Buzhinsky, Igor; Pakonen, Antti; Vyatkin, Valeriy

Synthesis-Aided Reliability Assurance of Basic Block Models for Model Checking Purposes

Published in:
Proceedings of the 2018 IEEE 27th International Symposium on Industrial Electronics, ISIE 2018

DOI:
10.1109/ISIE.2018.8433793

Published: 10/08/2018

Document Version
Peer reviewed version

Please cite the original version:
Buzhinsky, I., Pakonen, A., & Vyatkin, V. (2018). Synthesis-Aided Reliability Assurance of Basic Block Models for Model Checking Purposes. In *Proceedings of the 2018 IEEE 27th International Symposium on Industrial Electronics, ISIE 2018* (Vol. 2018-June, pp. 669-674). [8433793] (Proceedings of the IEEE International Symposium on Industrial Electronics). IEEE. https://doi.org/10.1109/ISIE.2018.8433793

Synthesis-Aided Reliability Assurance of Basic Block Models for Model Checking Purposes

Igor Buzhinsky[1, 2], Antti Pakonen[3], Valeriy Vyatkin[1, 4]
[1] Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
[2] Computer Technology Department, ITMO University, St. Petersburg, Russia
[3] VTT Technical Research Centre of Finland Ltd., Espoo, Finland
[4] Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden
igor.buzhinskii@aalto.fi, antti.pakonen@vtt.fi, vyatkin@ieee.org

***Abstract*—In the Finnish nuclear industry, model checking, a formal verification technique, is used as an additional means of safety assurance for instrumentation and control (I&C) system design. Since the code of vendor-specific basic function blocks used in I&C is commonly closed, these blocks need to be modeled manually based on available specification. This modeling introduces an additional source of human factor into the verification process. To increase the reliability of the library of basic blocks used in nuclear I&C verification, we apply formal synthesis techniques, which can construct finite-state models of reactive systems from behavior examples and temporal properties. Since these techniques have computational limitations and synthesized models are hard to understand even by an analyst, we do not use them in the final verification process. Instead, in an iterative process, behavioral differences between a synthesized model and a manual model implementation are identified and used to create a list of features of manual implementations which either violate the specification or show that the specification is ambiguous.**

***Index Terms*—model checking, formal verification, formal synthesis, nuclear I&C systems.**

I. Introduction

As safety-critical automation systems need to be proven to be reliable before they are put in use, traditional verification approaches such as testing and simulation are not sufficient. Model checking [1], [2], a formal verification approach, is a way to exhaustively explore the state space of the model of the system in order to check whether certain functional requirements are satisfied. In particular, since 2008 model checking has been applied in the Finnish nuclear industry for verifying instrumentation and control (I&C) system designs [3].

Nowadays, a common activity related to model checking is manual preparation of formal models of the system's components. This introduction of the human factor into model checking reduces its reliability. One possible remedy is to obtain formal models directly from the source code of the system [4], but this is not possible (1) when the modeled components describe the plant, not the controller, and (2) when the source code is closed to the analysts [5]. An alternative approach is to apply formal synthesis techniques [6]–[11], but they have computational limitations which often make them inapplicable in practice, and their output is difficult for a human to understand.

In this study, we focus on the problem of reliability assurance of basic function block models, a network of which is the ultimate target for model checking. For many major I&C system vendors, such blocks are proprietary, non-standard and closed-source, and thus need to be modeled manually based on available natural language specifications. On the other hand, if such specifications are formalized, formal synthesis such as LTL synthesis [7], [8] becomes possible. By performing iterated equivalence checks of synthesized and manual implementations, possible issues in the latter can be identified, which comprise both alternative ways of implementing the original specification (in the cases of specification ambiguity), and situations where the original specifications is violated.

We apply this approach on several function block models taken from the Apros continuous simulator. The problem of computational complexity of synthesis is overcome by placing limitations on the input and output parameters of the synthesized model. The results of our study show that even with such limitations multiple issues of manual block model implementations can be revealed.

