Explicit-State and Symbolic Model Checking of Nuclear I&C Systems: A Comparison

Igor Buzhinsky, Antti Pakonen, Valeriy Vyatkin

Citation:

Buzhinsky I., Pakonen A., Vyatkin V. Explicit-State and Symbolic Model Checking of Nuclear I&C Systems: A Comparison. 43rd Annual Conference of the IEEE Industrial Electronics Society (IECON). October 29 – November 01, Beijing, China, pp. 5439–5446. IEEE, 2017.

DOI: https://doi.org/10.1109/IECON.2017.8216942

Explicit-State and Symbolic Model Checking of Nuclear I&C Systems: A Comparison

Igor Buzhinsky[1, 2], Antti Pakonen[3], Valeriy Vyatkin[1, 4]
[1] Department of Electrical Engineering and Automation, Aalto University, Finland
[2] Computer Technology Department, ITMO University, St. Petersburg, Russia
[3] VTT Technical Research Centre of Finland Ltd, 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 some fields of industrial automation, such as nuclear power plant (NPP) industry in Finland, thorough verification of systems and demonstration of their safety are mandatory. Model checking is one of the techniques to achieve a high level of reliability. The goal of this paper is practical: we explore which type of model checking – either explicit-state or symbolic – is more suitable to verify instrumentation and control (I&C) applications, represented as function block networks. Unlike previous studies, in addition to the common open-loop approach, which views the controller model alone, we consider closed-loop verification, where the plant is also modeled. In addition, we present a procedure to translate block networks to the language of the SPIN explicit-state model checker.**

I. Introduction

The use of formal verification methods, and in particular of model checking [1], is especially justifiable in safety-critical domains like the nuclear industry [2]–[5]. This is due to both the need to avoid accidents, but also the practical need to demonstrate to regulators that the systems are sufficiently safe.

Model checking is a formal verification approach which analyzes the state space of the system under verification. In open-loop model checking, only the controller's model is analyzed, whereas in closed-loop model checking [6] the model of the controlled plant is also taken into account. Modeling feedback from the controller's environment usually reduces the state space of the model [7]. On the other hand, from the safety point of view, limiting behavior of the model by introducing the plant causes concern. Still, if closing the loop means that we can analyze applications that would otherwise be too complex, the approach would be very useful to *supplement* the more common open-loop assessment. Another practical advantage of closed-loop models is filtering out unrealistic feedback from the environment. Analysts would therefore avoid spending excess time on interpreting counterexamples dealing with irrelevant, even physically impossible scenarios.

Previously [8], [9], we verified nuclear I&C applications represented as block diagrams using a framework based on symbolic verifier NuSMV [10]. Usually, this verifier performed quite fast in the open-loop case, but not in the closed-loop case: while performing verification with explicit-state plant models generated automatically using the approach from [9], we noticed that such models significantly increase time required for model checking, which may be caused by their complexity. These results comply with the ones presented in [11]. Moreover, within a reasonable time limit, NuSMV was unable to process some models even in the open-loop scenario.

To mitigate the aforementioned problems, this paper explores the case of model checking block networks using the explicit-state model checker SPIN [12]. According to previous studies [13], [14], while suffering from the so-called state space explosion problem [15], SPIN scales better to harder verification problems than symbolic model checkers. Our primary interest is its application in closed-loop verification, whose application in the nuclear I&C domain we are exploring. In particular, the approach [9] generates plant models with small state spaces, which must reduce verification time compared to the open-loop case if an explicit-state model checker is used. Thus, we expect SPIN to handle larger plant models in realistic time, allowing us to consider more detailed (and hence potentially more reliable) plant models.

The key contribution of this paper is the comparison of explicit-state and symbolic model checking on the problem of verifying an I&C block network. The comparison is performed in both open-loop and closed-loop cases. In addition to performing the comparison, the paper proposes a procedure to translate block networks to the language of the SPIN verifier so that they could be processed efficiently.

The paper is organized as follows. In Section II, necessary details concerning model checking are provided. In Section III, the way how we model I&C applications is described. Section IV compares the results of using SPIN and NuSMV on our model checking problem. In Section V, our study is compared with prior research. Section VI concludes the paper and outlines future work.

II. Model checking

Model checking [1] is a formal verification technique which checks certain properties about the formal model of the considered system by performing exhaustive state space analysis. In the industrial context, model checking is used to ensure PLC program correctness [16], [17]. The use of model checking in safety assessment of nuclear I&C systems is considered in [2]–[5], [18].

