steps are identified on-the-fly. The coordination and optimisation of these executions is achieved by an agent-based approach. Each aspect of the testing approach advocated here is shown to either be consistent with existing standards for development and V&V of safety-critical transportation systems, or it is justified why it should become acceptable in future revisions of the applicable standards.

1 Introduction

Motivation

Autonomous transportation systems (ATS) in the automotive, avionic, or railway domains are highly complex and at the same time safety-critical. It is a widely accepted belief that the verification and validation (V&V) effort (including the test effort) for assuring an acceptable degree of safety and reliability in complex ATS will become so high that it can no longer be performed with conventional methods [20]. In particular, it cannot be expected that all the necessary system tests will be executable on the integrated target system (vehicle, train, or aircraft). Instead, a major portion of the tests needs to be executed concurrently in simulation environments. This approach, however, needs special attention from the perspective of applicable safety-related standards and certification rules: for good reasons, it has to be justified why

*Kerstin Eder was supported in part by the "UKRI Trustworthy Autonomous Systems Node in Functionality" under grant number EP/V026518/1. Wen-ling Huang and Jan Peleska were supported in part by the German Ministry of Economics, Project "HiDyVe – Highly Dynamic Virtual and Hybrid Validation and Verification" under grant agreement 20X1908E.

M. Farrell & M. Luckcuck (Eds.):
Third Workshop on Formal Methods
for Autonomous Systems (FMAS2021)
EPTCS 348, 2021, pp. 54–72, doi:10.4204/EPTCS.348.4

© Eder, Huang, and Peleska
This work is licensed under the
Creative Commons Attribution License.

simulation environments are sufficiently trustworthy "replicas" of the real target systems and their operational environments, so that certification credit can be obtained for these tests, though they have not been executed with the original equipment and the real operational environment.

Objectives

This contribution is a position paper: we outline a novel approach to testing complex ATS and justify each building block of this approach by references to existing theories from the field of formal methods or motivate them at least by means of illustrating examples. The novelty consists in a new combination of existing theories and technologies, and in a careful consideration of applicable standards and pre-standards in the automotive, railway, and avionic domains [3, 18, 19, 35]. Specifically, our approach is as follows.

(1) It is proposed to combine so-called *complete* software test strategies on the module level with scenario-based system tests. A test suite generated according to a specific strategy is complete, if it guarantees under certain hypotheses that every correct implementation will pass all test cases and every faulty implementation will fail at least one test case. Correctness is either defined by means of a conformance relation (refinement, equivalence, or variants thereof) to a given reference model, or by means of a set of property specifications to be fulfilled by the implementation. Here, the model-based approach is used. On the system level, test scenarios are created from more comprehensive system models whose behavioural semantics can be represented by symbolic finite state machines (SFSM) [31], extended by control state invariants for time-continuous and discrete variables. This extension of SFSMs can be interpreted as a restricted variant of hybrid automata, as introduced in [11]. The system-level models can be traced back to module-level models, and this relationship can be exploited to obtain meaningful coverage values for system tests.

(2) On the system level, tests are performed concurrently, following the *online testing* paradigm [23], where input data to the *system under test (SUT)* are calculated on the fly from a system model for each test step which is part of a test case. Also, the SUT's reactions are checked in real-time against the system model. A portion of these concurrent system tests will be performed using the original equipment in the real operational environment, while the rest is executed in cloud-based simulation environments. To coordinate this concurrent effort, an agent-based approach is used – we use the term *agent-based system testing (ABST)*. As pointed out in [22], the main advantage of using agents in testing is their ability to perform autonomous actions. They can decide to "push" test executions into specific directions, pursuing different goals, such as coverage maximisation, prioritisation, or investigation of critical functional aspects of the SUT.

Main Contributions

To the best of our knowledge, the approach discussed in this paper considers the following aspects for the first time. (1) The combination of complete testing strategies on the module level with complementary system tests whose degree of completeness can also be measured; (2) the agent-based approach to maximise system test coverage during online testing; (3) the investigation of the impact of applicable existing and future standards on the admissibility of cloud-based tests for the purpose of achieving certification credit; (4) the exploitation of test models created during module testing for the purpose of system test coverage assessment.

Overview

In Section 2, an example is presented, modelling an autonomous freight train controller. This example will be used in subsequent sections to illustrate the aspects of the comprehensive test approach advocated in this paper. In Section 3, we discuss complete test methods and their applicability to module tests in the cloud. We present our approach to agent-based system testing in the cloud and on the original target systems in Section 4 and show how results from module testing can be used to calculate coverage. In each of these sections, the certification-related aspects are discussed where appropriate. Conclusions and plans for future work are presented in Section 5.

Throughout the paper, we refer to related work where appropriate. The technical report [7] contains a comprehensive section on other work related to the approach advocated here. Moreover, this technical report contains the full model of the autonomous train controller presented in Section 2.

2 Running Example – Autonomous Freight Train Control System

System Description

Consider a control system for an autonomous freight train, with interfaces as depicted in Fig. 1. The controller is only active when powered ($\texttt{pwr} = 1$). Resetting the controller is performed by switching the power off and on again. The controller acts on the train engine with a simplified interface a carrying three commands a_- (negative acceleration, brake the train), 0 (no acceleration, keep current velocity – this state is called *coasting*), and a_+ (accelerate the train). For the sake of simplicity, only one deceleration and one acceleration value is considered. The decisions about braking or accelerating depend on several inputs. The *radio block centre (RBC)* sends a *movement authority (MA)* to proceed up to track coordinate x_B which is greater than the train's starting position x_A. The train is expected to proceed until x_B and stop there.[1] Conversely, the train transmits its current position estimate x to the RBC. Depending on this position, the RBC sends the actual maximum speed allowed (v_{Max}) to the train. An obstacle detection sensor sets controller input ω to 1 if an obstacle is detected on the track. In this situation, the train is expected to brake until it has come to a halt and/or until the obstacle has been removed. Three position sensors[2] provide their actual location estimates $x_i \in [x_A, x_B],\ i = 1,2,3$, together with confidence values $c_i \in [0,1]$. The train controller calculates a fourth location estimate based on its last location estimate, last velocity, and last acceleration. We assume that this estimate is associated with a constant confidence value c_4.

The train controller operates in processing cycles of constant time duration Δt (a typical value would be $\Delta t = 0.1\ \texttt{s}$). It manages a state tuple $(x, c, x_4, v, a, x_{\text{Stop}}, x_B)$. Here, x denotes the actual aggregated position estimate with its overall confidence value c. It is calculated from the sensor values x_1, x_2, x_3 and from variable x_4 according to Formula (5). Variable x_4 is the current position estimate derived from the physical equation (4): x_4 estimates the new position based on last position, speed, acceleration, and the time which has passed since the last estimate. Due to the wheel slip of trains, x_4 is only an approximation of the train's true position, and further sensors estimating the position from other sources are required. Variable v is the current velocity derived from the physical motion equation (6). Output variable a carries the current acceleration value in $\{a_-, 0, a_+\}$. Variable x_{Stop} stores the current estimate where the

[1] This part is slightly simplified: according to the ETCS standard [8], new movement authorities $x'_B > x_B$ may be received while the train is driving, so that a stop at x_B is not required.

[2] For our example, we assume three sensors obtaining location information, for example, from GPS, Balise, and distance radar.

