Complete Property-Oriented Module Testing

Felix Brüning, Mario Gleirscher, Wen-ling Huang, Niklas Krafczyk, Jan Peleska(✉), and Robert Sachtleben

Department of Mathematics and Computer Science, University of Bremen, Bibliothekstrasse 1, 28359 Bremen, Germany
{fbrning,mario.gleirscher,huang,niklas,peleska,rob_sac}@uni-bremen.de

Abstract. We present a novel approach to complete property-oriented white box module testing: a finite test suite, created and extended online (that is, during test execution), in combination with model learning and model checking allows to prove or disprove that a software module fulfils an arbitrary LTL property. The approach is applicable for modules with possibly infinite input and output domains. The testing strategy is based on the concept of black box checking proposed by other authors and on a complete model-based equivalence testing strategy developed previously by the authors of this paper. Since the white box approach allows for static analyses, basic information about internal states, guards and assignment expressions can be extracted from the module code. With this information at hand, the approach effectively performs a proof whether the implementation satisfies the specified property. The "classical" black box checking method is accelerated by means of coverage-guided fuzzing, in combination with effective methods for learning, failure monitoring, and conformance testing. This combination allows to reduce the overall effort for proving that the software fulfils the desired property in a considerable way.

Keywords: Property-oriented testing · Module testing · Linear Temporal Logic · Model learning · Formal verification

1 Introduction

Objectives. In this paper, we apply the concept of *black box checking*, as originally presented by Peled et al. [21,22], in the context of white box module testing. Given an LTL property φ, tests are executed for learning the true behaviour of an implementation under test (IuT) which is a software module I; this behaviour is expressed by means of an initially unknown symbolic finite state machine B.

Niklas Krafczyk is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 407708394. Felix Brüning, Wen-ling Huang, and Jan Peleska are funded by the German Ministry of Economics, Grant Agreement 20X1908E.

Published by Springer Nature Switzerland AG 2023
S. Bonfanti et al. (Eds.): ICTSS 2023, LNCS 14131, pp. 183–201, 2023.
https://doi.org/10.1007/978-3-031-43240-8_12

While trying to learn the true representation of B from the test cases executed so far, violations of φ are detected either by means of a monitor checking the reactions of I to the test case inputs, or during model checking the model increments $B = M_1, M_2, \ldots$ learnt so far. If a complete test of I against $B = M_k$ proves the language equivalence between I and B, the verification campaign terminates: I fulfils φ if and only if B fulfils this property. This "proof by testing and property checking" holds under certain hypotheses about the maximal number n of distinguishable states in I, and the guard expressions and output assignments used by I. This information can be extracted from the IuT by means of static analyses. Since these analyses are fairly simple and do not require the full understanding of the programming language semantics, this approach to property verification is a suitable method for testing modules programmed using complex programming languages like C++, Java, C#, where software model checkers accepting the complete syntax do not exist.

Background: Black Box Checking. In the original work by Peled et al. [21, 22], model learning was performed using Angluin's L^* algorithm [1]: under the assumption that I has at most n distinguishable states, the black box B can be reconstructed incrementally by executing finitely many tests against I.

Some tests serve to elaborate a new hypothesis about B (say, $B = M_i$), other tests serve to verify or falsify that I is language-equivalent to the current version of B. For the latter task, the W-Method [9,26] was used in [21,22]. This is a *complete* testing method in the sense that, under the hypothesis that I has at most n states, I passes the tests generated by the W-Method if and only if it is language-equivalent to M_i. Failed test cases can be used by the L^*-algorithm to modify and extend M_i, in order to create a refined model version M_{i+1}.

Using model checking, each new version of B is verified against φ. To this end, the product of B and a Büchi-automaton P accepting $\neg\varphi$ is constructed. If the language of product automaton $B \times P$ is non-empty, this indicates the existence of a *counterexample*, that is, an infinite input/output sequence π violating φ [2]. For safety properties, the violation of φ can already be demonstrated on a finite prefix π' of π [24]. For liveness properties, omega regularity implies that the infinite counterexample π can be written as $\pi_1\pi_2^\omega$ (infinitely many copies of π_2 are appended to π_1), with finite input/output sequences π_1, π_2 [2]. Since I is assumed to have at most n states, it accepts $\pi = \pi_1\pi_2^\omega$ if and only if it accepts $\pi_1\pi_2^n$, since the latter already implies the existence of a "lasso" [4] starting with π_1 and ending in a loop endlessly repeating π_2. Therefore, either π' or $\pi_1\pi_2^n$ are run against I. If the counterexample is accepted by I, an error has been found, and the combined learning and testing process can be aborted. If I does not accept the counterexample, this information can again be used to update B via continued learning. If the latest increment $B = M_k$ passes the check against φ, and the complete test suite proves that I and B are language-equivalent, the black box testing campaign has *proven* that I satisfies φ, under the hypothesis that I has at most n distinguishable states.

Contributions. In this paper, we refine and optimise the black box checking approach in several ways and specialise it for the purpose of white box software module testing, including tool support. For B, we admit (nondeterministic) *symbolic finite state machines (SFSM)* with finite state space, input and output variables over arbitrary primitive data types (including infinite types like $\mathbb{Z}$ and $\mathbb{R}$) and transitions labelled by guard conditions over input variables and output expressions over output and input variables. We advocate a white-box approach which is quite realistic for software in safety-critical systems, where source code needs to be verified by independent verification teams [28]. This allows us to determine upper state bounds n and identify the guard and assignment expressions used in the code by means of static analyses. These static analyses ensure that a passed black box checking suite corresponds to an actual *proof* that I satisfies property φ.

Regarding methodological contributions, the application of black box checking to software with conceptually infinite input and output types is enabled by an equivalence class partitioning method previously developed by the authors [14]. Otherwise black box checking would be infeasible, due to the large size of the alphabets involved, when using interface variables of type `double`, `float`, `int` directly.