978-1-5386-1127-2/17/$31.00 ©2017 IEEE

A. Temporal logics

Properties to be checked are formulated in formal languages such as *linear temporal logic* (LTL) [19], *computation tree logic* (CTL) [20], and *property specification language* (PSL). In this paper, we will consider only LTL specifications. In LTL, in addition to Boolean statements about the state of the system (e.g. certain variables have certain values), the user is allowed to use *temporal operators*. These operators allow writing properties checking the validity of system behaviors, which are represented as infinite sequences of states. A temporal formula is satisfied for the formal model if and only if it is satisfied for each behavior specified by the model.

The most commonly used temporal operators are $\mathbf{G}$, $\mathbf{F}$, $\mathbf{X}$ and $\mathbf{U}$. Assume that f and g are temporal formulas. Temporal formula $\mathbf{G}\,f$ means that f is satisfied globally along the behavior of the system. Then, $\mathbf{F}\,f$ states that f is satisfied at least once along the behavior, $\mathbf{X}\,f$ means that it is satisfied in the next state of the behavior, and $f\,\mathbf{U}\,g$ requires g to become true eventually, and f to be true until this moment.

B. Explicit-state model checking and SPIN

Explicit-state model checking is based on explicit state space analysis: that is, each processed reachable state of the system is represented in memory explicitly. The drawback of explicit-state model checking is *state space explosion* [15]: time and memory required to verify the system grows linearly with the number of its possible states. In turn, the number of possible states can be exponential of the number of bits used to store a single state of the system.

SPIN [12] is an explicit-state model checker which supports LTL. The textual language employed by SPIN is called Promela. In Promela, the user can build a formal model as a number of processes, which can execute asynchronously or adhering to explicitly defined means of synchronization. Processes execute statement by statement, and statement executions can interleave between processes unless a group of statements is declared as `atomic`. Furthermore, the keyword `d_step` merges multiple deterministic statements in a way that their intermediate states are not stored in memory, which makes model processing faster. Below, we provide an example of Promela process type which is responsible for introducing a unit delay: the output is defined by the value of the variable `last`, which stores the input value from the previous execution of the process.

```
bool INPUT, OUTPUT, last = 0;
proctype UnitDelay(chan ret) {
  OUTPUT = last;
  last = INPUT;
  ret ! false;
}
```

C. Symbolic model checking and NuSMV

Symbolic model checking [21] was introduced as a means of mitigating the state space explosion problem. In it, state subsets are encoded implicitly as Boolean formulas represented by binary decision diagrams (BDDs). The reason why the state space explosion problem is mitigated is, according to [15], that "in many practical situations space requirements for Boolean functions are exponentially smaller than for explicit representation." The key Boolean relations used in symbolic model checking are the ones representing the initial state (*init*) and possible state transitions (*trans*) of the system. Unfortunately, BDD processing is done using heuristics, and thus symbolic model checking may still represent a computational challenge for sufficiently complex systems.

NuSMV is a symbolic verifier capable of working with modular systems. It supports LTL, CTL and PSL. LTL and PSL can be checked not only using BDDs, but also with bounded model checking (BMC) [22]. Unlike SPIN, all NuSMV modules execute synchronously (although asynchrony can be modeled by forcing some modules to keep their states unchanged). NuSMV modules are formed not of sequential code, but of the declarations of *init* and *trans* relations. Below, a NuSMV module representing the unit delay is given as an example. Notice that the transition relation is specified implicitly by assigning the next value of the variable `last` using the keyword `next`.

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

D. Open-loop and closed-loop model checking

When industrial controllers are verified, this can be done in either open loop or closed loop. Open-loop model checking considers the model of the controller alone: its inputs are allowed to take any possible values (within certain ranges or value sets). In contrast, in the closed-loop case [6], the model of the controller is connected with the model of the plant in a feedback loop. This allows modeling the controller in the environment where it is supposed to operate in production. The presence of the plant model also allows verifying properties that involve plant variables unobservable by the controller. For properties which do not involve such variables, model checking outcomes may be different in open-loop and closed-loop cases, since in the first case plant behaviors are unrestricted.

A comparison of both the techniques is done in [11], and one of the conclusions of [11] states that these techniques are complementary. However, in [11] closed-loop model checking is slower than open-loop one. Two reasons, both of which are related to the use of symbolic verification and NuSMV, may explain such a result. First, the state of a closed-loop model is composed not only of the controller state but also of the plant state. This means a larger number of Boolean variables in BDDs, which, all else being equal, increases the computational complexity of working with them. Second, the length of the model's textual description is larger when the plant model is considered, which makes NuSMV process larger BDDs.

