Buzhinsky, Igor; Vyatkin, Valeriy

Testing automation systems by means of model checking

Published in:
Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017

DOI:
10.1109/ETFA.2017.8247579

Published: 04/01/2018

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Buzhinsky, I., & Vyatkin, V. (2018). Testing automation systems by means of model checking. In *Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017* (Vol. Part F134116, pp. 1-7). (Proceedings IEEE International Conference on Emerging Technologies and Factory Automation). IEEE. https://doi.org/10.1109/ETFA.2017.8247579

Testing Automation Systems by Means of Model Checking

Igor Buzhinsky[1, 2], Valeriy Vyatkin[1, 3]
[1] Department of Electrical Engineering and Automation, Aalto University, Finland
[2] Computer Technologies Laboratory, ITMO University, St. Petersburg, Russia
[3] Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden
{igor.buzhinskii, valeriy.vyatkin}@aalto.fi

***Abstract*—Industrial automation systems are commonly obliged to comply with correctness requirements and safety standards. Testing and simulation are traditionally used to ensure this compliance. For mission-critical applications, formal verification and model checking in particular are also used, but such techniques are computationally intensive and difficult to apply in practice. This paper searches for synergies between testing and model checking by generalizing an earlier proposed formal test modeling framework. It presents a technique of testing automation systems with the use of model checking, which now supports multiple model checking environments and a more generic test case representation. The proposed technique is applied on a case study involving a simple safety-critical system with timing requirements. Experiments show that the technique is fast despite the use of formal methods and at the same time has several benefits compared to usual testing.**

I. Introduction

Verification and validation are common procedures for industrial automation systems to ensure their reliability and compliance with safety standards such as IEC 61508 [1]. These activities are commonly implemented by means of testing and simulation. In testing [2], the behavior of control software is checked on predefined scenarios, known as test cases. In simulation [3], the behavior of the simulation model of the closed-loop system (that is, the one composed of the controller and the plant under control) is examined. As for formal techniques, which are less popular due to their complexity, model checking [4], [5] can be applied to prove or disprove specifications by exploring the full state space of the system under verification. However, for large-scale systems, model checking is hard to apply due to its high computational complexity.

On the one hand, testing is a popular and ubiquitously used methodology unlike model checking. On the other hand, model checking offers a more comprehensive analysis of the considered automation system. These observations motivate the development of a hybrid approach, which would be as fast and easy to apply as testing, but would provide better means for system analysis.

This work continues the research direction established in [6], where a testing framework based of the net condition/event systems (NCES) [7] formalism has been proposed. Unlike [6], the approach presented in this paper is generic with respect to the employed formal language. In particular, we apply it together with wide-spread model checking environments SPIN[1] and NuSMV[2]. NCES are not considered due to their limited tool support, but the proposed solution can be applied to formal models of this type as well. Moreover, we use a more general test representation as finite-state machines (FSMs) which is not limited to linear sequences of inputs and required outputs.

The evaluation of the technique on a case study involving a simple safety-critical system in performed on verifiers employing two different model checking approaches: explicit-state (SPIN) and symbolic (NuSMV) model checking. While using explicit model checking for testing is fast, symbolic model checking is shown to be slower, but it scales well in the cases of long test cases and multiple tested controllers. The evaluation also deals with the question of checking test case correctness.

The rest of the paper is structured as follows. Section II reviews related research. Then, Section III explains the proposed approach. This approach is further applied on a case study in Section IV. Finally, Section V concludes the paper.

II. Related work

This paper is a direct continuation of the work [6], where a framework for formal modeling of the testing process for automation systems is proposed. In [6], testing is modeled using the formalism of NCES. While the basic idea of [6] is similar to the one of the present paper, the latter develops the former in two aspects: in terms of supported model languages (the ones with better tool support are used) and by considering a more general test case representation.

In the rest of the section, two related research topics are reviewed: known formal notations to represent test cases and test suites, and model checking together with related concepts.

A. Formal test case representations

The simplest possible representation of a test case is a finite sequence of elements. In [8], test cases are represented as finite sequences of symbols from a chosen alphabet. Such an approach is sufficient in [8], since the tested entities in [8] are FSMs without any separation of inputs and outputs. Test

[1] http://spinroot.com/
[2] http://nusmv.fbk.eu/

suites are represented as trees. A similar representation of test cases is employed in [6], except that inputs and output are now separated.