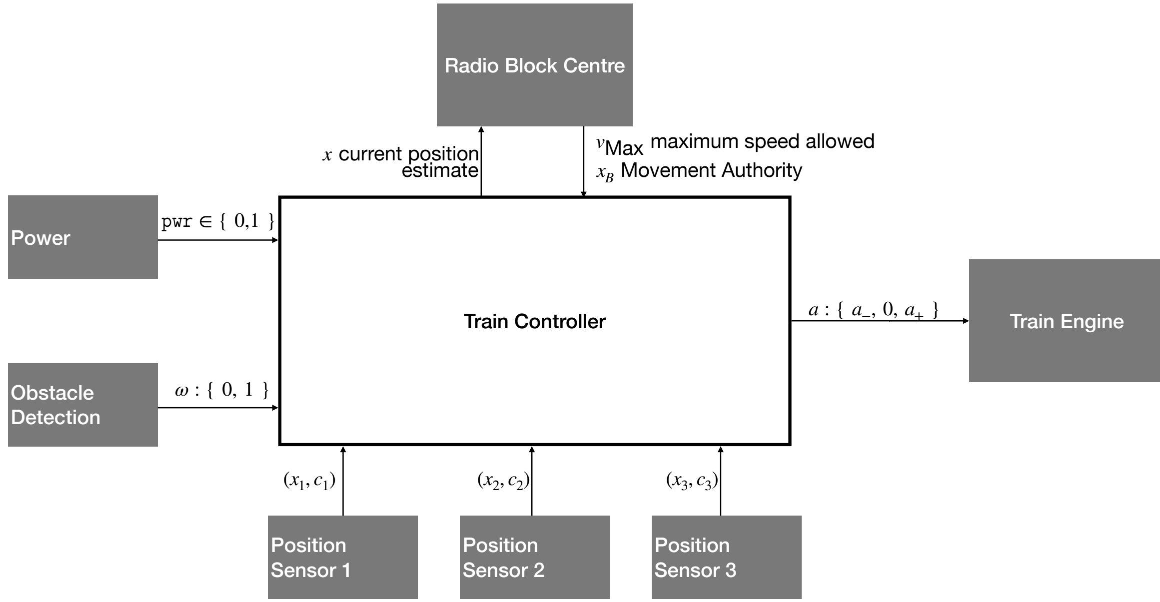

Figure 1: Train controller, location sensors, and engine interface.

train would stop, if braking would be started in the next processing cycle. Finally, x_B stores the current movement authority value. The initial state is $(x = x_A, c = 1, x_4 = x_A, v = 0, a = 0, x_{\text{Stop}} = x_A, x_B = x_A)$. As soon as a movement authority $x_B > x_A + \alpha$ is received, the train controller accelerates the train by setting $a = a_+$. The value α is a constant small distance representing the precision of a train's stopping position: if $|x_B - x| \leq \alpha$, the train is considered to be "close enough" to the position x_B and, therefore, loses its movement authority. When initialising the system, x_B is set to x_A, because no movement authority has been received yet. Conversely, condition $x_B > x_A + \alpha$ indicates that the distance to reach the authorised destination x_B is sufficiently far away from starting point x_A, so that the train should be put in motion. After each processing cycle, the location estimate x_4 and the speed estimate v are updated according to the physical equations for motion with constant acceleration:

$$\Delta x = v \cdot \Delta t + \frac{a}{2} \cdot \Delta t^2, \tag{1}$$

$$\Delta v = a \cdot \Delta t, \tag{2}$$

which imply

$$v + \Delta v = 2 \cdot \frac{\Delta x}{\Delta t} - v. \tag{3}$$

Accordingly, the controller assigns new values to x_4 by

$$x_4 := \underline{x} + v \cdot \Delta t + \frac{a}{2} \cdot \Delta t^2, \tag{4}$$

where $\underline{x}$ is the last overall position estimate x, calculated at the beginning of the Δt period.

The overall position estimate is calculated from the current position sensor values x_1, x_2, x_3 and x_4, taking into account their different confidence values by assigning:

$$x := \frac{\Sigma_{i=1}^{4} c_i \cdot x_i}{\Sigma_{i=1}^{4} c_i}. \tag{5}$$

The new speed estimate is calculated according to Equation (3) with $\Delta x = x - \underline{x}$, and assigned to v as

$$v := 2 \cdot \frac{x - \underline{x}}{\Delta t} - \underline{v}, \tag{6}$$

where $\underline{v}$ denotes the previous speed estimate. The overall confidence value for the position value obtained according to Equation (5) is

$$c = \frac{\Sigma_{i=1}^{4} c_i}{4}. \tag{7}$$

If the confidence value is acceptable ($c \geq c_{\text{Min}}$), then the train is accelerated until the maximum speed v_{Max} has been reached. After that, the train starts coasting. If the confidence value is too low ($c < c_{\text{Min}}$), the train is slowed down until it has reached a safe lower speed v_{Safe} until the position confidence is acceptable again.

With the values for position and speed at hand, the earliest possible stopping position x_{Stop} is calculated under the assumption that the train would keep its current acceleration in the actual processing cycle and start braking in the next cycle. For this calculation, the following assignment can be applied, using an auxiliary variable Δ_{Stop} indicating the duration until the train comes to a stop.

$$\Delta_{\text{Stop}} := -\frac{v + a \cdot \Delta t}{a_-}, \tag{8}$$

$$x_{\text{Stop}} := x + v \cdot \Delta t + \frac{a}{2} \cdot \Delta t^2 + (v + a \cdot \Delta t) \cdot \Delta_{\text{Stop}} + \frac{a_-}{2} \cdot \Delta_{\text{Stop}}^2 \tag{9}$$

$$= x + v \cdot \Delta t + \frac{a}{2} \cdot \Delta t^2 - \frac{a_-}{2} \cdot \Delta_{\text{Stop}}^2. \tag{10}$$

To understand Assignment (8), recall from Equation (2) that the speed changes to $v' = v + a \cdot \Delta t$ in the current processing cycle, where a is the current acceleration. From the next cycle on, the constant deceleration a_- will be applied, so the speed changes according to $\Delta v = a_- \cdot \tau$ over duration τ. Resolving formula $v' + \Delta v(\tau) = 0$ to τ, yields $\tau = \Delta_{\text{Stop}}$ from Assignment (8) for the duration to come to a halt from velocity v'. For understanding Assignment (10), recall from Equation (1) that the position x changes to $x' = v \cdot \Delta t + \frac{a}{2} \cdot \Delta t^2$ in the current processing cycle. After that, the speed v' has been reached, and deceleration a_- is applied. Applying Equation (1) with $v = v'$, $a = a_-$, and for the duration Δ_{Stop}, yields the stopping position used in Assignment (9). Observing that $(v + a \cdot \Delta t) = -\Delta_{\text{Stop}} \cdot a_-$, yields Equation (10).

When the estimated stopping position is only δ meters away from the destination x_B, further acceleration is forbidden. As soon as forecast x_{Stop} reaches the value of x_B, the train brakes until a very low speed v_{Min} has been reached, from where the train can stop "immediately", that is, within less than $\alpha = 0.6\ \mathtt{m}$. If the train has closed in on x_B within 0.6 m, it is braked again until it stops. Then, the train waits for a new movement authority to continue its journey.

Typical values for the constants and boundary variables mentioned in the requirements above are

$$\begin{array}{lllllllllll} a_+ & = & 1\ \mathtt{m/s}^2, & a_- & = & -1\ \mathtt{m/s}^2, & \Delta t & = & 0.1\ \mathtt{s}, & c_{\text{Min}} & = & 0.9 \\ v_{\text{Safe}} & = & 8\ \mathtt{m/s}, & v_{\text{Max}} & = & 22\ \mathtt{m/s}, & v_{\text{Min}} & = & 1\ \mathtt{m/s}, & \delta & = & 200\ \mathtt{m} \\ \alpha & = & 0.6\ \mathtt{m} \end{array}$$

It can be assumed that v_{Max} is always greater than v_{Safe}.

