Timed model checking of fault-tolerant nuclear I&C systems

Buzhinsky, Igor; Pakonen, Antti

Published in:
Proceedings of 18th IEEE International Conference on Industrial Informatics, INDIN 2020

DOI:
10.1109/INDIN45582.2020.9442188

Published: 20/07/2020

Document Version
Peer reviewed version

Timed model checking of fault-tolerant nuclear I&C systems

Igor Buzhinsky[1, 2], Antti Pakonen[3]
[1] Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
[2] Computer Technologies Laboratory, ITMO University, St. Petersburg, Russia
[3] VTT Technical Research Centre of Finland Ltd., Espoo, Finland
igor.buzhinskii@aalto.fi, antti.pakonen@vtt.fi

***Abstract*—Certain safety-critical systems, such as nuclear instrumentation and control (I&C) systems, must be ensured to be correct. One of the approaches of doing this is formal verification and, in particular, model checking, which thoroughly examines the state space of the formal model of the system. To make model checking computationally feasible, many simplifying assumptions, often referred to as abstractions, are made. One of such abstractions is the assumption of discrete time. However, when I&C systems are considered working in the real world, where communication delays and failures are possible, this assumption becomes less realistic, calling for the need for richer formalisms. In this paper, using timed automata, we extend our previous model checking approach for nuclear I&C systems to account for continuous time. We apply our approach to a reactor protection system case study and show that continuous-time verification is in general feasible, although proving the satisfaction of certain system properties still remains a computational challenge.**

I. Introduction

Nuclear instrumentation and control (I&C) systems are among safety-critical systems that deserve much attention to make them reliable. The defense-in-depth design principle assures this reliability on multiple levels, from personnel training to I&C hardware and software. In Finland, one of the techniques to ensure the correctness of these implementations is model checking [1], [2], a formal verification [3] technique capable of exhaustive state space exploration.

While many works, such as [4]–[7], were devoted to software-only verification, the works [8]–[10] also accounted for possible hardware failures. However, all these works operated with discrete time, which corresponds to the number of performed state transitions of the model. This assumption is well-justified when the system to be model-checked is a cyclically executed controller, such as a programmable logic controller (PLC), in which case a cycle can be associated with a discrete time unit. However, when several such controllers communicate via a network, the cycles of these controllers may not be aligned in time. Moreover, a real-world communication delay may not be a multiple of the cycle time and vice versa, making the discrete time semantics either less realistic or difficult to apply computationally.

This paper addresses the aforementioned problems by proposing a real-time modeling and model checking approach for redundant failure-tolerant I&C systems. This approach (1) makes the model of communication within the I&C system more precise, and (2) enables checking of specifications with explicitly specified response time durations. This is achieved based on the use of timed automata and the capabilities of the nuXmv [11] model checker.

We evaluate our approach by modeling a reactor protection system case study and model-checking regular and timed temporal properties for this model. Our results show that such properties can be checked with an imprecise approach, but precise model checking still remains a challenge especially for timed temporal properties.

The rest of the paper is organized as follows. In Sections II and III, we introduce the concepts used throughout the paper. In Section IV, the proposed modeling approach is outlined. In Section V, this approach is evaluated on a case study. In Section VI, it is compared with similar approaches. Section VII concludes the paper and discusses future work.

II. Nuclear I&C systems

A. Running example

In this paper, we use a part of the case study from our previous work [10]—the reactor protection system (PS), which is a typical nuclear I&C system. More specifically, this is a fault-tolerant safety function whose purpose is to initiate reactor trip (shutdown) as a response to certain sensor measurements.

The PS case study is based on the proposed U.S. EPR nuclear power plant (NPP) [12], with the details not found in [13] based on our own assumptions. To enable fault tolerance, the PS consists of four redundant, identical *divisions* that are separated electrically and placed in different rooms. In each division, the application logic is distributed between two types of *function units*:

1) *Acquisition and processing units* (APUs) obtain inputs from the sensors of the I&C system and process them (e.g., compare with thresholds, wait for certain values to be present for a specified time).
2) *Actuation logic units* (ALUs) collect the outputs of the APUs, perform voting (e.g., require that at least 2 out of 4 APUs vote for actuation) and, also accounting for additional human operator inputs, generate the values that can be sent to actuators or other I&C systems. We regard the outputs of the ALUs as the outputs of the PS.