Furthermore, we reduce the number of situations where tentative models $B = M_i$ need to be checked by means of a complete testing method. In particular, our strategy allows to check tentative models *later*, after many distinguishable states (say, ℓ) of the IuT have already been discovered. This significantly reduces the exponential term $p^{n-\ell+1}$ influencing the size of the complete test suite, where $n \geq \ell$ is the upper bound of potential distinguishable states in I, and p is the number of input/output equivalence classes derived from guard conditions and output expressions extracted from the code, as described below. Instead of the "classical" L^*-algorithm, we use a novel, highly effective state machine learning algorithm proposed by Vaandrager et al. [25]. For generating complete test suites, a variant of the complete H-Method [12] is used, which needs significantly fewer test cases than the W-Method in the average case [13]. We have modified the H-Method for *online testing*: this means that the test case generation is incremental and interleaved with the test execution, so that it is unnecessary to create a complete suite, when tests of I against the current version of B fail early. We apply the monitor concept proposed by Bauer et al. [3] for detecting safety violations on the fly, during tests intended for model learning. This reduces the need to perform complete model checking runs of $B \times P$ against φ. To speed up the learning process and to avoid having to create complete suites for too many intermediate increments of B, we apply coverage-guided fuzz testing [5,17] for finding many distinguishable states of the implementation at an early stage. Again, this leads to small exponents $n - \ell + 1$ in the term $p^{n-\ell+1}$ dominating the number of test cases to perform for a complete language equivalence test.

While these techniques for effort reduction cannot improve the worst case complexity that was already calculated by Peled et al. [21,22], their combination significantly improves black box checking performance in the average case.

We confirm this by several experiments verifying control software from the automotive domain. These experiments also show that the property testing approach described in this paper is effectively applicable for testing modules performing control tasks of realistic size and complexity. Therefore, the approach advocated here is an interesting alternative to proving code correctness by means of code-based model checkers or proof assistants. From the perspective of standards for software development in safety-critical systems [8,16,28], our approach even has a significant advantage in comparison to "pure" code verification, since tests are actually *executed* against the IuT. The standards emphasise that verification may never be based on static analyses (model checking, formal proof) alone: it is always necessary to perform dynamic tests of the integrated HW/SW system as well.

To the best of our knowledge, the approach presented here is the first to use equivalence class abstractions for enabling complete property testing of source code with large interfaces, using black box checking in combination with fuzzing.

Regarding the implementation of the approach, we present the open source library `libsfsmtest` for complete model-based or property-oriented module testing[1], whose latest version supports the module testing strategy described in this paper. For users only interested in the application of the library for practical testing, a cloud interface[2] is provided, supporting both test suite generation and module test execution.

Related Work. Meng et al. [18] confirm that fuzz testing can be effective for testing software against properties specified in LTL. However, their approach does not provide any completeness guarantees: the tool LTL-FUZZER created by the authors is to be used as an effective bug finder.

Pferscher et al. [23] also combine model learning and fuzzing, but with the objective to check whether an implementation conforms to a reference model, while our focus here is on property-oriented testing. The fuzzer is not guided by the code coverage achieved, as in our approach, but by the coverage of a reference model. Since the latter has not been validated with respect to completeness and consistency, the testing process can only reveal discrepancies between reference model and implementation, but not a correctness proof.

The model learning aspect of black box checking has received much attention since Angluin's seminal paper [1], and a comprehensive overview about improvements and alternative approaches to automata learning is given by Vaandrager et al. [25]. We could have made use of the LearnLib library [15] for the model learning part in our Algorithm 2 (see Sect. 3). However, we would not have used the W-Method or Wp-Method implemented there for equivalence testing and finding counter examples, since our own library `libfsmtest` provides methods like the H-Method that requires far less test effort in the average case. Moreover, the new data structure and associated algorithms for learning that has been pro-

[1] https://gitlab.informatik.uni-bremen.de/projects/29053.
[2] https://fsmtestcloud.informatik.uni-bremen.de.

posed by Vaandrager et al. [25] is not yet available in LearnLib, and it seemed particularly attractive with respect to maintainability and performance to us.

An alternative to LearnLib is AALpy by Aichernig et al. [19]. While its Python implementation seems less attractive to us, due to the better performance of C++, AALpy uses a strategy for disproving conformance between preliminary model versions and an implementation that is an interesting alternative to our current implementation: AALpy tries to avoid the generation of unnecessary complete conformance test suites by combining random testing with the W-Method, expecting to find early discrepancies between a preliminary version of the model and the implementation by means of random testing. In our approach, we prefer to focus the application of random testing in an initial phase using coverage guided fuzzing with the objective to find an initial candidate for machine B with as many states as possible. After that, we relay on conformance tests without randomisation, but create the cases of the H-Method incrementally, which also avoids the creation of a full conformance test suite as long as B and I do not conform.

Waga [27] presents a black box checking approach that is complementary to ours in several ways. (1) The main objective is bug finding for cyber-physical systems, while we focus on *complete* property checks for software modules. (2) Waga applies signal temporal logic, while we apply LTL. (3) Waga does not use any means of abstractions comparable to the equivalence class abstractions we consider to be crucial for complete property checking. Summarising, Waga's approach performs well for the purpose of bug finding on system level, while the method advocated here provides complete checks on module level.

Overview. In Sect. 2, we summarise the foundations required for the combined testing and black box checking approach described in this paper. In Sect. 3, the methodological main result is presented. In Sect. 4, a short summary of the available tool support is given. In Sect. 5, the application of our approach with this tool platform is described, and performance data is presented. Section 6 contains a conclusion.

2 Theoretical Foundations

2.1 Black Box Checking

The strategy for combined learning, model checking, and testing proposed by Peled et al. [22] is shown in Algorithm 1, with some adaptations for the notation used in this paper. The strategy uses two sub-functions for learning and testing: (1) As the first sub-function, Angluin's L^*-algorithm [1] is invoked (lines 6, 25) for learning the internal structure of the black box B representing the true behaviour of implementation I. The L^*-algorithm is called in Algorithm 1 with three parameters (I, M_i, π): I is the implementation, and the L^*-Algorithm may execute additional tests against I, in order to produce a new model. Parameter M_i specifies the latest assumption for the representation of B, and π is a word

representing a counterexample that is either accepted by M_i, but not by B, or vice versa. Based on this information, L^* returns a more refined model M_{i+1}.

