Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study

Karl Meinke(✉)

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

Abstract. Learning-based testing (LBT) is a paradigm for fully automated requirements testing that combines machine learning with model-checking techniques. LBT has been shown to be effective for unit and integration testing of safety critical components in cyber-physical systems, e.g. automotive ECU software.

We consider the challenges faced, and some initial results obtained in an effort to scale up LBT to testing co-operative open cyber-physical systems-of-systems (CO-CPS). For this we focus on a case study of testing safety and performance properties of multi-vehicle platoons.

Keywords: Cyber-physical system · System-of-systems · Platooning · Model-based testing · Learning-based testing · Machine learning · Requirements testing

1 Introduction

A *cooperating cyber-physical system-of-systems* can be characterised by the use of wireless communication, multiple stakeholders, dynamic system definitions, and unpredictable operating environments. Such systems-of-systems have been termed *Cooperative Open Cyber-Physical Systems* (CO-CPS) [33]. It is assumed that no single stakeholder has overall system responsibility, and that cooperation relies on wireless communication to perform safety-relevant functions.

CO-CPS are emerging around the world, due to rapid progress in telecommunications, robotics and AI. Many examples can be found in Cooperative Intelligent Transport Systems (C-ITS) and intelligent manufacturing. However, they represent a great challenge to the software quality assurance (SQA) community. Not least, the cyber-physical character of CO-CPS means that the impact of safety and security incidents (malicious or unintended) is potentially very high. However, if we survey the range of current technologies available for SQA, we can find significant limitations in many current approaches to quality assurance of CO-CPS.

On the one hand, the dynamic and heterogeneous nature of CO-CPS makes a full static analysis technically difficult. The sheer scale of many proposed CO-CPS suggests that a full system-of-systems analysis would even be technically

P. Reinecke and A. Di Marco (Eds.): EPEW 2017, LNCS 10497, pp. 135–151, 2017.
DOI: 10.1007/978-3-319-66583-2_9

infeasible. Furthermore, it is unclear (for commercial reasons) whether all source code in a CO-CPS would ever be made available for this. Static analysis of the individual components by their vendors might be technically feasible. However, it is difficult to see how such low-level component analysis could take into consideration unpredictable environment factors and high-level emergent phenomena (such as physical collisions). For this reason, software testing, laboratory simulations and field tests are the de-facto SQA standard used in industry today. Here the problem is that software testing traditionally focuses on unit, integration and system level testing. Simulation and field testing can be reliable and decisive at the level of systems-of-systems, but tend to be slow and unsystematic in their coverage. There is thus a great need to perform systematic and fully automated requirements testing on CO-CPS.

The scalability problem for quality assurance of CO-CPS might be made more tractable by taking a *model-based approach*, using judicious abstraction to suppress irrelevant technical detail. However, one is still faced with the fact that not all software vendors will take a model-driven approach, let alone exchange their models, to protect intellectual property (IP). Therefore, in the worst case one would be left to perform a model based analysis where some component models are known, but others are missing, inconsistent with code, or out of date.

Against this background situation for CO-CPS, within the EU ECSEL project SafeCOP[1], we are evaluating the potential of a technology known as *learning-based testing* (LBT) [23,24]. LBT is a paradigm which combines techniques from *model-driven development* (e.g. model-based testing, model checking of safety requirements etc.) with *machine learning*. The basic idea is to use machine learning to *reverse engineer* a behavioral system model from runtime observations of a system under test (SUT). Since LBT is a black-box technique, it is code and platform independent, potentially scalable, and need not infringe upon component IP rights. The runtime SUT observations can be made either by laboratory simulation (e.g. software-in-the-loop SIL, hardware-in-the-loop HIL) or field testing. The learned model can then be used to analyse safety properties [11], and even security properties [14], by using appropriate tools such as model checkers. Potential system anomalies discovered during model analysis are confirmed by executing the corresponding test cases on the SUT.

We present here some initial results of applying LBT to a case study of testing co-operative vehicle platoons [4]. One reason for choosing this case study is because the problem size can be scaled up uniformly by adding more vehicles. This allows us to measure the influence of different factors on the scalability of LBT technology.

The case study of platooning presented here is a first attempt to address two important questions about state-of-the-art LBT technology:

(1) how well does recently developed multi-core based LBT technology scale up to testing complex CO-CPS scenarios;
(2) how do problem size and other factors affect scalability?

[1] See www.safecop.eu.

The organisation of this paper is as follows. In Sect. 2 we review fundamental concepts and the state-of-the-art in learning-based testing. In Sect. 3, we consider the architecture and functionality of platooning as a CO-CPS. In Sect. 4 we present our case study of LBT applied to a platoon model. In Sect. 5 we survey related work in the literature. Finally in Sect. 6, we draw conclusions from our initial results, and comment on future research directions.

2 Learning-Based Testing

In this section, we review some fundamental principles of learning-based testing as these have been implemented in our research tool LBTest. The earliest version of this tool (LBTest 1.x) has been described in [26]. Therefore we will focus on the latest tool architecture LBTest 3.x, presented in Fig. 1. In Sect. 2.1 we use this architecture to explain the basic principles of LBT. Then, in Sects. 2.2 and 2.3, we show how concurrent aspects of this architecture contribute towards solving tool scalability issues[2].