The architecture of the PS is shown in Fig. 1. Function units execute on their own devices and therefore their single

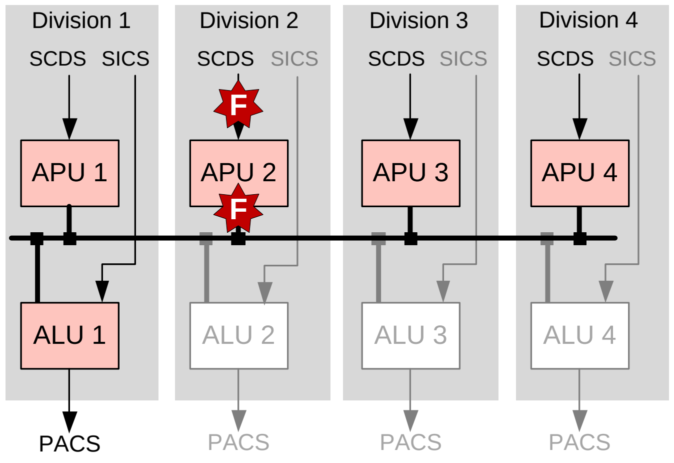

Fig. 1. Architecture of the PS (reproduced from [10]). Function units and connections that are omitted in the formal model are shown in grey. Failures are injected into division 2.

execution cycles can be considered atomic. The application logic running inside each function unit is specified using a network of interconnected *basic blocks* that model logical elements such as Boolean operators, flip-flops, delays and timers. These networks for the PS case study can be found in [10]. We also note that each signal in Fig. 1, as well as inside the APUs and the ALUs, is paired with its *fault status*. This status may be turned on for some failures and then used in processing, for example, by excluding knowingly unreliable signals from voting.

B. Hardware failures

According to the Finnish regulatory guides on nuclear safety [14], a safety function must be tolerant to arbitrary failures in a single division (N+1 requirement). Combining this assumption with redundancy and symmetry leads to the possibility of the following modeling assumptions, which were proposed in our previous works [9], [10]:

- A single division may be assigned as the one with the failure, and the failures can include arbitrary, self-announcing (indicated with fault statuses) and non-self-announcing corruptions of unit outputs and signals transmitted through connections. We select division 2 as the failing one.
- Only one non-failing ALU needs to be modeled. We select the ALU from division 1. The failing ALU from division 2 does not need to be modeled as well.

C. Network communication

There are two sorts of connections in the PS:

- The connections from the sensors and the human operators to function units are hardwired. They are encoded with 4..20 mA current, one signal per wire [13]. Failure statuses are not carried this way, but instead the signal can be marked as invalid by the APU if, e.g., the current drops below 4 mA.
- According to [12] (Chapter 11), each function unit has a Profibus communication module connected to an optical link module (OLM). OLMs are used to transfer signals through fiber-optic cables to other OLMs, this way achieving electrical isolation [12] (Chapter 6). As a result, transmitted data (with a possibility of grouping several signal values together to be transmitted atomically) is passed through several buffers on the way from an APU to an ALU. However, to reduce the computational complexity of modeling, we will assume that there is a single buffer between each APU and ALU with bounded transmission delays. These delays can be estimated as explained in [12] (Appendix B).

III. Model checking

Model checking is a formal verification technique to exhaustively explore the state space of the *formal model* of a system. It is performed automatically by tools called *model checkers*. Often, formal models are represented with different state machine formalisms, such as Kripke structures (for discrete-time model checking), timed transition systems [15] and timed automata [16] (for continuous-time model checking). In verification of timed automata, discrete and timed transitions are distinguished: the former instantly change the state of the model or its component, and the latter only increment the elapsed time and the values of clocks defined in the model.

Model checking is often resource consuming. One part of this problem comes from the need to process large state spaces. This problem is mitigated by the use of *symbolic model checking*. Among the verifiers that implement symbolic model checking are NuSMV [17] and nuXmv [11]. They implement model checking algorithms such as the ones based on binary decision diagrams (BDDs), bounded model checking [18] (BMC) and IC3 [19] (nuXmv only).

A. nuXmv model checker

While some model checkers, such as UPPAAL, allow the user to specify the models directly as state machines, other tools offer different languages. NuSMV [17] offers a language where a model is specified as an arrangement of interconnected nested modules, and each module has its own state variables, initialization and transition rules. The MODCHK [4], [5] tool, which is used in VTT for customer and research projects, in addition offers a graphical interface to specify the connections between the modules visually. The model can be verified directly in this tool (using MODCHK as a front-end for NuSMV) or exported as a NuSMV model.

