Use Case Testing: A Constrained Active Machine Learning Approach

Karl Meinke(✉) and Hojat Khosrowjerdi

School of Electrical Engineering and Computer Science,
KTH Royal Institute of Technology, 100 44 Stockholm, Sweden
karlm@kth.se

Abstract. As a methodology for system design and testing, use cases are well-known and widely used. While current active machine learning (ML) algorithms can effectively automate unit testing, they do not scale up to use case testing of complex systems in an efficient way.

We present a new parallel distributed processing (PDP) architecture for a constrained active machine learning (CAML) approach to use case testing. To exploit CAML we introduce a use case modeling language with: (i) compile-time constraints on query generation, and (ii) run-time constraints using dynamic constraint checking. We evaluate this approach by applying a prototype implementation of CAML to use case testing of simulated multi-vehicle autonomous driving scenarios.

Keywords: Autonomous driving · Constraint solving · Learning-based testing · Machine learning · Model checking · Requirements testing · Use case testing

1 Introduction

For the design and testing of complex software systems, the use case approach has a long history emerging from [20] with many proposed variations and refinements. A use case can be viewed as a recurring short story in the daily life of a system. The essence of use case driven software engineering (SE) is to focus on a limited number of commonly occurring scenarios whose correct design and reliable implementation can generate significant end user benefit. For example, for cyber-physical systems, a focus on recurrent high-risk use cases can benefit end user safety.

By modeling system interactions with external actors, use cases open the way to evaluating a system in different environments and scenarios. In general, there can be a vast number of potential contexts so parameter modeling can be crucial. An environment is discretised into agents known as actors, which can be humans or other software systems. Through modeling short dialogs between system and environment within constrained scenarios, use cases capture important context sensitive behavioural information that can be used to test system implementations.

© Springer Nature Switzerland AG 2021
F. Loulergue and F. Wotawa (Eds.): TAP 2021, LNCS 12740, pp. 3–21, 2021.
https://doi.org/10.1007/978-3-030-79379-1_1

The development of UML languages, such as sequence diagrams, has made it possible to bridge the gap between informal natural language models of use cases and precise machine readable formats suitable for automated test case generation (TCG). Consequently, there is a significant literature on TCG for use cases from UML models surveyed in [32].

However, for automated TCG, machine learning (ML) based approaches such as black-box checking [30] and learning-based testing [26] are also worthy of consideration. For unit testing, such methods have been shown to be both effective [10,18,22,23] and efficient [35] for finding errors in real-world systems. For use case testing, active ML offers the possibility to systematically and efficiently explore the variability inherent in different *use case parameters* as well as the *time dimension*. Furthermore, by reverse engineering a model of the system under test (SUT), ML can be easily combined with static analysis techniques such as model checking and constraint solving. Unfortunately, current active ML algorithms in the literature provide no support for use case constraints and therefore scale rather poorly to use case testing.

In this paper, we propose a new and more scalable ML-based testing approach suitable for use case testing. This approach is termed *constrained active machine learning* (CAML). It generalises the ML techniques for dynamic software analysis surveyed in [3] by inferring *chains of intersecting automaton models*. Our proposal combines three new techniques that improve scalability: (i) a parallel distributed processing (PDP) architecture that supports concurrent test execution, (ii) use case modeling constructs that sequence and constrain ML parameters at compile time, and (iii) use case modeling constructs that constrain and dynamically bound ML parameters at runtime (the training phase). While these new techniques can undoubtedly be extended and optimised for even better scalability, we will show that they suffice to tackle non-trivial testing problems such as advanced driver assistance systems (ADAS) in multi-vehicle use case scenarios.

The structure of the paper is as follows. In Sect. 2, we discuss the background, including scalability problems for current active ML algorithms. In Sect. 3, we describe the architecture of a constrained active ML approach to testing. In Sect. 4, we present a use case modeling language that makes available the capabilities of the CAML architecture. In Sect. 5, we present a systematic evaluation and benchmarking of a prototype implementation of CAML. We have integrated a CAML prototype with the commercial vehicle simulation tool ASM produced by dSPACE GmbH[1]. The resulting toolchain allows us to model and test four industry-standard use cases for an *adaptive cruise controller* (ACC) in multi-vehicle scenarios ranging from 2 to 4 vehicles. In Sect. 6, we discuss the results of this evaluation. In Sect. 7, we survey related approaches. Finally, in Sect. 8, we discuss conclusions and possible future extensions of our approach.

[1] See www.dspace.com.

2 Background and Problem Statement

2.1 Use Case Modeling

A use case describes a system in terms of its interaction with the world. In the popular account [13], a use case consists of a *main success scenario* which is a numbered list of steps, and optionally one or more *extension scenarios* which are also numbered lists. A *step* is an event or action of the system itself or of an interacting agent known as an *actor*. Structurally, main and extension scenarios are the same, i.e. an enumeration of actions describing an interaction between the system and external actors. The difference is simply interpretation: extensions are "a condition that results in different interactions from ... the main success scenario" [13]. The template approach to use cases of [8] is more expressive. It includes both preconditions and success guarantees. We model these two concepts as constraints in our approach, as they are relevant for both efficient TCG and test verdict construction. The sequence diagram language of UML [9] generalises these linear sequences of actions to allow branches, loops and concurrency. The live sequence chart (LSC) language of [16] goes even further than UML by integrating temporal logic concepts and modalities (so called hot and cold conditions). Our approach could be extended to cover these advanced features, but they are not the subject of this initial research. Nevertheless, we will borrow simple temporal logic constructs to constrain TCG using ML.

2.2 Active Automaton Learning

By active automaton learning (see e.g. [17,21]) we mean the use of heuristic algorithms to dynamically generate new queries and acquire training data online during the training phase[2]. This contrasts with passive learning, where an a priori fixed training set is used. Since pioneering results of [1,14], it has been known that active ML has the capability to speed up the training process compared with passive ML. Recently, active automaton learning algorithms such as Angluin's L* [1] have experienced renewed interest from the software engineering community. Active automaton learning can be applied to learn behavioural models of black-box software systems. Such models can be used for SE needs such as code analysis, testing and documentation. A recent survey of active ML for SE is [2].