Formal Controller Model

The informal system description given above is now modelled using UML state machines [26]. The formal model semantics can be specified, for example, by associating a variant of Kripke structures with state machines, as described in [14]. In the following, we explain the behaviour formalised with these machines in an intuitive way.

The expected controller behaviour is modelled by two state machines. The first updates the variable vector $(x, c, x_4, v, a, x_{\text{Stop}}, x_B)$ defined above every Δt according to the equations listed above. This state machine is not shown, since it consists of a single state with a self loop triggered every Δt with an action that just performs the necessary assignments specified above.

Concurrently, the hierarchic state machine TRAIN CONTROLLER with its top-level machine specified in Fig. 2 is executed. Its normal behaviour is to transit after power on into state ACTIVE described by the lower-level state machine in Fig. 3. In the special situation where the controller is started with an obstacle in front (condition $\omega = 1$), it transits into submachine state OBSTACLE PRESENT. If the train is still driving, it will brake in state BRAKE FOR OBSTACLE until it has come to a standstill and state HALTED is entered. State ACTIVE will always be left when an obstacle is detected. It is visited again as soon as the obstacle has been removed ($\omega = 0$).

When submachine ACTIVE is entered, one or more choice states will be visited to decide which stable state should be chosen. (1) If no movement authority is available ($x_B - x \leq \alpha$), state WAIT FOR MA is entered. There, a moving train is braked to stop (this and other subordinate state machine diagrams are not shown here). State WAIT FOR MA is left as soon as a movement authority is available ($x_B - x > \alpha$). (2) If a movement authority exists and the train is still far enough from its destination ($x_B - x_{\text{Stop}} > \delta$), the controller branches into submachine DRIVING. There, the train will be accelerated to its maximum speed v_{Max}. The train will be slowed down if it is too fast and accelerated if it is slower than the maximum speed allowed. If the position confidence value c is too low, the controller transits to state SAFE DRIVING, where the train is kept at velocity v_{Safe} until the position confidence is acceptable again.

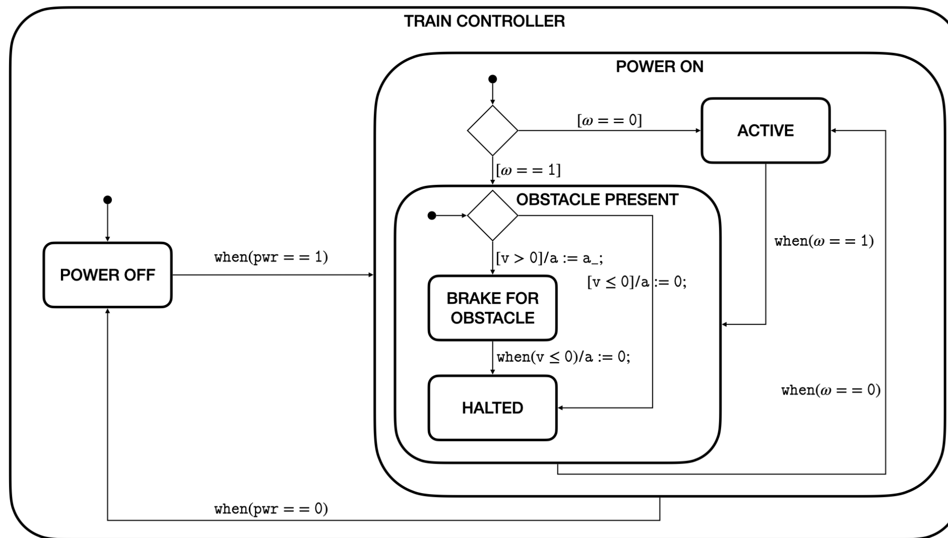

Figure 2: Train controller behaviour – top-level state machine.

(3) If the predicted stopping location comes as close as δ to x_B (change condition $x_B - x_{\text{Stop}} \in (0, \delta]$),

the train is not allowed to accelerate further. It will be kept at a constant velocity $v_{\text{const}} \le v_{\text{Max}}$ in state NO ACCEL (the associated submachine is not shown). (4) When it is time to brake ($x_B - x_{\text{Stop}} \le 0$), state BRAKE TO TARGET is entered, where the train is slowed down to a positive speed value v_{Min}. This positive speed value is maintained until the train is very close to its destination ($x_B - x \le \alpha$). Then, state STOP TRAIN is entered, where the train is slowed down to a halt.

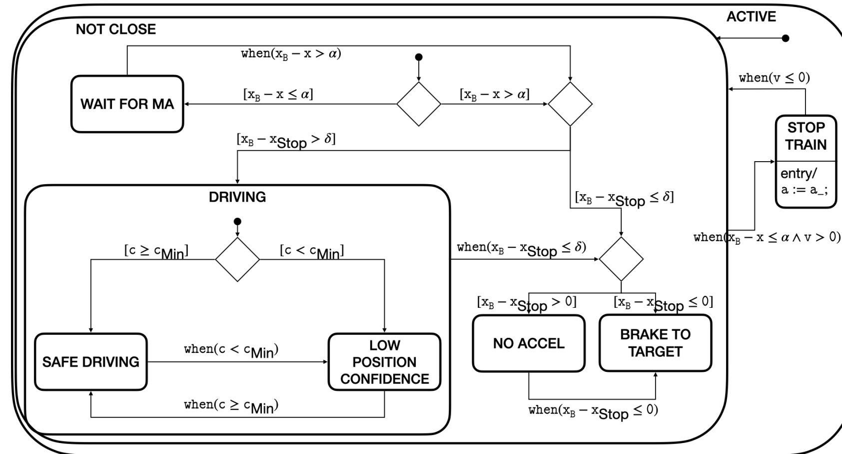

Figure 3: Train controller behaviour – active states.

3 Testing on the Module Level – Complete Methods

Complete Test Suites and Formal Methods

Since 2011, the standard RTCA DO-178C [35], which is applicable for the development and V&V of safety-critical avionic systems, contains guidelines for the application of formal methods, these are described in the supplement RTCA DO-333 [33]. This supplement lists model checking as one accepted formal verification method among others (e.g. theorem proving). The complete testing strategies advocated in this paper for module-level software verification can be regarded as model checking of code in two variants: (1) for *conformance checking*, *complete test suites* proving model conformance can be used. (2) For *property checking*, requirements for the given module need to be formalised (typically in a temporal logic, e.g. LTL). Then it can be verified again by means of complete test suites that the code fulfils the given formula, just as the reference model does. Completeness of a test suite asserts that every violation of model conformance or of the specified property will be uncovered, and that correct implementations will pass the test suite. Frequently, it suffices to apply *exhaustive* test suites. These guarantee to detect errors but may also reject correct implementations in certain situations.

In black-box testing, completeness or exhaustiveness can only be guaranteed under certain hypotheses, typically about the maximum number of states implemented in the SUT, and about the possible mutations of branching conditions and output expressions. For safety-critical systems development, however, software source code must be available for analysis. Therefore, the hypotheses about the SUT can be checked by means of various forms of static analysis. Consequently, complete or exhaustive test suites correspond to "real" code-level model checkers in the sense that the absence of failed test executions *proves* software correctness. Different from model checkers, the verification is performed by

dynamic execution of the test cases against the code, and not by static code analysis methods.

