Buzhinsky, Igor; Pakonen, Antti

Model-Checking Detailed Fault-Tolerant Nuclear Power Plant Safety Functions

Published in:
IEEE Access

DOI:
10.1109/ACCESS.2019.2951938

Published: 01/01/2019

Document Version
Publisher's PDF, also known as Version of record

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 17, 2019, accepted October 18, 2019, date of publication November 6, 2019,
date of current version November 18, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2951938

Model-Checking Detailed Fault-Tolerant Nuclear Power Plant Safety Functions

IGOR BUZHINSKY[1,2] **AND ANTTI PAKONEN**[3]

[1]Department of Electrical Engineering and Automation, Aalto University, 02150 Espoo, Finland
[2]Computer Technologies Laboratory, ITMO University, 197101 St. Petersburg, Russia
[3]VTT Technical Research Centre of Finland, 02044 Espoo, Finland

Corresponding author: Igor Buzhinsky (igor.buzhinskii@aalto.fi)

This work was supported in part by the Finnish Research Programme on Nuclear Power Plant Safety 2018-2022 (SAFIR 2022), in part by the Russian Ministry of Science and Higher Education by the State Task 2.8866.2017/8.9, and in part by the Government of the Russian Federation under Grant 08-08.

ABSTRACT Model checking has been successfully used for detailed formal verification of instrumentation and control (I&C) systems, as long as the focus has been on the application logic alone. In safety-critical applications, fault tolerance is also an important aspect, but introducing I&C hardware failure modes to the formal models comes at a significant computational cost. Previous attempts have led to state space explosion and prohibitively long processing times. In this paper, we present an approach to model and formally verify protection functions allocated to one or several I&C systems, accounting for hardware component failures and delays in communication within and between the systems. Formal verification is done with model checking, whose feasibility on such complex systems is achieved by utilizing the symmetry of I&C systems: the components of the overall model that do not influence the checked requirements are eliminated, and the failing components are fixed. Generation of such abstracted models, as well as subsequent verification of their requirements and symmetry with the NuSMV symbolic model checker, is handled by a software tool. In addition, we explore how to specify formal requirements for systems of the considered class. Based on a case study built around a semi-fictitious nuclear power plant protection system that achieves reliability by means of redundancy, we demonstrate how failure tolerance of even detailed system designs can be formally verified.

INDEX TERMS Formal verification, model checking, nuclear I&C systems, fault tolerance.

I. INTRODUCTION

Safety-critical systems, such as transport, spacecrafts and power plants, demand comprehensive efforts to ensure their reliability. Traditional verification and validation (V&V) approaches, such as testing and simulation, may not be sufficient due to their inability to cover all possible behavior scenarios. In contrast, *formal methods* such as static analysis [1], automated theorem proving [2], [3] and model checking [4], [5] are exhaustive. In model checking, which is largely used in this paper due to its focus on state space processing, this analysis is performed for all possible state sequences of the formal model of the system under verification. In Finland, model checking is used to verify the design of instrumentation and control (I&C) systems of nuclear power plants (NPPs) [6], [7].

Single failure tolerance is often a requirement for safety-critical I&C systems—the failure of any individual component shall not prevent the system from performing its function. Tolerance to single failures can be achieved with redundancy. Redundant subsystems, together capable of performing the desired tasks even if one of them fails, are used in application domains such as nuclear energy [8], aviation [9], [10], aerospace [11], [12], railway [13], and automotive [14] industries.

Verifying that the failure tolerance mechanisms actually work calls for the modeling of both the application logic and the different failure modes of the underlying hardware. Furthermore, distributing the logic between redundant, separated systems introduces communication delays and

The associate editor coordinating the review of this manuscript and approving it for publication was Junjian Qi.

asynchrony, which should also be addressed in the analyses. Previous attempts (e.g., [15], [16]) at using model checking to evaluate these aspects in one model have run into scalability issues, while logic-only models of similar complexity are not an issue for model checkers.

In this paper, we present a method for verifying the fault tolerance of I&C systems based on model checking. In addition to modeling the application logic, we account for the failure modes of the underlying I&C system hardware components, and the communication delay between distributed computers. As the case example, we use the reactor protection system of the proposed U.S. version of the European Pressurized Water Reactor (EPR) NPP [8]. We simplify the failure model by focusing strictly on the verification of single failure, as required by the Finnish regulatory guides on nuclear safety,[1] and demonstrate our approach using the symbolic model checker NuSMV. As verification of previously intractable formal requirements becomes possible, our results enhance the state-of-the-art of nuclear I&C verification. They may also be insightful in other aforementioned domains where fault tolerance is achieved with redundant architectures.

This paper is an extended version of the work [19]. Compared to [19], we supplement our verification approach with the ability to handle multiple interconnected I&C systems and evaluate it on a more complex case study—composed of two four-redundant systems, and a third two-redundant system, performing two different safety functions for reactor shutdown. In addition, we study different ways of expressing temporal requirements to be verified for such systems, and also perform formal checks of symmetry of function units with respect to their input variables, which is used as the basis for failure model simplification.

The rest of the paper is structured as follows. Section II introduces a running example and overviews used concepts and notations. Section III describes the proposed approach of hardware and software modeling and model checking. In Section IV, the proposed approach is evaluated on a case study. Finally, Section V analyzes related work and Section VI concludes the paper.