(2) As the second sub-function, the W-Method [9,26] $\mathrm{VC}(I, M_i, \ell, k)$ is used as a conformance test that is able to prove or disprove the language equivalence between M_i and I, under the hypothesis that I has no more than k distinguishable states. The algorithm is called with the implementation I to be used in the test, the currently learnt, minimised model M_i that may or may not be equivalent to I, the number ℓ of distinguishable states in M_i, and the currently assumed upper bound $k \leq n$ of distinguishable states in I. Note that the worst case estimate for the number of test steps to be executed for such a conformance test is $O(\ell^3 p^{n-\ell+1})$ [9].

Initially, the L^*-algorithm is set up with the empty machine (line 6). Then the implementation is tested until (a) either the learnt model B satisfies φ and has been shown to be language-equivalent to I by means of complete tests, under the hypothesis that I has at most n states (line 18), or (b) an approximation M_i of B has been learnt that *violates* φ on an infinite word $\pi_1 \pi_2^\omega$, and this word is accepted by the implementation (line 22).

Algorithm 1. Black box checking strategy, as proposed by Peled et al. [22].

```
1  function BlackBoxChecker(in I : Implementation;
2                           in φ : LTL formula to be fulfilled by I;
3                           in n : maximal number of states of I) : {pass, fail}
4  begin
5    P := Büchi-Automaton accepting ¬φ;
6    M_1 := L*(I,empty,−); -- initialise learning algorithm with empty machine
7    i := 1;
8    while ( true )
9    begin
10     X := M_i × P; -- Product of machine learnt so far and BA checking ¬φ
11     if L(X) = ∅ then -- M_i does not violate φ
12     begin
13       ℓ := number of states of M_i; k := ℓ;
14       do
15         (conforms, π) := VC(I, M_i, ℓ, k); -- apply the W-Method
16         k := k + 1;
17       while (k ≤ n ∧ conforms);
18       if ( conforms ) then return pass; -- Implementation conforms to M_i, and M_i
                fulfils φ
19     end
20     else begin -- current model M_i violates φ
21       let π_1, π_2 such that π_1 π_2^ω ∈ L(X); -- this word violates φ
22       if I passes test π_1 π_2^n then return fail;
23       else  π := shortest prefix of π_1 π_2^ω not accepted by I;
24     end
25     M_{i+1} := L*(I, M_i, π); -- extend model, using counterexample~π
26     i := i + 1;
27   end
28 end
```

Once a hypothetical model M_i has been proposed by the L^*-algorithm, its product with the Büchi-automaton P accepting $\neg\varphi$ is constructed (line 10). If

the language of this product is empty, this implies that M_i does *not* accept a word violating φ. Therefore, it is checked whether M_i is language equivalent to I, under the hypothesis that I does not have more than n states (lines 11—19). This is done incrementally over $k = \ell, \ldots, n$, in order to avoid superfluous tests if the non-equivalence can already be detected with a smaller value $k < n$. Therefore, the full number of $O(\ell^3 p^{n-\ell+1})$ test steps only needs to be executed if I conforms to M_i. If language equivalence between M_i and I can be established by the conformance tests, the strategy terminates with verdict 'pass', since I conforms to a mealy machine $B = M_i$ that fulfils φ.

If the language of the product $X := M_i \times P$ is non-empty, this means that M_i accepts a word satisfying $\neg\varphi$. Omega regularity implies that such a word can be written as $\pi_1 \pi_2^\omega$, with finite prefix π_1, followed by an infinite repetition of finite word segment π_2. To test whether the implementation accepts $\pi_1 \pi_2^\omega$, it suffices to check whether it accepts $\pi_1 \pi_2^n$, since I is assumed to have at most n distinguishable states. If I accepts the finite test $\pi_1 \pi_2^n$, we know that it accepts a word violating φ and can stop the procedure by returning 'fail' (line 22). There is no further need to look for a more refined model $B = M_{i+j}$ representing the true behaviour of I, since the implementation must be fixed anyway.

If, however, I rejects $\pi_1 \pi_2^n$, this implies that the implementation cannot be language-equivalent to the currently assumed representation M_i of B. Now we look for the shortest prefix π of $\pi_1 \pi_2^\omega$ that is rejected by I. This prefix is suitable as a "teacher's response" for the L^*-algorithm, to be used to construct a more refined version M_{i+1} of the true implementation behaviour (line 25).

Peled et al. prove ([22, Theorem 3]) that if the implementation satisfies φ, the worst-case time complexity of the strategy described above is $O(\ell^3 p^\ell + l^3 p^{n-\ell+1} + l^2 mn)$, otherwise (error case), it is $O(\ell^3 p^\ell + l^2 mn)$. The higher complexity in the no-error case given by term $l^3 p^{n-\ell+1}$ in the complexity sum is derived from the fact that the equivalence tests of the implementation against the learnt model B need to execute all test steps required for the conjecture that I has at most n states. In the error case, these tests can be aborted earlier.

2.2 Equivalence Class Construction for SFSM

We summarise here previously obtained results [14] that are relevant for the present paper. A symbolic finite state machine (SFSM) M is a state machine operating on a finite set of control states and input variables and output variables from a symbol set $V = I \cup O$. Variables are typed by some (possibly infinite) set D. A variable valuation is a function $\sigma \in D^V$, associating a value $\sigma(v)$ with each variable symbol $v \in V$. Given a quantifier-free first order expression e with free variables in V, we say that σ is a model for e (written $\sigma \models e$), if and only if the formula $e[v/\sigma(v) \mid v \in V]$, that is created from e by exchanging every occurrence of a variable symbol $v \in V$ by its valuation $\sigma(v)$, evaluates to true.

A transition relation $s_1 \xrightarrow{g/a} s_2$ connects certain control states s_1, s_2. The transition label g/a consists of a guard expression g, that is, a quantifier-free first order expression over variables from I, and update expressions a that are

Table 1. Construction method for I/O equivalence classes (from [14]).