E. Model checking of nuclear I&C applications

In the Finnish nuclear industry, model checking has been used for nearly a decade to evaluate the design of I&C application software. Tens of issues have been identified, leading to design changes in safety classified systems [4]. A systematic methodology for modeling function block based nuclear automation systems has been proposed in [3]. In practice, model checking has been utilized in the Finnish nuclear industry by the regulator (STUK) for evaluating the I&C design of the Olkiluoto 3 power plant under construction, by the utility Fortum in the I&C renewal project of the Loviisa plants, and by the utility Fennovoima for evaluating the functional I&C architecture of another new-build (Hanhikivi 1) being planned [4].

A key challenge in model checking is to avoid the state space explosion phenomenon, which is a particularly serious problem in software verification [1], [15]. For I&C application software, several factors increase the state space:

1) processing of large amounts of analogue (numerical) data;
2) the use of memory and delay components to deal with timing and sequencing [5];
3) feedback loops.

While nuclear I&C functionality – especially in safety classified systems – should be kept simple, the need for fault tolerance means that for each process variable, there may be up to four redundant sets of measurements. The different, physically isolated "channels" of the systems exchange information and perform voting over control actions. Signal validity is used to exclude "bad" data from voting, if measurements fail or communication is lost [18].

The complexity of typical nuclear safety I&C functions is usually not an issue for model checkers like NuSMV. Exceptions include applications that process and store numerical data to memory by including complex control logic and/or feedback loops. Even for strictly binary logic, the sheer number of redundant inputs can lead to prohibitively long analysis times.

III. Formal modeling of I&C applications

To model I&C applications formally in discrete-state model checkers such as NuSMV and SPIN, they need to be represented as finite-state models. We achieve such a representation by first modeling basic blocks manually in NuSMV and Promela, and then drawing a block network in the software tool called MODCHK [8], [18], developed by VTT. Previously, this tool has been successfully applied by VTT in several customer projects related to Finnish nuclear I&C industry. MODCHK diagrams can be further converted to NuSMV and then translated into Promela. All the mentioned procedures will be described in more detail further in this section.

A. Basic blocks

A *Mealy finite-state machine* is a tuple $(S, s_0, I, O, \delta, \lambda)$. Here, S in a finite set of *states* and $s_0 \in S$ is the *initial state*. Then, I is a set of *inputs*, each of which has a finite set of possible values (e.g. Booleans or integers). The set of all input values combinations will be referred to as $v(I)$. Similarly, O is a set of *outputs*, and $v(O)$ is the set of output value combinations. Finally, $\delta : S \times v(I) \to S$ is a *transition function* and $\lambda : S \times v(I) \to v(O)$ is an *output function*.

A *basic block type* is an arbitrary Mealy finite-state machine. As an example of a basic block type, consider the block type which calculates the conjunction of three Boolean arguments. In this case, $S = \{s_0\}$, that is, there is only one state. Then, I is the set formed of three Boolean inputs, $v(I)$ is the set of all Boolean triples, O is the singleton set of one Boolean output, $v(O) = \{\texttt{true}, \texttt{false}\}$, $\delta(s_0, (i_1, i_2, i_3)) = s_0$, and $\lambda(s_0, (i_1, i_2, i_3)) = i_1 \wedge i_2 \wedge i_3$. Often, state machines are represented graphically, but since we model basic block types by writing their code, such a representation is not needed in this paper. Another example of a basic block type is the unit delay, whose NuSMV and Promela code has been shown in Sections II-B and II-C.

In MODCHK, basic blocks are specified by using a tool dialog to define the block interface, and then a text editor to write the NuSMV code. Manual modeling is a necessity, since many vendors (especially in the nuclear domain) use non-standard, proprietary languages [8]. Block graphics are specified using Scalable Vector Graphics (SVG).

B. Block networks

A *basic block instance* is a pair of a basic block type and a unique name. A *block network* is a quadruple (B, I, O, C), where B is the set of basic block instances, I is the set of inputs, O is the set of outputs, and C is the set of *connections*. Each connection can join either an output of one basic block instance with an input of another basic block instance, an output of a basic block instance with an output of the block network, an input of the block network with an input of a basic block instance, or an input of a basic block instance with a constant.

Such a definition allows viewing the entire block network as a single Mealy machine. During a single execution cycle of such a machine, values propagate along the connections, allowing all basic block instances to execute. A problem, however, arises in cases of cyclic dependencies between basic blocks. In a broader context, this issue is discussed in [23]. The problem can be addressed by using unit delays to break such cycles and specifying the processing order explicitly.