2.1 Principles of LBT

LBTest uses active automaton learning aka. *regular inference* (see e.g. [13]) to generate queries about a black-box system, which can be used to infer a behavioral model in polynomial time [2].

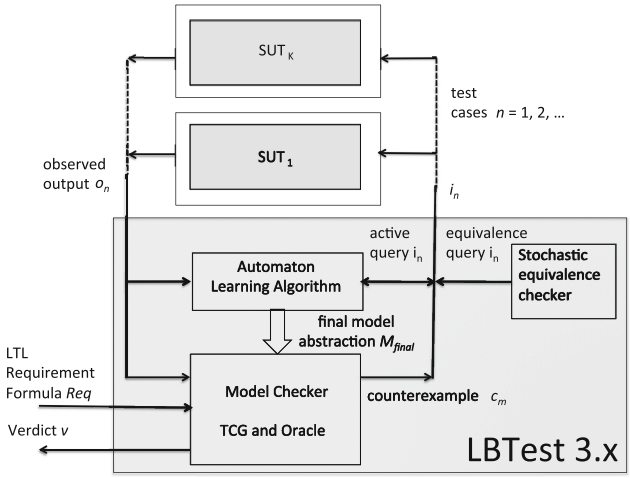

Fig. 1. LBTest 3.x concurrent learning architecture

[2] This architecture has been developed within the VINNOVA FFI project VIRTUES, http://www.csc.kth.se/~karlm/virtues/.

For requirements testing, partial models of the SUT can be subjected to model checking against a temporal logic requirement specification, even before the learning process is complete. In LBTest, propositional linear temporal logic[3] (PLTL) is used as a requirements modeling language. This particular logic has the advantage that test cases can easily be extracted from counterexamples generated by a model checker. LBTest makes use of a loosely integrated symbolic checker NuSMV [7]. We are also developing a more tightly integrated explicit state model checker for efficiency reasons. These two processes of learning and model checking may be interleaved, an idea first suggested in [27]. Then they incrementally build up a sequence $M_1, M_2, ...$ of models of the SUT, while generating and executing requirements test cases on each model M_i.

To separate true counter-examples (SUT errors) from false counter-examples (artifacts of an incomplete model) it is necessary to validate each counter-example derived from model checking. For this we can: (i) extract a test case representing the counter-example[4], (ii) execute it on the SUT, (iii) apply an equality test that compares the observed SUT behavior with the predicted bad behavior from the model, and (iv) automatically generate the test verdict (`pass, fail`) from step (iii). The soundness of this process relies on the soundness of the underlying model checker, and the soundness of equality testing.

The completeness of LBT relies on the completeness of the underlying model checker, as well as convergence results about the learning algorithms which are used (see [13]). However, within practical case studies of large complex systems it may not be possible for learning to be completed in any reasonable time frame (see e.g. [11]). This problem is significant for CO-CPS. Therefore, development of LBTest has focused on incremental learning algorithms that can generate incomplete approximating models of the SUT in small increments.

One measure of the coverage achieved by LBT is in terms of the behavioral accuracy of the final model. This accuracy could be defined in terms of trace inclusion between the model and the SUT. However, phenomena of both over and under approximation often occur within the same partial model, i.e. no strict trace inclusion holds either way. Nevertheless, by using a *probably exactly correct* (PEC) model of convergence, we can obtain a satisfactory black-box *convergence measure* as follows.

Figure 1 illustrates the stochastic equivalence checker used in LBTest 3.x. This checker empirically estimates the behavioral accuracy of the final learned model M_{final} for replicating the behavior of the SUT on a randomly chosen set of input sequences. For this, the input sequences are executed both on the SUT and the model. We then measure the percentage of behaviorally identical output sequences generated by both. This convergence model is related to the *probably approximately correct* (PAC) convergence model of [30], but for PEC

[3] Recall that propositional LTL extends basic propositional logic with the temporal modalities `G(`ϕ`)` (always ϕ), `F(`ϕ`)` (sometime ϕ) and `X(`ϕ`)` (next ϕ). Other derived operators and past operators may also be included. See e.g. [12] for details.

[4] Infinite counter-examples to LTL liveness formulas are truncated around the loop, and the weaker test verdict `warning` may be issued.

the probability of exact identity (not approximate equivalence) is estimated. PEC convergence aims at the needs of software safety analysis over the discrete data type partitions commonly employed in testing.

2.2 Towards Scalable LBT Architectures

From empirical studies such as [11,20,25] we have observed two important obstacles to scaling up LBT methods for large and complex SUTs. These are:

(i) *the tendency for learned model size to increase rapidly with SUT size*;
(ii) *the tendency for test latency (i.e. the time to execute a single test case) to increase with SUT size.*
Even worse, these two problems compound one another, leading to long test session times and low final convergence measures. In benchmarking the architectural proposal of [27] we have also observed another significant problem:
(iii) *model checking each member M_i of a converging sequence of models $M_1, M_2, ...$ is highly inefficient, and does not seem to improve the rate of model convergence.*

We will consider each of these issues, and how it can be addressed, in turn.