II. PRELIMINARIES

A. RUNNING EXAMPLE: REACTOR PROTECTION SYSTEM (PS)

To illustrate the concepts introduced in this section, we start with the fault-tolerant *reactor protection system* (PS) case study, which will be used as a running example. The PS is a safety I&C system responsible for reactor *trip* (i.e., shutdown). In Section IV, this case study will be extended by adding more systems and safety functions.

The PS with which we work in this paper is inspired by the design of the U.S. EPR NPP [8],[2] but partly based on our own invention. Based on Areva NP's TELEPERM XS technology, the PS is organized into four redundant, independent *divisions*, located in separate buildings [18] (see Fig. 1).

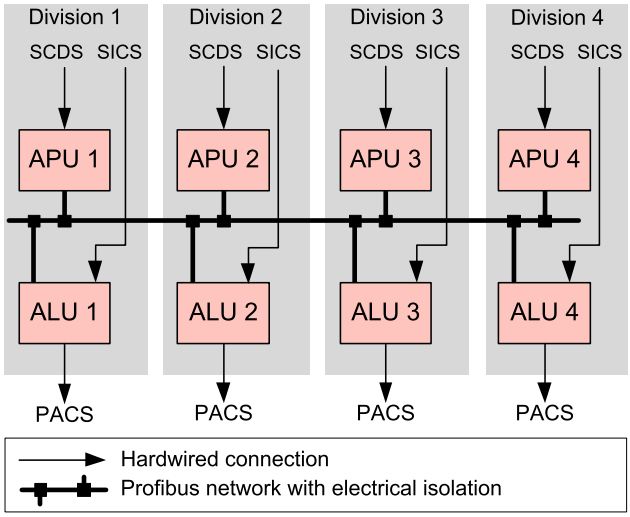

FIGURE 1. Architecture of the PS.

Two function units are located in each division [18]:

1) *Acquisition and Processing Units* (APUs) acquire signals from the process sensors and monitoring systems via the Signal Conditioning and Distribution System (SCDS, not modeled in this paper) using a hardwired connection, perform calculations and setpoint comparisons, and distribute the results to the ALUs for voting.
2) *Actuation Logic Units* (ALUs) perform voting over processing results and issue actuating results, taking into account operator control actions from the Safety Information and Control System (SICS, not modeled in this paper) user interface. The actuation orders are sent to the Priority and Actuator Control System (PACS, will be introduced in Section IV-B) via a hardwired connection.

Thus, all the divisions are redundant and have an identical structure. The communication between the divisions is relayed via a Profibus network, using fiber-optic cabling to achieve electrical isolation. [8] The safety function processing logic for the APUs and ALUs is shown in Fig. 2.

Within the PS, tolerance against a single failure (of an input sensor, a unit of the PS, or an actuator device) is

[1] To be exact, in Finland, the reactor protection system needs to fulfill the "N+2" requirement: in addition to single failure, the system still needs to perform its function even if any component of a redundant system is simultaneously out of operation for maintenance [17]. We assume that a "maintenance bypass" logic is built into the I&C system [18], meaning that requirements dealing with maintenance can be formulated and checked without the need to add more elements to the model, and a device that is placed out of operation will only issue actuation orders if it fails.

[2] No project is underway to construct an EPR in the U.S., but the U.S. Nuclear Regulator NRC has published parts of the plant supplier Areva NP's 2013 Final Safety Analysis Report (FSAR) for a suggested U.S. variant.