Block networks are drawn manually in MODCHK, mimicking the structure of the original function block based control application. Basic block instances are added to the networks in a drag-and-drop fashion, and connected by drawing wires. Specified function blocks are used to set up model inputs, outputs, and other monitors. If necessary, the modeling procedure also involves:

1) replacing continuous values with integers;
2) limiting possible values that integer inputs can have;
3) setting delay length parameters to sufficiently but not overly large values;
4) breaking any existing feedback loops with a unit delay block;

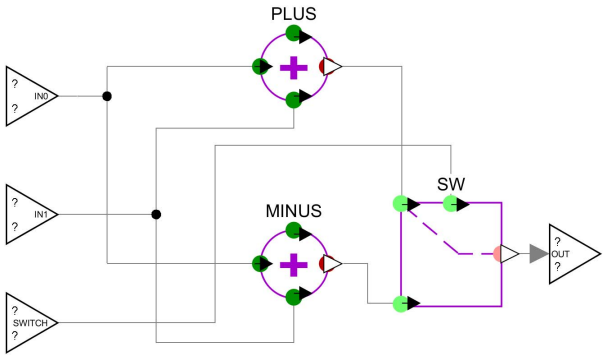

Fig. 1. MODCHK block network PlusMinus, which computes either the sum or the difference of inputs IN0 and IN1 depending on the value of the Boolean input SWITCH. The basic block instance MINUS has the same type as PLUS, but is configured to multiply the lower argument by -1 before adding.

5) specifying abstract representations for (or just omitting) complex function blocks (e.g. PID controllers).

Fig. 1 shows an example (hereinafter called the PlusMinus example) of a simple block network drawn in MODCHK, which will be further used for illustration purposes. Basic block types used in this model correspond to the ones of the Apros[1] continuous process simulator.

C. Translation to NuSMV

The tool MODCHK is able to generate NuSMV code for specified block networks automatically [18]. Below, we exemplify this translation on the PlusMinus block network. This code assumes that basic block types such as `BINARY_SWITCH` and `ADDER_3` are defined in the same NuSMV model. In the `VAR` section of the code, basic block instances of these types are declared. In addition to usual inputs and outputs, the code specifies processing of signal validity (which is not used in this model, but is very relevant in many nuclear applications for dealing with, e.g., measurement failure, or loss of communication between subsystems [18]) and the information about the presence of input connections.

```
MODULE PlusMinus(IN0, IN0_FAULT, IN0_CONNECTED, IN1,
  IN1_FAULT, IN1_CONNECTED, SWITCH, SWITCH_FAULT,
  SWITCH_CONNECTED)
VAR
  SW : BINARY_SWITCH(SWITCH, SWITCH_FAULT, TRUE,
    PLUS.ADDER_OUTPUT_SIGN,
    PLUS.ADDER_OUTPUT_SIGN_FAULT, TRUE,
    MINUS.ADDER_OUTPUT_SIGN,
    MINUS.ADDER_OUTPUT_SIGN_FAULT, TRUE, FALSE);
  PLUS : ADDER_3(0, FALSE, FALSE, IN0, IN0_FAULT,
    TRUE, IN1, IN1_FAULT, TRUE, 1, 1, 1);
  MINUS : ADDER_3(0, FALSE, FALSE, IN0, IN0_FAULT,
    TRUE, IN1, IN1_FAULT, TRUE, 1, 1, -1);
DEFINE
  OUT := SW.BSWITCH_OUTP_SIGN;
  OUT_FAULT := SW.BSWITCH_OUTP_SIGN_FAULT;
```

Another code fragment below shows how to specify input ranges for the network to form an open-loop model. Later, we will also consider closed-loop models, where the module `main` will also instantiate the plant module.

```
MODULE main()
VAR
  IN0 : -1000..1000;
  IN1 : -1000..1000;
  SWITCH : boolean;
  pm : PlusMinus(IN0, FALSE, TRUE, IN1, FALSE, TRUE,
    SWITCH, FALSE, TRUE);
DEFINE
  OUT := pm.OUT;
```

Finally, LTL requirements for the block network can be specified:

```
LTLSPEC G(!SWITCH -> (OUT = IN0 + IN1))
LTLSPEC G(SWITCH -> (OUT = IN0 - IN1))
```

D. Translation to Promela

The translation of block networks to Promela which is described in this section is novel and has not been reported previously, unlike the one to NuSMV. It has been implemented as a Java application[2] which accepts basic block types specified in Promela and the block network specified in NuSMV, and produces the code of the block network in Promela. Thus, except the aforementioned need to provide basic blocks in Promela (in our case, since the blocks were originally developed in NuSMV, we translated them manually), and the lack of automatic translation of temporal properties from NuSMV to Promela (which would not present any scientific novelty), the translation procedure is automatic.