In automaton learning, the task is to infer the behavior of an unknown black-box system, aka. the *system under learning* (SUL), as an automaton model, e.g. a finite state Moore machine[3] $A = (\Sigma, \Omega, S, s_0, \delta : \Sigma \times S \rightarrow S, \lambda : S \rightarrow \Omega)$. This model is constructed from a finite set of observations of the input/output

[2] Since the new training data is generated by heuristic algorithms alone, active ML is *not* the same as interactive ML which requires human intervention.

[3] Here Σ is a finite input alphabet, Ω is a finite or infinite output alphabet, S is a finite state set, $s_0 \in S$ is the initial state, δ is the state transition function and λ is the output function. δ is extended to a transition function on input sequences $\delta^* : \Sigma^* \times S \rightarrow S$ by iteration.

behaviour of the SUL. During the training phase, a single step consists of heuristically generating a finite input sequence $\overline{i} = (i_1, ..., i_n) \in \Sigma^*$ as a query about the SUL. This query $\overline{i}$ must be answered by the SUL online with a response $\overline{o} = (o_1, ..., o_n) \in \Omega^*$. By iterating this single step, the learning algorithm compiles a growing list of queries $\overline{i_1}, ..., \overline{i_k}$ and their responses, $\overline{o_1}, ..., \overline{o_k}$ for increasing $k = 1, 2, ...$. This is the training data for A. As the training data grows, increasingly accurate models[4] $A_i : i = 0, 1, ...$ of the SUL can be constructed[5]. Different active learning algorithms generate different query sets. For example, the L* algorithm [1] maintains an expanding 2-dimensional table of queries and responses, where new gaps in the table represent new active queries.

Note that each new hypothesis model A_i must be checked for behavioral equivalence with the SUL to terminate learning. Equivalence checking is a second source of active queries and there are well known algorithms for this e.g. [34]. Probabilistic equivalence checking, by random sampling, is a common black-box method and the basis for *probably approximately correct* (PAC) automaton learning [21].

Equivalence checking avoids the problem of premature termination of the training phase with an incomplete model. Thus, many active learning algorithms such as L* can be proved convergent and terminating in polynomial time under general conditions. This means that under reasonable assumptions about queries and the structure of the SUL, eventually some hypothesis A_i will be behaviourally equivalent to the SUL.

2.3 Problem Statement: Scalable ML

Active machine learning can be used to automate the software testing process, a technique known as *black-box checking* (BBC) [30] or more generally *learning-based testing* (LBT) [26]. These approaches leverage active query generation as a source of test cases, and the SUL role is played by the software *system under test* (SUT). They are very effective for unit testing (see e.g. [10, 18, 22, 23]) where the set of possible SUT inputs, and their temporal order, are very loosely constrained, if at all. They can achieve high test coverage and outperform other techniques such as randomised testing [35]. The BBC/LBT approaches both arise as a special case of our more general use case approach (c.f. Sect. 3.2), namely as a *single step use case* with the constant gate predicate `false`.

In contrast to unit testing, use case testing evaluates focused, temporally ordered and goal directed dialogues between the system and its environment (see e.g. [12]). Here, a test *fail* implies some non-conformity between the SUT and an intended use case model. Active machine learning can potentially automate use case testing, with the obvious advantages of test automation (speed, reliability, high coverage).

[4] The A_i grow in size during active learning. The relationship between k and i varies between learning algorithms.

[5] A unified algebraic view of different automaton construction methods is the quotient automaton construction. Further details can be found in [3].

However, in the context of use case testing, two assumptions used in current active automata learning algorithms (such as L*) fail. *Assumption 1:* every input value $i \in \Sigma$ is possible for every use case step. *Problem 1:* This assumption leads to a large number of irrelevant test cases since test values are applied out of context (i.e. relevant use case step). *Assumption 2:* Every sequential combination of input values $(i_1, ..., i_n) \in \Sigma^*$ is a valid use case test. *Problem 2:* This assumption also leads to a large number of irrelevant test cases since most sequential combinations of test values will not fulfill the final or even the intermediate goals of the use case.

The combination of test redundancy arising from Problems 1 and 2 leads to an exponentially growing test suite (in the length of the use case) with very many irrelevant and/or redundant test cases.

Problem Statement: *The key problem to be solved for applying active ML to use case testing is to constrain the training phase, so that a scalable set of scenario-relevant test cases is generated.*

We decompose our solution to this problem by solving Problem 1 using static (compile-time) constraints, and solving Problem 2 using dynamic (run-time or training) constraints. Our approach is an instance of applying ML for its *generative aspect* [11], i.e. the capability to generate and explore solutions to constraints by machine learning.

3 Constrained Active Machine Learning (CAML)

In this section, we introduce a generic architecture for use case testing by CAML. This architecture aims to overcome the scalability problems of active ML identified in Sect. 2.3.

3.1 Use Case Testing: An Example

We can motivate our CAML architecture from the modeling needs of a well-known embedded software application from the automotive sector.

An *adaptive cruise controller* (ACC) is an example of a modern ADAS application used as a component for semi- and fully autonomous driving. An ACC is a control algorithm designed to regulate the longitudinal distance between two vehicles. The context for use is that a host vehicle H (that deploys the ACC) is following behind a leader vehicle L. When the ACC is engaged, it automatically maintains a chosen safety gap (measured in time or distance) between H and L. Typically, a radar on H senses the distance to L, and the ACC monitors and maintains the inter-vehicle gap smoothly by gas and brake actions on H. An important use case for testing ACC implementations[6] is known as *cut-in-and-brake* (C&B). The C&B use case consists of four steps.

[6] Many ACC algorithms exist in the literature, see e.g. [37].

Step 1: Initially H is following L (actor 1) along one lane of a road. Along an adjacent lane, an overtaking vehicle O (actor 2) approaches H from behind and overtakes.
Step 2: After O achieves some longitudinal distance d ahead of H, O changes lanes to enter the gap between H and L.
Step 3: When O has finished changing lane, it brakes for some short time.
Step 4: O releases its brake and resumes travel.

