Formal Modeling of Testing Software for Cyber-Physical Automation Systems

Igor Buzhinsky*, Cheng Pang†, Valeriy Vyatkin†‡*
* Computer Technologies Laboratory, ITMO University, St. Petersburg, Russia
† Department of Electrical Engineering and Automation, Aalto University, Finland
‡ Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden
igor.buzhinsky@gmail.com, {cheng.pang.phd, vyatkin}@ieee.org

***Abstract*—The paper presents a framework which uses formal models for testing control software for industrial automation systems. The formalism called Net Condition/Event Systems (NCES) is applied to model the program under test, along with the system under control (plant) and the testing environment. The benefits of using the framework include the opportunities to test systems with time delays without the need to wait, to test parameterized sets of systems with a single execution of a test suite, and to check test suites for correctness. The use of the framework is illustrated on a simple system consisting of a lab-scale plant and a control application for it.**

I. Introduction

Control software for industrial automation systems undergoes thorough testing to ensure its correctness and safety. Various automated testing methods are applied in order to make this process more time and cost efficient. Still, testing has many limitations in its ability to guarantee the fulfillment of requirements in implemented systems.

Another verification technique that has been extensively studied in the literature is called formal verification. It promises much more comprehensive discovery of possible errors in software, but has problems with applicability and complexity. Formal verification implies that the model of a software system is created using a formal language, which requires specific knowledge from control engineers and suffers from the state explosions problem if applied to realistically sized software systems. If Net Condition/Event Systems (NCES) [1] are used as formal models, the problem of state explosion arises only on the analysis stage, though, and not during the composition stage. Testing is much faster and much more popular in industry, although it cannot truly guarantee the correctness of the tested system.

In this paper we attempt to combine these two techniques. We adopt an earlier developed formal modeling framework [2], which uses the modular place-transition formalism of NCES to represent hierarchically organized automation systems and to verify them with the model checking approach [3]. Using this framework, we model the software testing process, partially applying model-based testing (MBT) [4], a testing methodology in which formal models of specification (such as state diagrams) facilitate test generation. We suggest using NCES not only to represent the software and the plant, but also to model test suites and specifications. The proposed approach has several benefits, among which the possibility to test systems with delays without waiting, to test parameterized sets of systems in a single step, and to check test suites for correctness.

As demonstrated in [5] and [6], NCES can be applied to work with software conformant with modern industrial standards such as IEC 61131-3 [7] and IEC 61499-1 [8]. The latter one supports distributed systems, which is one of possible domains of applying the introduced framework. The mentioned work [6] also presents a software package capable of translating IEC 61499-1 applications into NCES automatically.

The remainder of this paper is organized as follows. In Section II we review the application of formal models in testing and explain the formalism of NCES. The latter allows us to present the NCES-based testing framework for industrial automation systems, which is done in Section III. The framework is illustrated with examples of its application for a simple system in Section IV. The paper is concluded in Section V.

II. Formal Models and Testing

In this section we review several approaches focused on testing, which employ various formal models. After that, we examine the formalism of NCES and address their previous applications.

A. The Use of Formal Models in Testing

The idea of using formal models in software testing is far not novel. Multiple formal techniques of representing implementations, specifications and test suites are known. A popular approach is to use finite-state models, such as deterministic finite-state machines (FSMs). For example, in [9] both software specification and implementation are assumed to be FSMs with equal input alphabets. The work [9] proposes a method of generating test suites (as sets of traces of the specification) that are able to distinguish the correct implementation from incorrect ones.

A more complex theory of labeled transition systems and input-output transition systems is developed in [10]. The paper also employs finite-state models and presents the **ioco** conformance relation between implementations and specifications, such that test suites generated in a specific way are able to check whether a particular implementation conforms to the

specification. This approach is modified in [11], where similar models (symbolic transition systems), which now explicitly involve variables, are obtained from specifications represented as sequential function charts (SFCs).

Timed automata and model checking are utilized in [12] to generate test suites for IEC 61131-3 function block diagrams. The paper shows how to perform this task if certain structural coverage criteria are required to be fulfilled for the test suite. The problem is solved by reducing it to a reachability property expressed as a CTL temporal formula.