Three translation problems have been identified prior to implementing the translation:

1) Order of basic block execution: in NuSMV, execution of different basic block instances is inseparable since all instances make a step synchronously. Thus, dependencies between blocks are resolved implicitly. In SPIN, however, the execution order needs to be specified explicitly.
2) SPIN does not support function calls, which would have been useful to execute basic blocks.
3) The exact way of specifying the model of the block network influences the results of verification. For example, intermediate states during the execution of the network and its basic block instances often must not be visible as parts of infinite behaviors during LTL model checking. Moreover, resulting Promela models must be equivalent to respective NuSMV ones to make the comparison between SPIN and NuSMV valid.

The first problem has been resolved by analyzing dependencies between basic blocks and organizing their execution in the order of topological sort, whose implementation is linear of the number of block instances. However, dependencies may be circular due to possible presence of feedback loops in the network. In Section III-B, we have made an assumption that

[1] http://www.apros.fi/en/

[2] https://github.com/igor-buzhinsky/modchk-to-spin

all such loops are explicitly broken by a unit delay block. Although formally there is a circular dependency between block instances even when such blocks are used, these dependencies can be resolved by executing unit delay blocks before any other blocks, and then executing the rest of the blocks in the topological order.

A known solution to resolve the second problem is to use process types for each function (which, in our case, would contain the code specifying the basic block). For example, such a solution has been used in [13], and a possible implementation of this solution for the unit delay block has been given in Section II-B. However, for our problem, in which basic blocks do not contain any function calls, a better solution exists: for each basic block instance, we inline its code into the place where it must be executed (variable names used in the definition of the block type are substituted with unique names for each inlined block instance). This solution has the benefit of being able to wrap the whole block network code into a single `d_step` sequence – that is, make SPIN treat it as an indivisible statement, which is computationally efficient.

The third problem, which relates to the way the Promela model is organized in general, is solved by applying the following pattern. Since no processes are needed to execute basic block instances, the whole system executes within the single default process `init`. In this process, model execution is organized as an infinite loop where first inputs to the block network are selected (in the closed-loop case, this involves executing the plant model, which receives the previous outputs of the controller), and then the block network executes. The body of the loop is marked as `atomic` to make its intermediate steps invisible in model checking. The overview of the described pattern is provided below:

```
// <Variable declarations>
init { do :: atomic {
  // <Nondeterministic controller input selection>
  d_step {
    // <Block network execution>
  }
} od }
```

Promela models following this pattern are equivalent to NuSMV ones in the following sense: the sets of possible executions of Promela and NuSMV models (represented as sequences of controller inputs and outputs) are equal, provided that the first dummy behavior element (where none of the variables are initialized) is removed from Promela behaviors. The reasons why this happens are that only steps between each cycle of plant and controller execution are included into behaviors (like in NuSMV), and that basic block executions are scheduled in the order of dependencies between them (this makes each cycle execute like in NuSMV). This equivalence holds under the assumption that all Promela and NuSMV basic blocks are equivalent in a similar sense.

An example of applying the described translation procedure to the PlusMinus example in the open-loop case is shown below. The original generated code was shortened: less relevant parts have been simplified, and some declarations have been omitted (shown as "...").

```
// Basic block instance output declarations:
int OUTPUT_PLUS_ADDER_OUTPUT_SIGN;
int OUTPUT_MINUS_ADDER_OUTPUT_SIGN;
int OUTPUT_SW_BSWITCH_OUTP_SIGN;
// ...
// Controller input declarations:
int INPUT_IN0;
int INPUT_IN1;
int INPUT_SWITCH;
// ...
init { do :: atomic {
  // Controller input selection:
  select(INPUT_IN0 : -1000..1000);
  select(INPUT_IN1 : -1000..1000);
  select(INPUT_SWITCH : 0..1);
  // Controller execution:
  d_step {
    // Executing PLUS:
    #define INPUT_PLUS_ADDER_INPUT_SIGN1 0
    // ...
    OUTPUT_PLUS_ADDER_OUTPUT_SIGN =
      (INPUT_PLUS_ADDER_INPUT_SIGN1 *
      INPUT_PLUS_ADDER_COEFFICIENT1) +
      (INPUT_PLUS_ADDER_INPUT_SIGN2 *
      INPUT_PLUS_ADDER_COEFFICIENT2) +
      (INPUT_PLUS_ADDER_INPUT_SIGN3 *
      INPUT_PLUS_ADDER_COEFFICIENT3);
    // Executing MINUS:
    #define INPUT_MINUS_ADDER_INPUT_SIGN1 0
    // ...
    OUTPUT_MINUS_ADDER_OUTPUT_SIGN =
      (INPUT_MINUS_ADDER_INPUT_SIGN1 *
      INPUT_MINUS_ADDER_COEFFICIENT1) +
      (INPUT_MINUS_ADDER_INPUT_SIGN2 *
      INPUT_MINUS_ADDER_COEFFICIENT2) +
      (INPUT_MINUS_ADDER_INPUT_SIGN3 *
      INPUT_MINUS_ADDER_COEFFICIENT3);
    // Executing SW:
    #define INPUT_SW_BSWITCH_CONTROL_S INPUT_SWITCH
    // ...
    OUTPUT_SW_BSWITCH_OUTP_SIGN =
      (INPUT_SW_BSWITCH_CONTROL_S ->
      INPUT_SW_BSWITCH_INP_SIGN_2 :
      INPUT_SW_BSWITCH_INP_SIGN_1);
  }
} od }
#define OUT OUTPUT_SW_BSWITCH_OUTP_SIGN
ltl p_plus { [] (INPUT_SWITCH ||
  (OUT == INPUT_IN0 + INPUT_IN1))) }
ltl p_minus { [] (!INPUT_SWITCH ||
  (OUT == INPUT_IN0 - INPUT_IN1))) }
```