The autonomous train controller example introduced in Section 2 has typical characteristics occurring in many ATV control applications: inputs may be associated with conceptually infinite domains, such as real values for train position, but outputs are discrete, as the accelerate/coast/brake outputs to the train engine.[3] For this class of systems, complete test suites can be constructed in four steps [14, 15].

Step 1 The input equivalence classes are constructed.

Step 2 The system model is abstracted to a finite state machine (FSM) in Mealy style. The inputs of this FSM correspond to the input equivalence classes of the original system model, and the FSM outputs correspond to the discrete system outputs.

Step 3 A complete FSM test strategy is applied, resulting in a test suite with input sequences of equivalence class identifiers and expected outputs from the domain of system outputs.

Step 4 This FSM test suite is translated into a test suite for the original system, by selecting concrete representatives for each of the classes referenced in an FSM test case. The resulting test suite has the analogous completeness properties as the FSM test suite (this is the main result shown in [14, 15]).

Example 1. For the autonomous train controller, we consider two modules: the cyclic update machine C_0 providing new position and speed values on every Δt cycle, and the proper controller C_1 exercising the acceleration commands to the engine according to the SysML state machines shown in Fig. 2 and subsequent sub-machine models.

For controller module C_1, the SysML state machine model is associated with a formal behavioural model by means of a translation to a flattened *symbolic finite state machine (SFSM)* [21, 31]. These can be considered as a simpler subset of the more general Kripke structures used in [14, 15] for the elaboration of the complete testing theory. The SFSM representation has the advantage that the input equivalence classes can be easily calculated by enumerating all positive and negated conjunctions of the SFSM guards, as long as these have a model. For the train controller module C_1 shown in Fig. 2 and its sub-machines, this results in 28 input equivalence classes $c_i,\ i = 1,\dots,28$, of which we show several examples here:

$$
\begin{aligned}
c_1 &\equiv \texttt{pwr} = 0 \\
c_3 &\equiv \texttt{pwr} = 1 \land \omega = 1 \land 0 < v \\
c_5 &\equiv \texttt{pwr} = 1 \land \omega = 0 \land x_B - x > \alpha \land x_B - x_{\text{Stop}} > \delta \land c \geq c_{\text{Min}} \land 0 < v < v_{\text{Min}} \\
c_9 &\equiv \texttt{pwr} = 1 \land \omega = 0 \land x_B - x > \alpha \land x_B - x_{\text{Stop}} > \delta \land c \geq c_{\text{Min}} \land v > v_{\text{Max}} \\
c_{27} &\equiv \texttt{pwr} = 1 \land \omega = 0 \land x_B - x \leq \alpha \land x_B - x_{\text{Stop}} \leq 0 \land v = 0
\end{aligned}
$$

Input equivalence class c_1 is specified by all input tuples where $\texttt{pwr} = 0$; all other inputs can have arbitrary values. This holds because the other inputs do not affect the behaviour of C_1, as long as the power is off. In contrast to this, the predicate specifying class c_9 consists of six conjuncts involving all inputs to C_1. It captures the situation where the train is still sufficiently far away from the target station and the position confidence is high, so that it may drive with maximum speed, but the train is currently overspeeding. Note that all conjuncts come from (possibly negated) guard conditions in the state machine.

[3] Other examples of systems of this kind are airbag controllers, thrust reversal controllers in aircraft, or safety controllers for robots interacting with humans.

Applying the abstraction principle described in [14, 15] results in an FSM with input alphabet $c_1,\dots,c_{28}$, output alphabet $a := a_0, a := a_-, a := a_+$, and six states. To this FSM, we apply the well-known complete H-Method [6]. This method produces test suites guaranteed to uncover all violations of observational equivalence, provided that an upper bound of the internal number of states in the implementation is known. Since we assume that the source code of C_2 is available (this would always be required for safety-critical control components), the number of control states used in the C_2 implementation can be extracted by means of static analysis. If the code is directly generated from the SFSM, it is guaranteed that the implementation also has only six non-equivalent control states. Under this assumption, application of the H-method results in approximately 600 test cases[4]. As an example of these generated test cases, consider the following one, consisting of two steps.

$$(c_{27}/\mathtt{a} := \mathtt{0}).(c_5/\mathtt{a} := \mathtt{a}_+)$$

Translation of this FSM test case to a concrete test case, which can be executed against the target system, requires to select representatives from input classes c_{27} and c_5 specified above. An example for such a concrete test case is (we assume that $x_A = 0$)

$$(\mathtt{pwr} = 1 \wedge \omega = 0 \wedge x_B = 0 \wedge x = 0 \wedge x_{\text{Stop}} = 0 \wedge v = 0/\mathtt{a} := \mathtt{0}).$$
$$(\mathtt{pwr} = 1 \wedge \omega = 0 \wedge x_B = 10000 \wedge x = 0 \wedge x_{\text{Stop}} = 0 \wedge c = 0.95 \wedge v = 0.01/\mathtt{a} := \mathtt{a}_+).$$

In the first step, the train controller is powered, and no obstacle is present. The speed of the train is zero, and there is no movement authority available ($x_B = 0$). Thus, it is expected that the requested acceleration is zero, i.e. $\mathtt{a} := \mathtt{0}$. In the second test step, a movement authority arrives ($x_B = 10000$), so it is expected that the train starts to accelerate, i.e. $\mathtt{a} := \mathtt{a}_+$.

The main results of [14, 15] imply that the test suite, where all test cases of the abstract FSM suite are made concrete as illustrated in the example case above, is also complete under certain alternative sets of sufficient conditions. The simplest condition set requires that the implementation does not introduce additional states, and that the implementation uses the same guard conditions as the SFSM. Both conditions can be checked by means of static analysis. Note that, if these conditions apply, it always suffices to select a single representative from each input equivalence class c, and use this representative for all concrete test cases where c is referenced in a test step.

The longest test cases of this complete test suite have only four steps: every state of the minimised FSM can be reached in at most two steps, when starting from the initial state. From every state, every input needs to be exercised; this adds one further step to the test case. Then, the target state reached needs to be distinguished from other states. This can be performed for the train controller by traces of length one. This adds up to test cases of maximum length four. □

It has been criticised in the past that complete test suites turn out to be too big to be applied in practice to systems of realistic complexity. We agree that complete test suites should not be applied to system level testing. However, for software components of a complex system, these test suites turn out to be feasible from today's perspective, for the following reasons: (1) As shown in [16, 30], the construction of equivalence classes reduces the test suite size significantly without impairing the suite's completeness properties. (2) For the application of complete test suites as described here, we need not formulate hypotheses about potential additional control states in the implemented software, because the number

[4]The number of test cases needed usually varies with the method implementation. We have used here the open source library `libfsmtest` for model-based testing with FSMs, which is available from https://bitbucket.org/JanPeleska/libfsmtest.

of these states can be extracted from the code by static analysis. Consequently, the test suite size only depends polynomially on the number of states [5] (the size would grow exponentially in the potential number of *additional* control states present in the implementation). (3) The execution of software tests can be performed fully automatically in the cloud, so that many test cases can be executed in parallel. For these reasons, the original criticism of complete test suites no longer applies today.

In summary, the complete test suites advocated here are suitable for verifying the logical correctness of the source code, but they are not intended for receiving certification credits regarding the correctness of the integrated hardware/software (HW/SW) system. In the approach advocated here, this integration aspect is shifted to the system test level.