The nuXmv verifier is a successor of NuSMV, whose recent version [11] supports timed model checking. Below, we provide an example of a nuXmv listing defining a single module, which represents a buffer that can be used to store a single Boolean value. This way, a delay in its transmission over the network can be modeled, and in our approach (see Section IV-B) we use buffers of this kind on the connections between function units.

```
MODULE BUFFER(IN0, SHOULD_EXEC)
VAR
  OUT0: boolean;
  timer: clock;
ASSIGN
  init(OUT0) := IN0;
  next(OUT0) := timer >= 50 & SHOULD_EXEC
    ? IN0 : OUT0;
  next(timer) := timer >= 50 & SHOULD_EXEC
    ? 0 : timer;
INVAR
  TRUE -> timer >= 0 & timer <= 100
```

In this example, if `timer` is between 50 and 100 (time units are abstract, but they can be mapped to a real-world duration, e.g., 1 ms) and an external execution permission `SHOULD_EXEC` is given, the buffer can update its stored version of input variable `IN0`, resetting the timer. Specifically, the `ASSIGN` declaration is used to specify possible initial values of state variables, and `INVAR` constrains the timer such that it starts from a value between 0 and 100 and cannot elapse beyond 100. Finally, fault statuses are not shown here for simplicity, but they can be treated as Boolean signals processed simultaneously with `IN0`.

B. Temporal logics

Linear temporal logic (LTL) [20] is a formal language that extends Boolean propositional logic with *temporal operators*, which can relate Boolean expressions at different discrete time instants of a single execution scenario. LTL temporal operators include $\mathbf{X}$ ("in the next state"), $\mathbf{F}$ ("eventually in the future") and $\mathbf{G}$ ("always in the future"). *Metric interval temporal logic* (MITL) [21] is similar to LTL, but facilitates timed model checking by annotating temporal operators with time intervals. In particular, in addition to LTL, nuXmv supports MITL operators $\mathbf{G}_{[a,+\infty)}$ (always, starting from a time units from the current moment), $\mathbf{G}_{[0,a]}$ (always during the entire next a time units), $\mathbf{F}_{[a,+\infty)}$ (eventually in the future, starting from a time units from the current moment), $\mathbf{F}_{[0,a]}$ (eventually during the next a time units). In these operators, a is a real number. For example, the property $\mathbf{G}((\mathbf{G}_{[0,10]}\ \texttt{IN0}) \to \mathbf{F}_{[0,10]}\ \texttt{OUT0})$ is satisfied for the `BUFFER` example above, meaning "always, if `IN0` is true for 10 future time units, then `OUT0` will also become true during the same time interval."

Usually, LTL or MITL formulas (or *temporal properties*) correspond to system specifications. If a property is violated, it is always possible to find a *counterexample*—a behavior scenario of the system showing the violation.

IV. Proposed modeling approach

A. Timed models

Broadly speaking, our modeling approach is based on the formalism of timed automata. A *timed automaton*[1] is a tuple (L, L_0, C, E), where L is a finite set of *locations*, $L_0 \subseteq L$ is a set of *initial locations*, C is a finite set of real-valued *clocks*, and $E \subseteq L^2 \times \mathbb{R}^{|C|} \times 2^C$ is a set of *edges*. A timed automaton can be interpreted as an executable model: its state is composed of a location and an assignment of the clocks, the clocks are initialized with zeros, an initial location is selected nondeterministically from L_0, and $(l_1, l_2, t_1, ..., t_{|C|}, C_r) \in E$ means that location l_1 can change to location l_2 when the values of clocks are $t_1, ..., t_{|C|}$, and all the clocks from the set C_r will be reset to zero as a result of this transition. In addition to such *discrete transitions*, *timed transitions* are possible, which synchronously increment all the clocks by some positive value. Below, we tailor this definition for our needs, essentially introducing variables, modularity, and associating each module with exactly one clock.

[1]We use a simplified definition compared to the classical one [16].