The rest of the paper is structured as follows. Section II introduces necessary concepts and used tools. Section III describes the proposed framework of synthesis-aided basic block model reliability assurance. In Section IV, this framework is applied on a number of basic block models. In Section V, it is compared with model checking. The results are discussed in Section VI.

II. Preliminaries

A. Model checking

Model checking [1], [2] is a formal method of checking (usually functional) requirements of various systems by means of state space exploration. To enable model checking, a formal model of the system must be prepared in either a graphical or textual form, depending on the used verifier. For example, popular model checkers NuSMV [12] and SPIN [13] utilize textual notations.

For reactive systems, requirements to be checked are most typically specified in *linear temporal logic* (LTL) or *property specification language* (PSL). These languages utilize the discrete time semantics. In LTL, which we use in this paper, *temporal operators* such as $\mathbf{X}, \mathbf{F}, \mathbf{G}$ allow expressing the desired behavior on the next time step of system execution, eventually, or always, respectively. For example, LTL property

$\mathbf{G}(x \rightarrow \mathbf{X}\, y)$ states that whenever predicate x (which is some condition over the variables of the system) is satisfied, another predicate y must be satisfied on the next time step.

B. Nuclear I&C verification

Model checking has been used to verify nuclear power plant I&C systems in the Republic of Korea [14] and Hungary [15]. In Finland, VTT has verified I&C system designs related to Olkiluoto 3 new-build, Loviisa 1&2 renewal, and the planned Hanhikivi 1 nuclear power plants. Since 2008, VTT has identified over forty design issues, leading to, e.g., design changes in the Loviisa Reactor Power Control System and Reactor Trip System [3].

VTT has also developed a graphical tool called MODCHK [5] for verifying function block diagram based application logics. Instead of relying on standard programming languages like IEC 61131-3, MODCHK allows the user to create vendor-specific basic function block libraries—also supporting features like signal validity, commonly used in the nuclear industry [16]. Having such a library, the user can construct a modular, hierarchical block diagram with a graphical editor, and generate the necessary input files for NuSMV. MODCHK then visualizes counterexamples by animating the modeled block diagram.

MODCHK currently requires the analyst to redraw the block diagram instead of supporting direct transformation from different I&C development and modeling tools. One of such tools is Apros, a dynamic process simulator used—among other domains—in the nuclear industry [17]. Apros also provides a set of function blocks for modeling the plant I&C systems. VTT and the Finnish utility Fortum have a common project for integrating Apros with MODCHK, allowing direct model transformation and verification. To that end, basic I&C blocks of Apros have been modeled with NuSMV.

C. Basic block models

Let I and O be the sets of input and output *variables*. Each of these variables can be either Boolean or integer. In the latter case, we assume that the value set of the variable is bounded. Informally, a *basic block model* is an entity which operates in discrete steps and is able to keep memory between steps. On each step, it deterministically relates given input values with corresponding output values. Basic block models can be specified in various formats, depending on the used model checking tool, such as NuSMV or SPIN.

As an example, consider the unit delay block with Boolean input and output variables. This block returns `false` on the first step, and the previous input value on each next step. Below, it is provided in the format of the NuSMV verifier:

```
MODULE UNIT_DELAY(INPUT)
VAR
    last: boolean;
DEFINE
    OUTPUT := last;
ASSIGN
    init(last) := FALSE;
    next(last) := INPUT;
```

Later, we will consider automatic synthesis of basic block models. Synthesis tools commonly assume that they are represented as finite-state machines. By $v(I)$ and $v(O)$ we denote the sets of all possible value combinations of input and output variables. Then, a *finite-state machine* (FSM) is a tuple $(S, s_0, v(I), v(O), \delta, \lambda)$, where S is a finite set of *states*, $s_0 \in S$ is the *initial state*, $v(I)$ and $v(O)$ are defined above, $\delta : S \times v(I) \rightarrow S$ is the *transition function* and $\lambda : S \times v(I) \rightarrow v(O)$ is the *output function*. On each step, the FSM accepts input values, changes its state according to δ and produces corresponding output values given by λ. Fig. 1 shows the unit delay block model as an FSM.