In this paper we attempt to develop new methods of using formal models in testing that are based on closed-loop representations of cyber-physical systems [13] with explicit modeling of the physical system, also referred to as the plant. Therefore, we require a formal language that is modular, so that the entire testing framework could be conveniently represented. We select Net Condition Event Systems (NCES) for this role.

B. Net Condition/Event Systems

NCES is a formal language that extends Petri nets [14] in a number of ways. An NCES has a set of *places* (denoted with circles) and *transitions* (denoted with boxes) between them. Each place can be either *marked* (in this case it has a so-called *token* in it indicated by a dot) or not marked, and the initial marking is a part of the NCES. Each transition has a set of input and output places and is connected with them by *flow arcs* (denoted with arrows). If all input places are marked, then the transition can fire: that is, unmark the input places and mark the output ones. In addition, transitions might have *condition arcs* (indicated by arrows with bold dots at their ends) coming from other places. Such arcs place additional restrictions on transition firing: all places in which the condition arcs originate must be marked as well. Finally, transitions can be either *spontaneous* or *forced*. Spontaneous transitions simply fire when all their firing conditions are met. In contrast, forced transitions have incoming *event arcs* (denoted with arrows with broken centers) from other transitions, and their activations are caused by the activations of event arc sources. A formal and detailed definition of NCES is given in [1].

For convenience, NCES can be viewed as modularized entities, such that each module has *event and condition inputs and outputs*. An example of an NCES module is shown in Fig. 1. It has two places `p_before` and `p_after`, and a transition `t1` between them. When the condition arc `enabled` is active and when the `switch` event comes, `t1` transports the token from `p_before` to `p_after` and emits the `switched` event. At the same time output condition arc `after` becomes enabled while arc `before` stops being enabled.

Fig. 2 shows an example of a composite module. Inside this module there are two instances of the module from Fig. 1 and an instance of the module `true`, which provides an always enabled condition output. When event `init` arrives, it first triggers the execution of `Module1` and then the execution of `Module2`. In the end, the output arc `done` becomes enabled.

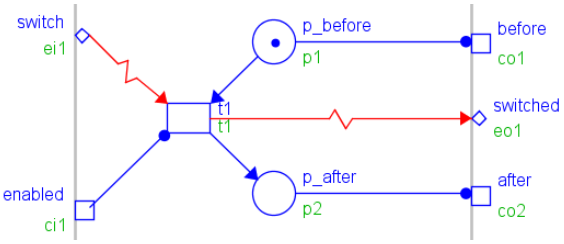

Fig. 1. An example of a basic NCES module, which models a one-way switch

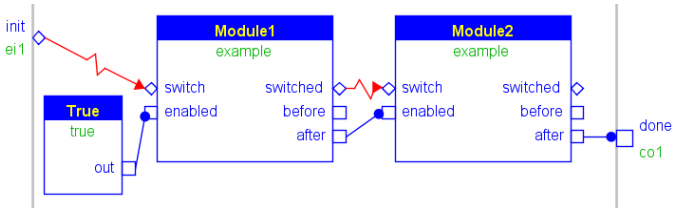

Fig. 2. An example of a composite NCES module

Modularization allows one to represent systems which can be decomposed into parts. For example, they can be used to represent IEC 61499-1 function blocks [15] (they can be implemented as separate NCES modules) and modular plants, thus being capable of representing cyber-physical systems [13]. The top-level module usually does not have any inputs and outputs and represents the interaction of the plant and the controller. The importance of closed-loop modeling is illustrated in [16].

C. Previous Applications of NCES

NCES were originally introduced in [17] in order to model discrete event dynamic systems. In [1] they were suggested to be used for modeling the behavior of software for programmable logic controllers (PLCs). In particular, the work [1] described how to transform programs written in IL ("instruction language") into NCES.

Following that, NCES were proposed to be used for applications conformant with international standards. In [5] it was shown how to represent IEC 61131-3 function blocks as NCES modules. Function blocks, entities which combine behavior and state, appeared to be quite similar to their NCES substitutes. Finally, in [6] distributed systems represented as IEC 61499-1 function blocks networks were also shown to be modeled as NCES.