Model-Conformance vs. Requirements Satisfaction – Traceability Issues

The supplement to RTCA DO-178C on model-based development and V&V states [34, MB.6.7]: *"Model coverage analysis does not eliminate the need for traceability analysis between requirements from which the model was developed and the model."* This has an implication on the model-based testing approach which – to the best of our knowledge – has not received sufficient attention in the research communities: traceability demands that requirements are related to the model portions realising these requirements. Then, the code has to be related to the model parts it implements. Finally, the test cases checking the code need to trace back to the requirements they intend to verify.[5] It does *not* suffice to argue that the formal model (in our case, the SysML state machines) has been reviewed (or even formally verified) and found to reflect all requirements correctly and then conclude that, since the code has been shown by complete test suites to be equivalent to the model, it must implement all requirements "somehow" in the proper way.

For modelling with SysML, traceability between model elements and requirements can be directly represented by means of the so-called «satisfy» relationship linking behavioural or structural model elements to requirements [25]. It has been shown in [29] how this information can be exploited to create formal representations of requirements in a temporal logic like LTL. The atomic propositions of the LTL formulae are quantifier-free first-order expressions over model variables, including inputs and outputs. The validity of a formula can be decided by abstracting model computations to the sequences of sets containing the formula's atomic propositions valid in the respective computation step, see Section 3.2.2 of [1]. In [21], it has been shown that for given sets P of atomic propositions exhaustive test suites can be constructed, which frequently require fewer test cases than needed for establishing full model equivalence. Exhaustiveness in this situation means that the test suite is guaranteed to fail for at least one test case, if the computations of the implementation, when abstracted to traces over sets of elements from P, contain traces that are not produced by the reference model. Therefore, passing the test suite implies that the implementation fulfils all LTL properties over atomic propositions from P that are also fulfilled by the reference model.

We conclude that the requirements-based verification tasks imposed by the RTCA DO-178C standard can also be handled by means of exhaustive test suites. We suggest that on the module level two complete test suites should be executed: the first, with the objective to prove model equivalence, enables us to calculate a meaningful test coverage measure for system tests, as will be explained in the next section. The second, requirements-driven test suite, is essential to satisfy the traceability requirements of the standard.

[5] This can be done either indirectly through the traceability chain *test case* $\longrightarrow$ *code* $\longrightarrow$ *model* $\longrightarrow$ *requirement* or directly from test case to requirement.

4 System-Level Testing

Meaningful System Tests

Besides checking the correctness of the overall system integration, a major objective of system testing is to verify the complete workflow of services and applications, potentially across several communicating controllers and the respective interfaces involved. Tests designed for this purpose are usually called *end-to-end (E2E) tests*. Analysing the module tests discussed in the previous section, it becomes apparent that they do *not* represent meaningful E2E tests. This is illustrated by the following example.

Example 2. Consider one of the module tests of maximum length four from the complete test suite discussed in Section 3. We present here the symbolic version with input equivalence classes, and not the concrete version, where atomic propositions have been resolved by selecting concrete values.

Step 1. $(\texttt{pwr} = 1 \land \omega = 0 \land x_B - x > \alpha \land x_B - x_{\text{Stop}} > \delta \land c \geq c_{\text{Min}} \land v = 0 / \texttt{a} := \texttt{a}_+)$.
Step 2. $(\texttt{pwr} = 1 \land \omega = 0 \land x_B - x \leq \alpha \land 0 < v / \texttt{a} := \texttt{a}_-)$.
Step 3. $(\texttt{pwr} = 1 \land \omega = 0 \land x_B - x > \alpha \land x_B - x_{\text{Stop}} > \delta \land c \geq c_{\text{Min}} \land v = 0 / \texttt{a} := \texttt{a}_+)$.
Step 4. $(\texttt{pwr} = 1 \land \omega = 0 \land x_B - x > \alpha \land x_B - x_{\text{Stop}} > \delta \land c \geq c_{\text{Min}} \land v = v_{\text{Min}} / \texttt{a} := \texttt{a}_+)$

The first test step initiates a power up in a situation where movement authority is already available ($x_B - x > \alpha$). The train is expected to accelerate. Step 2, however, initiates a robustness transition in the controller, because the train location appears to be close to the destination ($x_B - x \leq \alpha$), without having first fulfilled condition $x_B - x_{\text{Stop}} \leq \delta$ which would be expected in a "physically reasonable" execution. The train is expected to brake. While this transition is contained in the controller model to ensure its robustness, its occurrence is highly unlikely, so that it would not be selected for a "realistic" E2E test, but only for a robustness test. It would suffice, however, to check this aspect of robustness at the module level. In Step 3, a situation is described again where the train is further away from its destination – this is expressed by conditions $x_B - x > \alpha \land x_B - x_{\text{Stop}} > \delta$. If we just selected a smaller value of x to fulfil these conditions, this would again result in a robustness transition: the position sensors provide an improved position value, while the previous one for Step 2 was too high. Alternatively, this step could be realised by increasing the value of x_B. This would correspond to a realistic system test step, where a new movement authority is obtained by increasing the destination value x_B. Obviously, the distinction between a robustness test step and a normal behaviour test step suitable for an E2E test cannot be made on the basis of the equivalence class formulae alone. While this distinction is not relevant for module testing, it has high significance for the design of meaningful E2E tests. In the last step, the input is just updated according to the expected change of velocity: if the speed was zero before and the train keeps accelerating, the velocity will increase to v_{Min}.

The test case now ends while the train is still accelerating. For a suitable E2E test, we would expect a continuation with further test steps, leading the train to its destination x_B. Such continuations, however, do not exist in the module test suite, because every test case there starts from the initial controller state by powering the system.

Obviously, the physical model connecting acceleration, velocity, and position (see Section 2) needs to be considered for constructing meaningful system tests in a simulation environment: Step 3, for example, can only be executed when the train has come to a standstill ($v = 0$), and the point in time when this happens depends on the effect of the negative acceleration a_- and the actual speed value when the brakes had been triggered in Step 2.

For system tests on target hardware and in the real operational environment, these simulations of the physical world are unnecessary, but the trigger conditions for each new test step need to be monitored.

Here, Step 3 can only be triggered *after having observed* that the train has come to a standstill. Then, the new value of x_B can be provided to the SUT, so that $x_B - x > \alpha \wedge x_B - x_{\text{Stop}} > \delta$ is fulfilled. Moreover, we observe that system testing with the target system and its operational environment needs to be *intrusive*, if different confidence values of the position sensors and erroneous position values need to be tested. 'Intrusive' means that the behaviour of the position sensors can be changed for test purposes, for example, by corrupting position values and lowering the actual confidence values calculated by the sensors. Finally, the occurrence and removal of obstacles has to be controlled (for example, remotely controlled vehicles could be moved on the track and removed at a later point in time). □

Summarising the findings highlighted by this example, we conclude that (1) Module test suites are not well-suited to be turned into meaningful E2E tests. (2) The formula representations of input equivalence classes do not provide sufficient information about which concrete solutions would give meaningful test data for E2E testing. (3) Test environments for the simulated execution of system tests require components observing the applicable physical laws during test execution. (4) System tests with the "real" target in its operational environment require intrusive test equipment for some input interfaces to the SUT.

Finding number one can be justified more formally. When testing for model conformance, the complete strategies applied usually strive to reach every state of the SUT on the shortest path possible, and then to execute every possible input from this state. Consequently, breadth-first-search algorithms are applied. This has the effect that transitions that should only be performed in exceptional cases are frequently taken if they reside on the shortest path to such a state. Moreover, the objective to exercise every possible input (or input class) in a given state necessarily leads to frequent resets, leading to the start of a new test case. In contrast to this, E2E tests need to be identified by a depth-first search through a system model; this leads to longer test cases starting and ending in system states that are meaningful from the perspective of the tested application.