In [9], a formal model of testing is introduced. The implementation of the tested system, its specification and test cases, which are generated based on specification, are formally represented with labeled transition systems (LTS), a class of finite-state models. Essentially, test cases are defined as a sort of state machines, whose transitions are annotated with inputs and outputs and which have special states indicating successful and unsuccessful test completion. Test cases are further constrained to make cycles of ordinary states impossible, which guarantees that tests execution is finite. In contrast, in the present paper this restriction is omitted, which allows infinite test executions, which are treated as test failures unless the special accepting state is reached.

B. Temporal specifications and model checking

Temporal logics [4], [5] are formalisms which allow expressing specifications over system behaviors formally. The formulas of *linear temporal logic* (LTL) specify constraints which must be satisfied for every possible system execution. Being an extension of the Boolean propositional logic, LTL additionally includes *temporal operators*. Assume that system behavior is viewed as a sequence of states. Temporal operators include, but are not limited to the following ones: $\mathbf{X}$ refers to the next state, $\mathbf{G}$ requires that a certain formula is satisfied always, and $\mathbf{F}$ asserts that it is satisfied for some state in the future. As for *computation tree logic* (CTL), every its temporal operator is annotated with a quantifier $\mathbf{E}$ ("there exists a path") or $\mathbf{A}$ ("for all paths"). Consequently, the following CTL operators exist: $\mathbf{EX}$, $\mathbf{AX}$, $\mathbf{EG}$, $\mathbf{AG}$, $\mathbf{EF}$, $\mathbf{AF}$.

The problem of *model checking* refers to checking a temporal property for a chosen model. Formally, such a model is required to be a *Kripke structure*, but commonly used verifiers such as NuSMV, SPIN and UPPAAL hide these details and allow describing the system in a more natural way. In particular, such languages allow describing the system as a state machine or a system of connected state machines. Depending on the employed algorithms, model checking can be *explicit-state* (e.g. in SPIN and UPPAAL) and *symbolic* (NuSMV). Specifically for automation systems, model checking is subdivided into *open-loop* model checking, which checks only the controller model, and *closed-loop* model checking [10] which examines the entire system composed of the plant and the controller. A comparison of these techniques can be found in [11].

III. Proposed technique

In this paper, a technique of testing automation systems by means of model checking is proposed. Basically, the proposed technique is composed of the following steps:

1) **Controller modeling.** The source code of the controller to be tested is transformed to its formal model. This step can be potentially automated, for example using approaches such as [12]–[15].
2) **Plant modeling.** Optionally, a formal plant model is constructed. Many approaches of plant model construction, including automated ones, are known [16]–[19].
3) **Test case modeling.** Test cases for the system are formulated as FSMs. These state machines are assumed to be deterministic and representing particular behaviors of the plant. A test case is passed, if an *accepting* state is reached. A more simple test representation as sequences of inputs and outputs can also be represented this way, but the formalism of FSMs is more general since it can permit multiple behaviors of the controller. As mentioned in [6], such models can be employed to validate test cases.
4) **Formulating temporal specifications.** Optionally, temporal specification for the controller are formulated. Even if the plant model is absent, these specifications can be checked on test cases. Such a check is less general that the one with the plant model or the one with the controller model alone, but for explicit model checking it is fast: provided that tests are deterministic state machines, the state spaces to be checked are limited to linear sequences of states.
5) **Model checking.** For each test case, the closed-loop model composed of the controller and the test case is model checked: the accepting state of the test case is checked to be always reachable. In addition, temporal specifications formulated in stage 4 can be checked, and test cases can be validated against the plant model if the latter is present.

In the following subsections, we explain each step of the technique in more detail.

A. Controller modeling

At the first stage of the approach, the source code of the controller must be converted into a formal model. In this paper, we assume that the controller is a PLC program: that is, it comprises of a routine which is executed in cycles, the delay between which is constant. Such programs are commonly represented according to the IEC 61131-3 [20] international standard. Potentially, the proposed technique can be applied to distributed applications such as the ones complying with the IEC 61499 [21] standard: this would require considering non-cyclic execution.