After that, the work [2] presented a framework which suggested several solutions concerning the development process involving NCES. It suggested creating and editing these models in a special NCES editor, which is shortly described in [2]. This editor, which is adopted in this paper, employed an XML format for representing NCES, which is similar to the one of IEC 61499-1 function blocks. The framework also proposed to use NCES to represent closed-loop systems composed of the model of the plant and the model of the control application. It is then possible to analyze the *reachability graph* (or the *reachability space*) of the system. To construct

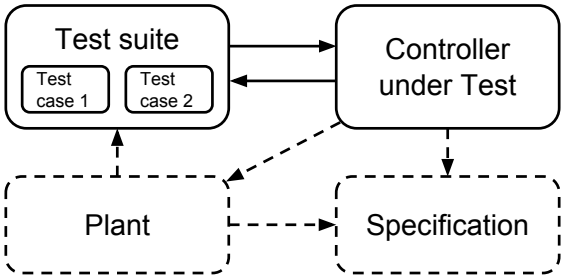

Fig. 3. The scheme of NCES models suggested in the framework

the graph, the modular model is first "flattened" by removing all module boundaries, and then the graph is formed out of all combinations of place markings possible in the system (nodes of the graph) and transitions between them (arcs of the graph). Such a graph can be visualized and verified with the model checking approach. The described framework, however, did not address testing. This paper deals with this issue and builds a new framework on top of the one from [2].

A more recent application of NCES is a comprehensive framework for closed-loop modeling and verification of IEC 61131-3 software [18]. NCES are also chosen for reactive system modelling in the book [19].

III. Proposed Framework

The proposed framework suggests using formal models of the controller, the test suite, the plant and the specification of the system, and representing the mentioned models as NCES. While the use of the models of the controller and the test suite are inevitable, the models of the plant and the specification are optional and mainly facilitate test suite validation. The suggested arrangement of models is shown in Fig. 3, where each NCES model is represented with a box, and data flows between them are shown with arrows. Optional parts are outlined in dashed lines. The purpose and contents of various parts of the scheme are as follows:

- The model of the control application (*controller under test*) is the one which is tested. It can be obtained automatically from the application's implementation [6], [18], [19]. The model receives sensor inputs and outputs actuator signals.
- The model of the *test suite* is connected to the controller. It can send sensor inputs to the controller, receive its outputs and compare them with expected ones. During initialization, the test suite nondeterministically selects a test case to be executed. This is done to make the reachability graph of the composed system contain the execution traces of all test cases in the test suite. In turn, a test case is composed of a sequence of *test elements*, where each element sets proper input values for the controller and waits for expected outputs. A test element is passed, if it receives the expected outputs from the controller. If it never receives such outputs, then subsequent elements are blocked and thus are not visited in the reachability graph. The test case is passed, if and only if all its elements are passed.
- The model of the *plant* can be connected to the controller to receive actuator signals from it. This allows to monitor the state of the plant during test execution. The diagram in Fig. 3 also shows that the test suite can receive data from the plant. This connection is relevant if one wishes to ensure that the test suite only produces inputs which can be generated by the plant. In this case, the test element not only needs to receive correct outputs from the controller to proceed, but also must ensure that the plant has been able to produce the inputs sent to the controller. The model of the plant is therefore a means of validating the test suite in the described sense.
- The model of the *specification* is the second tool to validate test suites: it monitors faults in the plant and the controller and does not alter the behaviour of the rest of the system.

Below we explain the use cases of the proposed framework.

1) Discrete time modeling: One of the properties of NCES is their ability to model time in a discrete fashion: the corresponding dialect is called Timed NCES, or TNCES. When the tested system includes delays, usual testing time will also include them. During usual software testing it is possible to skip such delays, but this solution is hard to implement when delays occur due to timers which are set and checked in different parts of the controller. In contrast, if time is modeled in NCES, there is no need to wait.

2) Testing families of related controllers: If there is a series of structurally similar controllers to be tested, it is possible to test this series in a single reachability graph construction. For example, if the controller has a Boolean parameter, it can be selected nondeterministically prior to test case execution. A more complex example is constructing the controller from a set of modules: in this case, for each module type a concrete module of this type will be selected prior to test case execution. In particular, we may consider the problem of verifying all products in a product line [20], where each product is constructed from a set of commonalities and variabilities.