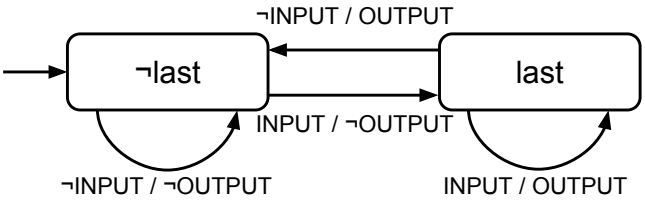

Fig. 1. Unit delay basic block model represented as a state diagram of an FSM. `false` and `true` values of Boolean variables are indicated by the presence or absence of the negation operator "¬"

D. Controller model synthesis

Two common types of input data are used for formal synthesis: behavior traces and LTL properties. A *behavior trace* is a finite sequence $e_1, ..., e_k$, where each $e_i, 1 \leq i \leq k$ is a pair $e_i = (\iota_i, \omega_i)$ of input $\iota_i \in v(I)$ and output $\omega_i \in v(O)$ values. If a behavior trace is included into the input data, it requires the synthesized model to produce $\omega_1, ..., \omega_k$ when it is started from s_0 and fed with $\iota_1, ..., \iota_k$. For example, the following trace is a partial specification for the unit delay block (Fig. 1): $(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true}), (\texttt{false}, \texttt{false})$.

Next, *LTL properties* are formal requirements which relate the states of the model in different time instants. While in model checking they are used as specifications to be checked, in synthesis they postulate the desired behavior: assuming that input values are freely chosen on each step, the synthesized FSM needs to produce output values such that all specified LTL properties are satisfied. For example, the following LTL property is necessary and sufficient to specify the input-output behavior of the unit delay block (Fig. 1): $\neg\texttt{OUTPUT} \wedge \mathbf{G}(\texttt{INPUT} \leftrightarrow \mathbf{X}\,\texttt{OUTPUT})$.

In the problem of *LTL synthesis*, input data is limited to a set of LTL properties. The works [6]–[8] are among the ones which propose solutions for this problem along with the tools which implement them. Then, note that operator $\mathbf{X}$ allows representing behavior traces in LTL, but for the purpose of computational efficiency it is possible to consider them separately [9]. Finally, there are approaches, such as [10], which treat behavior traces as the only kind of input data.

III. Proposed framework

The proposed framework of synthesis-aided basic block reliability assurance is based on iterating the following actions: (1) preparation of formal specification, (2) formal synthesis of basic block models which comply with this specification, and (3) equivalence checking of the behaviors of synthesized models with manually prepared models. All these actions need to be performed by an analyst who is familiar with LTL. The workflow of the framework is outlined in Alg. 1 and is explained below.

A. Input data

The following entities are used as the input data:

1) S, the textual specification of the basic block. Ideally, it is thorough and unambiguous, although these properties are not always achieved in reality.
2) M, the manual implementation of the basic block model in a formal language such as NuSMV. M has an interface (I, O) composed of input and output variables, which we assume to be correct, i.e. automatically synthesized models will have the same interface.

B. Output data

The output data of the framework is the *list of issues*, which is represented as a non-formalized text document. Each *issue* is a particular feature of M which either violates S or corresponds to a part of S which appears ambiguous. Each issue originates from a difference of behavior between M and a synthesized model. Its description is composed of an example showing the difference between these behaviors, the descriptions of a particular aspect of M's behavior which raises concern, and the condition leading to this behavior.

C. Intermediate data

During the workflow of the framework, the following additional entities are maintained:

1) L, the formalized version of S. L is composed of the interface (I, O, r) of the basic block to be synthesized, and a set of LTL properties F. The interface is the same as the one of M with the following difference: to enable formal synthesis, integer variables need to be annotated with their ranges, which are specified in the mapping $r : I \cup O \to \mathbb{Z}^2$. Initially, r places strict constraints on I and O, and then these constraints are weakened gradually. The rationale behind this approach is to allow formal synthesis computationally and simplify the manual analysis of its outcome: rather than dealing with multiple counterexamples related to multiple issues, the analyst will work only with the ones possible with the current r. Next, LTL properties within F are specified over the variables from I and O. Finally, behavior traces may be also included into L if they can be prepared based on S. Although we do not consider behavior traces further, their use will not alter the proposed framework.
2) $f_{\equiv}$, the LTL formula which checks behavioral equivalence of M and synthesized models $\{M_j\}$. Initially, $f_{\equiv}$ is set to $\mathbf{G}\left(\bigwedge_{i=1}^{|O|}\left(o_i\left(M_j\right) = o_i\left(M\right)\right)\right)$, where $o_i(M_j)$ and $o_i(M)$ are the output variables of M_j and M, respectively.

D. Workflow

Initially, the analyst formalizes S into L (line 1). This initial version of the LTL specification will be refined later. After that, the iterative procedure (lines 2–19) starts. On each iteration, the analyst first attempts to synthesize the models which comply with L (line 3). Since multiple synthesis tools are known [6]–[9], different tools can be used to obtain different models. In this paper, we use the following tools: Unbeast (https://www.react.uni-saarland.de/tools/unbeast), G4LTL-ST (https://sourceforge.net/projects/g4ltl), BoSy (https://github.com/reactive-systems/bosy), EFSM-Tools (https://github.com/ulyantsev/EFSM-tools). We created a wrapper tool (https://github.com/igor-buzhinsky/synthesis-aided-basic-block-assurance) which is able to run each of them assuming that the LTL specification is represented in a unified format and to output the produced state machine in NuSMV.

Each tool is run with a time limit. If such a time limit is violated, this means either that the current version of L is too complex for the tool, or that L is *unrealizable*, i.e. the synthesis problem has no solution. Distinguishing these cases can be done by running the tool in a mode which proves unrealizability (if the tool supports such a mode) or by disabling/changing some of the formulas in L.

If L is unrealizable (line 4), the analyst must understand the causes of unrealizability and correct L before running synthesis again (lines 5–7). Alternatively, when all the tools time out (line 8), the iterative process stops (line 9). However, if the analyst believes that some issues are not yet discovered, the analyst may instead decide to increase the time limit.

Otherwise, the tools will produce FSMs $\{M_j\}$ which comply with L (the number of these models equals to the number of tools which terminated within the time limit). In this case, each of $\{M_j\}$ is compared with M by combining them into a NuSMV model where M_j and M execute independently and input values are chosen freely, and then by checking $f_{\equiv}$ (line 11). The model checking algorithm can be chosen freely among the available ones, although we recommend using bounded model checking [18] as the one capable of producing minimum counterexamples, which are easier for manual analysis.

If all model checker executions report that $f_{\equiv}$ is satisfied (line 12), this means that no differences between input-output behaviors of $\{M_j\}$ and M were found. Hence, no issues can be discovered. Still, more issues can be potentially discovered with higher boundaries on input and output variables. For this reason, L is refined (line 13) and the new iteration of the framework starts.

Otherwise, a number of counterexamples are returned by the model checker. These counterexamples are examined by the analyst, who needs to understand their causes (line 15). As a means of simplifying counterexample analysis, we

use the counterexample visualization tool (https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer) developed by the authors based on the work [19]. Assuming that the synthesis tools and the model checker work correctly, the following situations are possible for each counterexample:

1) M_j shows behavior which violates the analyst's understanding of S. This means that the analyst has made a mistake while preparing L and this mistake needs to be fixed (line 17).
2) The behavior of M_j seems justified given S, but the behavior of M does not. In this case either a new issue is added to the list of issues (line 16), or an instance of an earlier created issue is recognized (this issue is potentially generalized to include new input conditions). This issue indicates the violation of S by M.
3) The behaviors of M_j and M are different, but the analyst cannot conclude that either of them is incorrect. The corresponding issue is created or updated (line 16). This issue indicates the ambiguity of S.