Ideally, the source code of the controller (which may be also represented with visual languages such as function blocks and ladder logic) can be transformed to its formal model automatically using approaches such as [12]–[15]. Unfortunately, such approaches are not yet implemented as open tools, so in Section IV of this paper we model the controller manually.

B. Plant modeling

We describe the stage of plant modeling before considering test case models since plant modeling for model checking is an established technique, and our model of test cases will maintain many properties of the plant model.

The model of the plant represents physical and mechatronic aspects of the plant, regardless of its controller. Such a model is required in closed-loop approaches [10], and many methods of its construction, including automated ones, are known [16]–[19]. In such approaches, the controller model and the plant model are connected together. Since the plant model is often nondeterministic, the closed-loop system becomes nondeterministic as well, which is common in model checking. For the case of PLC applications this means that on each cycle the controller and the plant models exchange values of controller inputs and outputs.

Within the proposed testing technique, the presence of the plant model is optional. The reason for this is the lack of necessity to consider it during the usual process of testing, and since the technique proposed in this paper models this process, the plant model is not required. Nevertheless, this technique is able to benefit from this model in the following ways:

1) Temporal specifications (see Section III-D) can be checked not only on test cases, but also on the entire plant model. Since test cases are assumed to represent only particular plant behaviors, the plant model is more general, and thus results of model checking with this model are more reliable. This is the common way of performing closed-loop model checking. On the other hand, if the computational complexity of usual model checking is too high, model checking on test cases only may be easier at least for explicit-state model checkers such as SPIN.
2) The plant model can be used to check the validity of existing test cases (see Section III-E).
3) It is possible to consider automatic procedures of generating test cases which represent some particular aspects of the plant model. For example, this can be achieved by resolving each nondeterministic choice in this model. This use of the plant model is out of scope of this paper, but may be considered in future work.

C. Test case modeling

As mentioned in Section II-A, multiple test case representations are known. Since in this paper we limit our attention to PLC software, a test case can be considered as an entity which interacts with the controller under test in the discrete manner, by submitting inputs to the controller and receiving its outputs. Thus, the test case model is similar to the plant model in terms of its interface (inputs and outputs), but there are two important differences between these two types of formal models:

- Test case models must be deterministic, which corresponds to real life.
- Test case models judge whether the controller's behavior is correct or not.

The simplest type of models which submits to these requirements is the linear sequence of inputs and required outputs, like the one in [6]. However, such models require concrete behavior from the controller for each input scenario and hence are incapable of checking incomplete specifications. The solution which is applied in this paper is to use deterministic Moore FSMs.

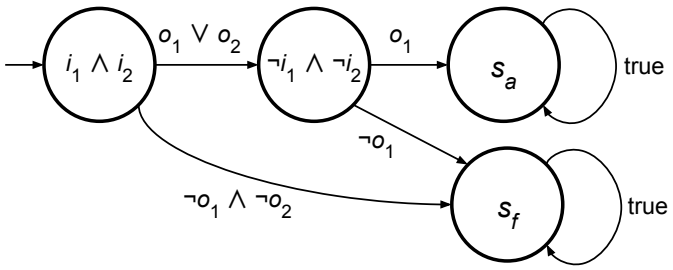

Fig. 1. An example of a test case modeled as an FSM. In this example, the controller has Boolean inputs i_1, i_2 and Boolean outputs o_1, o_2. In addition to the accepting state s_a, a failure state s_f is shown which indicates the failure of the test case. States are annotated with the values of the output function λ, and the transition function δ can be deduced from the edges marked with guard conditions. The values of λ in s_a and s_f are not shown since they have no effect on the outcome of the test case.

Formally, a *test case* is a tuple $(S, s_0, s_a, I, O, T, \lambda)$, where S is the finite set of *states*, $s_0 \in S$ is the *initial state*, $s_a \in S$ is the *accepting state*, I and O are the finite sets of *controller input value combinations* and *controller output value combinations* respectively, $\delta : S \times O \to S$ is the *transition relation*, and $\lambda : S \to I$ is the *output function*. Accepting states are commonly considered in deterministic finite automata (DFA) and not in FSMs, but their necessity here is explained by the need to check the controller's correctness: if and only if this state is eventually reached during the test-and-controller interaction, then the test case is *passed*. Otherwise, the test case is *failed*.