(i) Model Size. The size of a learned model is a function of the code complexity of the underlying SUT, as well as the number of parameters of the SUT which the learning algorithm tries to stimulate and observe.

One factor influencing model size is the number of SUT input variables and the number of test values chosen for each input variable. These parameters bound the number of *exit transitions* from each model state. The number of exit transitions is further influenced by the combinatorial strategy used to generate composite input test vectors from the individual input variable values. A judicious combinatorial choice is necessary to control the otherwise exponential explosion in the number of transitions. In LBTest 3.x, n-wise testing [17] is available as a combinatorial strategy.

Another factor influencing model size is the number of observed SUT output variables, and the number of output value partition classes for each output variable. These factors influence the number of *states* in a learned model, since more output variables and finer output partitions lead to more easily distinguished SUT states.

So, a judicious choice of model accuracy, combinatorial test strategy and model abstraction can all be applied to improve the efficiency of learning and testing.

Besides these test configuration parameters, the problem of large model sizes has also been ameliorated by new research into machine learning algorithms. Since Angluin's seminal algorithm [2], many new learning algorithms, that can learn a model with fewer and/or shorter queries, have been derived, e.g. [16].

(ii) Test Latency. Improvements in learning and model checking algorithms are scarcely able to overcome a distinctive feature of large complex SUTs which is the tendency towards long test latency or execution times. For CO-CPS, communication network delays also become significant. Test latency times can become a significant component of an LBT test session duration.

Test latency can be ameliorated by executing test cases concurrently. With this aim we have conducted research into parallelized learning algorithms on multi-core platforms. Already in [15] certain improvements in learning performance by parallelization have been reported. An important challenge is to systematically characterize such improvements in terms of problem size parameters. Our work contributes to this area by studying a *parameterized and uniformly scalable learning problem* namely platooning. As the size (i.e. number of vehicles) of a platoon of identical vehicles scales up, the problem parameters:

(i) total number of lines of code under test, and
(ii) total number of program registers determining the global state space,

both increase linearly. Thus it becomes meaningful to compare testing results for different platoon sizes (c.f. the similar curves in Fig. 4). Without such uniform properties, benchmark results across an ad-hoc collection of SUTs can be very difficult to interpret.

(iii) Model Checking Overheads. Incremental learning generates a convergent sequence of models $M_1, M_2, ...$ However, each model M_i will contain a good many structural features (states and transitions) that persist in model M_{i+1}. It is beyond the capability of any model checker we know of to identify these persistent features and avoid checking them twice in both M_i and M_{i+1}. Therefore, a long model generation sequence will contain significant redundant model checking effort. Our empirical observations with LBTest 2.x and NuSMV have shown that this redundant checking can consume more than 50% of the overall test session time. Furthermore, as reported in [21], model checker generated queries have not been observed to accelerate the convergence of learning in any case study so far.[5]

While it might be possible to introduce a sophisticated delta-oriented approach to model checking, the simplest solution seems to be to defer model checking until after machine learning.

2.3 Concurrent Multi-core LBT

Figure 1 illustrates a new architecture for LBT that significantly departs from the proposals of [23,27]. Two new features are prominent, and both are intended to counter the scalability bottlenecks described in Sect. 2.2.

[5] It seems possible to theoretically explain this observation for certain types of formulas by considering their semantics. However, this is outside the scope of our present discussion.

Firstly, the new architecture supports parallel execution of multiple instantiations of the SUT on a multi-core platform. The aim is to mitigate long SUT test latency. At the start of a test session, LBTest clones K versions of the executable SUT, each within its own external OS process. The value of K is chosen as a function of the number of SUT input values to be tested. Once started, each SUT process persists throughout the learning phase, and acts as a server to answer certain kinds of queries about SUT behavior. Different load balancing schemes on these query servers are used according to the learning strategy.

Of course, concurrent execution is a rather obvious solution to test latency. The real technical challenge here is to devise efficient parallel learning algorithms that can allow multiple threads to efficiently and safely perform concurrent updates on a single shared automaton model. At the same time we need to optimise multi-core usage on the hardware level. For this we have investigated concurrent implementations of Kearn's algorithm [19]. For reasons of space, these rather complex concurrent algorithms will be described elsewhere.

The second new feature of LBTest 3.x is its support for deferred model checking, as described in Sect. 2.2, using an iteration bound to terminate learning. Only when learning is terminated do model checking and counter-example validation of the final model M_{final} begin in a second phase. This minimises the redundant model checking identified in Sect. 2.2.

3 Platooning as a CO-CPS

In this Section we review some general features of platooning that characterise it as a CO-CPS. Then we discuss the particular platooning model that was tested in Sect. 4.

3.1 General Principles of Platooning

Platooning technology is sometimes called an "electronic towbar" between road vehicles, and this phrase gives much insight into the idea.