A *clock module* is a tuple $(V^{\text{in}}, V^{\text{s}}, V^{\text{out}}, R, L, L_0, c, E)$, where $V^{\text{in}} = \{v_1^{\text{in}}, ..., v_k^{\text{in}}\}$ is a finite set of *input variables*, $V^{\text{s}} = \{v_1^{\text{s}}, ..., v_m^{\text{s}}\}$ is a finite set of *state variables*, $V^{\text{out}} \subseteq V^{\text{s}}$ is a finite set of *output variables* (state variables that can be accessed by other parts of the modular model), $R : V^{\text{in}} \cup V^{\text{s}} \to \{\text{true}, \text{false}\} \cup 2^{\mathbb{Z}}$ is a *range function*, which returns the set of values for each variable, $L \subseteq R(v_1^{\text{s}}) \times ... \times R(v_m^{\text{s}})$ is a set of *locations*, $L_0 \subseteq L$ is the set of *initial locations*, c is a *clock*, and $E \subseteq L^2 \times \mathbb{R}^2 \times R(v_1^{\text{s}}) \times ... \times R(v_m^{\text{s}})$ is the set of *edges*: $(l_1, l_2, t_{\min}, t_{\max}, x_1, ..., x_k) \in E$ if and only if a transition from l_1 to l_2 is allowed when $t_{\min} \leq c \leq t_{\max}$ and $v_i^{\text{in}} = x_i$ for each $1 \leq i \leq k$. During clock module execution, we assume that initially $c \geq 0$, and c is reset to zero after each discrete transition. Thus, a *clock module* is a variant of a timed automaton that has only one clock.

A *clock unit* with execution cycle length r is a clock module such that each its edge has form $(l_1, l_2, r, r, x_1, ..., x_k)$, i.e., a transition is only possible when $c = r$. We require that the initial value of the clock of a clock unit is $0 \leq c \leq r$. A *clock buffer* with bounds $r_{\min}$ and $r_{\max} \geq r_{\min}$ is a clock module such that each its edge has form $(l_1, l_2, r_{\min}, r_{\max}, x_1, ..., x_k)$, i.e., a transition is only possible when $r_{\min} \leq c \leq r_{\max}$. Clock buffers are suitable for representing nondeterministic transmission delays in a network.

A *timed model* is a tuple (V, R_V, M, X), where V is a set of *system input variables*, R_V is the range function of these variables (defined as for clock modules), M is a set of clock modules, and X is a set of connections of the form (v_1, v_2), where v_1 belongs to V or V^{out} of some clock module, and v_2 belongs to V^{in} of another module. Thus, connections allow certain modules to read (but not set) system input variables or output variables of other modules. A timed model can be interpreted as a timed automaton, where (1) a location is an assignment of system input variables and state variables of each clock module, (2) initial locations correspond to arbitrary assignments of system input variables and possible initial locations of clock modules, (3) the set of clocks is composed of the clocks of clock modules, and (4) the set of edges corresponds to the following discrete transitions: system input variables change arbitrarily, and any positive number of clock modules make their own transitions and reset their clocks. If multiple clock modules can make discrete transitions, then these transitions can be performed synchronously or sequentially with all possible groupings. In a timed transition, clocks increment synchronously in all clock modules by some positive value. However, we limit this behavior with the following invariant: no clock can advance beyond r_{max} of any of the modules.

B. Implementing I&C systems as timed models

Generally, an I&C system consists of a number of function units and connections between them and sensor inputs. The I&C system can be modeled with a timed model as follows:

- Each sensor input in each unit is represented as a system input variable.
- Each function unit in each division is represented with a clock unit.
- Each connection is represented with a clock buffer, where the input variables of the buffer are the output variables of the sender or system input variables, and the input variables of the recipient are the output variables of clock buffers. Clock buffers are assigned with different r_{min} and r_{max} depending on the type of connection.

With this approach, the PS needs 5 clock units (we set $r = 50$, measuring time in ms), 13 clock buffers for hardwired connections (we set $r_{\text{min}} = 0$ and $r_{\text{max}} = 50$) and 4 clock buffers for Profibus connections (we set $r_{\text{min}} = 0$ and $r_{\text{max}} = 150$). A timeline of a possible execution of this model is shown in Fig. 2. Note that in this scenario, while APU 4 and ALU 1 execute concurrently, ALU 1 is unable to get the most recent output of APU 4 due to it being stored in the buffer. What is more, on the second cycle, ALU 1 receives the output which is even older than the previous one of the APU 4.