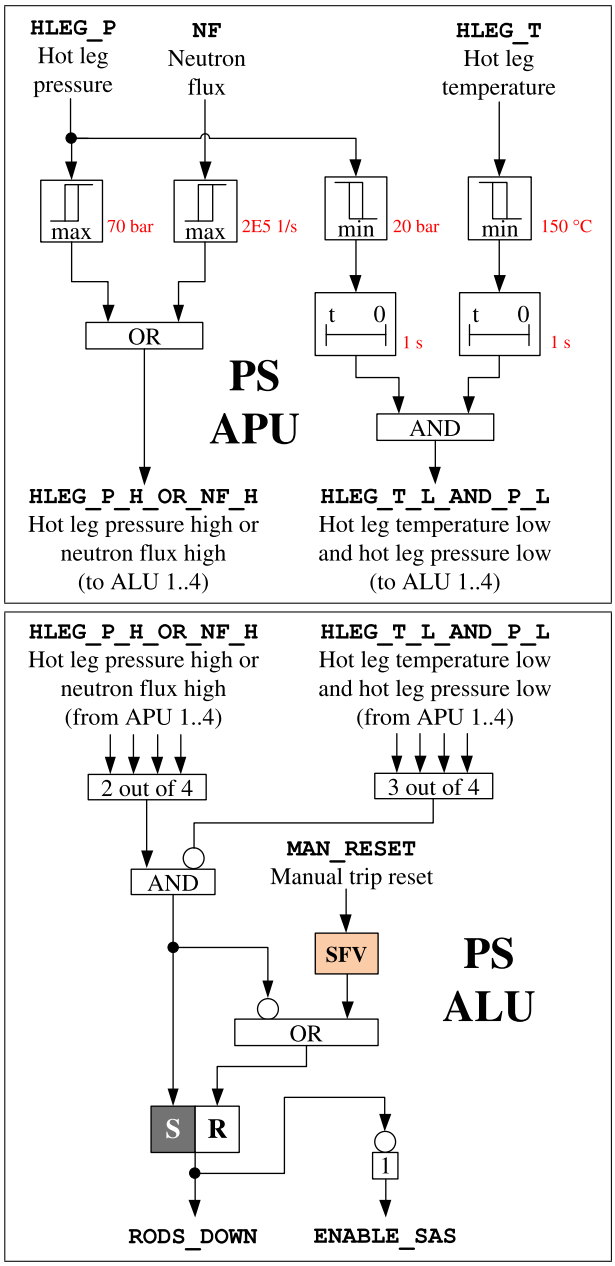

FIGURE 2. The processing logic for each of the four redundant PS APUs (top) and ALUs (bottom). Function block notations: a circle is a negation, `min` and `max` are "<" and ">" comparisons respectively, `t..0` blocks are on-delay (TON) timers, S R blocks are one-bit memory elements similar to flip-flops, SFV are faulty value substitution blocks, 1 is the identity function.

based on the four-division structure, and selection and voting functions in the APUs and ALUs. Each signal in the PS logic, in addition to its value, has a *status*, which can be set to "fault" by failures detected by input modules or function processors. The status is then used to exclude invalid signals in selection (e.g., second-maximum,[3] second-minimum) and voting (n-out-of-m) blocks. Even if a single failure disables an ALU logic performing the vote, the redundant ALUs can still actuate the reactor trip function. [8]

The PS case study approaches the complexity of detailed design, implementing signal status processing (partly based on publicly available information, partly on invention) within elementary blocks, and also in the shape of special blocks for fault status filtering. For the ALU logic, we added a block type of our own invention (SFV) to substitute the value of faulty control signals from the operator's interface with a default "false".

To avoid confusion, we note that the PS described above has the same structure as the one from [19], and its complexity is comparable, but its inputs, outputs and implementation (APUs and ALUs) are different as the PS used here is responsible for reactor trip rather than emergency core cooling.

B. MODEL CHECKING

Model checking [4], [5] is a verification technique that analyzes the system under verification by exploring its set of possible states. More precisely, it works with its *formal model*—the mathematical model of the system's behavior specified as states and state transitions. For this analysis, statements on the sequences of visited states, called *temporal properties* (also referred to as *temporal requirements*), may be specified for checking. They allow the analyst to formulate the requirements for the behavior of the system in time. For example, the *linear temporal logic* (LTL) is a language to specify a requirement to be satisfied for all reachable behaviors of the model. This is a discrete-time temporal logic, meaning that time is modeled as a sequence of discrete *steps*, or *cycles*, of the model. LTL extends the Boolean propositional logic with *temporal operators*, such as **X** (in the next state), **G** (now and always in the future), and **F** (now or at some point in the future). An extended version of LTL has past time operators, such as **O** (now or at some point in the past).

For example, suppose that the logic of the APU from Fig. 2 is represented with a formal model, where `HLEG_T` is an integer variable specifying the hot leg temperature, and `HLT_long` is the corresponding Boolean signal arriving to the `AND` block. Due to `min` threshold blocks and on-delay timers (`t..0` blocks), the following LTL property will be satisfied:

$$\mathbf{G}((\texttt{HLEG\_T}<150) \wedge (\mathbf{X}(\texttt{HLEG\_T}<150)) \rightarrow \mathbf{X}\,\texttt{HLT\_long}),$$

meaning that any occurrence of the condition $\texttt{HLEG\_T} < 150$ lasting for two consequent cycles will lead to `HLT_long` being true on the second of these cycles.

In *computation tree logic* (CTL), temporal operators are prefixed with *path quantifiers* **A** ("for all") and **E** ("exists"). This permits properties expressing reachability. For example, CTL property **AG EF** `HLT_long` specifies that in any reachable state of the model the aforementioned signal `HLT_long`

[3] A block selecting the remaining maximum number in a set, after excluding the maximum number.

can be true in the future. Finally, *property specification language* (PSL) is a notation with syntax inspired by regular expressions and LTL semantics.

C. MODEL CHECKERS AND NUSMV

A *model checker* is a tool to perform model checking automatically. In addition to verification outcomes (true/false), if requested, model checkers generate *counterexamples*, or error traces, for violated temporal properties.

NuSMV [20] is a model checker with its own textual language to specify formal models and temporal properties. NuSMV implements *symbolic model checking* [21], whose idea is to process the state space of the formal model implicitly, mitigating the *state space explosion* problem of explicit model checking algorithms: the state space of the model may grow too quickly with the growth of the model's textual representation. Conventional symbolic model checking is done by working with *binary decision diagrams* (BDDs). For LTL and PSL, *bounded model checking* (BMC) is also available, which is an imprecise yet often faster technique that only considers counterexamples whose length is bounded by $k + 1$ cycles, where $k \geq 0$ is chosen by the user.