1. Let $\Sigma = \Sigma_I \cup \Sigma_O \cup AP$ be the set of all first-order formulae occurring in guard conditions or output expressions of the IuT, or in the property specification φ.
2. For a set of formulae $P \subseteq \Sigma$, define a new first-order formula which is a conjunction of formulae from P and negated formulae from $\Sigma \setminus P$:

$$\phi_P \equiv \bigwedge_{e \in P} e \wedge \bigwedge_{e \in \Sigma \setminus P} \neg e. \tag{1}$$

3. Let $\mathbf{P}$ denote the set of all formulae ϕ that have been constructed according to Eq. (1) and that possess at least one valuation $\sigma \in D^V$ as model, so that $\sigma \models \phi$.
4. For each $\phi \in \mathbf{P}$, define an *input/output equivalence class* $\mathrm{io}(\phi)$ by

$$\mathrm{io}(\phi) = \{\sigma \in D^V \mid \sigma \models \phi\}.$$

5. Let $\mathcal{A} = \{\mathrm{io}(\phi) \mid \phi \in \mathbf{P}\}$ denote the set of all input/output equivalence classes.

first order expressions over at least one output variable and optional variables from I. The language of M is the set $L(M) \subseteq (D^V)^\omega$ of all infinite traces of valuations $\sigma_1, \sigma_2, \cdots \in D^V$, such that there exists a sequence of states $s_0, s_1, \ldots$ starting in the initial state and guard and output expressions $(g_1/a_1)(g_2/a_2)\ldots$, such that

$$\forall i > 0 \,.\, s_{i-1} \xrightarrow{g_i/a_i} s_i \quad \sigma_i \models g_i \wedge a_i.$$

The property testing approach described in this paper applies to all software modules whose input/output behaviour can be described by means of an SFSM. The class of real-world applications that can be modelled by SFSM is quite large, examples are airbag control modules, anti-lock braking systems (see Sect. 5) or train control systems.

An input/output equivalence class io is a set of valuations $\sigma \in D^V$ constructed according to the specification in Table 1. The intuition behind this specification is that the input/output equivalence classes partition the set D^V of valuations: two members of the same class are models for exactly the same conjunction over all guard conditions, output expressions, and atomic propositions occurring in the LTL property φ to be verified, each conjunct occurring either in positive or negated form. No valuation can be in more than one class, since two classes differ in the sign (positive/negated) of at least one conjunct.

Two sequences $\pi_1, \pi_2 \in (D^V)^\omega$ of valuations are equivalent if each pair of corresponding sequence elements $(\pi_1(i), \pi_2(i))$, $i = 1, 2, \ldots$ is contained in the same input/output equivalence class. The following properties of equivalent

traces $\pi_1, \pi_2 \in (D^V)^\omega$ are crucial in the context of this paper [14, Theorem 2][3]: (1) $\pi_1 \in L(M)$ if and only if $\pi_2 \in L(M)$. (2) π_1 and π_2, when contained in $L(M)$, cover the same sequences of states in M (there is only one uniquely determined state sequence if M is deterministic). (3) $\pi_1 \models \varphi$ if and only if $\pi_2 \models \varphi$.[4]

3 Optimisation of the Test Method

Based on black box checking (Algorithm 1), we propose the new white box module testing strategy specified in Algorithm 2 and incorporating several optimisations. This strategy is divided into three phases: (1) setup, (2) fuzzer-guided exploration, and (3) learning as explained below.

Algorithm 2. White box module testing strategy.

```
1  function FuzzingBlackBoxLearner(in I : Implementation;
2                                   in Σ_I : guard conditions;
3                                   in Σ_O : output expressions;
4                                   in φ : LTL property to be verified;
5                                   in n : maximal number of states of I;
6                                   in r_max : maximum number of rounds of fuzzing;
7                                   ) : {pass, fail}
8  begin
9    -- Phase 1: Setup
10   AP := atomic propositions of φ;
11   A := input/output equivalence classes based on Σ_I ∪ Σ_O ∪ AP;
12   H := set of input valuations σ_1, σ_2, ..., such that for each ψ ∈ A, there exists
          some σ ∈ H extendable to a valuation satisfying ψ;
13   T := {ε}; -- initialise a prefix-closed set of traces observed in I
14   P := construct a property monitor accepting ¬φ;
15
16   -- Phase 2: Fuzzer guided exploration
17   r := 0; -- number of performed fuzzing iterations
18   ℓ_T := 1; -- lower bound on the number of distinct states already observed
19   while ( r < r_max and l_T < n )
20   begin
21     b̄ := non−empty sequence of integers obtained from fuzzer;
22     x̄ := map each element b̄ to an element of H;
23           -- e.g. by selecting the (b mod |H| + 1)^th element of H
24     outputQuery(I, T, P, x̄); -- apply x̄ to I and update T with the observed
           output; return fail if P observes a violation of φ
25     r := r + 1;
26     ℓ_T := |maximalPairwiseDistinguishableSubsetOf(T)|
27   end
28
29   -- Phase 3: Learning using L#
```

[3] Note that this theorem has only been formulated for finite traces π_i in [14]. The proof, however, holds for infinite traces $\pi_i \in (D^V)^\omega$ as well, because $\pi_1, \pi_2 \in (D^V)^\omega$ are equivalent if and only if all finite prefixes of π_1, π_2 with identical length are equivalent.