An example of a test case model visually represented as an FSM is shown in Fig. 1. Later such FSMs will be expressed using the formal languages of SPIN and NuSMV verifiers.

D. Formulating temporal specifications

Normally, temporal specifications are formulated for either the controller or the closed-loop system to be checked on its entire state space. Model checking, however, is typically a time-consuming procedure. Within the testing technique proposed in this paper, such temporal specifications are suggested to be checked only on execution scenarios specified by test cases. This approach does not reach the level of dependability achieved by ordinary model checking, but allows checking temporal properties for complicated controllers. Depending on the employed modeling language, temporal properties can be formulated in formal notations such as LTL and CTL. They can be interpreted as an additional *test oracle* which determines whether the controller behaves correctly on the predefined test scenarios.

E. Model checking

The process of model checking, which can be automatically performed in environments such as NuSMV, SPIN and UPPAAL, is used to achieve three goals.

First, to check whether the selected test case is passed, the inevitability of reaching the accepting state of the test case is checked. This can be expressed as either LTL property $\mathbf{F}\, s_a$ or CTL property $\mathbf{AF}\, s_a$. If the test case to be checked is selected nondeterministically, the same properties will check whether the accepting state is reached in all the test cases. If either of the test cases is failed, the corresponding system execution path will be generated by the verifier as a counterexample.

Second, temporal specifications formulated according to Section III-D are checked on test cases. Similarly to the previous case, a counterexample will be generated if either of the test cases is failed.

Third, if the plant model is available, then the validity of a test case can be checked in the following sense: the test case is valid if it enforces the system to produce an input-output behavior which is possible if the test case is replaced with the nondeterministic plant model. To implement such a check, the system of three components (the plant, the controller and the test case models) is considered. The test case model is connected with the controller model with inputs and outputs, as usual, and the plant model is configured to passively accept the outputs of the controller. The test case is valid if and only if the following CTL property is violated:

$$\neg\, \mathbf{EG}\left(\left(i_1^{\text{test}} = i_1^{\text{plant}}\right) \wedge \ldots \wedge \left(i_n^{\text{test}} = i_n^{\text{plant}}\right)\right),$$

where n is the number of controller inputs, and $i_1^{\text{test}}, \ldots, i_n^{\text{test}}$ and $i_1^{\text{plant}}, \ldots, i_n^{\text{plant}}$ are controller inputs produced by the test and the plant model, respectively. Unfortunately, this property has no representation in LTL, which means that the SPIN model checker is unable to check it. Nevertheless, another explicit-state model checker UPPAAL would be able to check properties like the specified one.

IV. Case Study

The suggested technique has been applied to a case study comprising a fictitious traffic lights system where the controller must comply with certain timing specifications. This system is described below, after which the details of applying the proposed technique are presented.

A. Description and specification

We consider the example of a traffic lights device responsible for regulating traffic in all directions at a crossroads. The overview of the considered system is shown in Fig. 2. The two orthogonal directions will be further referred to as directions D1 and D2, and the systems of lights for each of the directions will be referred to as T1 and T2. Technically, each of these light systems has six lamps, where each of colors red, yellow and green is visible in opposite directions.

The device is operated with a single PLC. The PLC has two binary traffic intensity sensors `T1_INTENSE` and `T2_INTENSE` for directions D1 and D2. The rationale behind these inputs is to provide the traffic lights a means to favor the direction of the most intense traffic. Then, the PLC can be optionally accessed by a human operator, who can switch the system to the manual mode (modeled with binary input `MANUAL`) and send custom color signals `T1_RED_MAN`, `T1_YELLOW_MAN`, `T1_GREEN_MAN`, `T2_RED_MAN`, `T2_YELLOW_MAN`, and `T2_GREEN_MAN`. Finally, the PLC has six binary outputs `T1_RED`, `T1_YELLOW`, `T1_GREEN`, `T2_RED`, `T2_YELLOW`, and `T2_GREEN`: one for each pair of opposite color lamps of T1 and T2 (this means that from now on we can imagine that there are only six different lights, three for each direction). The specification for the system is as follows:

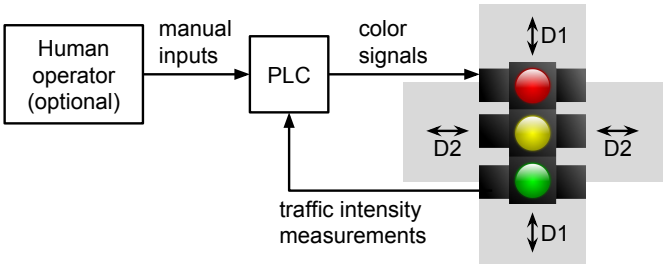

Fig. 2. Traffic lights system.

1) The system operates in rounds. The duration of each round is 50 seconds. In each round, traffic light colors change in the following order (we first provide the color of T1 and then the color of T2): (green, red), (yellow, red), (red, red), (red, green), (red, yellow), (red, red).
2) The duration of light combinations (yellow, red), (red, red) and (red, yellow) is one second.
3) If `T1_INTENSE` $\wedge\, \neg$`T2_INTENSE` holds at the beginning of the round, then in this round light combination (green, red) lasts at least 10 seconds longer than (red, green).
4) If `T2_INTENSE` $\wedge\, \neg$`T1_INTENSE` holds at the beginning of the round, then in this round light combination (green, red) lasts at least 10 seconds shorter than (red, green).
5) Otherwise, light combinations (green, red) and (red, green) last 23 seconds each.
6) In the manual mode, the rules above are ignored and the traffic lights show exactly the colors from the manual input. Once the manual mode is turned off, the system proceeds as if it was at the beginning of a round.

B. Controller

For simplicity we assume that the PLC cycle time equals one second. The controller code was implemented according to the IEC 61131-3 standard in the Structured Text language in CODESYS[3]. It involves six TON (on delay) timers, which are reset at the beginning of each period and are given proper delays. After that, these timers consequently fire and change the lights accordingly. If the program encounters the `MANUAL` input signal, the timers are reset and the controller displays the colors provided as manual inputs. A part of the described implementation is shown in Fig. 3.

After the PLC implementation was prepared, the controller has been manually modeled in the languages of NuSMV and SPIN verifiers. In the NuSMV model, each new time step

[3]https://www.codesys.com/

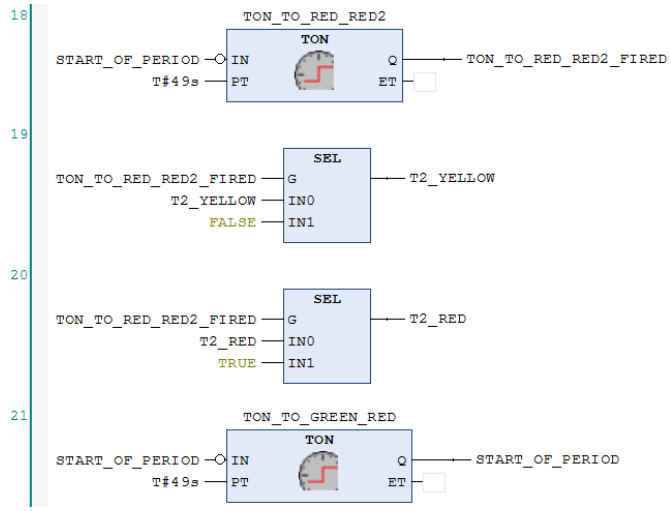

Fig. 3. A part of the PLC implementation of the traffic lights controller.

refers to the next PLC cycle, and variable assignments within a cycle are implemented using the technique of static single assignment. In the SPIN model, this is not the case. As a consequence, temporal specifications were altered to make them be effectively checked only after the controller finishes each of its cycles.

C. Plant model

The plant model corresponds to the environment where the PLC operates, which in our case means two entities from Fig. 2: the traffic lights device and the human operator. We consider the simplest possible plant model: traffic intensities (coming from the traffic lights device) and manual control signals (coming from the operator) can change arbitrarily each time the controller reads its inputs. Virtually, this means that model checking with such a model would be open-loop since the plant model does not constrain controller inputs. Using such a model means that every test case which complies with the controller's interface is valid with respect to this plant model.