3) Test suite validation: This use case has already been partially described while explaining the rationale behind considering the models of the plant and the specification. Test suite validation is important when tests are not generated automatically, like in MBT. Even when MBT is applied, it might be reasonable to additionally ensure that the translation of abstract test cases to executable ones is correct. To validate a test suite, one needs to have the model of the plant and, optionally, the model of the specification. The requirement that the plant can output sensor signals equal to the ones encoded in test cases may be checked with the help of the plant's model, as previously described. Another possible requirement, which now requires the model of the specification to be present, is to check whether neither the plant nor the controller enters an invalid state. Such checks are possible if the fault can be deduced from their outputs. Each requirement in the specification is represented with two places: the initial one,

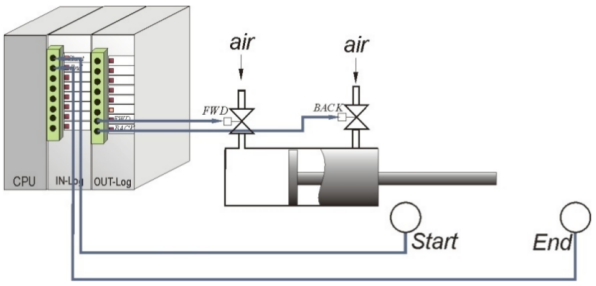

Fig. 4. The scheme of the cylinder

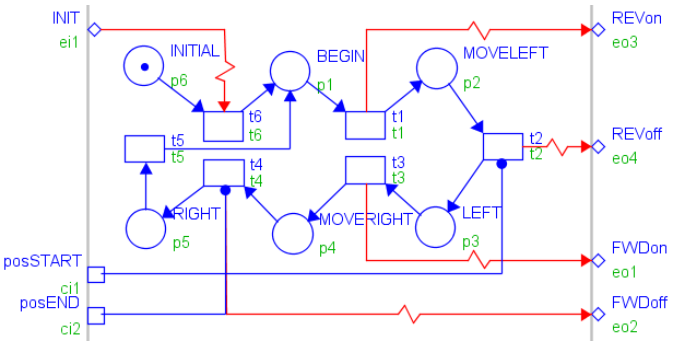

Fig. 5. The controller's logic

signifying no violation of the requirement, and the faulty one. If the state space of the entire system contains a state where at least one of such faulty places is marked, then there is an error either in the test case, in the controller or in the plant's model.

IV. Examples

In this section, the proposed framework is applied to a simple example: the model of a linear drive implemented using a pneumatic cylinder, and a control application for it. The cylinder, shown in Fig. 4, has two sensors (`Start` and `End`) and two control signals (`FWD` and `BACK`). The NCES model of the cylinder consists of its moving status (back, forward and stopped), its position (start, end and intermediate) and two Boolean variables for its sensors. Initially, the cylinder is in its end position and is stopped. The model of the controller, in turn, is composed of the controller's logic and two Boolean variables for its control signals. The controller's logic, shown in Fig. 5, is a loop which emits output signals to change the controller's outputs `FWD` and `BACK` and waits for input signals `Start` and `End` in proper moments. The plant and the controller are interconnected, forming a closed-loop system.

A. Test Case

Fig. 6 shows an NCES model of a test case (a trivial case of a test suite) consisting of three elements, which is connected to the controller. The left column of the network shows the initializer of the system (needed to generate the initial event), test elements and the final signal receiver, which reports (by marking a certain place) that the test was passed, when it receives an event. The second column of the network is