[4] Recall that LTL formulae over free variables from V have infinite sequences of valuations in D^V as models [10].

```
30  M_1 := L^#(I, H, A, T, P, −) -- start learning using input alphabet H, output
        alphabet A, and the observations T observed during fuzzing
31  i := 1;
32  while ( true )
33  begin
34    X := M_i × P; -- Product of machine learnt so far and BA checking ¬φ
35    if L(X) = ∅ then -- M_i does not violate φ
36    begin
37      ℓ := number of states of M_i;
38      (conforms, π) := H(I, M_i, T, P, ℓ, n); -- apply an online H-Method
39      if ( conforms ) then return pass;
40            -- Implementation conforms to M_i, and M_i fulfils φ
41    end
42    else begin -- current model M_i violates φ
43      let π_1, π_2 such that π_1 π_2^ω ∈ L(X); -- this word violates φ
44      if I passes test π_1 π_2^n then return fail;
45      else  π := shortest prefix of π_1 π_2^ω not accepted by I;
46    end
47    M_{i+1} := L^#(I, H, A, T, P, π); -- learn more elaborate model,
48                                      -- based on counterexample π
49    i := i + 1;
50  end
51 end
```

Phase 1: Setup. In the first phase, we exploit white box knowledge on the IuT in order to abstract from its possibly infinite input and output domains to finitely many equivalence classes. To this end, the algorithm uses the two input parameters Σ_I and Σ_O, denoting the guard conditions and output expressions occurring in the IuT, respectively. Together with the atomic propositions AP occurring in the LTL property φ to check, these are employed in computing input/output classes $\mathcal{A}$ (lines 10 and 11) using the techniques described in Sect. 2. These classes could then already serve as symbolic inputs. However, since multiple input/output classes may share the same input valuations, this could introduce superfluous inputs. Thus, line 12 of the algorithm attempts to minimise the number of inputs by only selecting sufficiently many input valuations $\sigma \in \mathcal{H}$ to provide input representatives of all input/output classes. In the following, we use elements of $\mathcal{H}$ both as symbols and as concrete input valuations.

The first phase concludes by initialising a tree T representing a prefix-closed set of symbolic traces observed in the IuT (line 13), as well as a property monitor P constructed as proposed by Bauer et al. [3] (line 14) to accept $\neg\varphi$. This monitor detects violations of safety properties φ observed during the subsequent execution of Algorithm 2. Since violations of liveness properties can only be determined on infinite traces, these are accepted, but do not lead to failure indications by the monitor.

Phase 2: Fuzzer-Guided Exploration. In the second phase, coverage-guided fuzzing is employed to quickly reach a large number of distinct states in the IuT and record observations on the behaviour of the IuT, with the aim of speeding up the subsequent learning phase. Experiments confirming the efficacy of this approach are discussed in Sect. 5.

Fuzzing is used for several iterations (lines 19—27). In each iteration, a non-empty sequence of integers $\bar{b}$ is obtained from the fuzzer and translated into a sequence of input symbols $\bar{x}$ by mapping each integer to an element of $\mathcal{H}$ (lines 21 and 22). Thereafter, $\bar{x}$ is applied to the IuT (line 24). All such invocations of the IuT in Algorithm 2 occur via calls to procedure outputQuery$(I, T, P, \bar{x})$. These reset the IuT and P to their initial states and initialise a symbolic trace $\gamma = \epsilon$. The following steps are then performed for each input symbol x in $\bar{x}$ in turn: First, x is translated into the concrete input valuation σ_I it symbolises, which is then applied as input to the IuT. Next, the outputs σ_O observed in response are used to create a valuation $\sigma_I \cup \sigma_O$ ranging over all input and output variables. This valuation belongs to exactly one input/output class $y \in \mathcal{A}$, which is considered as the output symbol observed for input symbol x. Thereafter, x/y is appended to observation γ, which is then added to T. Finally, P is used to check whether γ violates φ, in which case Algorithm 2 returns fail.

The fuzzer-guided exploration terminates as soon as one of the following conditions is satisfied: (1) Fuzzing has been performed for r_{max} iterations, where r_{max} is another input parameter of the algorithm, and the number of iterations is tracked in variable r (lines 17 and 25), or (2) fuzzing has identified n distinct states in the IuT. A lower bound on the number of identified distinct states in the IuT is tracked in variable l_T (lines 18 and 26), which is updated after each iteration. This is realised via function maximalPairwiseDistinguishableSubsetOf(T) as follows: First, pairs of traces are identified that are distinguishable in T.[5] From these, a maximal set $S \subseteq T$ is selected such that any pair of distinct traces in S is distinguishable.[6] Variable l is then set to $|S|$, as distinct traces in S must reach distinct states in IuT I and hence $|I| \geq |S|$.

Phase 3: Learning. The third phase (lines 30—50) finally implements learning in analogy to Algorithm 1. It differs from the latter in two aspects: First, instead of Angluin's L^* algorithm [1], learning is performed using an adaptation of the efficient $L^\#$ algorithm proposed by Vaandrager et al. [25] (lines 30 and 47). $L^\#$ follows the same *minimally adequate teacher* framework as L^* in generating hypothesis state machines and providing these to the teacher to check for equivalence with the IuT; hence it can directly replace the original calls to L^* in Algorithm 1. In contrast to line 6 of Algorithm 1 and also differing from the original description of $L^\#$, which starts without prior knowledge, line 30 of

[5] Traces α, β are distinguishable in T if there exists $\alpha.(\bar{x}/\bar{y}), \beta.(\bar{x}/\bar{y}') \in T$ with $\bar{y} \neq \bar{y}'$.

[6] Note that finding the largest such set is equivalent to finding the largest clique [6] in an undirected graph with vertexes T where traces are adjacent if and only if they are distinguishable. This constitutes a computationally expensive problem, so that we currently apply a greedy heuristic.

Algorithm 2 provides initial knowledge to the learning algorithm in the form of T, the previously observed traces.

The second difference consists in the conformance testing strategy employed to check whether the current hypothesis M_i is language-equivalent to the IuT (line 38). Instead of the W-Method [9,26], we employ the H-Method [12]. While both strategies exhibit the same worst case behaviour in terms of test steps, the H-Method has been observed in practice to require on average significantly fewer test steps [13]. We adapted the H-Method for online testing. That is, instead of computing the entire test suite and only thereafter applying it to the IuT, we interleave test case generation and application in an attempt to find failures early. This is particularly effective if the current hypothesis contains fewer than n states, as the term $|\mathcal{H}|^{n-\ell+1}$ dominates the number of test cases to consider.

Finally, recall that all interactions with the IuT within Algorithm 2 are performed via function outputQuery first called in the second phase. Thus, for every query to the IuT performed by the H-Method or by $L^{\#}$ (lines 30, 38, 47), property monitor P continues to check for violations of φ.