D. Temporal specifications

Formulated temporal specifications include the requirements for the automatic (i.e. non-manual) mode that: certain color combinations are prohibited; each traffic light always shows exactly one color; colors change in permitted order. For the manual mode, the traffic lights are obliged to show the manually specified colors. These requirements were formulated in LTL for SPIN and in CTL for NuSMV. Exact timing information specified in Section IV-A is not included into the temporal specification since it is difficult to formulate in LTL and CTL. Timing requirements will be checked by test cases described in the next subsection. Several examples of specifications formulated in CTL are provided below:

1) $\mathbf{AG}(\neg\texttt{MANUAL} \rightarrow \neg(\texttt{T1\_GREEN} \wedge \texttt{T2\_GREEN}))$: "in the automatic mode, both lights shall not show green";

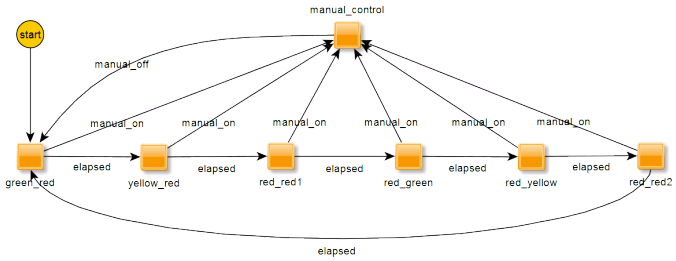

Fig. 4. Abstract specification for the traffic lights controller.

2) $\mathbf{AG}(\neg\texttt{MANUAL} \rightarrow \texttt{T1\_GREEN} \vee \texttt{T1\_YELLOW} \vee \texttt{T1\_RED})$: "in the automatic mode, the first light shall show at least one color";
3) $\mathbf{AG}(\neg\texttt{MANUAL} \rightarrow \neg(\texttt{T1\_GREEN} \wedge \texttt{T1\_YELLOW}))$: "in the automatic mode, the first light shall not show green and yellow simultaneously";
4) $\mathbf{AG}(\texttt{T1\_GREEN} \wedge \texttt{T2\_RED} \rightarrow \mathbf{AU}(\texttt{T1\_GREEN} \wedge \texttt{T2\_RED}, \texttt{MANUAL} \vee \texttt{T1\_YELLOW} \wedge \texttt{T2\_RED}))$: "in the automatic mode, color combination (green, red) shall only change to (yellow, red), and this change shall eventually happen";
5) $\mathbf{AG}(\texttt{MANUAL} \rightarrow (\texttt{T1\_YELLOW} = \texttt{T1\_YELLOW\_MAN}))$: "in the manual mode, the first light shall show yellow if and only if the corresponding manual signal is on".

E. Test cases

Test cases for the traffic lights example were prepared based on the methodology of model-based testing (MBT) [22], wherein test cases are derived from models of system specification, which are prepared independently from the system implementation. Fig. 4 shows the abstract specification of the system represented as a state machine: the traffic lights change their colors in the predefined order, which can be interrupted by manual mode activation. Using the graphwalker[4] tool, a test suite of five abstract test cases was generated to ensure the edge coverage of the state machine from Fig. 4. Since abstract test cases are only paths in the state machine, they were further concretized to incorporate traffic intensity information and concrete light colors to show in the manual mode. During this concretization, each test case was subdivided into four ones depending on the employed traffic intensities in both directions.

Test cases obtained using MBT, however, were short and did not exercise the system for more than two rounds. Due to this reason, for each combination of traffic intensities, an additional test case was specified manually that required the system to be executed for 250 rounds. Later these tests will be referred to as "long" ones. Considering the length of each round, the length of each "long" test is 12500 PLC cycles, which is roughly equal to 208 minutes. In the next subsection, we will see how model checking speeds up the testing process significantly.