In the last two cases, two solutions are possible regarding exclusion of the current issue from consideration (if multiple issues have been identified on the current iteration, then only one issue is excluded before starting a new iteration):

1) Update L to match the discovered aspect of M's behavior.
2) Modify $f_{\equiv}$ to exclude situations causing the issue. Such modifications can, for example, include omitting checks of equality of particular output values, treating certain different output values as equivalent, or adding a condition on possible input/output value sequences which weakens the equality check (if this condition is represented by temporal formula c, then $f_{\equiv}$ is changed to $c \rightarrow f_{\equiv}$).

Since solution (2) can potentially hide issues different from the current one, solution (1) is preferred. However, if the logic of M is too laborious to represent in L, solution (2) may be a reasonable alternative.

IV. Case study

Although the ideal scenario of showing the applicability of the proposed framework would have been to use vendor-specific data, by now the authors faced difficulty with getting required permissions from the vendors.

On the other hand, MODCHK has a library of function block models (written in NuSMV) which correspond to the basic blocks of Apros, a process simulator which is used to model nuclear power plant I&C. For these basic blocks, manuals are available in Apros which were used as specifications while preparing the models. These manuals, although they are not as precise as industrial specifications, can be formalized in LTL. Moreover, in the Apros case, the original implementation of function blocks are available as Fortran code, which enables additional analysis which is impossible for closed-source blocks. Thus, we decided to use several Apros blocks in the case study.

Algorithm 1: Workflow of the synthesis-aided model reliability assurance framework

Data: textual specification S for the basic block, its manual implementation M
Result: list of issues in M and S

```
1  create L, the initial version of the formal specification, based on S;
2  while true do
3      attempt to synthesize block models M_1, ..., M_k using L as specification with available LTL synthesis tools;
4      if L is unrealizable then
5          understand the causes of unrealizability;
6          correct L;
7          continue;
8      else if all synthesis tools timed out then
9          break;
10     end
11     check equivalence of M with each of {M_j} using model checking with f_≡;
12     if all equivalence checks passed then
13         update L to support higher parameter boundaries;
14     else
15         understand the causes of counterexamples;
16         potentially discover as issue in M and/or S;
17         correct L or f_≡ to exclude such counterexamples;
18     end
19 end
```

A. Chosen basic block models

Among available Apros blocks, we focused on the ones whose behavior is time-dependent (unlike, e.g., the one of blocks implementing logical operations) since such behavior is much more error-prone according to the industrial experience of VTT. Four basic blocks were used in our study:

1) **Flip-flop** is a block capable of storing a single bit of memory. This bit can be set to `true` or `false` by `set` and `reset` signals respectively. This is the only used basic block whose variables are limited to Boolean ones.
2) **Binary delay** applies a delay to its binary input signal. However, the behavior of the block is not equivalent to shifting the signal in time, but instead the pulses of the input signal whose duration is less than the delay are ignored. The delay can not only be constant, but also can be specified with an additional continuous (integer in the case of formal models) signal.
3) **Pulse** generates its output signal with pulses which happen during rising edges of the input signal. Additional input signals define the behavior (with four distinct modes) and the length of pulses.
4) **Timer** is a block which produces continuous (integer in the case of formal models) output signals: the one linearly increasing over time since the start of the timer, and the one linearly decreasing and reaching zero when

the maximum specified time is reached. An additional Boolean output signal indicates whether the timer has elapsed. The timer can be reset or temporarily stopped by input Boolean signals.

These blocks are visible in the left part of Table I, where the numbers of their input and output variables are also provided.

B. Analysis

The framework was applied to each of the basic blocks mentioned in Section IV-A. The time limit of synthesis tools was set to five minutes, and their executions were performed on the Intel Core i7-4510U CPU with the clock rate of 2GHz.

For flip-flop, the process terminated in around one person-hour, and changes in L were limited to fixing the errors done by the analyst. Once a realizable version of L was prepared, the behavior of $\{M_j\}_{j=1}^{4}$ matched the behavior of M, and hence no issues were identified. Since this block does not have integer variables, line 13 of Alg. 1 was inapplicable.

During the analysis of binary delay, three issues were identified, all of which were connected with the imprecision of the manual. First, the manual did not specify the initial state. Second, the manual did not specify the effect of changing the delay value while the delay is being processed. Third, the manual was ambiguous regarding whether input pulses with length strictly equal to the delay value need to be visible in the output. The workflow involved 11 major updates of L (related to its generalization for larger delays and mimicking the behavior of M) and took around 8 person-hours.