C. Implementing timed models in nuXmv

A timed model can be naturally specified in a timed model checker suitable for verification of asynchronous systems, such as UPPAAL [22]. Motivated by the works [6], [23], we tried implementing the timed model of the PS in UPPAAL but faced computational intractability of this model. Instead, we used the nuXmv model checker, where we implement timed models as follows:

1) The continuous time domain is selected (`@TIME_DOMAIN continuous`).
2) All system input variables and clock modules are declared in the `main` module.
3) Each clock module is implemented as a nuXmv module with a separate clock `timer` defined and an invariant of the form `INVAR TRUE -> timer >=` r_{min} `& timer <=` r_{max}. The part "`TRUE ->`" is included due to syntactic restrictions of the nuXmv language.
4) An *execution mask* $e_1, ..., e_{|M|}$, where $|M|$ is the number of clock modules, is specified as a Boolean array. If e_i is false during a discrete transition, then clock module i preserves its state variables unchanged. The constraint $e_1 \vee ... \vee e_{|M|}$ eliminates discrete transitions without system execution progress.
5) Clock buffers are implemented similarly to the code demonstrated in Section III-A, except that they are capable of handling several signals transmitted simultaneously.
6) Clock units are also implemented similarly, but with a notable distinction: instead of memorizing a number of values, a clock unit needs to execute a function block diagram that can be expressed as a modular (two-level in the case of the PS) nuXmv code without timed features. To support execution suspension (point 4 above), we modify nuXmv modules that contain state variables (i.e., whose output variables cannot be expressed as functions of input variables) to explicitly preserve their state when an additional input signal `SHOULD_EXEC` is false.

A slightly altered version of our model is available online[2]. The modification involves simplifying the fault signal processing logic to better match publicly available information.

V. Experiments

A. Temporal specifications

We formulated 48 temporal properties that are mostly related to different response-request and absence of spurious actuation [5] requirements. These properties are almost the same as the black-box requirements used in our previous work [10] under the presence of communication delays. Specifically, the same natural language requirement is often specified with multiple temporal properties of variable strength. However, unlike in our previous works, 28 of these requirements contained time-bounded MITL operators that specified concrete response durations. For example, the property

$$(\mathbf{G}\,\neg\texttt{MAN\_RESET}_1) \to \mathbf{G}((\mathbf{G}_{[0,150]} \vee_{i=1}^{4}(\texttt{HLEG\_P}_i > 70)) \to \mathbf{F}_{[0,300]}\,\texttt{RODS\_DOWN}_1)$$

means "if manual reset of ALU 1 is never requested, then always, if the hot leg pressure (`HLEG_P`) is above 70 bar in any division during then next 150 ms, then ALU 1 will produce the `RODS_DOWN` command at least once within the next 300 ms." This property is violated for the PS, but replacing "$\vee$" with "$\wedge$" makes it satisfied.

B. Experimental setup

For each temporal property independently, nuXmv was run with a time limit of one hour and a memory limit of 16 GB, with command-line options "`-coi -dynamic -df -int -time`". As verification algorithms, we used BMC (`timed_check_ltlspec -b` nuXmv batch command) and IC3 (`timed_check_ltlspec`). Both BMC and IC3 increase their internal *bound* k during processing. However, while IC3 is always able to find the outcome of a temporal property for some value of this bound, BMC is only able to detect property violations (but may be more efficient in doing this compared to IC3). All runs were performed twice: assuming 0 and 1 hardware failures. Experiments were done on a single CPU core (2.67 GHz) of a computational cluster.