The C&B use case is clearly hazardous for both H and O, with highest collision risk during Steps 2 and 3. Safety critical parameters such as d above may be explicit or implicit in a use case description, and their boundary values are often unknown. These may need to be identified by testing [4]. Active ML is a powerful technology for such parameter exploration.

Extensive testing of use cases such as C&B is routinely carried out in the automotive industry. A *test case* for C&B consists of a *time series* of parameter values for vehicle actuators such as gas, brake and steering, to control the trajectories of H, O and L. The lengths of each individual Step 1–4 are not explicitly stated by the use case definition above. These constitute additional test parameters. Chosen parameter values must satisfy the constraints of Steps 1–4 to make a valid C&B scenario. Notice that H is longitudinally autonomous as long as the ACC is engaged, and can be fully autonomous on straight road sections. So only the trajectory parameters of L and O can be directly controlled in this case. Clearly random testing, i.e. randomised choice of test parameter values, is not useful here. Most random trajectories for L and O do not satisfy the criteria for C&B, and would represent extremely haphazard driving, uncharacteristic of real life. For a given use case U, valid test cases are *constrained time series*, and we must address efficient constraint satisfaction in any practical ML solution.

3.2 A Parallel Distributed CAML Architecture

Following the connectionist or parallel distributed processing (PDP) paradigm, we introduce a pipeline architecture for CAML in Fig. 1. This architecture consists of a linear pipeline of alternate *active automaton learning modules* L_i and *model checking modules* MC_i. Each learner L_i conducts online active ML on a *cloned copy* SUT_i of the SUT.

For use case testing, the basic idea is to dedicate each learning algorithm L_i to the task of learning Step i, for all the $i = 1, ..., n$ steps of an n-step use case U. We will show later, in Sect. 4, how the use case U is modeled by constraints. Here we focus on explaining and motivating the PDP architecture of Fig. 1.

Each learner L_i has the task to infer an automaton model A_i of Step i in U by actively generating queries[7] $in_\alpha = in_{\alpha,1}, ..., in_{\alpha,l(\alpha)} \in \Sigma_i^*$ and executing them on SUT_i. We may refer to A_i as the *state space model* for Step i. Constraining the input for SUT_i to the input alphabet Σ_i in Step i at compile time significantly reduces the search space for finding valid use case tests for U as whole. This

[7] The queries have variable length $l(\alpha)$.

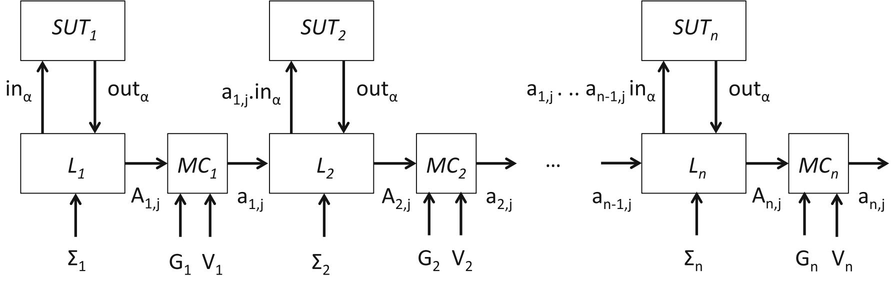


Fig. 1. A constrained active ML architecture

addresses Problem 1 of Sect. 2.3. Each query in_α is executed locally on SUT_i. The observed output behaviour $out_\alpha = out_{\alpha,1}, ..., out_{\alpha,l(\alpha)} \in \Omega_i^*$ of SUT_i is integrated by L_i into the current version $A_{i,j}$ of A_i to incrementally generate a sequence of approximations $A_{i,1}, A_{i,2}, ...$ that converge to A_i, as described in Sect. 2.2.

We can observe in the use case C&B that the end of each Step i is characterised by a Boolean condition G_i that must become *true* to enter the next Step $i + 1$ or else to finish the use case. For example: we leave Step 1 of C&B and start Step 2, once the gap between O and H exceeds d and not before. To constrain and connect each adjacent pair of state space models A_i and A_{i+1}, constructed independently by L_i and L_{i+1}, we model G_i as a Boolean constraint $G_i \subseteq \Omega_i$ which is a *predicate* on state values $\lambda(s) \in \Omega_i$. We term G_i the *gate condition* for Step i. The gate condition G_i can be seen as both the *success guarantee* for leaving Step i and the *precondition* for entering Step $i + 1$ (c.f. Sect. 2.1). In particular, G_n is a success guarantee for finishing the entire use case U.

Figure 1 shows a second Boolean constraint or predicate $V_i \subseteq \Omega_i$ called the *verdict condition*. This will be discussed later in Sect. 4.3.

The gate condition G_i is evaluated on each approximation $A_{i,j}$ of A_i, for $j = 1, 2, ...$ by the model checker MC_i (c.f. Fig. 1). Model checking [7] is a general constraint solving technique for Boolean and temporal logic formulas on automaton models. The model checker MC_i incrementally analyses each $A_{i,j}$ to identify a new state $s_{i,j} \in S_{i,j}$ for $A_{i,j}$ (not previously seen in $A_{i,j-1}$) that satisfies the gate G_i, i.e. G_i is *true* as a predicate on $\lambda(s_{i,j})$. The state $s_{i,j}$ will become an initial state of A_{i+1}. In this way, adjacent models A_i and A_{i+1} intersect, and $A_1, ..., A_n$ collectively build a complete and connected chain of automaton models of U.

Now, a guaranteed condition of automaton learning algorithms such as L* is that every learned state $s \in S_{i,j}$ is *reachable* in $A_{i,j}$ by at least one *access sequence* $a = a_1, ..., a_m \in \Sigma_i^*$, i.e. $\delta_i^*(a, s_0) = s$. The model checker MC_i can return such an access sequence $a_{i,j}$ for state $s_{i,j}$ satisfying gate G_i. This access sequence $a_{i,j}$ is a valid *test case solution* for Step i of U and hence a *partial solution* to a complete and valid test case for U. Dynamic constraint solving using MC_i at runtime further constrains the size of the state space to

be searched in building valid test cases for U. This approach addresses Problem 2 of Sect. 2.3.