A *platoon* consists of a sequence of road vehicles $V_1, ..., V_n$ which (by means of sensors, wireless V2V communication and control algorithms for longitudinal or distance control) are able to maintain a fixed distance x_r between one another and a relative velocity $v_r = 0$ under normal cruising conditions. (See Fig. 2, adapted from [5].) The *lead vehicle*, V_1, is under manual control by a qualified platoon leader who needs to have the necessary technical skills to control the platoon. The vehicles $V_2, ..., V_n$ are its *followers*, and may be autonomous or semi-autonomous, depending on the extent to which lateral control (i.e. steering) is automated.

A platoon may be heterogeneous, consisting of different models from different vendors carrying different payloads. It should be possible to add and remove vehicles dynamically during a journey, and there are many safety critical use cases, such as lane change, emergency braking etc.

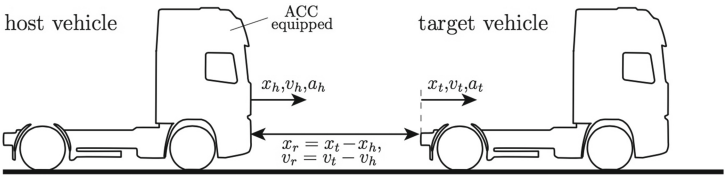

Fig. 2. Platoon vehicle pair: V_{i+1} (left) and V_i (right)

The interest in platooning technology, lies in the possibility to reduce fuel consumption and corresponding CO2 emissions, as well as to improve road usage and safety while reducing traffic congestion (see e.g. [29]). Platoons exploit the reduced aerodynamic drag that arises with short inter-vehicle distances. There is an important trade-off between fuel efficiency and safety in platoon design, since drag is reduced by shorter inter-vehicle spacing. System response times, component reliability, road hazards and the effects of safety critical uses cases such as emergency braking on the platoon and its environment all need to be evaluated during software design.

3.2 A Simple Platooning Model

For pragmatic reasons, our study of LBT scalability was restricted to software-in-the-loop (SIL) testing of a basic platoon simulator. The simulation is 1-dimensional, meaning that no steering model is used. The simulator is therefore only able to analyze certain use cases, such as straight-line cruising and emergency braking. Other use cases need a more complex simulation model, and this is the subject of ongoing research and industrial collaboration. However, our model includes many important physical characteristics such as maximum engine and brake torque, vehicle mass, aerodynamic drag etc. defined using a *point-mass* modeling approach. (See e.g. [34] for an introduction to vehicle modeling.)

The simulator consists of about 2000 Java LOC. However, to get a clearer impression of the underlying SUT complexity we provide here some details about its structure and function.

The block architecture of a single vehicle in the platoon simulator is illustrated in Fig. 3. This depicts a *brake-by-wire* BBW subsystem augmented with a *co-operative adaptive cruise controller* CACC. The latter is connected to an *odometry* unit ODOM (providing host vehicle position and velocity) and a *wireless communication* WCOM unit (relaying host and target positions and velocities). Odometry is based upon host velocity measurements[6]. The WCOM unit simulates a 2 ms inter-vehicle wireless message delay, without any transmission error model.

The CACC controller is a crucial component that provides longitudinal control of each follower vehicle. It dynamically issues accelerator and brake torque

[6] In practise, GPS localisation would be relied upon for greater accuracy.

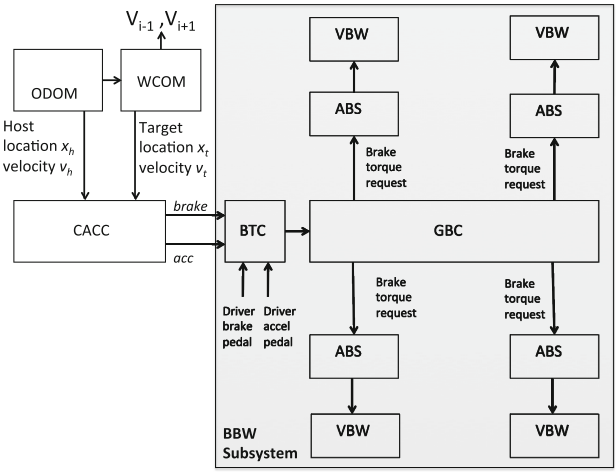

Fig. 3. Software architecture for platoon vehicle V_i

requests to maintain the position of the host vehicle within maximum and minimum distances from the target vehicle in front. A wide variety of CACC algorithms have been proposed in the literature. The controller tested here is a simple PD control algorithm with adaptive parameters, taken from [5]. For a general introduction to PID control theory one may consult e.g. [10]. The function of any PID controller in the context of an ACC problem is to maintain the relative position of the host vehicle V_{i+1} within the boundaries $x_{r,d,max}$ and $x_{r,d,min}$ (metres) from the target V_i, where

$$x_{r,d,max} = t_{hw} \cdot v_h + x_{r,0}, \qquad x_{r,d,min} = (t_{hw} - t_{hw,\delta}) \cdot v_h + x_{r,0}.$$