[2]https://github.com/igor-buzhinsky/ps-timed-model

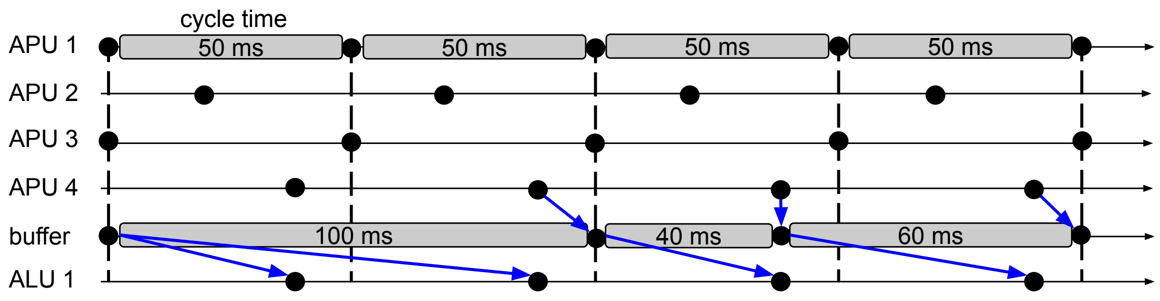

Fig. 2. A timeline showing a possible execution scenario of the PS. All units and the buffer between APU 4 and ALU 1 are shown. A black circle indicates a discrete transition (single execution) of a clock module. Each function unit executes with a cycle length of 50 ms, and in this example the executions of APU 1 and APU 3, and APU 4 and ALU 1 are synchronized. Unlike function units, the buffer updates its knowledge about the output of APU 4 at irregular but bounded times. Information transmission between APU 4 and ALU 1 is shown with blue arrows.

C. Results

The results of model checking are shown in Table I. Precise model checking with IC3, unfortunately, was unable to handle most of temporal properties, and in particular almost all timed properties. Checking of timed properties failed due to reaching the time limit. For a subset of these properties, we observed that increasing the time limit up to four hours was also insufficient. A possible explanation of this outcome is that nuXmv translates timed verification problems to untimed ones, and the length of timed temporal properties is increased several times when they are transformed to untimed ones.

On the other hand, BMC finds violations of both untimed and timed temporal properties almost instantly. In all unchecked cases, the time limit was reached. This happens either when the property is actually satisfied (BMC will increment its bound k infinitely in such cases), or showing their violation requires longer counterexamples. There are visible gaps between k reached for violated and unchecked properties. For example, for timed properties for a one-failure model, found counterexamples have at most $k = 15$, but all unprocessed cases reached at least $k = 19$.

The obtained results comply with our works [9], [10] on checking similar systems with discrete time semantics. Checking a model with communication delays is more computationally difficult and usually susceptible to BMC only. For non-timed temporal properties, we did not observe any changes of verification outcomes compared to discrete-time verification of the same properties.

VI. Related work

In [9], [10], an approach was proposed to verify nuclear I&C systems under the presence of hardware failures, asynchrony and communication delays. In Section II-B, we already mentioned the key modeling ideas of this approach that are essential for this paper. Unlike the present work, in [9], [10] modeling of asynchrony and communication delays was done with discrete modules capable of nondeterministic signal delays of up to several cycles. With this discrete-time approach, real-time specifications cannot be checked. On the other hand, the case study considered in [10] was a superset of the one used in this paper, but the approach of the latter can be easily generalized to I&C system models with a more complex structure than the one of the PS.

In [6], [23], a timed model checker UPPAAL [22] was applied to model a stepwise reactor shutdown system. Unlike in our work, timed automata were applied to model not network communication (it was not considered in [6], [23]), but low-level software components (basic blocks) such as timers. In our paper, internal contents of even time-dependent units are modeled with discrete time. This is a valid assumption since, for basic blocks that we consider, time durations that are not multiple of cycle time can be rounded without a change in the behavior of the block. Finally, formal models under verification in [6], [23] were limited to one function unit.

In [24], a fault-tolerant aerospace embedded system was modeled in UPPAAL with timed automata. The developed model does not involve network communication and is designed for a safety architecture different from ours.

VII. Conclusions

In this paper, we presented an approach to model and verify redundant failure-tolerant I&C systems under real-time semantics. This approach makes modeling more realistic and enables verification of timed (i.e., with specific time durations specified) temporal specifications for I&C systems that are distributed across multiple devices. As a result, response time requirements can be checked for such systems.

Our approach utilizes the nuXmv model checker, which was chosen due to its high performance and compatibility with our verification framework based on NuSMV and MODCHK. Nonetheless, due to its formulation in terms of timed automata, it can be implemented in other model checkers, although at the moment we are not aware of a better alternative.

The approach has several limitations. Due to computational complexity of verification, we are able to check response time requirements only with BMC. Also, due to the license of nuXmv, the approach cannot be applied in commercial projects. Other limitations may be addressed in future work:

1) At the moment, we neglect the fact that data transferred from APUs to ALUs is passed through more than one buffer. The used communication model could be made more detailed and thus even more realistic. However,

TABLE I
MODEL CHECKING RESULTS.