The active learners $L_1, ..., L_n$ and model checkers $MC_1, ..., MC_n$ collaborate to construct valid test cases for the whole n-step use case U as follows.

For each $j = 1, 2, ...$ and for each $1 \leq k < n$, all k access sequences (partial solutions) $a_{1,j}, ..., a_{k,j}$ coming from $MC_1, ..., MC_k$ (which satisfy the gates $G_1, ..., G_k$ respectively) are passed to learner L_{k+1} where they are concatenated into a *setup sequence*[8] $(a_{1,j}, \ldots, a_{k,j})$. This setup sequence is used as a prefix, and appended in front of every active query $in_\alpha \in \Sigma_{k+1}^*$ generated by L_{k+1}. A complete active query for SUT_{k+1} therefore has the form:

$$(a_{1,j}, \ldots, a_{k,j}, in_\alpha).$$

From the corresponding output sequence $out_\alpha \in \Omega_{k+1}^*$ returned by SUT_{k+1} only the final suffix of length $|in_\alpha|$ is retained by L_{k+1} to construct A_{k+1}. This suppresses all SUT output due to the setup sequence $a_{1,j}. \ldots .a_{k,j}$. So the state space model A_{k+1} only contains information about Step $k+1$ of U, and we avoid duplication of effort between the parallel learners.

Finally the n access sequences (partial solutions), which emerge periodically from $MC_1, ..., MC_n$, are concatenated to form

$$a_j = (a_{1,j}, \ldots, a_{n,j}).$$

Thus a_j represents the j-th complete test case for U, as a concatenation of the j-th partial solutions. The test case a_j satisfies all of the guards $G_1, ..., G_n$, in particular the final goal G_n of U. Moreover, in each of the steps $a_{i,j}$ all actions are constrained to Σ_i^*. So a_j is a valid test case for U.

4 A Use Case Modeling Language for CAML

We can now introduce a constraint-based modeling language for use cases that exploits the CAML architecture of Sect. 3.2. A constraint model U will capture an informal use case description in terms of parameters and constraints suitable for using in the CAML architecture. These include: Σ_i, G_i and V_i for each step $i = 1, ..., n$.

4.1 Input/Output Declaration

Recall the running example of the C&B use case from Sect. 3.1. The actors are the three vehicles H (with its ACC), L and O. The first modeling step is to decide what actor parameters we need to control and observe. Much automotive application testing is performed within the safety of a virtual environment such as a multi-vehicle simulator. Whatever the context, we can assume the existence

[8] This terminology comes from testing theory and is used to denote an initialisation sequence bringing SUT_{k+1} into a state where in_α can be applied.

of a test harness or wrapper around the SUT which exposes the SUT API in a standardised and symbolic way, as a set of variable names and their types: float, integer, enumeration, Boolean, etc.

This modeling activity for C&B identifies the following minimum sets[9] of relevant input and output parameters and their types:

```
input_variables = [SpeedL:enum, SpeedO:enum, SteerO:enum];
```

This statement declares three test input variables (from the SUT API) of enumeration type `enum` that will control the leader vehicle speed, the overtaker speed and the overtaker steering[10]. So a *test input vector* to the SUT is an ordered triple of `enum` values (x_1, x_2, x_3). A complete *use case test input* is a finite sequence of test input vectors (c.f. Fig. 2(a)) $((x_1^1, x_2^1, x_3^1), ..., (x_1^n, x_2^n, x_3^n))$.

For the output variables, the model declaration is:

```
output_variables = [Crash:boolean, O2HDist:float, TimeDev:float];
```

This statement declares three test output variables (from the SUT API) for crash detection, O-to-H longitudinal distance and time gap deviation (as a percentage error) between the intended ACC time gap[11] (H-to-L) and the observed time gap (H-to-L). A *test output vector* from the SUT is an ordered triple of values (y_1, y_2, y_3), where y_1 ranges over `Boolean` and y_2 and y_3 over `float` values. A *use case test output* is a finite sequence of test output vectors (c.f. Fig. 2(b)) $((y_1^1, y_2^1, y_3^1), ..., (y_1^n, y_2^n, y_3^n))$.

4.2 Sequencing, Static and Dynamic Constraints

Next we declare the four steps of the C&B use case in terms of: (i) compile time constraints on the input alphabets Σ_i and (ii) runtime constraints on the gate predicates G_i.

```
input_values[1] = { 50,55:SpeedL, 55,60,65:SpeedO, 0:SteerO };
gate[1] = when( O2HDist >= 5.0 & O2HDist <= 40.0 );
input_values[2] = { 50:SpeedL, right_100_4:SteerO, 50:SpeedO };
gate[2] = when( time >= 4.0 );
input_values[3] = { 60:SpeedL, 25,30,35:SpeedO, 0:SteerO };
gate[3] = when( TimeDev <= 5.0 );
input_values[4] = { 50:SpeedL, 60:SpeedO, 0:SteerO };
gate[4] = when( time >= 5.0 );
```

Each declaration `input_values[`i`]` symbolically declares Σ_i, the input values for Step i in the notation of Sect. 3.2. In general, values for Σ_i are sampled within

[9] Our example is pedagogic only. A more realistic model for C&B has more parameters and values.

[10] Recall that H is autonomous, hence only L and O are controllable in this scenario.

[11] The intended time gap is here assumed to be a fixed nominal value for every test case, typically around 1.5–2.5 s. It is often assignable by the driver of H.

the typical range of values (e.g. vehicle speeds) characteristic for each step of the use case (e.g. an acceleration, steady or deceleration step). For example, in Step 1 above, variable `SpeedL` has possible values `50,55`, `SpeedO` has possible values `55,60,65` but `SteerO` takes only the value `0`. The steering value `0` is a neutral command (i.e. straight ahead) in Steps 1, 3 and 4. However in Step 2 (the lane change step for overtaker O), the steering value `right_100_4` generates a sigmoidal right curve for O across 100% of the lane width in 4 time steps[12]. Notice that the declared speed of O drops from 50 in Step 2 to 25, 30 or 35 in Step 3. This implements the braking action of O in Step 3 (which need not even be a constant deceleration).