4 Tool Support: `libfsmtest` and `libsfsmtest`

We have implemented the described approach as a C++ framework based on `libFuzzer`[7], the coverage-guided fuzzing engine distributed as a part of the LLVM project and on `ltl3tools`[8] supporting the generation of runtime monitors for LTL properties [3].

The implementation I is integrated into a *test harness*, which contains an implementation of Algorithm 2. Alphabets Σ_I and Σ_O and the LTL property φ are read from files using the `libsfsmtest`[9] library, which also extracts the atomic propositions AP occurring in φ. From these, the equivalence classes over $\Sigma_I \cup \Sigma_O \cup AP$ and a propositional abstraction of φ are constructed, from which `ltl3tools` can construct a runtime monitor in a specific pseudo code representation. Using `libsfsmtest` again, this monitor is transformed into an executable version reading and checking input/output valuations observed on I. We have implemented the $L^{\#}$ algorithm in `libfsmtest`[10]. The fuzzer invokes the test harness, which orchestrates the translation from fuzzer inputs to input equivalence classes, the application of inputs to I and the feedback of the observations on I to the runtime monitor for φ and the learning algorithm.

Libraries `libfsmtest` and `libsfsmtest` are available as open source under MIT license. If users are not interested in obtaining the source code, they can perform the whole testing approach described here by using a cloud service.[11]

[7] https://llvm.org/docs/LibFuzzer.html.
[8] https://ltl3tools.sourceforge.net/.
[9] https://gitlab.informatik.uni-bremen.de/projects/29053.
[10] https://bitbucket.org/JanPeleska/libfsmtest/.
[11] https://fsmtestcloud.informatik.uni-bremen.de.

5 Experiments

For evaluation of the property testing approach described in this paper, we re-implemented an *anti-lock braking system (ABS)* for cars with lane stability control, as designed and published by Bosch GmbH [11]. The full functionality described there has been reduced to ABS for the front-left wheel only, and we do not consider gravel road conditions.

The ABS system implements two fundamental tasks: (1) locking a wheel should be avoided if the driver brakes too hard or brakes on slippery roads. The ABS controller prevents wheel locking by alternately holding, reducing and increasing the brake pressure for each wheel individually so that each wheel rotates recurringly while braking, in order to keep the car steerable. (2) The ABS controller implements a lane stability control to prevent the car from swerving on asymmetric road conditions during braking with straight steering angle. The ABS controller then adjusts the braking force in a car for all wheels, to facilitate the steering intervention by the driver, while still applying the maximal possible braking force. The ABS controller measures constantly the wheel velocity v_U and calculates the brake slip λ_B for each wheel, relative to the vehicle target speed v_R, which in this example is measured at the car powertrain. The equation to calculate the slip is [11]

$$\lambda_B = \frac{v_U - v_R}{v_R}.$$

The ABS controller evaluates the momentary acceleration α of each wheel to detect each wheel's tendency to lock. If α falls below the threshold $-a < 0$, a possible wheel lock is detected and the input valve `VI` (in front of the brake fluid inlet of the wheel brake cylinder) is closed, as well as the output valve `VO` (after the brake fluid outlet of the wheel brake cylinder) to hold the current brake pressure. The additional brake pump `P` to artificially increase the brake pressure is set to mode `OFF`. Consequently, the negative wheel acceleration α is not reduced any further. Then, if the brake slip falls below the maximum slip threshold, the output valve is opened again. Thus, the brake pressure decreases again, and α and the slip increase.

When $\alpha = -a$, the valves are switched to hold pressure (both valves closed, pump off). Now, the acceleration increases and can exceed two thresholds $+a, +A$ satisfying $+a < +A$. In the first iteration, the brake pressure will be increased when $\alpha > +A$, by setting `VI` := `OPEN`, `VO` := `CLOSED`, and `P` := `ON`. After a certain time, α decreases and reaches $+A$, so that the ABS controller switches again to hold pressure. The braking pressure is held until the $+a$ threshold is reached. At this point, the second iteration begins (henceforth repeating) and the brake pressure is slowly increased until $-a$ is reached. In the following cycles, α is kept between the two thresholds $-a$ and $+a$ by the ABS controller. If the ABS controller receives a signal from the yaw sensor that the car rotates around the z axle during braking, an asymmetric road condition is detected. If the car rotates to the direction of the current wheel, the driver is braking and the steering angle is in direction straight ahead, the controller then tries to facilitate the steering intervention by the driver by alternately reducing and increasing the

brake pressure of the current wheel but applying maximum possible braking force in threshold $-a_{\text{GMA}}$ (slightly higher than $-a$) and $+a$ until the rotation is within the yaw threshold again.

The C++ implementation consists of one model with approx. 700 lines of code. It processes 6 input variables of type `double` and writes to three output variables with small enumeration types. The module behaviour depends on 11 (not necessarily pairwise distinguishable) internal control states. Table 2 shows the LTL properties we tested on the example implementation.

The atomic propositions in the LTL formulae, together with the guards and update expressions contained in the code, result in up to 784 input/output equivalence classes $\text{io}_i \in \mathcal{A}$ that were calculated in about 203 s.

For each LTL property, we created one mutant that violates that property. For properties 1, 2 and 4 we did so by manually applying one of the five mutation operators[12] described in [20] at random locations in the program, until we found a mutant that violated that property. For each mutant we determined how fast it could be found with our approach for different numbers of fuzzing rounds. The time it took for a mutant to be killed for each run is shown in Table 3.

From this small set of data we can already conclude that the fuzzing can enable a learning-based approach to be used on problems where it would otherwise be not a sensible choice for economic reasons: While a purely learning-based approach was able to find the property violations for properties 2 and 3, it ran out of memory space in the other cases. This happens when equivalence queries are done with too large of a difference between the number of discovered states and the specified upper bound on the number of states. In all cases, we noticed that some amount of fuzzing usually drastically reduced the runtime of the approach. However, we also see that there is a trade-off to be made between making sure that learning does not start too early, as can be seen for the case where we used 100 fuzzing rounds for property 1, and taking too much time fuzzing, as can be seen for the other approaches where 100 fuzzing rounds found the violation faster than most other settings. Obviously, a violation for property 2 was rather easy to find on the corresponding mutant, and we attribute the runtime differences in the different fuzzing round configurations to runtime noise in the execution setup.