A NuSMV model is a nested arrangement of synchronously executed *modules* that roots from the `main` module. An example of a NuSMV module `TON` that represents an on-delay timer within the APU logic in Fig. 2 (`t..0` blocks) is given below:[4]

```
1  MODULE TON(BIN_IN)
2  VAR
3      ticks: 0..2;
4  ASSIGN
5      init(ticks) := BIN_IN ? 1 : 0;
6      next(ticks) := case
7          !next(BIN_IN): 0;
8          ticks = 2: 2;
9          TRUE: ticks + 1;
10     esac;
11 DEFINE
12     BIN_OUT := ticks = 2;
```

The state of this module is defined by the value of a single bounded integer variable `ticks` defined on line 3. Then, on lines 5–10, the evolution of its value is described: it increases by 1 (but never exceeds 2) if the `BIN_IN` input is true and resets otherwise. The output of the timer is defined on line 12. Note that the actual modules with which I&C systems are modeled in this paper also operate with fault statuses in addition to regular input/output data (Section II-G); they are omitted here for simplicity.

[4]In this paper, a single cycle does not correspond to a predetermined physical duration. This is done for computational efficiency and at the same time is a limitation of the approach. For example, from the given implementation of `TON` it may appear that a cycle corresponds to one second, but blocks with larger delays may have lower execution frequencies, while in reality all blocks execute with much higher frequency.

D. MODEL CHECKING OF NUCLEAR I&C SYSTEMS

Model checking I&C application logics based on function block diagrams has been an active research topic for several years, with different areas of application [22]. It has been proven applicable for ensuring the correctness of industry-sized PLC programs [7], [23], but, nevertheless, is not yet a wide-spread industry practice in any application area.

I&C logics can be verified using either *open-loop* or *closed-loop* modeling. Open-loop model checking only considers the model of the "controller" logic, while in the closed-loop model feedback from the controlled plant is taken into account [24]. Closing the loop can help limit the state space of the model [24], but analysis times can actually increase when symbolic model checking is used [25], [26]. What is more, generating a realistic plant model can be a challenge, and limiting model behavior might accidentally filter out scenarios relevant to safety.

Since 2008, VTT has successfully used model checking to verify both early (functional) and detailed design of safety I&C systems for Finnish NPPs [7]. A graphical tool called MODCHK [27], [28] is used to manually define a collection of vendor-specific elementary function blocks, model the function block diagrams with a graphical editor, specify the properties with a text editor, generate the necessary input files for NuSMV, and visualize counterexamples produced by NuSMV with an animated view of the function block diagram. A screenshot of MODCHK with the diagram of PS ALU (Fig. 2) is shown in Fig. 3.

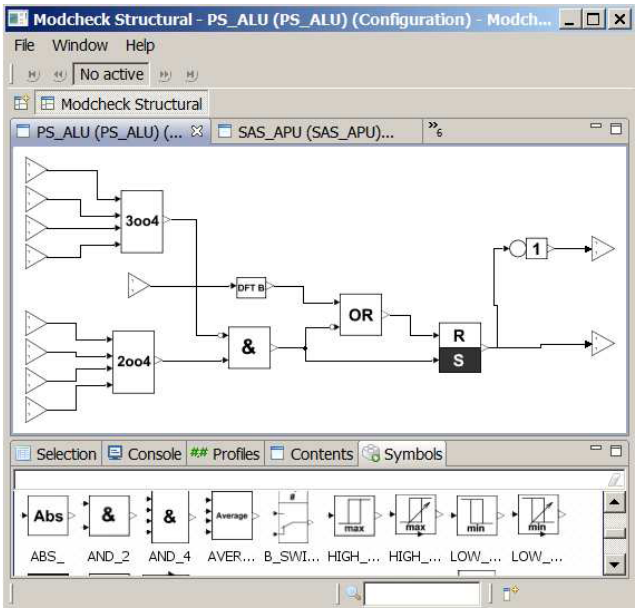

FIGURE 3. Screenshot of PS ALU logic in MODCHK.

E. BASIC BLOCKS AND BLOCK DIAGRAMS

In this subsection, we formalize the structure and behavior of systems like the PS, starting from basic elements and proceeding with modular structures.

1) FINITE-STATE MACHINES

A *finite-state machine* (FSM) is a tuple $(S, S_0, I, O, D, T, \Lambda)$, where:

1) S is a finite, non-empty set of *states*;
2) $S_0 \subseteq S$ is a non-empty set of *initial states*;
3) I and O are finite sets of *input and output variables* respectively;
4) $D : I \cup O \to 2^{\mathbb{Z}} \cup \{\text{false}, \text{true}\}$ is a *domain function* that returns the set of possible values—either a finite set of integers or the set of Booleans—for the given input or output variable;
5) $T \subseteq S \times V_{I,D} \times S$ is a *transition relation*, where $V_{I,D} = \times_{v \in I} D(v)$ is the set of all combinations of values of input variables (assuming their fixed ordering);
6) $\Lambda \subseteq S \times V_{I,D} \times V_{O,D}$ is an *output relation*, where $V_{O,D} = \times_{v \in O} D(v)$ is the set of all combinations of values of output variables (assuming their fixed ordering).

An FSM is *deterministic* if $|S_0| = 1$ and for all $s \in S$, $\iota \in V_{I,D}$ there exist unique $s' \in S$ and $\omega \in V_{O,D}$ such that $(s, \iota, s') \in T$ and $(s, \iota, \omega) \in \Lambda$, i.e., the initial state is unique and exactly one transition and combination of output values exist for each state and combination of input values. A deterministic FSM is thus a Mealy machine [29].