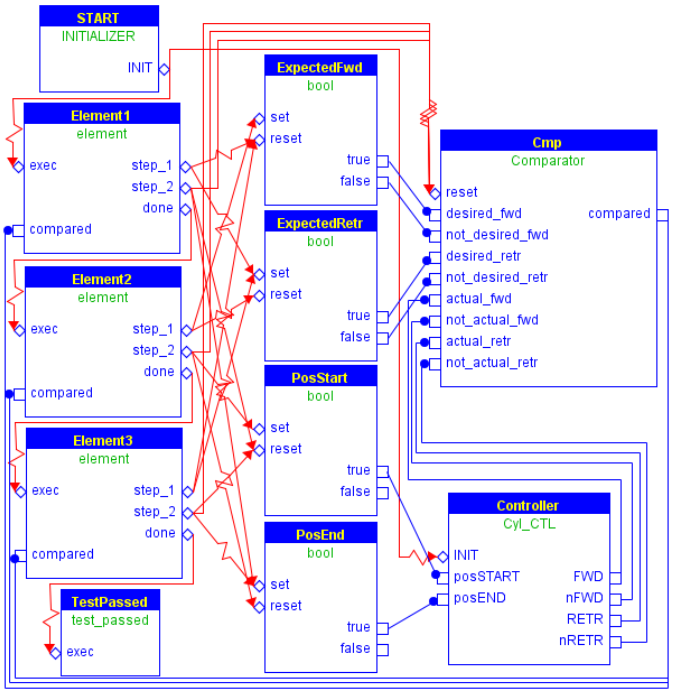

Fig. 6. The NCES model of a test case consisting of three elements

TABLE I
Inputs and outputs of the test case shown in Fig. 6

Index	PosStart	PosEnd	ExpectedFwd	ExpectedRetr
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1

devoted to variables: the ones representing the values expected by the test (`ExpectedFwd` and `ExpectedRetr`) and the ones encoding the input parts of test elements (`PosStart` and `PosEnd`). Input and output values of this test case are also shown in Table I for clarity. Lastly, the third column contains the tested model of the controller and the comparator used to compare the outputs of the plant to the expected outputs.

After the initialization, the test case executes element by element in the following way. At the first step, the element prepares expected variable values. At the second step, it sets sensor inputs of the controller, and the controller reacts to these changes and possibly updates its outputs. Both the expected outputs and the controller's outputs are connected to the comparator, which allows the test sequence to proceed once the expected inputs are equal to the produced ones. If the controller does not produce correct outputs, then the system reaches a deadlock state and the test case is failed. Otherwise, if the controller always responds with the expected values, the `TestPassed` module is activated, and the final state of the system (also a deadlock one) signifies that the test was passed.

B. Testing a Controller with Delays

This example illustrates the first use case of the framework. Imagine that the controller's logic has been enhanced with

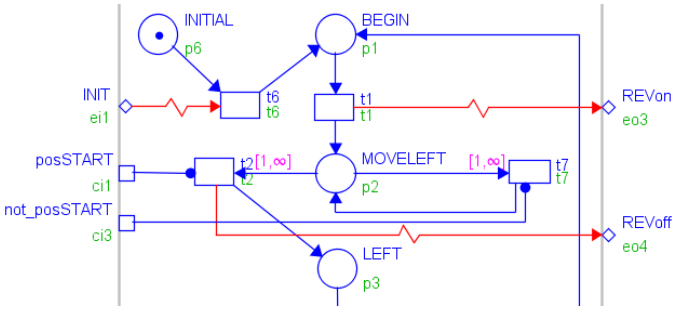

Fig. 7. The controller's logic with delays in the place `MOVELEFT` (places that follow `LEFT` are omitted)

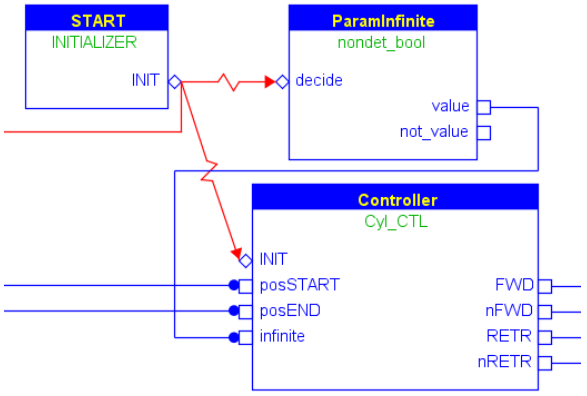

Fig. 8. A system with a Boolean parameter, the value of which is chosen in the beginning of the system's operation (only a part of the system is shown)