Here t_{hw} (seconds) is the *time headway* between V_{i+1} and V_i, and $t_{hw,\delta}$ causes a small difference in headway. The parameter $x_{r,0} > 0$ (m), maintains a safe relative inter-vehicle distance at $v_h = 0$ (m/s), to support so called *stop-and-go* functionality. The host position is maintained by two PD equations:

$$acc = K_{ACC}(k_{x_r} \cdot (x_r - x_{r,d,max}) + k_{v_r} \cdot v_r),$$

$$brake = K_{ACC}(k_{x_r} \cdot (x_r - x_{r,d,min}) + k_{v_r} \cdot v_r),$$

governing requested accelerator and brake torque. In the above formulas: (i) K_{ACC} (dimensionless) is a constant overall gain parameter. (ii) $x_r = x_t - x_h$ (metres) and $v_r = v_t - v_h$ (metres/second) are the relative distance and velocity to the target vehicle (c.f. Fig. 2). (iii) k_{x_r} is the P action: this gain is tuned to regulate the distance error to zero ($x_r - x_{r,d,max} = 0$ for *acc* and $x_r - x_{r,d,min} = 0$ for *brake*). (iv) k_{v_r} is the D action and the regulated error is v_r. (v) Since *acc* is smaller than *brake* (due to a different desired distance), it takes some time before the brakes are activated after the accelerator is released.

In this PD controller design, k_{x_r} and k_{v_r} are dimensionless *adaptive parameters*:

$$k_{x_r} = k_{x_r,1}(v_h) \cdot k_{x_r,2}(x_r - x_{r,d,max}), \qquad k_{v_r} = k_{v_r}(x_r - x_{r,d,max}).$$

All forces acting on the vehicle, both positive and negative, are resolved at each wheel individually.

To inject *behavioral faults* into our platooning model for testing, we replaced the non-linear adaptive parameter functions $k_{x_r,1}, k_{x_r,2}, k_{v_r} : \mathbb{R} \to \mathbb{R}$ of [5] with highly simplified piecewise linear approximations. These linear approximations to non-linear functions make the brake and accelerator control responses, *acc* and *brake*, less smooth with both over- and under-compensation for change, as we show in Sect. 4.2.

For each follower vehicle, the BBW subsystem takes the accelerator and brake torque requests from CACC, and translates these into forces on the four vehicle body wheels VBW[7]. The brake torque controller BTC calculates the global brake torque request (in Newton metres)

$$torqueRequest = (brake/100) \cdot maxBrakeTorque$$

and the global brake controller GBC distributes this brake request to each anti-locking brake system ABS_i, which controls wheel VBW_i.

The fundamental simulation cycle corresponds to 1 ms of real-world time, while the various architectural components have execution cycle times varying between 2 and 20 ms. Normally, vehicle software components would communicate periodically (but not necessarily deterministically) using the vehicle's CAN bus network, while the vehicles themselves communicate asynchronously. However, it is common industrial practise to perform SIL testing using a simplified *synchronous composition* of components to ensure reproducibility of test results. So our platoon simulator is also based on a synchronous composition of all architectural components, as well as the platoon vehicles themselves.

4 Test Experiment Design and Results

In this section, we first describe our testing experiment conducted on the platooning simulator described in Sect. 3, using the LBT tool architecture described in Sect. 2.3. We then describe the test results obtained, and interpret these from the perspective of LBT scalability.

4.1 Test Experiment Design

To test the primary use case of *high-speed cruising* for a platoon configuration of n vehicles, we focused on emulating the lead driver behavior, since in our

[7] For the lead vehicle, CACC is disabled and accelerator and brake pedal values are used by BBW instead. See Fig. 3.

simulator all follower vehicles autonomously adapt to this. Thus, each test case tc for an n-vehicle platoon consisted of a sequence $tc = (r_1, r_2, ..., r_\lambda)$ of lead driver accelerator and brake torque requests r_j. The continuous input spaces for each of these two input variables (accelerator and brake pedal angles) were sampled at 10% intervals, yielding $K = 21$ symbolic input values $0, a_1, ..., a_{10}, b_1, ..., b_{10}$ ranging from 0% to 100% pedal depression[8]. No assumptions were made about lead driver behavior, so both excessive and sporadic acceleration and braking could occur. The time headway t_{hw} between each successive pair of vehicles was nominally set to 2.0 s. A time headway of this size is normally quite safe for commercial CACC algorithms (see e.g. [5]).

For each test case $tc = (r_1, r_2, ..., r_\lambda)$, the length λ and torque requests r_j were chosen dynamically both by the learning algorithm and the equivalence checker. In the experiments of Sect. 4.2, λ typically took an average value around 12. The test case tc was then submitted to one of $K = 21$ SUT server processes S_p executing an n vehicle platoon simulator instance. The communication wrapper around S_p loaded and executed the request sequence $(r_1, r_2, ..., r_\lambda)$ sequentially. Each torque request value r_j was maintained constantly for a nominal 5 s (5000 simulation cycles). Thus the length of the simulation corresponding to tc was 5λ virtual seconds. The values chosen for λ were sufficient to reach high cruising speeds, in excess of 110 km/h.