To investigate the runtime of the approach when there is no property violation found, we also ran it for the unmutated implementation which satisfies all four properties. In this case, the full model for the implementation, which has 11 states, has to be learnt. Due to the differing amounts of input/output equivalence classes for the properties, the runtimes can differ significantly for runs with different properties. We ran the approach with 5000 fuzzing cycles once for each property and logged the runtimes, number of applied input sequences and number of applied inputs of the fuzzing and learning portions of the approach, separately. For the number of applied inputs and input sequences we also separately noted how many were applied during equivalence queries. Table 4 shows the average, minimum and maximum numbers determined this way over all

[12] ABS, AOR, LCR, ROR, UOI.

Table 2. The set of LTL properties checked on the example implementation.

LTL formula	Description
$\mathbf{G}((\text{driverBrakes} \wedge v_R \geq v_{\min} \wedge \lambda_B \leq \phi \wedge \alpha \leq -a \wedge \lvert\text{yaw}\rvert \leq \theta \wedge \lvert\beta\rvert \leq \xi) \implies (\neg\texttt{VI} \wedge \texttt{VO} \wedge \neg\texttt{P}))$	Whenever the driver brakes while the velocity is above the minimum activation velocity $v_{\min}$ and when there is negative slip that is less than threshold ϕ, the wheel circumference is decelerating and the car is not yawing to either side more than θ radians per second while the driver is not steering more than ξ radians to either side, then valve `VI` shall be closed, `VO` opened, and the brake pump `P` shall be off to release brake pressure.
$\mathbf{G}((\text{driverBrakes} \wedge v_R \geq v_{\min} \wedge \text{yaw} < -\theta \wedge \lvert\beta\rvert < \xi) \implies (\neg\texttt{VI} \wedge \texttt{VO} \wedge \neg\texttt{P}))$	Whenever the driver brakes while the velocity is above the minimum activation velocity $v_{\min}$, the car is yawing to the left more than θ radians per second while the driver is relatively straight (not more than ξ radians to either side), then valve `VI` shall be closed, `VO` opened, and the brake pump `P` shall be off to release brake pressure.
$\mathbf{G}(\texttt{P} = \texttt{SLOW} \wedge \mathbf{X}(\text{driverBrakes} \wedge v_R \geq v_{\min} \wedge \alpha > -a \wedge \lvert\beta\rvert < \xi \wedge \text{yaw} \geq -\theta) \implies \mathbf{X}(\texttt{P} = \texttt{SLOW}))$	Whenever the brake pump is increasing the pressure slowly, it will continue to do so if the pressure is still too low for the acceleration α of the wheel's circumference to be below $-a$ and if the driver continues braking and steering straight ahead, the road conditions stay symmetric and the vehicle is moving fast enough for the system to be active.
$\mathbf{G}((\alpha < -a \wedge \text{driverBrakes} \wedge v_R \geq v_{\min} \wedge \lvert\beta\rvert < \xi \wedge \text{yaw} \geq -\theta) \implies ((\neg\texttt{VI}) \mathbf{U} \neg(\alpha < -a \wedge \text{driverBrakes} \wedge v_R \geq v_{\min} \wedge \lvert\beta\rvert < \xi \wedge \text{yaw} \geq -\theta)))$	Whenever the acceleration of the wheel's circum- ference is less than $-a$ while the driver is braking, the vehicle velocity is above the minimum activation velocity, the driver is steering relatively straight ahead (no more than ξ radians to either side), and the vehicle is not turning to the left more than θ radians per second, the brake pressure will not be increased until any of these conditions change.

Table 3. Program runtimes for the example described above. These were recorded on a kubernetes cluster with 1 CPU core and 16 GiB of RAM allocated to the task. *OOM* denotes that the property violation was not found during fuzzing, and the learning approach ran out of memory. r_{max} denotes the maximal number of fuzzing rounds performed.

r_{max}	Execution time			
	Property 1	Property 2	Property 3	Property 4
0	OOM	310 ms	12.3 s	OOM
100	OOM	5 ms	4.1 s	7.5 s
5000	21.4 s	10 ms	8.0 s	11.7 s
10000	39.1 s	4 ms	23.5 s	28.4 s

properties. We performed these experiments with the same fixed seed initializing the random choices for the fuzzer, starting with seed = 1. This was incremented by one only when the fuzzer would not discover enough states for the learning

to be able to learn the rest of the states without posing equivalence queries with too few discovered state. For the four properties tested on the conforming IuT, we succeeded with seed 1 twice, and with seeds 2 and 3 once. While this could seem inconvenient, we found that simply launching several fuzzing runs with different seeds was inexpensive and fast enough to still be practical.

Table 4. Runtime, number of applied input sequences and number of inputs applied during testing all properties against the implementation satisfying all properties.

	Prop. 1	Prop. 2	Prop. 3	Prop. 4
I/O Eq. Classes	600	600	784	714
Input Eq. Classes	120	117	138	124
Input Seq. Fuzzing	5000	5000	5000	5000
Input Seq. Learning	3311	3022	3764	5377
Input Seq. Equivalence Queries	38956	27862	32654	29522
Inputs Fuzzing	15245	19229	15595	17889
Inputs Learning	12409	10642	14217	19954
Inputs Equivalence Queries	47856	39325	45259	33577
Runtime Fuzzing	22.7 s	37.2 s	25.8 s	36.6 s
Runtime Learning	29.6 s	36.7 s	42.4 s	41.3 s

Compared to testing the mutants, testing the conforming implementation takes significantly longer, which matches the complexity results. In our set of problems, the equivalence queries are consistently the most expensive part of the whole approach which is also supported by the complexity results reported in Sect. 2.1. Furthermore, some of the runtime variations can be explained by the variations in input/output equivalence classes caused by the atomic propositions of the respective properties.[13]

6 Conclusion

In this paper, a novel white box module testing strategy based on learning has been presented. This strategy is complete: given an LTL property φ and an implementation under test I, it decides whether I satisfies φ under the assumption that a representation of I as a symbolic finite state machine contains at most n states and employs only guard and assignment expressions contained in a set of expressions Σ. As n and Σ can be determined from static analysis of I, the strategy effectively performs a proof whether I satisfies φ.