IV. Comparison on a case study

The described techniques of modeling block networks in NuSMV and SPIN have been applied on a case study, which is based on an Apros model of an NPP with a pressurized water reactor (PWR), hereinafter referred to as the generic PWR model. This model has been provided by Fortum Power and Heat Oy,[3] a power utility with NPP operation license in Finland, and incorporates main NPP process components and corresponding automation devices.

Due to the large size of the generic PWR model, we considered only eight control subsystems, each represented by an Apros automation diagram. To avoid disclosure, the

[3]http://www.fortum.com/

names of these subsystems were masked and, from now on, are referred to as S1, ..., S8. These subsystems are responsible for protection functions (S1–S4), reactor power and turbine trip (i.e. shutdown) control (S5), pressurizer control (S6, S7) and feed water tank water level control (S8). Executing the Promela model generation technique for all control subsystems took only two seconds in total.

A. Preparation of formal models

All the considered I&C subsystems have been modeled in the MODCHK tool, and then converted to NuSMV and Promela. To allow closed-loop model checking, plant models were generated automatically. Manual plant modeling was not attempted due to the complexity of the case study, and automatic translation of the Apros model to a formal one was impossible due to the lack of tool to perform it.

As an algorithm of plant model construction, the procedure from [9] was used. To apply it, we prepared a set of behavior traces collected based on simulations in Apros. This set included 3000 traces, each capturing the behavior of the generic PWR model along a four-minute interval with a sampling rate of one second (thus, each trace contained 240 elements). Although the work [9] is devoted to plant model synthesis from behavior traces and LTL properties using satisfiability solvers, it also describes a simplified and more scalable procedure which only works when behavior traces is the only type of input data for plant model construction. The latter procedure was applied.

Table I shows the complexity of obtained formal subsystem models in terms of inputs, outputs, and the number of internal basic block instances. It also shows the number of states in generated plant models. This number varies greatly among subsystems, corresponding to various degrees of abstraction, which are primarily connected with different numbers of controller inputs rather than subsystem complexity.

Plant models were generated as explicitly represented nondeterministic Moore machines, which means that their outputs (that is, inputs for the controller) depend only on the state. Each continuous output actually represented a range of values. Considering such ranges (i.e. allowing the actual value to be selected nondeterministically within the range) in SPIN models would have led to dramatic state space expansion – thus, only the middle value (rounded to an integer) was used for each interval. In NuSMV, state space expansion itself is not a major problem, so the mentioned extra nondeterminism was considered in addition to the case of fixed output values. In the latter case, the NuSMV model was equivalent to the Promela one and thus model checking was expected to yield identical results.

B. Temporal requirements

Temporal requirements for subsystems were formulated in LTL. All of them were also possible to be equivalently formulated in CTL, which was done since NuSMV often checks CTL requirements faster than equivalent LTL ones (assuming that BDD-based model checking is used). The requirements were elicited based on the documentation of the generic PWR model and its Apros implementation. The most common requirement types specified:

1) an eventual or one step delayed response of the block network to a satisfaction of a certain condition on inputs;
2) the lack of such response when the condition was not satisfied.

The numbers of temporal requirements elicited for each of the system are reported in Table II.