The informal meaning of `gate[i] = when( state_predicate );` is that once SUT execution has entered Step i, it stays in this step until a state is encountered that satisfies `state_predicate`. At this point SUT execution may pass to the next Step $i + 1$. Thus `gate[1] = when( O2HDist >= 5.0 & O2HDist <= 40.0 );` captures the transition from overtaking in Step 1 to lane change in Step 2 by setting specific minimum and maximum boundary values for d of 5.0 and 40.0 metres (c.f. the C&B description of Sect. 3.1). A gate condition can also take account of time, for example `gate[2] = when( time >= 4.0 );` ensures that we maintain the steering command of Step 2 for 4 time steps, relative to the start of Step 2. This ensures the steering action is completed.

The formalised C&B model above illustrates some of the variety of CAML capabilities for modeling a single step of a use case. These capabilities range from a single action that must be performed exactly once (Step 2 above) to a set of possible actions that can be executed in non-deterministic order over a time interval that is either: (i) unspecified, (ii) constant, (iii) finite and bounded or (iv) unbounded. Steps 1, 3 and 4 above illustrate some of these options. Each single step activity is defined by a judicious combination of input alphabets, gate constraints and step ordering. We have not attempted to be exhaustive in modeling all possible single step capabilities, and further extensions are possible (see Sect. 8).

4.3 Automated Test Verdict Construction

Recalling the discussion of Sect. 3.1, we can say informally that a (4 step) use case test input $a_j = a_{1,j}. \dots .a_{4,j}$ for C&B has a *pass* verdict if none of the vehicles O, L or H collide. Otherwise a_j has a *fail* verdict. The model checkers MC_i automate *test verdict construction* for each use case test input a_j as follows.

A use case test input $a_j = (a_{1,j}. \dots .a_{n,j})$ for an n-step use case U has the verdict *pass* if, and only if $v_{i,j} = pass$ for each $i = 1, ..., n$, where $v_{i,j} \in \{pass, fail\}$ is the *local verdict* for the test case step $a_{i,j}$ (which is an access sequence). Each model checker MC_i is used to evaluate its local verdict $v_{i,j}$ on $a_{i,j}$ in a distributed manner. In general, $v_{i,j}$ is based on a specific local criterion $V_i \subseteq \Omega_i$ for Step i as a predicate or constraint on state values $\lambda(s)$ for $s \in S_i$ a

[12] The time step length is also a fixed nominal value for all test cases.

state in the automaton model A_i. Figure 1 shows how the verdict predicates V_i are integrated into the CAML architecture. For C&B we are mainly interested in vehicle crashes in Steps 2 and 3 as the most hazardous steps. We can therefore extend the use case model of Sect. 4.2 with local verdict constraints for Steps 2 and 3 as follows:

```
verdict[2] = always( !crash );
verdict[3] = always( !crash & TimeDev <= 50.0);
```

The informal meaning of `verdict[i] = always( state_predicate );` is that `state_predicate` should remain *true* throughout Step i and if it becomes false at any point during Step i then both Step i, and the whole test case fail. For example, in `verdict[3]` for Step 3 above, when O is braking, we add to the no-crash requirement the additional verdict requirement that the observed time gap deviation `TimeDev` does not exceed 50%. This increases the safety margin of the ACC.

For the i-th access sequence $a_{i,j} = a_{i,j,1}, ..., a_{i,j,m} \in \Sigma_i^*$ of a_j, the model checker MC_i evaluates the verdict predicate V_i on $\lambda(s_{i,j,k})$ for each of the corresponding states $s_{i,j,1}, ..., s_{i,j,m} \in S_{i,j}$ traversed by $a_{i,j}$ in $A_{i,j}$. Here $s_{i,j,1}$ is the initial state of $A_{i,j}$ and $s_{i,j,m}$ is the final state that satisfies the gate condition G_i. If $\lambda(s_{i,j,k})$ satisfies V_i for each $k = 1, ..., m$ then $v_{i,j} = pass$ otherwise $v_{i,j} = fail$.

5 Evaluation and Benchmarking

To evaluate our CAML architecture for machine learning and its associated use case modeling language, we implemented these in a prototype TCG tool. This prototype was then integrated with the commercial vehicle software simulator ASM to provide a complete toolchain for testing driving scenarios in a virtualised road environment.

We conducted an evaluation of the complete toolchain to benchmark the speed and effectiveness of the CAML approach. For evaluation purposes, we chose use cases for an ACC-equipped semi-autonomous vehicle driven in multi-vehicle scenarios.

5.1 ROBOTest: A CAML Implementation

We implemented a prototype of the CAML architecture of Sect. 3, termed ROBOTest, on top of the ML-based testing tool LBTest [27]. LBTest has previously been successfully used in unit testing of automotive ECU software [22,23], as well as other domains including web and finance [36]. LBTest supports important features necessary for realistic testing case studies, such as infinite and continuous test parameter types (including integers, strings, floating point numbers), multi-threaded learning for high data throughput, and configuration files for job specification and test session management. In particular, a ROBOTest use case model of the type presented in Sect. 4 is simply added to an LBTest configuration file. During a testing session, multiple instances of LBTest `Learner` and `ModelChecker` classes implement the PDP architecture of Sect. 3.2.

Table 1. ML-based testing results for four ADAS use cases

No.	Use case	Use case steps	Use case vehicles	Ego vehicle autonomy	Executed test cases	Total execution time	Errors found	Learned model size
1	Following lead	2	2	Full	227	1 h 0 min 10 s	No	140 states
2	Cut-in	4	3	Full	761	20 min 29 s	Yes	177 states
3	Cut-out	4	3	Full	36	2 min 50 s	No	122 states
4	Overtake	5	4	Semi	1654	5 h 21 min 15 s	Yes	1022 states

5.2 Integration of ROBOTest and ASM

The ASM vehicle simulator from dSPACE GmbH provides the capability to perform software in the loop (SiL) testing of automotive applications. It can be used to produce realistic simulations of automotive applications in multi-vehicle scenarios. The ego vehicle parameters, road and environment parameters and the numbers and types of traffic objects are all configured before a simulation starts. The basic approach to ROBOTest and ASM tool integration was to expose key attributes of a parameterised ASM traffic model through a lightweight wrapper. By communicating indirectly with ASM through the wrapper, ego vehicle and traffic object commands could be accessed from the ROBOTest use case model contained in a configuration file. Such commands include parameterized commands to the ego vehicle and traffic objects for steering, gas, brake etc. Several command examples can be seen in the C&B use case of Sect. 4.