Temporal properties		Non-timed (total: 20)				Timed (total: 28)			
Model checking algorithm		BMC	BMC	IC3	IC3	BMC	BMC	IC3	IC3
Number of divisions with failures		0	1	0	1	0	1	0	1
Checked properties	Total	9	12	12	11	18	22	1	0
	Satisfied	0 [a]	0 [a]	9	7	0 [a]	0 [a]	0	0
	Violated	9	13	3	4	18	22	1	0
	Median processing time (s)	2.1	0.5	120.7	89.0	12.6	2.2	1014.1	—
	Median bound processed	9	4	12	12	11	11	21	—
	Maximum bound processed	11	11	12	12	15	15	21	—
Unchecked properties	Total	11	7	8	9	10	6	27	28
	Unchecked due to time limit	10	6	8	9	10	6	27	28
	Unchecked due to memory limit	1	1	0	0	0	0	0	0
	Median bound processed	40	45	190	243	22.5	22.5	65	92
	Minimum bound processed	23	32	23	198	18	19	15	15

[a] BMC can only prove that a property is violated.

we expect this change to increase the computational complexity of model checking.

2) Timed modeling and model checking could be tried on a larger case study, such as the one in [10]. This could be done by extending the tool proposed in [10] to support generation of timed formal models.

ACKNOWLEDGMENTS

This work was financially supported by the Finnish Research Programme on Nuclear Power Plant Safety 2019-2022 (SAFIR2022), and by the Government of the Russian Federation (Grant 08-08). We acknowledge the computational resources provided by the Aalto Science-IT project.

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] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.
[4] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2013, pp. 1–6.
[5] 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)*. CRC Press, 2017, pp. 3189–3196.
[6] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Reliability Engineering & System Safety*, vol. 105, pp. 104–113, 2012.
[7] A. Pakonen, J. Valkonen, S. Matinaho, and M. Hartikainen, "Model checking for licensing support in the Finnish nuclear industry," in *International Symposium on Future I&C for Nuclear Power Plants (ISOFIC)*. Korean Nuclear Society, 2014.
[8] J. Lahtinen, "Hardware failure modelling methodology for model checking," VTT, Tech. Rep. VTT-R-00213-14, 2014.
[9] A. Pakonen and I. Buzhinsky, "Verification of fault tolerant safety I&C systems using model checking," in *2019 IEEE International Conference on Industrial Technology, (ICIT)*. IEEE, 2019, pp. 969–974.
[10] I. Buzhinsky and A. Pakonen, "Model-checking detailed fault-tolerant nuclear power plant safety functions," *IEEE Access*, vol. 7, pp. 162 139–162 156, 2019.
[11] A. Cimatti, A. Griggio, E. Magnago, M. Roveri, and S. Tonetta, "Extending nuXmv with timed transition systems and timed temporal properties," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 376–386.
[12] Areva NP. (2012) U.S. EPR Protection System, Technical Report ANP-10309NP, Revision 4. [Online]. Available: https://www.nrc.gov/docs/ML1216/ML121660317.html
[13] ——. (2013) U.S. EPR Final Safety Analysis Report. [Online]. Available: https://www.nrc.gov/reactors/new-reactors/design-cert/epr/reports.html
[14] STUK. (2013) YVL B.1 Safety design of a nuclear power plant. [Online]. Available: https://www.stuklex.fi/en/ohje/YVLB-1
[15] T. A. Henzinger, Z. Manna, and A. Pnueli, "Timed transition systems," in *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer, 1991, pp. 226–251.
[16] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.
[17] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 359–364.
[18] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu *et al.*, "Bounded model checking," *Advances in Computers*, vol. 58, no. 11, pp. 117–148, 2003.
[19] A. Cimatti and A. Griggio, "Software model checking via IC3," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 277–293.
[20] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (SFCS)*. IEEE, 1977, pp. 46–57.
[21] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM (JACM)*, vol. 43, no. 1, pp. 116–146, 1996.
[22] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *3rd International Conference on the Quantitative Evaluation of Systems (QEST)*, Sept 2006, pp. 125–126.
[23] K. Björkman, J. Frits, J. Valkonen, J. Lahtinen, K. Heljanko, I. Niemelä, and J. J. Hämäläinen, "Verification of safety logic designs by model checking," in *6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT)*, 2009, pp. 5–9.
[24] M. Zhang, Z. Liu, C. Morisset, and A. P. Ravn, "Design and verification of fault-tolerant components," in *Methods, models and tools for fault tolerance*. Springer, 2009, pp. 57–84.