2) TRACES

FSMs have a notion of execution, during which *traces* are produced. A *full trace* of an FSM $(S, S_0, I, O, D, T, \Lambda)$ is a finite or infinite sequence $\tau = ((s_0, \iota_0, \omega_0), (s_1, \iota_1, \omega_1), \ldots)$ such that:

1) $s_0 \in S_0$;
2) each element of τ belongs to Λ;
3) for each pair of consequent elements (s_i, ι_i, ω_i), $(s_{i+1}, \iota_{i+1}, \omega_{i+1})$ of τ, $(s_i, \iota_i, s_{i+1}) \in T$.

Each full trace t has a corresponding *input-output trace* (*IO trace*) obtained from t by removing the first element (i.e., state information) from each tuple. Note that multiple full traces may correspond to a single IO trace. The set of all input traces of an FSM M will be denoted as $\tau_{\text{IO}}(M)$.

3) BASIC BLOCKS AND BLOCK DIAGRAMS

Formally, we will speak of *basic blocks* as of just FSMs, but by using this term we will emphasize that these are elementary parts of the formal model, not having any internal structure. In our case studies, a basic block will always correspond to some NuSMV module. For example, on Fig. 2, all rectangular shapes correspond to basic blocks, and among them is `TON`, whose NuSMV code was given in Section II-C. For convenience, when models are developed in MODCHK, basic blocks are defined once and reused possibly multiple times, with each block having its own state and behavior, e.g., like the two `TON` blocks in Fig. 2 (top).

Interconnected arrangements of FSMs, like the APU and the ALU from Fig. 2, form block diagrams. Formally, a *block diagram* is a tuple $N = (I, O, E, C)$, where I and O are defined as in the case of an FSM, E is a finite set of *elements* of N, where each $e \in E$ is an FSM, and C is a set of *connections* of three kinds:

1) input variables of N are connected to input variables of elements;
2) output variables of elements are connected to input variables of other elements;
3) output variables of elements are connected to output variables of N.

C is assumed to be constrained such that each input variable of an element and each output variable of N has exactly one corresponding connection from some other variable. In practice, missing incoming connections are handy, but the definition above already covers them as we can imagine trivial FSMs generating the same constant value to be connected in place of such missing connections.

The execution semantics of N is defined as follows: the states of N correspond to the combinations of states of its elements, and on the initial cycle all the elements are set to some of their initial states. Then, on each cycle, all $e \in E$ execute synchronously, following the semantics of NuSMV. For connections between the elements, we adopt MODCHK semantics: connections are zero-delay, meaning that one cycle is sufficient for a signal to propagate along a sequence of elements if each of these elements reacts to inputs on the same cycle. The reason for this assumption is the preference of shorter counterexamples: they are computationally cheaper to consider in model checking and easier for the analyst to understand. However, if a set of elements lies on a feedback loop, this loop needs to be broken with a unit delay block to forbid infinite flow of information. This problem of synchronous block diagrams is discussed in detail in [30]. Finally, as the aforementioned semantics allows representing block diagrams as FSMs, the former may be nested, meaning that systems like the one shown in Fig. 1 can also be modeled as block diagrams.

F. SYMMETRY OF FSMS WITH RESPECT TO INPUT VARIABLES

Let $M = (S, S_0, I, O, D, T, \Lambda)$ be an FSM and $\{v_1, \ldots, v_s\} \subseteq I$ be a set of s different input variables of M whose value sets $D(v_1), \ldots, D(v_s)$ are all equal. If p is a permutation of numbers $1, \ldots, s$ and $\iota \in V_{I,D}$, then by $p(\iota)$ we denote another input of M derived from ι by permuting the values of selected variables $v_1, \ldots, v_s$ with p.

M is *symmetric with respect to input variables* $v_1, \ldots, v_s$ if and only if for each permutation p of numbers $1, \ldots, s$:

$$((\iota_1, \omega_1), (\iota_2, \omega_2), \ldots) \in \tau_{IO}(M) \Leftrightarrow ((p(\iota_1), \omega_1), (p(\iota_2), \omega_2), \ldots) \in \tau_{IO}(M),$$

that is, permuting the input values of $v_1, \ldots, v_s$ does not lead to the change of possible output values of the FSM. If M is deterministic, the meaning of this statement is stronger: permuting the corresponding values does not alter the unique output sequence of M.

In nuclear I&C systems, symmetries with respect to input variables are common due to the idea of voting (n-out-of-m blocks) over signals from redundant divisions. The following symmetry observations can be made for the ALU of the PS (also note that the APU has no symmetries):

- the ALU is symmetric with respect to four input variables `HLEG_P_H_OR_NF_H`$_i$ (“hot leg pressure high or neutron flux high”), $1 \le i \le 4$;
- the ALU is symmetric with respect to four input variables `HLEG_T_L_AND_P_L`$_i$ (“hot leg temperature low and hot leg pressure low”), $1 \le i \le 4$.

G. SIGNAL FAULT/VALIDITY STATUSES

In MODCHK, each signal has both a *value* (Boolean or integer) and a fault/validity *status* (a Boolean variable). The status processing is explicitly defined for each basic block, and the counterexample animation feature uses a dashed line to show a faulty/invalid signal [27]. In nuclear applications, the status processing feature has been very relevant, as it is not just used in TELEPERM XS (see Section II-A), but also in a similar way in Rolls-Royce’s Spinline platform [31]. Status processing logic has also played a role in about 12% of the design issues VTT has identified [28].