C. Experiments

All experiments were performed on the Intel Core i7-4510U CPU with the clock rate of 2GHz. The used versions of NuSMV and SPIN were 2.6.0 and 6.4.5 respectively. For each subsystem, six model checking runs were performed in total. They included verification in SPIN with fixed controller input values and verification in NuSMV with both fixed input values and input value ranges (see Section IV-A). The experiments were done both in the open-loop and in the closed-loop case.

In NuSMV, we considered only BDD-based model checking: BMC was not applied since it is parameterized by a bound which influences both verification complexity and results and hence would complicate the comparison of NuSMV with SPIN. CTL model checking in NuSMV was started right after running the tool (optimization flags "-df", "-coi" and "-dynamic" were applied). The situation was different in SPIN, which first generated and then compiled the C code for the Promela model (optimization level O2 was used). Time limit for model checking was set to ten minutes per temporal requirement in both NuSMV and SPIN.

D. Results

While analyzing the results, we decided to focus on model checker performance rather than the meaning of the results in terms of the generic PWR model. This focus is connected with our primary goal – to investigate the efficiency of SPIN in nuclear I&C verification. The results of experiments in terms of total model checking time are shown in Table II. The following conclusions can be made by observing the data:

1) Open-loop model checking in SPIN often fails to terminate within the time limit. This is not surprising, provided that the state space of the model is at least as large as the number of possible input combinations, which is exponential of the number of inputs (see Table I).
2) In contrast, open-loop model checking in NuSMV is fast, which mostly complies with our previous experience. It appears, however, that input ranges are more difficult for NuSMV to process. The results of verification with ranges instead of fixed inputs are more reliable since the block network is checked on a larger set of possible inputs.
3) The same difference between fixed inputs and input ranges holds for closed-loop verification in NuSMV.
4) Closed-loop verification time in SPIN is always lower than the one in NuSMV, and in some cases the ratio

TABLE I
PARAMETERS OF THE SUBSYSTEMS OF THE GENERIC PWR MODEL

Subsystem of the generic PWR model		S1	S2	S3	S4	S5	S6	S7	S8
Inputs	Real-valued	4	6	1	7	11	13	2	2
	Boolean	0	2	24	2	6	0	0	0
Outputs	Real-valued	3	0	0	1	1	3	9	6
	Boolean	0	12	24	4	2	0	0	0
Basic block instances		49	41	38	22	29	27	16	18
States in the plant model		14	1355	1206	192	4906	1904	20	100

TABLE II
MODEL CHECKING TIME (IN SECONDS). TIME LIMIT VIOLATION IS INDICATED AS "TL"

Subsystem of the generic PWR model		S1	S2	S3	S4	S5	S6	S7	S8
Number of temporal requirements		9	24	26	15	10	18	11	8
Open-loop time	SPIN, fixed inputs	54	TL	TL	TL	TL	TL	3	8
	NuSMV, fixed inputs	5	1	11	11	1	21	1	2
	NuSMV, input ranges	106	48	31	369	2	TL	2	3
Closed-loop time	SPIN, fixed inputs	3	44	277	98	256	148	3	3
	NuSMV, fixed inputs	2611	137	769	TL	718	1104	268	8
	NuSMV, input ranges	1773	TL	2069	TL	1428	TL	347	12

of these times approaches or even exceeds 100. Such a result can be explained using the arguments given in Section II-D while describing the results of work [11]. The difference of our case is that, due to plant models encoded as explicit state machines, the length of this encoding is roughly proportional to the number of states, resulting in plant models significantly larger than controller ones in half of the cases. This might have decelerated NuSMV even more than in [11].

5) SPIN model checking time is higher for subsystems with larger plant models. This can be explained by the larger state space which needs to be explored.
6) The corresponding dependency for NuSMV cannot be established: for example, high verification times were observed for S1 and S4, which are relatively small in terms of the number of states in the plant model. On one hand, such a dependency was anticipated since, in the case of generated explicit-state plant models, a higher number of states means longer NuSMV model representation, which, in turn, means larger BDDs for NuSMV to process. On the other hand, performance of symbolic model checking is highly unpredictable [15]. It may depend not only on the length of the textual description of the model, but also on dependencies between variables. A larger case study and a more thorough exploration may be needed to understand on what aspects of formal models symbolic model checking performance depends.

The following additional notes need to be made concerning the results:

1) Model preparation time in SPIN may comprise a significant share of total verification time in the case of closed-loop verification with a large plant model (i.e. a thousand of states or more). The most demonstrative cases are S2 (1355 states in the plant model, the preparation phase lasted 88% of the total verification time) and S5 (4906 states, 92%).
2) Verification results in SPIN and NuSMV matched each other in the case of fixed inputs. An opposite result would have meant that the proposed translation of block diagrams to Promela does not produce models equivalent to NuSMV models generated by MODCHK.