debouncing: when the controller waits for a `Start` signal, it ensures that this signal was present for the entire last second. Such a situation can be modeled in NCES with timed arcs. Fig. 7 shows a part of the controller's logic from Fig. 5, in which delays were inserted. Timed arcs are inscribed in the form $[t, \infty]$, where t is the number of time units to wait. More formally, the input place of a timed arc has a clock which increases its value when this place has a token and when there are no other enabled transitions without incoming timed arcs. The transition from place `MOVELEFT` to place `LEFT` is activated only if the input signal `posSTART` was enabled for a time unit. Otherwise, another transition leading to the same place `MOVELEFT` resets the clock in `MOVELEFT`. If the test case from the previous subsection is applied to such a modified controller, then the results of its execution do not change: it is passed.

C. Testing a Parameterized Controller

This example is the demonstration of the second use case of the framework. Imagine that the controller has a Boolean parameter `infinite`, which denotes whether the controller has to perform only one cycle of the cylinder movement (the signal is off), or it should move it backward and forward eternally (the signal is on), as assumed before. The test case from Fig. 6 should pass for the controller with the enabled signal and fail for the controller with the disabled signal.

To apply this test case for the described set of two controllers simultaneously, we need to introduce the parameter into our NCES model. Since both values of `infinite` must be checked and these values do not change during the cylinder's operation, the parameter's value can be chosen nondeterministically during the system's initialization. This choice is modeled with a simple module `nondet_bool` shown in Fig. 8 together with the controller and the initializer.

After the system is modified, the reachability graph of the system will split into two paths, corresponding to different values of the parameter. The final state of the path where `infinite` is off reports that the test case is failed, and the other path reports success. In a more general situation, when there are much more parameter values or combinations of parameter values to check, there is no need to examine the reachability graph manually. If one needs to check whether the test case is passed for all versions of the controller, then it is sufficient to model check the graph using the CTL formula `EF(TestPassed.passed)`, where `TestPassed.passed` signifies the presence of a token in the place inside the `TestPassed` module (see Fig. 6) which is activated when all test elements are properly executed. This formula just checks that for every path of the model's execution (including every choice of parameters) the test case is eventually passed.

The classical alternative to the framework is separate testing of every controller in the family. The simple approach to fix parameter values prior to test case execution does not provide any benefits in comparison with this alternative, but we believe that it is possible to reduce the time of testing a family of controllers by applying some ideas of the work [21], which presents techniques to verify programs with variabilities.

D. Test Case Validation

This example will illustrate the third way of using the framework. We first modify the controller in the following way: it will now send both actuator signals independently, each one according to the value of a sensor signal. That is, the output signal `FWD` is on if and only if the input signal `Start` is received, and the output signal `BACK` is on if and only if the input signal `End` is on. This controller is not equivalent to the initial one: it will produce both outputs if both its inputs are on and none of the outputs if all its inputs are off.

Now imagine a test case which at some point sends both `Start` and `End` and expects both `FWD` and `BACK`. This test case is obviously incorrect, though the corresponding test element will pass for the described controller. At this stage we introduce an observing specification into the system. It monitors the state of the plant and triggers a requirement violation due to the simultaneous values of `FWD` and `BACK` received by the plant. The scheme of the described NCES model is visualized in Fig. 9. The controller's outputs are not only connected to the comparator (these connections go upwards and end outside the figure), but also to the plant, and the plant's output `input_error` is received by the `Observer` module, which represents the specification.

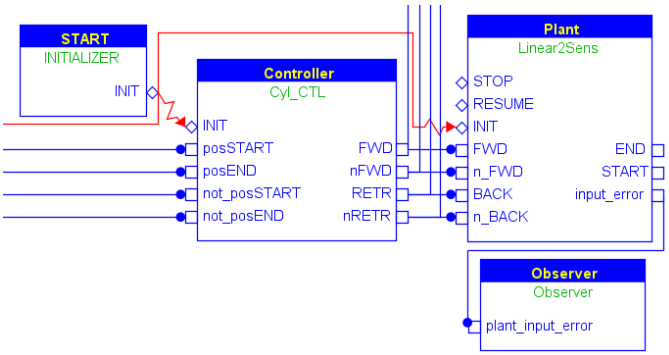

Fig. 9. A system with a plant and a specification, which monitors that the plant is in a valid state (only a part of the system is shown)