Maintaining the torque request over a fixed number of seconds is a *temporal abstraction* technique necessary to achieve a balance between long simulation times and small final model size. This abstraction can be adjusted in the simulator. It also has the advantage that we can easily calculate the cumulative virtual simulation time for an entire test session.

The principle SUT output recorded for the test case tc was the time sequence of inter-vehicle gaps $x^i_{r,0}, \ldots, x^i_{r,\lambda}$, for each vehicle $i = 1, \ldots n-1$. Here, the time sequence term $x^i_{r,t}$, for $0 \leq t \leq \lambda$, represents the gap between the host-target pair, V_i and V_{i+1} measured at the end[9] of $5t$ virtual seconds (i.e. $5000t$ simulation cycles). The continuous values of each distance observation $x^i_{r,t}$ were partitioned within the communication wrapper into three discrete equivalence classes:

`tooClose`, `tooFar`, `good`,

based on the (host velocity dependent) distance boundaries $x^i_{r,d,min}$ and $x^i_{r,d,max}$. Thus the symbolic output `good` for $x^i_{r,t}$ represented the output partition class $x^i_{r,d,min} \leq x^i_{r,t} \leq x^i_{r,d,max}$.

To gain further insight into the physical state space covered by testing we also observed the lead vehicle velocity values $v^1_0, \ldots, v^1_\lambda$ and acceleration values $a^1_0, \ldots, a^1_\lambda$ at the same observation times. These continuous valued observations were partitioned into 1 km/h and 1 km/h^2 equivalence classes.

[8] Thus a_{10} represents 100% accelerator depression, a_9 represents 90% depression, etc. Simultaneous depression of both pedals is handled as a brake request by the BBW component.

[9] It is also possible to use SUT observations between the output cycles by thresholding. This can yield greater accuracy, but this approach was not taken here.

With regard to system-of-systems requirements, the most fundamental requirement is that all n platoon vehicles should always maintain a safe but fuel efficient distance between each other. This test requirement could be represented in PLTL for an $n+1$-vehicle platoon (where $n \geq 1$) by the safety formula:

$$\mathtt{G}(\,\mathtt{Distance_1} = \mathtt{good}\,\&\,\mathtt{Distance_2} = \mathtt{good}\,\&\,\ldots\,\&\,\mathtt{Distance_n} = \mathtt{good}\,). \quad (*)$$

Here $\mathtt{Distance_i}$ represents the discretized gap between vehicles V_i and V_{i+1} corresponding to measurements $x^i_{r,t}$.

One experimental goal was to try to observe the injected errors in the CACC component, (described in Sect. 3.2) as violations of the test requirement (*). The other goal was to characterise the scalability of the tool.

4.2 Test Experiment Results

The test experiment described in Sect. 4.1 was conducted for platoon sizes $n = 2, \ldots, 6$ to investigate the scalability of the testing tool. To uniformise the results, each platoon vehicle in each configuration had identical physical parameters[10]. We measured the final model size for different platoon sizes and different test session durations. While test session duration is a platform dependent measurement[11], it was felt that this value gave good insight into tool usability and potential future improvements.

Figure 4 shows the growth of model size over time for platoon sizes $n = 2, \ldots, 6$ using concurrent learning. To analyse the benefit of concurrency, Fig. 4 also shows model growth for $n = 3$ under sequential learning. Note that the y-axis is in thousands of states (Kstates). The graph shows the effects of increasing test latency as the platoon size increases. The largest inferred model (for $n = 6$) had over 64,600 states and 1.35 million transitions achieved after 20 h and 25 min of learning. During this time, 1.5 million test cases tc were executed, with an average test case length of $\lambda = 10.6$. Since each step in tc corresponds to 5 virtual seconds, the total virtual testing time was over 22,000 h.

Notable in Fig. 4 is the gradual slowdown in rates of model growth over time. However, there is no sharp fall in tool performance. Furthermore, the vertical intervals between the curves are very similar, both for increasing n and t. These two characteristics seem to suggest good scalability properties for our approach as a function of the problem size n.

With regard to requirements errors, NuSMV developed a segmentation fault already with the smallest of our models for $n = 2$ (8826 states, 185 K transitions). However, using our explicit state model checker on the largest model for $n = 6$ (64,671 states, 1.35 million transitions), the error `tooFar` was found to occur in 50,076 states (77% of all states), while the error `tooClose` was found in just 101

[10] Non-homogeneous platoons could also be tested using our approach.

[11] The actual platform used was a 4-core MacBook Pro, Mid 2014, running Yosemite OS-X 10.10.5 with 2.8 GHz Intel Core i7, 16 GB 1600 MHz DDR3 and 1 TB static disk flash storage.