[4]http://graphwalker.github.io/

F. Test case execution

The principal part of the evaluation of the proposed approach comprised test case execution by performing model checking. In SPIN, all tests were successfully passed. Among them, the execution time of tests obtained using MBT was around one second, and the execution time of "long" tests was around 2.6 seconds. These results indicate that, despite performing model checking, it is possible to execute tests much faster than in conventional testing.

In NuSMV, the execution of the same test cases took significantly more time. The median execution time of all 24 considered test cases was 18.4 minutes, and the executions of eight tests did not finish within the time limit of one hour. Apparently, the reason for such slow performance is the nature of symbolic model checking, which is performed by NuSMV. While the determinism of the tested formal model makes explicit-state model checking process only one execution path, symbolic model checking has no means to benefit from model's determinism, and hence its execution time still depends on the complexity of the formal model. On the other hand, no significant difference of execution time for long and short tests was observed.

G. Additional checks

Model checking of the temporal properties specified in Section IV-D have been performed on the test cases. In SPIN, each property has been checked for each test case independently. All the checks succeeded, and model checking execution times varied between 1.5 and 2.0 seconds for MBT-generated test cases and between 2.4 and 4.0 seconds for "long" test cases. In NuSMV, for each test case, all temporal properties were checked in one run of the verifier since it is able to reuse information obtained during property checking. However, only 7 out of 24 NuSMV executions terminated within one hour. When ordinary closed-loop model checking was then performed as well, its execution time was only 45 seconds. Since at the same time model checking with the plant model is more dependable, we conclude that model checking on test cases is not reasonable for symbolic model checkers such as NuSMV.

Then, test validation has been run as described in Section III-E. Due to the lack of CTL support in SPIN, it was performed only in NuSMV. This time, the median model checking time was impossible to estimate since more than 12 NuSMV executions violated the time limit of one hour. We conclude that performing such a check in a symbolic model checking is not worth its purpose; either an explicit-state model checking like UPPAAL should be used, or other means of test validation should be searched for.

Finally, we consider the scenario of testing simultaneously several controllers with small differences. This idea has been inspired by the work [23]. The application of this scenario in real life is testing product lines, where different products are composed of common modules. In our case, we applied a simpler approach: 25 different controllers were obtained by altering two delays of the timers, each of which was configured to have five distinct values. In SPIN, such a check for MBT-generated tests lasted around 1.4–1.8 seconds, and "long" test execution time was around 53–55 seconds. Thus, the time required for model checking grows with the number of checked controllers, which is an expected result for explicit-state model checking. In NuSMV, the median test execution time was 33.6 minutes, and the executions of nine tests did not finish within one hour. Compared to the execution of single controller tests and accounting for the number of tested controllers, these results indicate that symbolic model checking is more suitable for testing controller families than single controllers.

V. Discussion and conclusions

In this paper, which continues the research direction of the work [6], a technique of testing automation systems using model checking has been proposed. Performing testing on the level of formal models has potential benefits of analyzing the entire system execution if a test case is failed, validating test cases, testing multiple controllers simultaneously, checking temporal specifications on specific system execution paths, and testing timed systems (like the one considered in the case study) without the need of simulating the actual delays in the system.

Among the shortcomings of the technique are the substitution of the original tested controller with its formal model, which might be imprecise, and the lack of open access tools which are able to generate such models automatically. Controller modeling is a general problem of formal methods applied to industrial automation. If the controller is modeled manually, a modeling error can be occasionally made. Then, even if this model is generated automatically (with approaches such as [12]–[15]), the issue of discretizing real values may arise.

The proposed technique has been evaluated on a case study using two types of model checkers: an explicit-state one (SPIN) and a symbolic one (NuSMV). It has been shown that, despite the use of model checking, which is commonly time consuming, testing using an explicit-state model checker is fast. For a symbolic model checking, test execution times are significantly larger, which makes the application of the technique in this case doubtful. On the other hand, if test cases are long but can be represented in a compact way, and if multiple similar controllers are tested simultaneously, then symbolic model checking is more scalable.

To make the proposed technique useful in industrial practice, the level of automation of its steps must be increased. This concerns automating the generation of formal models of controllers, plants and test suites. Although approaches for this do exist, they are not yet available as user-friendly tools with graphical interface. The very process of testing as model checking needs profound tool support as well – otherwise, the simplicity of testing, which is potentially reachable for the proposed approach, will not be achieved.