The presented type of specification can identify errors not only in the test suite, but also in the controller and in the plant's model. There is another solution, which focuses only on the test suite. The scheme from Fig. 9 can be extended with a second comparator (similar to the one shown in Fig. 6), which will compare input values of test elements with the ones emitted by the plant, and the test case will fail if either of the comparators does not emit `compared` at some stage (i.e. there is a failed comparison). If there is no observing specification, such an extension is sufficient to identify the error in the considered test case. On the other hand, if there is an observing specification which reports an error and the test case is passed in the presence of the second comparator, this means that the test case is correct (it produces inputs possible for the plant and requires outputs matching the controller's behaviour), but the error is rather caused by either the controller or the plant.

V. Conclusion

In this paper a framework has been proposed which uses NCES to support testing of industrial automation software. The framework has appeared to be useful for several tasks. Among them there are ones unachievable by usual testing: immediate testing of systems with delays, testing multiple related systems in a single step, checking the correctness of test suites. It also allows a deeper analysis of existing tests without the need to prepare temporal specifications for formal verification.

The mentioned potential benefits of the framework should be shown to be practically applicable, though. This is the main direction of the future work. First, we can apply the framework to more complex examples, including purely distributed ones. Next, we wish to show how to test families of controllers and to find means of doing this faster than testing all the controllers separately. It is also possible to investigate more on representing specifications as NCES. Finally, we believe that the framework can be applied for the problem of coverage test generation by formulating this problem in terms of the reachability graph.

Acknowledgements

This work was financially supported, in part, by the S-Step project (granted by FIMECC, Finland, Aalto University p/n 2115492) and by the Government of Russian Federation, Grant 074-U01.

References

[1] H.-M. Hanisch, J. Thieme, A. Luder, and O. Wienhold, "Modeling of PLC behavior by means of timed net condition/event systems," in *6th International Conference on Emerging Technologies and Factory Automation Proceedings (ETFA), 1997*. IEEE, 1997, pp. 391–396.

[2] V. Vyatkin, H.-M. Hanisch, and T. Pfeiffer, "Object-oriented modular place/transition formalism for systematic modeling and validation of industrial automation systems," in *IEEE International Conference on Industrial Informatics (INDIN), 2003*. IEEE, 2003, pp. 224–232.

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

[4] 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.

[5] J. Thieme and H.-M. Hanisch, "Model-based generation of modular PLC code using IEC61131 function blocks," in *IEEE International Symposium on Industrial Electronics (ISIE), 2002*, vol. 1. IEEE, 2002, pp. 199–204.

[6] V. Vyatkin and H.-M. Hanisch, "Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems," in *8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2001*, vol. 2. IEEE, 2001, pp. 113–118.

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

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

[9] 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.

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

[11] S. von Styp and L. Yu, "Symbolic model-based testing for industrial automation software," in *Hardware and Software: Verification and Testing*. Springer, 2013, pp. 78–94.

[12] E. P. Enoiu, D. Sundmark, and P. Pettersson, "Model-based test suite generation for function block diagrams using the UPPAAL model checker," in *6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2013*. IEEE, 2013, pp. 158–167.

[13] E. A. Lee, "Cyber physical systems: Design challenges," in *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008*. IEEE, 2008, pp. 363–369.

[14] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[15] C. Pang and V. Vyatkin, "Automatic model generation of IEC 61499 function block using net condition/event systems," in *6th IEEE International Conference on Industrial Informatics (INDIN), 2008*. IEEE, 2008, pp. 1133–1138.

[16] 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.

[17] 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.

[18] C. Gerber, S. Preuße, and H.-M. Hanisch, "A complete framework for controller verification in manufacturing," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. IEEE, 2010, pp. 1–9.

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

[20] I. Schaefer, D. Gurov, and S. Soleimanifard, "Compositional algorithmic verification of software product lines," in *Formal Methods for Components and Objects*. Springer, 2012, pp. 184–203.

[21] S. Soleimanifard and D. Gurov, "Algorithmic verification of procedural programs in the presence of code variability," in *Formal Aspects of Component Software*. Springer, 2014, pp. 327–345.