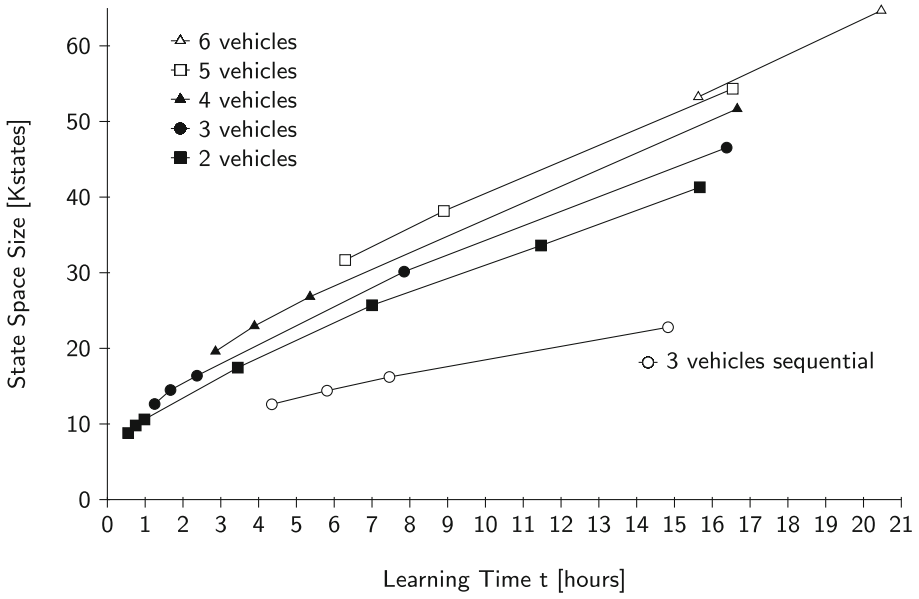

Fig. 4. Rates of model growth (state space size) over time for different platoon sizes.

states (0.0015% of all states) after 32.4 s of model checking. All errors proved to be valid SUT errors when corresponding test cases were executed on the SUT. The error `tooClose` was found only at low velocities, mainly at $v^1 = 0$, which seems to confirm the thesis of [5] that stop-and-go functionality is rather difficult to implement correctly. For the smallest model of $n = 2$ (8826 states), the error `tooFar` could also be found after 19 ms of model checking, but not error `tooClose`.

Through runtime monitoring, we estimated long term multi-core usage to range between 85%–95% over the problem size range $n = 2, \ldots, 6$, with approximately 10% fluctuations short term[12]. At peak core usage, CPU idle time was less than 1%, implying that further cores would have been of benefit.

For the experiments described in Fig. 4, the platoon models reached maximum convergence values of 9.4%, 9.4%, 8.8%, 7.1% and 6.0% for $n = 2, \ldots, 6$ respectively.

5 Related Work

The application of machine learning to testing has a somewhat long history, beginning with [32]. The architecture used in LBTest 2.x first appeared in [27] and was independently proposed in [24]. However, scalability and the effect of model checking on convergence, were not originally considered. Recently, the

[12] Based on 1 s sampling.

literature on machine learning applied to software engineering has become quite extensive. Known techniques use models based on deterministic automata [14, 16,23,28,31], non-deterministic finite automata [21], and extended finite state machines [6]. The emphasis ranges from unit and integration testing to software documentation. A state-of-the-art survey is [3]. Our experience [22] suggests that machine learning of hybrid automata would be too slow to deal with complex continuous state CO-CPS such as platoons.

To our knowledge, only one other study of parallelized machine learning for testing exists, namely [15]. This shares our premise that parallel learning is important to mitigate test latency. However, it evaluates only synthetic SUT latency obtained by inserting a 5 ms busy waiting loop into each SUT call. Model checking and requirements testing are not considered. The authors investigate speedup of learning randomly generated SUTs of different state space sizes in the range $1, \ldots, 256$ states. They conclude that under an increasing number of cores, a saturation point is met, where adding more cores yields no benefit[13]. By contrast, we have varied a much larger problem size $8K, \ldots, 64K$ states, keeping the core number fixed.

Platooning has been widely studied in the C-ITS literature. A survey of platooning research is [4]. An account of traditional SIL and HIL testing of a 3 vehicle platooning system is [1]. This work has very similar safety concerns to our own. Examples of static analysis applied to platooning are [8,9,18] where it is shown that verifying vehicle code does not scale to the whole system-of-systems, and a mixed top-down and bottom up strategy are applied.

6 Conclusions and Future Work

We have presented an initial assessment of the scalability of multi-core learning-based testing technology to cyber-physical systems-of-systems (CO-CPS). For this we have conducted testing experiments on a vehicle platooning simulator, where we have injected faults that violate safety and fuel efficiency requirements. Extensive testing experiments over different platoon sizes have demonstrated that learned model size scales well over the experimental time horizon and different platoon sizes. However, unsurprisingly perhaps, model convergence is low, at least according to the current PEC metric. Nevertheless effective testing, capable of finding valid SUT errors (both common and rare) was possible by learning large but incomplete models.

Future research needs to address several issues. Learning efficiency needs to be further improved to enhance coverage. Our study could be generalized by using more advanced simulators to test other use cases. We will also further consider how to scale up LBT to many-core platforms. Can the saturation effects cited in [15] be observed or avoided? The reliability questions surrounding incomplete model learning warrant further attention, e.g. the optimal choice of a learning convergence metric is an open question. Finally, equation (*) of

[13] Unfortunately our limited computing platform did not provide an opportunity to evaluate this result.