The wrapper was delegated the responsibility to translate ML generated use case tests into timed sequences of vehicle commands, and dispatch these sequences to the simulator. Key simulator variables were then logged by ASM and recovered by the wrapper. The resulting observation sequences were returned to ROBOTest for learning.

As the target language for test case translation, we used the ASM scenario language to specify the detailed actions of the ego vehicle and traffic objects. This was done in the scenario editor of the ASM ModelDesk application. ModelDesk also takes care of the road environment definitions and downloading configuration parameters into the ASM VEOS platform.

5.3 ACC Use Case Descriptions

To evaluate the toolchain resulting from integrating the two tools ROBOTest and ASM, we chose a set of use cases for an ACC application bundled with the ASM license. The choice was guided by the need for different use case lengths, complexity and number of actors. The following four use cases for an ACC-equipped ego vehicle in a multi-vehicle traffic environment were chosen.

1. Following Lead. The ego-vehicle follows a lead vehicle in the same lane, i.e. it is tracking the lead as its target. The lead vehicle accelerates and decelerates

within given speed bounds. The ego vehicle should adapt its speed and maintain its predefined time-gap.

2. Cut-in (c.f. Sect. 4). The ego-vehicle follows a lead vehicle (aka. leader1) in the same lane that has a constant speed. A cut-in vehicle (aka. leader2) drives behind the ego vehicle in an adjacent lane. The cut-in vehicle overtakes the ego vehicle and then performs a cut-in maneuver with constant speed, while leader1 maintains its constant speed. The cut-in vehicle should be selected as target when it has crossed the lane marking. The ego vehicle ACC should re-establish the intended time gap with cut-in as the new lead vehicle (leader2).

3. Cut-out. The ego-vehicle follows a cut-out vehicle in the same lane with constant speed. The cut-out vehicle (aka. leader1) follows another vehicle leader2 in the same lane. The cut-out vehicle speed is not faster than leader2. The cut-out vehicle changes to an adjacent lane and speeds up to overtake leader2. The ego vehicle ACC should re-establish the time gap to leader2 as the new target vehicle to be followed.

4. Overtake. The ego-vehicle follows a lead vehicle leader1 in the same lane. The ego vehicle performs a manual lane change to the adjacent lane, and then speeds up to overtake leader1. Another vehicle leader2 is already driving ahead in the adjacent lane and lies front of the ego vehicle after its lane change. The ego vehicle ACC should re-establish the time-gap with leader2. After the ego vehicle passes leader1, and if there is sufficient gap between leader1 and leader3 (which lies ahead of leader1 in the same lane), the ego vehicle switches back to its original lane. The ego vehicle ACC should then re-establish its time-gap with leader3.

5.4 ACC Test Objectives

The objective of testing all four uses cases, was to look for violations of two global safety requirements. The first was a basic no crash/collision requirement which is considered safety critical. The second safety requirement is that the observed time gap deviation should never vary by more than 20% of the selected time gap. We modeled these safety requirements in ROBOTest as follows:

```
verdict[i] = always(collision = false &
   timeGap <= 2.2 & timeGap >= 1.8)
```

6 Results

Each of the four use cases presented in Sect. 5.3 was formally modeled as an n-step sequence of input and gate constraints (for appropriate n) using the modeling language presented in Sect. 4. Each constraint model was then embedded into its own ROBOTest configuration file, and the safety requirements of Sect. 5.4 were added as verdict constraints. The configuration file was then run in a test session on the integrated ASM-ROBOTest toolchain. Table 1 shows the results of the four test sessions.

(a) Input test case

Test vector#	TO1Vel	TO2Vel	TO3Vel	egoSteer	Test vector#	TO1Vel	TO2Vel	TO3Vel	egoSteer
1	50	55	50	0	10	50	60	50	0
2	50	55	50	0	11	50	70	50	0
3	50	55	50	left[100_2]	12	50	60	50	0
4	50	55	50	left[100_2]	13	50	50	50	right[100_2]
5	50	70	50	0	14	50	50	50	right[100_2]
6	50	70	50	0	15	0	50	40	0
7	50	70	50	0	16	0	50	30	0
8	50	70	50	0	17	0	50	30	0
9	50	70	50	0					