The strategy improves previous checking strategies based on learning [22] in several aspects. First, it performs input/output abstraction and hence allows

[13] To reproduce these results, our implementations of Algorithm 2 and of the ABS experiment can be accessed at https://doi.org/10.5281/zenodo.8143283.

checking of implementations with possibly infinite input and output domains. Next, it employs fuzzing in order to quickly reach distinct states in I, speeding up subsequent learning. Thereafter, it applies the efficient $L^{\#}$ learning algorithm [25] and reduces the number of required test cases for equivalence checks by using the H-Method [12]. Finally, throughout the algorithm, violations of φ observed in interactions with I are efficiently detected using a monitor [3]. The efficacy of these optimisations has been demonstrated in experiments with modules performing control tasks of significant size and complexity

For future work, we plan to fully implement the LTL model checking performed in Algorithm 2 in our tool and to pursue several further optimisations. These include the use of parallelisation within computationally extensive tasks such as the construction of input/output equivalence classes or applications of the H-Method. Furthermore, we plan to evaluate various heuristics for tasks such as the selection of input valuations (line 12 of Algorithm 2). Additionally, we plan to lift the restriction that our current implementation of Algorithm 2 supports only deterministic IuT (the underlying theory already covers nondeterministic IuT behaviour). Finally, we plan to develop an argument for the tool qualification [7] of our implementation based on the idea that if the strategy claims that I satisfies φ, then the final hypothesis $B = M_i$ of I's model representation can be used as reference model for and independent model-based testing algorithm to be executed against I.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
2. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011). https://doi.org/10.1145/2000799.2000800
4. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods Comput. Sci. **2**(5), November 2006. https://doi.org/10.2168/LMCS-2(5:5)2006, http://arxiv.org/abs/cs/0611029, arXiv: cs/0611029
5. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016, pp. 1032–1043. ACM (2016). https://doi.org/10.1145/2976749.2978428
6. Bomze, I.M., Budinich, M., Pardalos, P.M., Pelillo, M.: The maximum clique problem. In: Du, D., Pardalos, P.M. (eds.) Handbook of Combinatorial Optimization, pp. 1–74. Springer (1999). https://doi.org/10.1007/978-1-4757-3023-4_1
7. Brauer, J., Peleska, J., Schulze, U.: Efficient and Trustworthy Tool Qualification for Model-Based Testing Tools. In: Nielsen, B., Weise, C. (eds.) ICTSS 2012. LNCS, vol. 7641, pp. 8–23. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34691-0_3
8. CENELEC: EN 50128:2011 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems (2011)

9. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **SE-4**(3), 178–186 (1978)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
11. Dietsche, K.H., Reif, K.: Kraftfahrtechnisches Taschenbuch, 2nd edn. Springer Vieweg (2018)
12. Dorofeeva, R., El-Fakih, K., Yevtushenko, N.: An improved conformance testing method. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 204–218. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_16
13. Endo, A.T., da Silva Simão, A.: Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. Inf. Softw. Technol. **55**(6), 1045–1062 (2013). https://doi.org/10.1016/j.infsof.2013.01.001
14. Huang, W.l., Krafczyk, N., Peleska, J.: Model-Based Conformance Testing and Property Testing With Symbolic Finite State Machines - Technical Report. Zenodo, October 2022. https://doi.org/10.5281/zenodo.7267975. https://zenodo.org/record/7267975, to appear in Science of Computer Programming SCP (Part I) and Proceedings of the 10th IPM International Conference on Fundamentals of Software Engineering FSEN 2023 (Part II)
15. Isberner, M., Howar, F., Steffen, B.: The Open-Source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
16. ISO/DIS 26262–6: Road vehicles - functional safety - Part 6: Product development: software level (2009)
17. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: a survey. IEEE Trans. Software Eng. (2019). https://doi.org/10.1109/TSE.2019.2946563
18. Meng, R., Dong, Z., Li, J., Beschastnikh, I., Roychoudhury, A.: Linear-time temporal logic guided greybox fuzzing. In: Proceedings of the 44th International Conference on Software Engineering, ICSE 2022, pp. 1343–1355. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3510003.3510082
19. Muskardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: Aalpy: an active automata learning library. Innov. Syst. Softw. Eng. **18**(3), 417–426 (2022). https://doi.org/10.1007/s11334-022-00449-3
20. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. (TOSEM) **5**(2), 99–118 (1996)
21. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems. IAICT, vol. 28, pp. 225–240. Springer, Boston (1999). https://doi.org/10.1007/978-0-387-35578-8_13
22. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. J. Automata Lang. Combinatorics **7**(2), 225–246 (2002). https://doi.org/10.25596/jalc-2002-225
23. Pferscher, A., Aichernig, B.K.: Stateful black-box fuzzing of bluetooth devices using automata learning. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NFM 2022. LNCS, vol. 13260, pp. 373–392. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06773-0_20
24. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects Comput. **6**(5), 495–511 (1994). https://doi.org/10.1007/BF01211865. http://link.springer.com/article/10.1007/BF01211865

25. Vaandrager, F., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness. In: TACAS 2022. LNCS, vol. 13243, pp. 223–243. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_12
26. Vasilevskii, M.P.: Failure diagnosis of automata. Kibernetika (Transl.) 4, 98–108 (July-August 1973)
27. Waga, M.: Falsification of cyber-physical systems with robustness-guided black-box checking. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Contro, HSCC 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3365365.3382193, https://doi.org/10.1145/3365365.3382193
28. WG-71, R.S.E.: RTCA DO-178C - Software Considerations in Airborne Systems and Equipment Certification. 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011