Sect. 4.1 represents a safety requirement that could be captured by a suitable *spatio-temporal logic*. Further study of spatio-temporal logics and model checking might be fruitful for CO-CPS use case testing.

This research has been funded by VINNOVA FFI project 2013-05608 VIRTUES and the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 692529 project SafeCOP.

References

1. Aki, M., Zheng, R., Yamabe, S., Nakano, K., Suda, Y., Suzuki, Y., Ishizaka, H., Kawashima, H., Sakuma, A.: Safety testing of an improved brake system for automatic platooning of trucks. Int. J. Intell. Transp. Syst. Res. **12**(3), 98–109 (2014)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Bennaceur, A., Giannakopoulou, D., Hähnle, R., Meinke, K.: Machine learning for dynamic software analysis: potentials and limits (Dagstuhl seminar 16172). Dagstuhl Rep. **6**(4), 161–173 (2016)
4. Bergenhem, C., Shladover, S., Coelingh, E., Englund, C., Shladover, S., Tsugawa, S.: Overview of platooning systems. In: Proceedings of the 19th ITS World Congress, Vienna, October 2012
5. van den Bleek, R.: Design of a hybrid adaptive cruise control stop-&-go system. Master's thesis, Technische Universiteit Eindhoven, Department of Mechanical Engineering (2007)
6. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Form. Asp. Comput. **28**(2), 233–263 (2016)
7. Cimatti, A., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). doi:10.1007/3-540-45657-0_29
8. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Using CSP||B components: application to a platoon of vehicles. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 103–118. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03240-0_11
9. El-Zaher, M., Contet, J., Gruer, P., Gechter, F., Koukam, A.: Compositional verification for reactive multi-agent systems applied to platoon non collision verification. Stud. Inform. Univ. **10**(3), 119–141 (2012)
10. Engelberg, S.: A Mathematical Introduction to Control Theory. Imperial College Press, London (2015)
11. Feng, L., Lundmark, S., Meinke, K., Niu, F., Sindhu, M.A., Wong, P.Y.H.: Case studies in learning-based testing. In: Yenigün, H., Yilmaz, C., Ulrich, A. (eds.) ICTSS 2013. LNCS, vol. 8254, pp. 164–179. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41707-8_11
12. Fisher, M.: An Introduction to Practical Formal Methods Using Temporal Logic. Wiley, Hoboken (2011)
13. De la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, Cambridge (2010)
14. 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, pp. 22–23 (2014)

15. Howar, F., Bauer, O., Merten, M., Steffen, B., Margaria, T.: The teachers' crowd: the impact of distributed oracles on active automata learning. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) ISoLA 2011. CCIS, pp. 232–247. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34781-8_18
16. 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). doi:10.1007/978-3-319-11164-3_26
17. Jorgensen, P.C.: Software testing (2008)
18. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal verification of autonomous vehicle platooning. CoRR abs/1602.01718 (2016)
19. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
20. Khosrowjerdi, H., Meinke, K., Rasmusson, A.: Automated behavioral requirements testing for automotive ECU applications. In: Proceedings of the 5th International Workshop on Model Based Safety Analysis. In: IMBSA 2017. LNCS. Springer (2017, to appear)
21. Meinke, K.: Recent progress in learning-based testing. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits: Proceedings of Dagstuhl Workshop, vol. 16172. Springer (2017, to appear)
22. Meinke, K., Niu, F.: An incremental learning algorithm for hybrid automata (2013)
23. 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). doi:10.1007/978-3-642-21768-5_11
24. Meinke, K.: Automated black-box testing of functional correctness using function approximation. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), pp. 143–153. ACM Press (2004)
25. Meinke, K., Nycander, P.: Learning-based testing of distributed microservice architectures: correctness and fault injection. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9509, pp. 3–10. Springer, Heidelberg (2015). doi:10.1007/978-3-662-49224-6_1
26. Meinke, K., Sindhu, M.A.: Lbtest: a learning-based testing tool for reactive systems. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST 2013), pp. 447–454. IEEE Computer Society (2013)
27. 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). doi:10.1007/978-0-387-35578-8_13
28. 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). doi:10.1007/978-3-540-77966-7_13
29. Sjöberg, K.: Platooning - challenges and opportunities (2016). https://docbox.etsi.org/Workshop/2016/201603_ITS_WORKSHOP/S04_TWDS_ACCIDENT_FREE_AUTOMATED_DRIVING
30. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)

31. 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). doi:10.1007/978-3-642-16573-3_10
32. Weyuker, E.: Assessing test data adequacy through program inference. ACM Trans. Program. Lang. Syst **5**(4), 641–655 (1983)
33. Willke, T., Tientrakool, P., Maxemchuk, N.: A survey of inter-vehicle communication protocols and their applications. IEEE Commun. Surv. Tutor. **11**(2), 3–20 (2009)
34. Özguner, U., Acarman, T., Redmill, K.: Autonomous Ground Vehicles. Artech House Publishers, Boston (2011)