For pulse and timer blocks, the process involved 13 and 11 major updates of L respectively, and around 8 person-hours were spent on each of these blocks. The analysis revealed multiple issues of manual implementations. For this reason, the second versions of these blocks were prepared based on the Fortran code of Apros blocks, and the analysis was repeated. Since the repeated analysis largely reused previously prepared L and $f_{\equiv}$, it required much less time.

The majority of identified issues were related to the following implementation aspects:

1) initial memory (such as initial values of previous input and output signals);
2) immediate vs. one-step-delayed reactions to input signal changes;
3) behavior during runtime changes of input variables whose runtime changes are unlikely and thus were not accounted for during model development, e.g. block modes and delays;
4) handling of clocks.

Table I summarizes the identified issues. While classifying issues into the ones involving errors in manual NuSMV modeling (12 issues) and the ones involving specification ambiguity (8 issues), we assumed that the manuals are the ground truth. This assumption is the only possible option in the situation of closed-source blocks. In our case, however, original implementations were available, although they were not used to prepare the manual implementations. Using this additional knowledge, we found that 3 of 12 issues classified as specification violations were in fact related to incorrect descriptions of the actual implementations in the manuals.

Another consequence of the availability of the original basic block code is the possibility to prepare versions of block models based on this code and not the manuals. As mentioned above, we applied this approach to pulse and timer block models since the numbers of implementation issues in them were high. As visible from the table, the compliance of such versions with the manuals was much higher. Yet, had these models been used as initial targets of the framework, some of specification ambiguity issues would have still been revealed.

V. Comparison with model checking

An alternative, more traditional quality assurance approach is model checking. The models used in the case study were already model-checked by their developer. A similar examination by a different analyst might have revealed some of the issues mentioned in Section IV-B. Still, while being a much simpler and faster approach, model checking of manually developed models has limitations:

1) constructive examples of alternative behaviors are unavailable to the analyst, hence understanding issues becomes more difficult;
2) the formal specification used in model checking may be unrealizable, in which case discovered issues may mention impossible behavior as desired or alternative;
3) the same specification may be incomplete, in which case some ambiguity issues can be missed (the same limitation of the synthesis-based approach is compensated by exploring different synthesized models);
4) some temporal formulas used in model checking may be incorrect and result into `true` instead of `false`, in which case some issues can be missed;
5) the analyst is not forced to consider different formalizations of the textual specification, which reduces the probability of discovering some issues.

VI. Discussion and conclusions

In this paper, we have proposed a way to improve the nuclear I&C formal verification methodology [5] wherein the verified model is represented as a network of basic block models, and each of such models is prepared manually based on natural language specification due to the source code being closed for the analysts. The suggested solution involves iterated application of formal synthesis approaches to explore behavioral differences between the manual and synthesized basic block implementations.

Subtle ambiguity in the natural language block specifications is not an issue limited to the Apros block manuals that we used in our research. In the practical industry projects carried out by VTT, it has at times been necessary to ask for clarification from the vendor. In particular, the specifications for time- and/or signal validity dependent processing do not necessarily describe—in explicit terms, or with sufficient accuracy—the intended response in unlikely or unintuitive scenarios.

TABLE I
APROS BASIC BLOCK MODELS AND ISSUES IDENTIFIED IN THEM

Basic block	$\|I\|$	$\|O\|$	Total issues	Issues related to specification ambiguity	Issues related to specification violation by	
					the initial model	the updated model
Flip-flop	4	2	0	0	0	N/A
Binary delay	3	1	3	3	0	N/A
Pulse	3	2	7	2	5	0
Timer	5	3	10	3	6	1
All			20	8	11	1

Then, timing diagrams are a user-friendly way to describe time-dependent blocks' behavior, but it is challenging to draw diagrams that include every possible combination of input sequences and block parameters. They were not used in our study. Yet, if they are present, they can be easily represented as behavior traces or LTL properties of the form $\mathbf{G}(f_i \rightarrow f_o)$, where f_i and f_o are predicates over input and output variables (possibly capturing several behavior steps).