This means that every basic block accepts and outputs status signals in addition to regular ones. In the simplest case, input statuses are just ignored or passed as output statuses. For example, the definition of `TON` from Section II-C may actually look like below:

```
1 MODULE TON(BIN_IN, BIN_IN_FAULT)
2 ... (like in the previous listing)
3 DEFINE
4     BIN_OUT := ticks = 2;
5     BIN_OUT_FAULT := BIN_IN_FAULT;
```

H. HARDWARE MODELING ASSUMPTIONS

Below, we define some concepts related to failure tolerance, using the Finnish regulatory guides on nuclear safety and security (YVL)[5] as a guideline.

Single failure criterion, which is the primary assumption of this paper, means that the system shall be able to perform its function even if any single component designed for the function fails. Protection against single failures is commonly achieved using several (potentially identical), redundant subsystems placed in physically separated *divisions*.

Consequential failure refers to “a failure caused by a failure of another system, component or structure or by an internal or external event at the facility” [17]. For example, a failure of a power supply system or a ventilation system can result in the subsequent total failure of several I&C system devices, and still be considered a single failure that shall be tolerated. In this paper, consideration of consequential failures is enabled by including simultaneous failure of different devices of the same division.

Common cause failure (CCF) refers to a “failure of two or more structures, systems and components due to the same single event or cause” [17]. Protection against CCF can be achieved using diverse backup systems (e.g., a different supplier, technology, or operating principle). CCFs are not addressed in this paper.

Passive failure means that the system fails to produce the required response. *Active failure* (or “spurious actuation” [28]) refers to inadvertent actuation without a real demand. Passive and active failures are both covered by the proposed approach.

III. PROPOSED APPROACH

The overview of the proposed verification approach is shown in Fig. 4. The approach starts with manual preparation of three kinds of artifacts:

1) Formal models of units (such as the APU and the ALU of the PS) represented as block diagrams. We create them in MODCHK (see Fig. 3), but they also can be written directly in NuSMV.
2) A *modular configuration* that describes how the units are connected and arranged into multiple divisions, together comprising one or more fault-tolerant safety functions. These configurations are explained in Section III-A.
3) *Temporal requirements* to the modular configuration, subdivided into *black-box* and *white-box* ones depending on their purpose (see Section III-C).

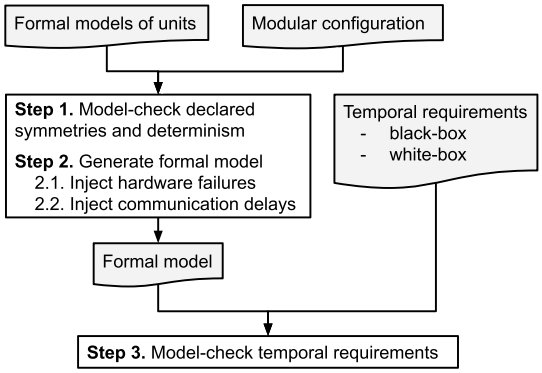

FIGURE 4. Overview of the proposed verification approach.

[5]https://www.stuklex.fi/en/yvl-ohje

These artifacts are processed by three automated steps:

1) Symmetry declarations specified in the modular configuration, as well as the determinism of all declared units are formally verified (see Section III-B).
2) Based on the formal models on units and the specified modular configuration, a NuSMV formal model of the overall system is automatically generated that includes both software and hardware aspects. The creation of this model involves injection of hardware failures (Section III-D) and communication delays (Section III-E). The basic instructions for such injection (such as the divisions to inject failures) are supplied within the modular configuration.
3) The prepared formal model is model-checked against the temporal requirements. This is done by simply executing NuSMV on these artifacts.

The approach is implemented in a software tool, which is available online.[6] Due to confidentiality of basic block models involved in our case study (Section IV), we provide a different fictitious example on which the tool can be executed.

A. MODULAR CONFIGURATIONS

A *modular configuration* is a textual way to specify a block diagram modeling the I&C system to be formally verified. We designed a simple domain-specific language (DSL) to describe such modular configurations. It supports declarations of the following entities:

1) A *unit group* g is a container to group block diagrams (such as the ones of the APU and the ALU from Fig. 2), inputs and outputs thereof in a specified number of divisions $d(g)$ (e.g., $d(g) = 4$). One of the divisions may be assigned as the failing one (see Section III-D). Each unit group corresponds to a single I&C system, which may correspond to a single safety function or a part thereof. It is possible to think of a unit group as of a block diagram, although due to the need to connect units of different unit groups with each other it is easier to define input and output variables for the whole modular configuration (see points 3–4 below).
2) A *unit* u is a block diagram consisting only of basic blocks, belonging to the specified unit group $g(u)$ and having the number of divisions specified by this group: $d(u) = d(g(u))$. The APU and the ALU of the PS (Fig. 2) are examples of units. Each unit is associated with NuSMV code specifying basic blocks and connections between them. The code for the latter is generated by MODCHK from a graphically specified block diagram, but may also be written directly in NuSMV. For each unit, it is possible to introduce communication delays up to the specified number of cycles (see Section III-E) and retain only one specified division in the formal model (see Section III-D). What is more, discovered symmetries (Section II-F) can be listed with separate `symmetry` declarations.
3) An *input* is a set of input variables that are duplicated for the number of divisions of the specified unit group to which they belong and are only distinguishable by the index of this division, i.e., their names, sets of possible values and semantics are the same. For example, `MAN_RESET` is a four-division input of the PS that corresponds to identically named input variables of the ALUs (Fig. 2).
4) An *output* is the same as an input except that it is a set of output variables. For example, `RODS_DOWN` is a four-division output of the PS (Fig. 1, 2) that corresponds to identically named output variables of PS ALUs (Fig. 2).
5) A *single connection* is a usual block diagram connection as defined in Section II-E. Its end points are specified by names with division indices. If the start point is an output variable of a unit, then this variable is addressed with the names of the unit group, the unit, the output, and the division index, e.g., `PS.ALU.ENABLE_SAS.1`. If the end point is an input variable of a unit, the declaration is similar, except that the input name is omitted, but all connections are listed in the order of input variables in the code of the unit.
6) A *parallel connection* represents a group of connections between entities having the same number n_d of divisions, such that division i ($1 \leq i \leq n_d$) of the connection's source is connected to the same division i of the destination. For example, input `MAN_RESET` of the modular configuration is connected in parallel to the identically named input of PS ALU (Fig. 2), and each of these inputs corresponds to four variables for divisions 1..4. Parallel connections are specified similarly to single connections, but concrete division indices are omitted.
7) An *all-to-all connection* represents a group of connections between entities having n_1 (source) and n_2 (destination) divisions. It is a compact way to declare n_1 n_2 single connections between each division of the source and each division of the destination. For example, APUs and ALUs from Fig. 1 are connected this way. All-to-all connections are specified similarly to parallel connections.

A more verbose description of the DSL is supplied with the software tool. The listing of the DSL definition of the PS is given in Fig. 5.

B. SYMMETRY AND DETERMINISM VERIFICATION

As symmetry is used to simplify the overall formal model by omitting certain divisions of certain units and by localizing failures in particular divisions (see Section III-D), formal verification of symmetries improves the reliability of the overall verification. Suppose that FSM $M = (S, S_0, I, O, D, T, \Lambda)$,

[6] https://github.com/igor-buzhinsky/hw-sw-model-builder

```
unit_group name=PS divisions=4 failing_division=2

unit in=PS name=APU filename=ps_apu.smv nusmv_module_name=PS_APU
     nusmv_outputs=HLEG_P_H_OR_NF_H:boolean;HLEG_T_L_AND_P_L:boolean
     max_delay=3 single_division_to_retain=NA
unit in=PS name=ALU filename=ps_alu.smv nusmv_module_name=PS_ALU
     nusmv_outputs=RODS_DOWN:boolean;ENABLE_SAS:boolean
     max_delay=6 single_division_to_retain=1

input in=PS name=HLEG_P     nusmv_type={0,50,80}
input in=PS name=HLEG_T     nusmv_type={20,400}
input in=PS name=NF         nusmv_type={0,180000,250000}
input in=PS name=MAN_RESET nusmv_type=boolean

output in=PS name=RODS_DOWN
output in=PS name=ENABLE_SAS

symmetry group=PS unit=ALU input_variable_indices=1,2,3,4
symmetry group=PS unit=ALU input_variable_indices=5,6,7,8

parallel_connection   from=input.HLEG_P             to=PS.APU
parallel_connection   from=input.HLEG_T             to=PS.APU
parallel_connection   from=input.NF                 to=PS.APU
all_to_all_connection from=PS.APU.HLEG_P_H_OR_NF_H to=PS.ALU
all_to_all_connection from=PS.APU.HLEG_T_L_AND_P_L to=PS.ALU
parallel_connection   from=input.MAN_RESET          to=PS.ALU
parallel_connection   from=PS.ALU.RODS_DOWN         to=output.RODS_DOWN
parallel_connection   from=PS.ALU.ENABLE_SAS        to=output.ENABLE_SAS
```

FIGURE 5. PS definition in the DSL.

which corresponds to some declared unit, has a `symmetry` declaration with respect to input variables $\{v_1, \ldots, v_s\} \subseteq I$. If M is deterministic, the following procedure based on model checking can prove or refute this symmetry.

Two copies of M, M_1 and M_2, are included into the overall formal model (e.g., module `main` in NuSMV). For each non-trivial permutation p of numbers $1, \ldots, s$, M_1 accepts the original input variables and M_2 accepts permuted input variables. If $O = \{o_1, \ldots, o_k\}$ (fault/validity statuses included), then temporal property

$$\mathbf{AG}\left(\left(o_1^{M_1} = o_1^{M_2}\right) \wedge \ldots \wedge \left(o_k^{M_1} = o_k^{M_2}\right)\right)$$

is checked. If all checks are passed, then the symmetry of M with respect to $v_1, \ldots, v_s$ is proven, otherwise it is refuted. This procedure requires $s! - 1$ model checking runs. In our case, such checks terminate in a few seconds since s never exceeds four (the number of divisions in the PS) and unit models are less complex than overall modular configurations.