V. RELATED WORK

In [13], explicit-state and symbolic model checking were compared on the problem of verifying distributed disc controller software. The model of the software was parameterized with the used number of processes. Explicit-state and symbolic model checking was done in SPIN and RuleBase (a model checker originating from SMV, the predecessor of NuSMV), respectively. The verified software contained function calls involving non-tail recursion, and thus function execution had to be emulated using processes. Applying SPIN led to a state space explosion, and thus it was able to analyze only a tiny fraction of the state space. On the other hand, RuleBase failed to verify the software model with more than two processes, while SPIN was able to handle a larger configuration. In a more recent study [14], NuSMV was compared with SPIN on the problem of commercial flight guidance systems verification. The conclusions of [14] comply with the ones of [13]: symbolic model checking is able to properly solve simpler verification problems, but SPIN is more scalable and is able to partially solve harder problems (although its verification results were unsound due to the use of bit-state hashing).

The main difference of our study is the consideration of closed-loop verification in additional to the more conventional

open-loop verification. In the case of closed-loop verification in NuSMV, the increase of complexity is connected not with a larger number of processes, like in [13], [14], but with a more complex model of a single process. Then, our verification problem comes from a different field of industry.

Both closed-loop and open-loop verification have been previously compared in [11]. The comparison was performed only using NuSMV. Thus, our study extends the results of [11] by also considering an explicit-state model checker.

VI. Conclusions and future work

In this paper, we have performed a comparison of explicit-state and symbolic model checking techniques on the problem of nuclear I&C application verification. In the open-loop scenario, symbolic model checking with NuSMV has been found clearly superior. The more interesting conclusion is related to the closed-loop case: if the model of the plant is represented with an explicitly specified state machine, the use of explicit-state verification performed by SPIN has been found to be beneficial despite that symbolic model checking is believed to be more efficient in practical cases. Thus, similarly to open-loop and closed-loop verification being complementary to each other, there may also be no "best" model checking tool.

To enable the comparison, we have developed an efficient technique to generate block diagram models in the SPIN language. Although we applied it to nuclear I&C systems only, block diagrams are commonly used to represent automation systems in general. For example, a somewhat more complex model of function block interaction is considered in the IEC 61499 standard [24].

The results of the performed comparison are especially valuable in the case of closed-loop verification with generated plant models [9]: making verification faster may enable it for larger, more detailed and reliable plant models. On the other hand, in this study we have not considered manually prepared plant models, which are not commonly expressed as large explicit state machines. This may lead to different results, and hence the scope of our conclusions is limited.

Performing a case study with manually created plant models may be considered in future work. Different case studies and a more thorough experiment design may also help understand in more detail what slows symbolic model checking in the closed-loop case. Another direction to improve the case study is to check a wider range of temporal properties, including the ones which require the plant model to be present.

Acknowledgments

This work was financially supported by the SAUNA project (funded by the Finnish Nuclear Waste Management Fund VYR as a part of research program SAFIR2018) 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] J. Yoo, S. Cha, and E. Jee, "A verification framework for FBD based software in nuclear power plants," in *15th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2008, pp. 385–392.

[3] 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.

[4] A. Pakonen, T. Tahvonen, M. Hartikainen, and M. Pihlako, "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)*. American Nuclear Society, 2017, pp. 1342–1352.

[5] A. Pakonen, C. Pang, I. Buzhinsky, and V. Vyatkin, "User-friendly formal specification languages – conclusions drawn from industrial experience on model checking," in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. IEEE, 2016, pp. 1–8.

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

[7] 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.

[8] 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)*. IEEE, 2013, pp. 1–6.

[9] 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.

[10] 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.

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

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

[13] C. Eisner and D. Peled, "Comparing symbolic and explicit model checking of a software system," in *International SPIN Workshop on Model Checking of Software*. Springer, 2002, pp. 230–239.

[14] Y. Choi, "From NuSMV to SPIN: Experiences with model checking flight guidance systems," *Formal Methods in System Design*, vol. 30, no. 3, pp. 199–216, 2007.

[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics*. Springer, 2001, pp. 176–194.

[16] G. Frey and L. Litz, "Formal methods in PLC programming," in *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, vol. 4. IEEE, 2000, pp. 2431–2436.

[17] B. F. Adiego, D. Darvas, E. B. Vinuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[18] 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)*. Taylor & Francis Group, London, UK, 2017, pp. 3189–3196.

[19] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977, pp. 46–57.

[20] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[22] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193–207, 1999.

[23] S. Tripakis, "Compositionality in the science of system design," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 960–972, 2016.

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