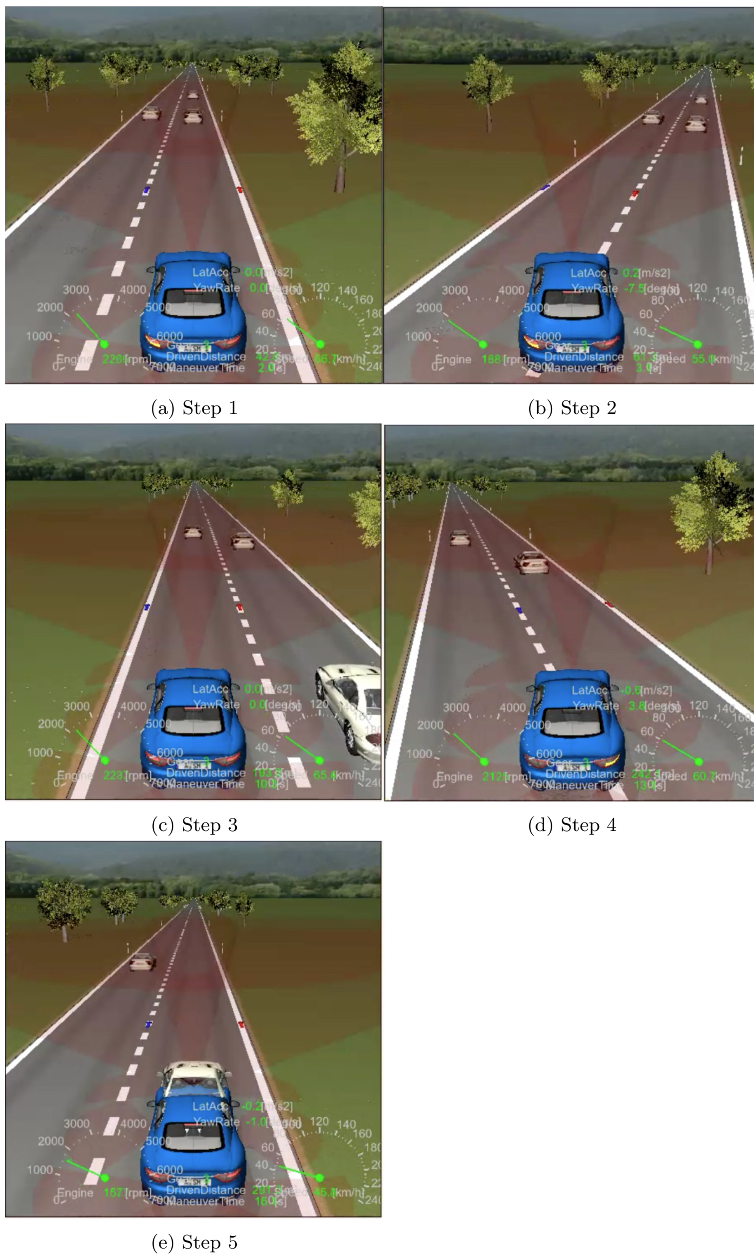
(b) Output observations

Time (s)	collision	ego2TO1Dist	ego2TO2Dist	ego2TO3Dist	timeGap	accTarget	TO1Spd	TO2Spd	TO3Spd	egoSpd
0.0	False	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	False	38.1	39.4	68.1	1.8	1.0	50.0	55.0	50.0	77.2
2.0	False	31.9	34.6	61.9	1.7	1.0	50.0	55.0	50.0	67.2
3.0	False	28.9	33.0	58.9	0.0	0.0	50.0	55.0	50.0	55.9
4.0	False	27.9	33.5	57.9	0.0	0.0	50.0	55.0	50.0	54.4
5.0	False	25.3	36.3	55.3	0.0	0.0	50.0	70.0	50.0	64.7
6.0	False	20.0	36.5	50.0	1.8	2.0	50.0	70.0	50.0	71.3
7.0	False	14.2	36.3	44.2	1.8	2.0	50.0	70.0	50.0	70.7
8.0	False	8.5	36.1	38.5	1.9	2.0	50.0	70.0	50.0	70.2
9.0	False	3.0	36.2	33.0	1.9	2.0	50.0	70.0	50.0	69.8
10.0	False	-2.3	33.7	27.7	1.8	2.0	50.0	60.0	50.0	67.1
11.0	False	-6.4	35.1	23.6	2.0	2.0	50.0	70.0	50.0	63.9
12.0	False	-10.2	34.1	19.8	1.9	2.0	50.0	60.0	50.0	63.4
13.0	False	-13.6	30.9	16.4	0.0	0.0	50.0	50.0	50.0	60.8
14.0	False	-16.5	28.0	13.5	0.8	3.0	50.0	50.0	50.0	62.5
15.0	False	-33.2	24.8	7.6	0.5	3.0	0.0	50.0	40.0	58.9
16.0	False	-48.0	23.9	1.2	0.1	3.0	0.0	50.0	30.0	46.9
17.0	True	-59.1	26.6	-1.5	0.0	0.0	0.0	50.0	30.0	34.2

Fig. 2. A failed test case for the Overtaking Scenario 5.3: (a) test inputs from ROBOTest, (b) test outputs from ASM

Table 1 shows that errors were found in two of the four use cases. It was easy to visually inspect the failed test cases reported by ROBOTest and confirm that the safety requirements were indeed violated (c.f. Fig. 2(b)). Furthermore, failed test cases could be played back through the ASM simulator in real time to visualise the full details. Figure 2 shows a complete failed test case for overtaking, consisting of 17 test vectors for the 4 input parameters that drive a 17 s simulation. Still images from replaying this test case in ASM can be seen in Fig. 3, where the ego (i.e. ACC host) vehicle is dark blue. Figure 3(e) shows the collision in Step 5. Such visualisations can yield further explanatory insight into *why* a test failure occurs. In this case, the test failures were mainly collision errors when a sudden speed change occurred.

Although use case errors were found in the SUT, the models in Table 1 were not fully converged (i.e. learning was incomplete) This was due to the relatively low data throughput of a single simulator license. Further research is needed to evaluate whether multi-threaded machine learning, using more than one simulator, can achieve full convergence (i.e. a completely learned model) in a reasonable time.



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

(e) Step 5

Fig. 3. (a),...,(e): ASM simulator images for all 5 use case steps in the failed overtaking use case test of Fig. 2 (Color figure online)

7 Related Work

Active automaton learning for testing is surveyed in [3], where the applications are mainly unit and integration testing. Our work represents the first attempt to apply ML to use case testing. The commonest models for automaton learning are deterministic automata [18,19,27,31,35], non-deterministic finite automata [5], and extended finite state machines [6]. Our work seems to represent the first attempt to use chains of intersecting finite automata.

There is a significant literature on TCG for use cases from UML models surveyed in [32]. UML sequence diagrams are sometimes seen as the canonical use case modeling language, and are prominent in the UML literature on TCG, e.g. [29]. The linear step ordering (see Sect. 2.1) common to both UML sequence diagrams [29] and informal models [8] is faithfully reflected in our CAML approach. UML state machine models are used in [33] for use case testing. By contrast, the CAML approach reverse engineers state machine models using ML, and thus avoids the effort of manual model construction and maintenance. Several authors have understood the need for constraints to automate use case testing e.g. [29], [24]. The UML object constraint language (OCL) has typically been used for this. By contrast, our constraints are based on linear temporal logic (LTL) and are conceptually closer to the live sequence charts of [16].

Testing semi- and fully autonomous vehicle software is a technically challenging emerging field where use case modeling languages such as OpenScenario [28] are currently under development. The case studies presented here extend previous research into automotive use case testing such as [4,25]. CAML addresses similar problems to the fuzz testing approach of [15]. However, our constraint-based approach to modeling and verdicts has wider scope and is more precise than the randomised approach of [15].