As a by-product of the study, we have shown a practical application of formal synthesis techniques, which are currently not sufficiently developed to handle large industrial problem instances. Namely, if a problem of model development cannot be solved by formal synthesis due to the presence of integer parameters, synthesis can still be utilized for reliability assurance of corresponding manually prepared models if these integer parameters are made bounded. With such bounds, in our study, G4LTL-ST [6] and EFSM-Tools [9] were the tools which performed adequately when the number of LTL properties in the input specification L was high.

Currently, the automation of the framework is only partial. The following steps are automated: running multiple LTL synthesis tools based on LTL specifications written in a unified format, conversion of synthesized models from tool-specific formats to NuSMV, visualization of counterexamples. However, the overall workflow shown in Alg. 1 is not tool-supported, which makes it laborious when multiple iterations are needed. Such a tool support may be a direction of future work. On the other hand, the workflow would be simplified if some of the issues are known beforehand—specification issues can be revealed already at the stage of manual model development.

Furthermore, the framework should be applied on vendor-specific function blocks, as this would correspond to its actual domain of application in nuclear I&C verification. In particular, signal validity processing is an important aspect of systems used in industry (e.g. by Areva and Rolls-Royce).

ACKNOWLEDGMENT

This work has been funded by the Finnish Research Programme on Nuclear Power Plant Safety 2015–2018 (SAFIR 2018), and by the Ministry of Education and Science of the Russian Federation, project RFMEFI58716X0032.

REFERENCES

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

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

[3] A. Pakonen, T. Tahvonen, M. Hartikainen, and M. Pihlanko, "Practical applications of model checking in the Finnish nuclear industry," in *10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC & HMIT 2017)*. American Nuclear Society, 2017, pp. 1342–1352.

[4] D. Darvas, I. Majzik, and E. B. Viñuela, "PLC program translation for verification purposes," *Periodica Polytechnica. Electrical Engineering and Computer Science*, vol. 61, no. 2, pp. 151–165, 2017.

[5] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *18th IEEE Conference on Emerging Technologies & Factory Automation (ETFA 2013)*. IEEE, 2013, pp. 1–6.

[6] C.-H. Cheng, C.-H. Huang, H. Ruess, and S. Stattelmann, "G4LTL-ST: Automatic generation of PLC programs," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 541–549.

[7] R. Ehlers, "Unbeast: Symbolic bounded synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 272–275.

[8] P. Faymonville, B. Finkbeiner, and L. Tentrup, "BoSy: An experimentation framework for bounded synthesis," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 325–332.

[9] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 1, pp. 35–55, 2018.

[10] G. Giantamidis and S. Tripakis, "Learning Moore machines from input-output traces," in *Formal Methods: 21st International Symposium (FM 2016)*. Springer, 2016, pp. 291–309.

[11] 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, Aug 2017.

[12] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[13] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[14] J. Yoo, S. Cha, and E. Jee, "Verification of PLC programs written in FBD with VIS," *Nuclear Engineering and Technology*, vol. 41, pp. 79–90, 2009.

[15] E. Németh and T. Bartha, "Formal verification of safety functions by reinterpretation of functional block based specifications," in *Formal Methods for Industrial Critical Systems (FMICS 2008). LNCS 5596*. Springer Berlin Heidelberg, 2009, pp. 199–214.

[16] A. Pakonen and K. Björkman, "Model checking as a protective method against spurious actuation of industrial control systems," in *27th European Safety and Reliability Conference (ESREL 2017)*. Taylor & Francis Group, London, UK, 2017, pp. 3189–3196.

[17] J. Näveri, T. Tahvonen, and P. Hakasaari, "Testing and utilization of Loviisa full scope Apros model in engineering and development simulator," in *International Youth Nuclear Congress (IYNC 2010)*, 2010.

[18] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[19] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," *Formal Methods in System Design*, vol. 40, no. 1, pp. 20–40, 2012.