As a consequence of these observations, we see that it is generally impossible to "glue" several module tests together and come out with a meaningful E2E test. We further observe that the optimal error detection capabilities of complete test suites are unrelated to their end-to-end test quality.

System Test Scenarios

System tests of autonomous trains need to be executed in different *scenarios*, where each scenario is specified by a route through the rail network and a set of events (low confidence positions, obstacles, overspeeding) to occur at certain points while traversing the route. This concept is similar to scenario-based testing of autonomous road vehicles [17], but considerably simpler, since the train movements are restricted by the rail network, and the absence of collision hazards involving other trains is already ensured by the interlocking system.

To allow for automated test case derivation for E2E system tests, we introduce a data structure called *symbolic scenario test tree (SSTT)*. These trees are interpreted as *hybrid systems* [11] which are simplified in the sense that our outputs are always concrete assignments instead of being implicitly specified through jump conditions. The use of these trees is exemplified with the tree fragment shown in Fig. 4.

Just as the control modes of hybrid system models, the nodes of an SSTT represent invariants over both discrete and time-continuous variables. The invariants involving discrete variables can be directly derived from the state machine models created for the controller modules. The time-continuous evolution of physical inputs, such as location and velocity in relationship to acceleration and passing time, need to be derived from the physical laws applicable in the system environment. The time-dependent invariants shown in the states of Fig. 4 are all implied by the physical laws presented in Section 2.

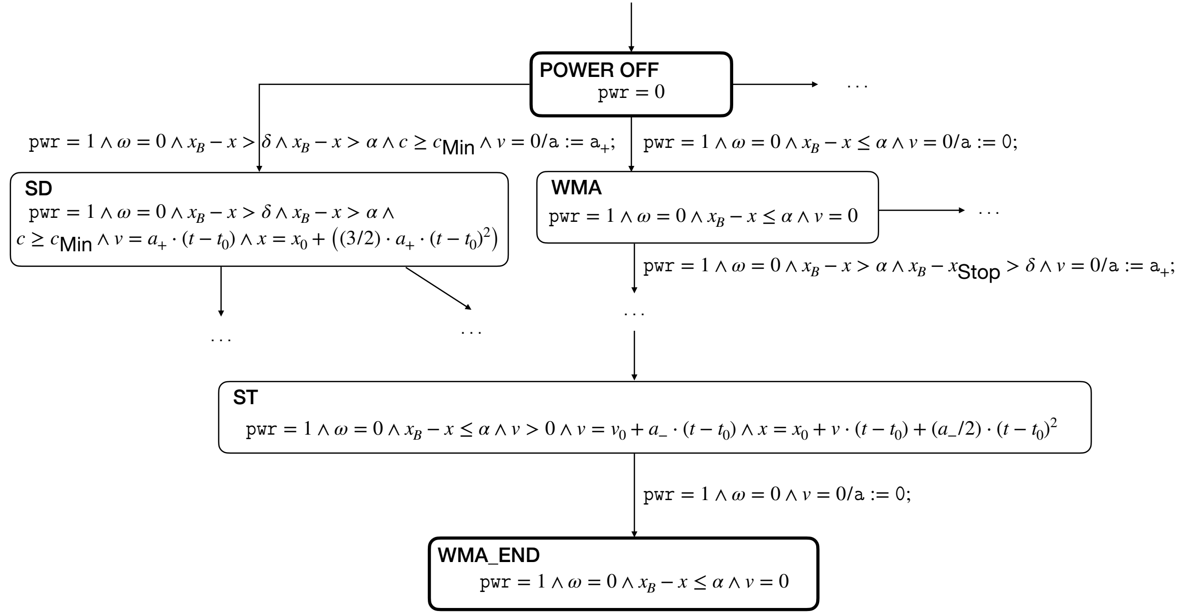

Figure 4: Partial representation of a symbolic scenario test tree (SSTT) for the autonomous train control system.

The edges of an SSTT are labelled with guard conditions and output assignments, as in symbolic finite state machines. In Fig. 4, the root of the tree is represented by node POWER OFF, which only has the invariant $\mathtt{pwr} = 0$. From there, edges lead to different child nodes, depending on the guard conditions labelling these edges. The edge to node WMA is taken when the power is switched on, no obstacle is present, the train is without movement authority ($x_B - x \leq \alpha$), and it is not moving. Therefore, the acceleration can be set to 0, and the system resides in state WMA until a movement authority arrives. There are several situations to distinguish: the movement authority might arrive with an obstacle present, or it might arrive in a position that is already quite close to the destination. The downward edge shown in Fig. 4 specifies the situation where no obstacle is present, and the train is not yet required to brake for the target destination ($x_B - x_{\mathrm{Stop}} > \delta$). The edge from node POWER OFF to node SD specifies the situation where the train is powered, while a movement authority is already available, so that the train is allowed to accelerate, directly leaving its standstill position. If the invariants need to refer to variable values at node entry, these are denoted by $< \mathtt{variable} >_0$. In node SD, for example, $(t - t_0)$ denotes the time which has passed since entering the node. Towards the leaves of the tree, the nodes describe the situations where the train approaches the destination x_B and brakes until it comes to a standstill. This is exemplified here by nodes ST ("stop train") and WMA_END ("wait for next movement authority").

As can be seen in this example, the SSTT contains the necessary information about physical laws to be applied. These were missing in the module test models. Moreover, the SSTT is created in such a way that robustness transition sequences that are better tested on the module level are left out, and that the leaves of the tree represent suitable target states from the perspective of E2E testing. The example also indicates that these trees can become quite large. In part, they can be automatically derived from the state machines specified for the controller modules in combination with rules about applicable control laws.

The information about sequences of transitions that are only taken in robustness situations, however, need to be provided by system experts. Moreover, the suitable termination states for E2E testing need to be manually provided. In [29], methods for automated elaboration of requirements formulae from SysML models have been presented. With these (LTL-) formulae at hand, it is at least possible to check automatically whether the SSTT covers a certain requirement. This, however, does not yet imply that the respective path through the tree also represents a meaningful E2E test. For these reasons, we expect that SSTT creation can only be partially automated and will always require inputs from experts.

Cloud-based Tests vs. Tests on the Target System – System Test Coverage

Cloud-based testing significantly increases the availability of hardware resources for test campaigns, and improves the possibilities to extend or shrink these resources according to the demands of such campaigns. It has to be discussed, however, how certification credit may be obtained for tests performed in the cloud, since cloud platforms do not allow for execution of SUTs with their original hardware. The standard supplement RTCA DO-333 regulating the use of formal methods [33, FM.6.3.1.c] states that formal verification results can also verify aspects of compatibility of software and hardware, if the underlying formal models also cover the hardware platform. For specific verification objectives like worst-case execution time analysis, this possibility has already been successfully exploited [37]. For system testing in the cloud, more general hardware models are needed, so that control applications can be executed on these models with their original machine code and address maps, as used on the real target. Such models have recently been developed; they are called *virtual prototypes* [12] and represent more advanced variants of the simulators that have been used for quite some time. We expect that certification credit for cloud-based system tests or HW/SW integration tests can be achieved if the software is executed in such a variant of virtual prototypes. This option, however, requires a new version of the standards and would certainly require *tool qualification* [36] for the virtual prototypes, to justify that they really represent a faithful model of the target HW.