8 Conclusions and Future Work

We have introduced a constrained active machine learning (CAML) architecture that fully automates use case testing. This architecture can overcome the scalability problems associated with current active automaton learning algorithms such as L* when applied to highly constrained situations such as use case testing. We have benchmarked the CAML approach on typical use cases for an embedded automotive ADAS application, and demonstrated its efficiency and effectiveness. For this we implemented a prototype of CAML which was integrated with the industrial vehicle simulator ASM.

There is considerable scope for extension and improvement of our approach. Future research topics include: (i) additional constraints on use case models for greater ML efficiency and reduced automaton sizes, (ii) extensions of the constraint language for wider scope of use case and verdict modeling, and (iii) interfacing our constraint modeling language to open standards such as UML, LSC and OpenScenario.

Acknowledgement. We acknowledge the collaboration of Fengco AB in making available an ASM vehicle simulator license within the Vinnova funded project 2017-05501 Chronos-2. We also acknowledge the assistance of M. Höglund to connect CAML with ASM.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
2. Bennaceur, A., Hähnle, R., Meinke, K. (eds.): Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8
3. Bennaceur, A., Meinke, K.: Machine learning for software analysis: models, methods, and applications. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026, pp. 3–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_1
4. Bergenhem, C., Meinke, K., Ström, F.: Quantitative safety analysis of a coordinated emergency brake protocol for vehicle platoons. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018, Part III. LNCS, vol. 11246, pp. 386–404. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03424-5_26
5. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, pp. 1004–1009 (2009)
6. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–216 (2016)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2001)
8. Cockburn, A.: Writing Effective Use Cases, 1st edn. Addison-Wesley Longman Publishing Co. Inc., Boston (2000)
9. Cook, S., et al.: Unified modeling language (UML) version 2.5.1. Standard, Object Management Group (OMG), December 2017. https://www.omg.org/spec/UML/2.5.1
10. Fiterău-Broştean, P., Howar, F.: Learning-based testing the sliding window behavior of TCP implementations. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) FMICS/AVoCS 2017. LNCS, vol. 10471, pp. 185–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67113-0_12
11. Foster, D.: Generative Deep Learning: Teaching Machines To Paint, Write, Compose, and Play. O'Reilly, Sebastopol (2019)
12. Fournier, G.: Essential Software Testing. CRC Press, Boca Raton (2009). pB
13. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
14. Gold, E.M.: Complexity of automaton identification from given data. Inf. Control **37**, 302–320 (1978)
15. Han, J.C., Zhou, Z.Q.: Metamorphic fuzz testing of autonomous vehicles. In: ICSEW 2020: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. Association for Computing Machinery, New York (2020)
16. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-19029-2

17. De la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York (2010)
18. Hossen, K., Groz, R., Oriat, C., Richier, J.: Automatic model inference of web applications for security testing. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014 Workshops Proceedings, Cleveland, Ohio, USA, March 31 - April 4 2014, pp. 22–23. IEEE Computer Society (2014). https://doi.org/10.1109/ICSTW.2014.47
19. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
20. Jacobson, I., Magnus, C., Jonsson, P., Övergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley Longman Publishing Co. Inc., Harlow (1992)
21. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
22. Khosrowjerdi, H., Meinke, K., Rasmusson, A.: Learning-based testing for safety critical automotive applications. In: Bozzano, M., Papadopoulos, Y. (eds.) IMBSA 2017. LNCS, vol. 10437, pp. 197–211. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64119-5_13
23. Khosrowjerdi, H., Meinke, K., Rasmusson, A.: Virtualized-fault injection testing: a machine learning approach. In: 11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, 9–13 April 2018, pp. 297–308. IEEE Computer Society (2018). http://doi.ieeecomputersociety.org/10.1109/ICST.2018.00037
24. Li, B., Li, Z., Qing, L., Chen, Y.: Test case automate generation from UML sequence diagram and OCL expression. In: 2007 International Conference on Computational Intelligence and Security (CIS 2007), pp. 1048–1052 (2007)
25. Meinke, K.: Learning-based testing of cyber-physical systems-of-systems: a platooning study. In: Reinecke, P., Di Marco, A. (eds.) EPEW 2017. LNCS, vol. 10497, pp. 135–151. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66583-2_9
26. Meinke, K., Niu, F., Sindhu, M.: Learning-based software testing: a tutorial. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) ISoLA 2011. CCIS, pp. 200–219. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34781-8_16
27. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 134–151. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21768-5_11
28. Menzel, T., Bagschik, G., Isensee, L., Schomburg, A., Maurer, M.: From functional to logical scenarios: detailing a keyword-based scenario description for execution in a simulation environment. In: 2019 IEEE Intelligent Vehicles Symposium, IV 2019, Paris, France, 9–12 June 2019, pp. 2383–2390. IEEE (2019)
29. Nayak, A., Samanta, D.: Automatic test data synthesis using UML sequence diagrams. J. Object Technol. **9**(2), 115–144 (2010)
30. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII/PSTV XIX'99, IFIP TC6 WG6.1, pp. 225–240 (1999)
31. Raffelt, H., Steffen, B., Margaria, T.: Dynamic testing via automata learning. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 136–152. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77966-7_13

32. Shirole, M., Kumar, R.: UML behavioral model based test case generation: a survey. SIGSOFT Softw. Eng. Notes **38**(4), 1–13 (2013)
33. Shirole, M., Suthar, A., Kumar, R.: Generation of improved test cases from UML state diagram using genetic algorithm. In: Proceedings of the 4th India Software Engineering Conference, pp. 125–134 (2011)
34. Vasilevski, M.P.: Failure diagnosis of automata. Cybernetic **9**(4), 653–665 (1973)
35. Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J.: Increasing functional coverage by inductive testing: a case study. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16573-3_10
36. Wong, P., et al.: Testing abstract behavioral specifications. Int. J. Softw. Tools Technol. Transf. **17**(1), 107–119 (2015)
37. Özguner, U., Acarman, T., Redmill, K.: Autonomous Ground Vehicles. Artech House Publishers, Boston (2011)