Apart from working on the aforementioned practical aspect, future work may involve adding the support of the UPPAAL

model checker, modeling timing aspects of systems with timed automata (with the help of UPPAAL), generating test cases based on the plant model, and applying the approach to distributed event-based automation systems, including the ones represented in the IEC 61499 standard.

ACKNOWLEDGMENTS

This work was financially supported by the Ministry of Education and Science of the Russian Federation, project RFMEFI58716X0032 and by the Engineering Rulez project funded by Tekes – the Finnish Funding Agency for Innovation (Decision 3095/31/2016).

REFERENCES

[1] *Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems (IEC 61508)*. Geneva: International Electrotechnical Commission, 2005.

[2] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[3] J. Banks, *Handbook of simulation: principles, methodology, advances, applications, and practice*. John Wiley & Sons, 1998.

[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[5] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[6] I. Buzhinsky, C. Pang, and V. Vyatkin, "Formal modeling of testing software for cyber-physical automation systems," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3. IEEE, 2015, pp. 301–306.

[7] M. Rausch and H.-M. Hanisch, "Net condition/event systems with multiple condition outputs," in *INRIA/IEEE Symposium on Emerging Technologies and Factory Automation (EFTA), 1995*, vol. 1. IEEE, 1995, pp. 592–600.

[8] A. Simão, A. Petrenko, and N. Yevtushenko, "Generating reduced tests for FSMs with extra states," in *Testing of Software and Communication Systems*. Springer, 2009, pp. 129–145.

[9] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing*. Springer, 2008, pp. 1–38.

[10] S. Preuße, H. Lapp, and H. Hanisch, "Closed-loop system modeling, validation, and verification," in *17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2012, pp. 1–8.

[11] J. Machado, B. Denis, and J.-J. Lesage, "Formal verification of industrial controllers: with or without a plant model?" in *7th Portuguese Conference on Automatic Control (CONTROLO)*, 2006, pp. 341–346.

[12] D. Darvas, B. Fernández Adiego, and E. Blanco Viñuela, "PLCverif: a tool to verify PLC programs based on model checking techniques," in *15th International Conference on Accelerator & Large Experimental Physics Control Systems*, 2015, pp. 911–914.

[13] D. Soliman, K. Thramboulidis, and G. Frey, "Function block diagram to UPPAAL timed automata transformation based on formal models," *IFAC Proceedings Volumes*, vol. 45, no. 6, pp. 1653–1659, 2012.

[14] S. Patil, V. Dubinin, and V. Vyatkin, "Formal modelling and verification of IEC61499 function blocks with abstract state machines and SMV-execution semantics," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2015, pp. 300–315.

[15] O. Pavlovic and H.-D. Ehrich, "Model checking PLC software written in function block diagram," in *3rd International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 439–448.

[16] M. Roth, L. Litz, and J.-J. Lesage, "Identification of discrete event systems: Implementation issues and model completeness," in *7th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, vol. 3, 2010, pp. 73–80.

[17] J. Machado, B. Denis, and J.-J. Lesage, "A generic approach to build plant models for DES verification purposes," in *8th International Workshop on Discrete Event Systems (WODES)*. IEEE, 2006, pp. 407–412.

[18] S. Preuße, *Technologies for Engineering Manufacturing Systems Control in Closed Loop*. Logos Verlag Berlin GmbH, 2013, vol. 10.

[19] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1521–1530, 2017.

[20] *International Standard IEC 61131-3: Programmable controllers – Part 3: Programming languages, Second edition*. Geneva: International Electrotechnical Commission, 2003.

[21] *International Standard IEC 61499-1: Function Blocks – Part 1: Architecture, Second edition*. Geneva: International Electrotechnical Commission, 2012.

[22] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-based testing of reactive systems: advanced lectures. Lecture Notes in Computer Science*. Springer, 2005, vol. 3472.

[23] S. Soleimanifard and D. Gurov, "Algorithmic verification of procedural programs in the presence of code variability," *Science of Computer Programming*, vol. 127, pp. 76–102, 2016.