For good reasons, we can also expect that in the future at least some tests will be required to be performed on the "real" target system with its integrated hardware and software: RTCA DO-333 states [33, FM.6.7] *"Tests executed in target hardware are always required to ensure that the software in the target computer will satisfy the high-level requirements ... "*. We expect that system-level tests performed on the target system will be regarded as more valuable than module tests on target HW, since system tests exercise more aspects of drivers and firmware than module tests that often only cover software interfaces and do not stimulate drivers and firmware at all. Since the current standard ISO 26262 applicable in the automotive domain refers to RTCA DO-178 when suggesting V&V methods, we expect that the verification approaches accepted for the avionic domain will also remain admissible for the automotive domain in the future. In the railway domain, the applicable standard EN 50128 [3] also regulates (even requires) the use of formal methods, but is less explicit than RTCA DO-178C with respect to formal hardware models and the amount of testing to be performed on the target system. With these standard-related facts in mind, we conclude that the most specific guidelines for a formal approach to testing in simulated and in target environments are given by the avionic standards.

According to these standards, system requirements shall all be covered by system-level tests. As discussed in Section 3, requirements can be transformed into LTL formulae, and the symbolic scenario test tree introduced earlier contains sufficient information about valid atomic propositions along each path to decide with an automated procedure which formulae are fulfilled on each path.[6] Uncovered

[6] Note that this procedure requires a *vacuity analysis* [2]: a requirement of the form $\mathbf{G}(\varphi \Rightarrow \psi)$ may be satisfied on a path

requirements may both lead to new branches to be added into a scenario test tree and to the creation of new scenarios, if the existing ones cannot accommodate paths where a given formula can be checked non-vacuously.

The new aspect not covered by current standards is the fact that a major portion of the system tests will be executed in the cloud, and not on the target within its operational environment. We expect that future revisions of the standards will demand that "a reasonable portion" of these tests should be executed on the target. Then, the question that remains to be answered is which sub-collection of system tests represents such a reasonable portion. When the approach advocated in this paper is followed, a sound answer to this question can be given: system testing on the target should cover the normal behaviour transitions of the SFSM models used for module-level testing. This is justified since we have shown that the software conforms to its SFSM models, so SFSM transition coverage implies software transition coverage. Consequently, transition coverage of SFSMs implies that all relevant transitions of the software have been exercised *on the target system*. This is a very comprehensive check of the HW/SW integration correctness. We suggest to exempt robustness transitions from this coverage requirement, since many robustness aspects can be better tested on the module level, avoiding too many variants of intrusive tests on the system level. Another suggestion is to let the required degree of transition coverage depend on the *design assurance level (A — D)*, that is, the SUT criticality. This concept is currently adopted in [35] for the code coverage required to get certification credit.

Agent-based Approach and Online Testing

The preceding discussion of system tests suggests a testing environment where multiple concurrent components are deployed that are responsible for the different aspects identified above. (1) Coverage analysis and coordination of concurrent test executions. (2) Control of an individual system test execution. (3) Simulation of the physical environment in real time or simulated time (for cloud-based system tests). (4) Interfacing to the target system in its operational environment (transmission of movement authorities, intrusive influence of the position sensors, and obstacle creation and removal).

For realising these testing environments, we apply agent-based system testing (ABST). To this end, a multi-agent system (MAS) is realised according to the paradigm of *mixed initiative control*. As stated in [9], this form of agent collaboration is characterised by *"allowing a supervisor and group of searchers to jointly decide the correct level of autonomy for a given situation"*. In [9], this paradigm was intended to regulate the collaboration between multiple robots and a human operator. In our context here, a coordinator test agent can monitor the transition coverage achieved so far and delegate the respective test steps suitable for covering missing transitions to agents executing tests on the target and its operational environment. Two further classes of test agents simulate the physical environment and interface to the SUT hardware for tests on the target. It has already been shown in [4], how MAS technology helps to increase the test strength of system test cases by letting agents execute strategies for selecting test data which is most effective in the current situation of a system test execution.

In [23], the authors introduced the term *online testing* (or *on-the-fly testing*) for model-based tests combining the generation of test steps of a test case with the actual execution and checking of expected results.[7] Online testing is very well-suited for the objectives outlined in this paper, because in the presence of potentially millions of system test cases to be executed, a separate test generation phase preceding

---

π of the tree just because φ never becomes true. In this case, the path π would not be a suitable witness for testing this requirement.

[7] This concept has already been used in 1996 in the VVT-RT Tool [27] whose commercial version is called RT-Tester today www.verified.de.

the test execution might take too much time, whereas online testing delivers information about passed and failed test steps right from the start.

Moreover, online testing helps to let the SSTT grow incrementally, instead of creating the complete tree beforehand. The coordinator test agent can initialise the SSTT with just as many different paths as test execution agents are available. Monitoring the module transition coverage achieved with these system test scenarios, the agent can let the SSTT "grow" by adding new paths that are suitable to cover just the unvisited module transitions.

5 Conclusion and Future Work

In this paper, we have proposed a novel approach for efficient module testing and system testing of autonomous transportation systems (vehicles, trains, aircraft). Techniques of this kind are of considerable importance, because the number of tests to be performed on autonomous systems is so much higher in comparison to test suites for conventional systems. Therefore, these test campaigns cannot be performed in acceptable time with conventional methods, since the latter impose system tests to be executed with original equipment only. The key characteristics of the approach proposed here are (1) the use of complete test strategies on the module level, (2) concurrent online systems tests executed on original equipment and in cloud-based simulation environments, and (3) the use of test agents to coordinate the system test effort. The consistency of this approach with applicable standards has been explained.

Two important aspects of testing autonomous systems were beyond the scope of this paper. The first is the fact that many autonomous systems are so complex that they can no longer be represented by comprehensive models. Instead, they are specified in scenario libraries, and this leads to the questions of scenario completeness and consistency [10, 28]. The second aspect concerns the assurance to be provided for applications based on neural networks and multi-agent systems, because their behaviour cannot be specified and verified with conventional methods, and their behaviour may evolve over time, due to machine learning effects and on-the-fly modification or even creation of plans [32].

Future work will focus on the detailed evaluation of the approach advocated in this paper. At first, an in-depth analysis for the autonomous train control system presented here will be performed. Next, applications from the robotic domain (human-robot collaboration) and the avionic domain, such as autonomous taxiing, take-off, and landing or formation flight with the objective of fuel saving will be investigated. Future work will also cover distributed autonomous systems under test, based on existing results such as [13, 24].

References

[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT Press.

[2] Thomas Ball & Orna Kupferman (2008): *Vacuity in Testing*. In Bernhard Beckert & Reiner Hähnle, editors: *Tests and Proofs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 4–17.

[3] CENELEC (2011): *EN 50128:2011 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*.

[4] Greg Chance, Abanoub Ghobrial, Séverin Lemaignan, Tony Pipe & Kerstin Eder (2020): *An Agency-Directed Approach to Test Generation for Simulation-based Autonomous Vehicle Verification*. In: *IEEE International Conference On Artificial Intelligence Testing, AITest 2020, Oxford, UK, August 3-6, 2020*, IEEE, pp. 31–38, doi:10.1109/AITEST49225.2020.00012.

[5] Tsun S. Chow (1978): *Testing Software Design Modeled by Finite-State Machines*. *IEEE Transactions on Software Engineering* SE-4(3), pp. 178–186.

[6] Rita Dorofeeva, Khaled El-Fakih & Nina Yevtushenko (2005): *An Improved Conformance Testing Method*. In Farn Wang, editor: *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, *Lecture Notes in Computer Science* 3731, Springer, pp. 204–218, doi:10.1007/11562436_16.

[7] Kerstin Eder, Wen-ling Huang & Jan Peleska (2021): *Complete Agent-driven Model-based System Testing for Autonomous Systems – Technical Report*, doi:10.5281/zenodo.5203111.

[8] European Railway Agency (2012): *ERTMS – System Requirements Specification – UNISIG SUBSET-026*. Available under http://www.era.europa.eu/Document-Register/Pages/Set-2-System-Requirements-Specification.aspx.

[9] Benjamin Hardin & Michael A. Goodrich (2009): *On Using Mixed-Initiative Control: A Perspective for Managing Large-Scale Robotic Teams*. In: *Proceedings of the 4th ACM/IEEE International Conference on Human Robot Interaction*, HRI '09, Association for Computing Machinery, New York, NY, USA, p. 165–172, doi:10.1145/1514095.1514126.

[10] Florian Hauer, Tabea Schmidt, Bernd Holzmüller & Alexander Pretschner (2019): *Did We Test All Scenarios for Automated and Autonomous Driving Systems?* In: *2019 IEEE Intelligent Transportation Systems Conference, ITSC 2019, Auckland, New Zealand, October 27-30, 2019*, IEEE, pp. 2950–2955, doi:10.1109/ITSC.2019.8917326.

[11] T.A. Henzinger (1996): *The theory of hybrid automata*. In: *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, pp. 278–292.

[12] Vladimir Herdt, Daniel Große, Pascal Pieper & Rolf Drechsler (2020): *RISC-V based virtual prototype: An extensible and configurable platform for the system-level*. *J. Syst. Archit.* 109, p. 101756, doi:10.1016/j.sysarc.2020.101756.

[13] Robert M. Hierons (2016): *A More Precise Implementation Relation for Distributed Testing*. *Comput. J.* 59(1), pp. 33–46, doi:10.1093/comjnl/bxv057.

[14] Wen-ling Huang & Jan Peleska (2016): *Complete model-based equivalence class testing*. *Software Tools for Technology Transfer* 18(3), pp. 265–283, doi:10.1007/s10009-014-0356-8.

[15] Wen-ling Huang & Jan Peleska (2017): *Complete model-based equivalence class testing for nondeterministic systems*. *Formal Aspects of Computing* 29(2), pp. 335–364, doi:10.1007/s00165-016-0402-2.

[16] Felix Hübner, Wen-ling Huang & Jan Peleska (2019): *Experimental evaluation of a novel equivalence class partition testing strategy*. *Software & Systems Modeling* 18(1), pp. 423–443, doi:10.1007/s10270-017-0595-8. Published online 2017.

[17] Hardi Hungar (2018): *Scenario-Based Validation of Automated Driving Systems*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*, Lecture Notes in Computer Science, Springer International Publishing, pp. 449–460.

[18] ISO (2021): *ISO/DIS 21448: Road vehicles – Safety of the intended functionality*. ICS: 43.040.10, Draft International Standard.

[19] ISO/DIS 26262-4 (2009): *Road vehicles – functional safety – Part 4: Product development: system level*. Technical Report, International Organization for Standardization.

[20] Nidhi Kalra & Susan M. Paddock (2016): *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, Santa Monica, CA, doi:10.7249/RR1478.

[21] Niklas Krafczyk & Jan Peleska (2021): *Exhaustive Property Oriented Model-based Testing With Symbolic Finite State Machines (Technical Report)*, doi:10.5281/zenodo.5151778. Funded by the Deutsche Forschungsgemeinschaft (DFG) – project number 407708394.

[22] Pavithra Perumal Kumaresen, Mirgita Frasheri & Eduard Paul Enoiu (2020): *Agent-Based Software Testing: A Definition and Systematic Mapping Study*. In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 24–31, doi:10.1109/QRS-C51114.2020.00016.

[23] Kim Guldstrand Larsen, Marius Mikucionis & Brian Nielsen (2004): *Online Testing of Real-time Systems Using* UPPAAL. In Jens Grabowski & Brian Nielsen, editors: *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, *Lecture Notes in Computer Science* 3395, Springer, pp. 79–94, doi:10.1007/978-3-540-31848-4_6.

[24] Bruno Lima, João Pascoal Faria & Robert M. Hierons (2020): *Local Observability and Controllability Analysis and Enforcement in Distributed Testing With Time Constraints*. *IEEE Access* 8, pp. 167172–167191, doi:10.1109/ACCESS.2020.3021858.

[25] Object Management Group (2015): *OMG Systems Modeling Language (OMG SysML), Version 1.4*. Technical Report, Object Management Group. Http://www.omg.org/spec/SysML/1.4.

[26] Object Management Group (2017): *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.5.1*. Technical Report, OMG.

[27] Jan Peleska (1996): *Test Automation for Safety-Critical Systems: Industrial Application and Future Developments*. In Marie-Claude Gaudel & Jim Woodcock, editors: *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*, *Lecture Notes in Computer Science* 1051, Springer, pp. 39–59, doi:10.1007/3-540-60973-3_79.

[28] Jan Peleska (2020): *New Distribution Paradigms for Railway Interlocking*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, *Lecture Notes in Computer Science* 12478, Springer, pp. 434–448, doi:10.1007/978-3-030-61467-6_28.

[29] Jan Peleska, Jörg Brauer & Wen-ling Huang (2018): *Model-Based Testing for Avionic Systems Proven Benefits and Further Challenges*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, *Lecture Notes in Computer Science* 11247, Springer, pp. 82–103, doi:10.1007/978-3-030-03427-6_11.

[30] Jan Peleska, Wen-ling Huang & Felix Hübner (2016): *A Novel Approach to HW/SW Integration Testing of Route-Based Interlocking System Controllers*. In Thierry Lecomte, Ralf Pinger

& Alexander Romanovsky, editors: *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* 9707, Springer, pp. 32–49, doi:10.1007/978-3-319-33951-1_3.

[31] Alexandre Petrenko (2016): *Checking Experiments for Symbolic Input/Output Finite State Machines*. In: *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, IEEE Computer Society, pp. 229–237, doi:10.1109/ICSTW.2016.9.

[32] Youcheng Sun, Hana Chockler, Xiaowei Huang & Daniel Kroening (2020): *Explaining Image Classifiers Using Statistical Fault Localization*. In Andrea Vedaldi, Horst Bischof, Thomas Brox & Jan-Michael Frahm, editors: *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXVIII*, *Lecture Notes in Computer Science* 12373, Springer, pp. 391–406, doi:10.1007/978-3-030-58604-1_24.

[33] RTCA SC-205/EUROCAE WG-71 (2011): *Formal Methods Supplement to DO-178C and DO-278A*. Technical Report RTCA/DO-333, RTCA Inc, 1150 18th Street, NW, Suite 910, Washington, D.C. 20036-3816 USA.

[34] RTCA SC-205/EUROCAE WG-71 (2011): *Model-Based Development and Verification Supplement to DO-178C and DO-278A*. Technical Report RTCA/DO-331, RTCA Inc, 1150 18th Street, NW, Suite 910, Washington, D.C. 20036-3816 USA.

[35] RTCA SC-205/EUROCAE WG-71 (2011): *Software Considerations in Airborne Systems and Equipment Certification*. Technical Report RTCA/DO-178C, RTCA Inc, 1150 18th Street, NW, Suite 910, Washington, D.C. 20036-3816 USA.

[36] RTCA SC-205/EUROCAE WG-71 (2011): *Software Tool Qualification Considerations*. Technical Report RTCA/DO-330, RTCA Inc, 1150 18th Street, NW, Suite 910, Washington, D.C. 20036-3816 USA.

[37] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat & Per Stenström (2008): *The worst-case execution-time problem - overview of methods and survey of tools*. *ACM Trans. Embed. Comput. Syst.* 7(3), pp. 36:1–36:53, doi:10.1145/1347375.1347389.