If M is nondeterministic (or, more precisely, can produce different output sequences in response to the same input sequence), this can be revealed by running model checking on a model created as explained above, but for the identity permutation, i.e., by simply running M against itself. Determinism given a fixed input sequence is a common assumption in nuclear I&C modeling, and thus it is beneficial to check it for M even if no symmetries with respect to input variables exist (this may be possible, e.g., for the APUs). Such a check can be simply enforced by requesting a symmetry check with respect to any single variable (e.g., with `input_variable_indices=1`).

C. SPECIFYING TEMPORAL REQUIREMENTS

Temporal (LTL, PSL or CTL) requirements are defined separately for different units whose outputs are checked. Depending on the target unit, custom modular configurations are prepared to (1) exclude parts of the system's model that do not influence the specified temporal properties and (2) specify single divisions for failure injection based on symmetry observations (see Section III-D).

1) BLACK-BOX REQUIREMENTS

A temporal requirement f to the overall model is a *black-box requirement* for division $i \in \{1, \ldots, d(u)\}$ of unit u if the following restrictions are satisfied:

1) Division i of unit u is selected as a non-failing one (see Section III-D).
2) At least one output variable of division i of unit u is referred to in f.
3) The internal variables of units (i.e., the ones not being input or output ones) are not referred to in f.
4) The output variables of other units in $g(u)$ are not referred to in f. Output variables of units outside $g(u)$ may be referred to in f only if they serve as input variables to u.

TABLE 1. Examples of black-box temporal requirements for division 1 of PS ALU. Division indices are shown in subscripts. Requirements outcomes are shown for the no-delay case.

Requirement type	Temporal requirement(s)	Explanation	Verification outcome	
			No failures	One failure
Invariant	1. $\mathbf{G}\,\neg(\texttt{RODS\_DOWN}_1 \wedge \texttt{ENABLE\_SAS}_1)$	RODS_DOWN and ENABLE_SAS commands cannot be given simultaneously.	True	True[a]
Request-response	2. $\mathbf{G}(\neg\texttt{MAN\_RESET}_1 \wedge (\mathbf{X}\,\texttt{MAN\_RESET}_1) \rightarrow \mathbf{X}\,\neg\texttt{RODS\_DOWN}_1)$	The trip signal RODS_DOWN can be manually reset by the operator.	False[b]	False
Absence of spurious actuation[c]	3. $\mathbf{G}(\texttt{RODS\_DOWN}_1 \rightarrow \mathbf{O}(\texttt{1-out-of-4}(i = 1..4, (\texttt{HLEG\_P}_i > 70) \vee (\texttt{NF}_i > 2{\cdot}10^5))))$	If a RODS_DOWN command is produced, then at least once in the past either the hot leg temperature HLEG_P or the neutron flux NF must be high in at least **one** division.	True	True
	4. $\mathbf{G}(\texttt{RODS\_DOWN}_1 \rightarrow \mathbf{O}(\texttt{2-out-of-4}(i = 1..4, (\texttt{HLEG\_P}_i > 70) \vee (\texttt{NF}_i > 2{\cdot}10^5))))$	The same as above, but the condition must be satisfied at least once for **two** out of four divisions simultaneously.	True	False
	5. $\mathbf{G}(\texttt{RODS\_DOWN}_1 \rightarrow \mathbf{O}(\texttt{3-out-of-4}(i = 1..4, (\texttt{HLEG\_P}_i > 70) \vee (\texttt{NF}_i > 2{\cdot}10^5))))$	The same as above, but the condition must be satisfied at least once for **three** out of four divisions simultaneously.	False	False
Global possibility	6. $\mathbf{AG}\,\mathbf{EF}\,\texttt{RODS\_DOWN}_1$	In any reachable state, both true and false values of RODS_DOWN can be potentially generated.	True	True
	7. $\mathbf{AG}\,\mathbf{EF}\,\neg\texttt{RODS\_DOWN}_1$		True	True

[a]As visible from Fig. 2 (PS ALU), ENABLE_SAS is produced as a negation of RODS_DOWN. This cannot be influenced by an upstream failure.

[b]Violated since the actuation criterion must be met before the reset. [c]This case is illustrated with a family of division-parameterized requirements.

5) All divisions $j \in \Delta$ of u, where $\Delta = \{1, \ldots, d(u)\} \setminus \{i\}$, are *treated equally* in f in the following sense. Consider any input or output of u, which is a group of similar variables specified for each division. Then permuting these variables in f for divisions from Δ may not lead to the change of model checking outcome (satisfied/violated) of f regardless of how all the units are implemented.
6) If some other unit u' is connected to u, j_1 and j_2 are two different divisions of u', and u is symmetric with respect to its input variables that come from divisions j_1 and j_2 of u', then these divisions are treated equally in f as defined above.

This means that the requirement is formulated to check the output variables of division u that does not have an injected failure (otherwise, this requirement would likely be violated). Then, all units in the overall model are treated equally, except this selected division of u—thus, there are no prior assumptions on the location of injected failures in the model. And finally, motivating the name "black-box", the requirement does not refer to the actual implementations of any units (point 3 in the list above), nor to the outputs of other units in the same unit group (point 4)—thus, it is semantically closer to high-level functional requirements to the overall system, such as correct execution of a safety function subject to possible failures.

Examples of black-box requirements for division 1 of PS ALU, assuming a modular configuration defined in Fig. 5, are given in Table 1. In particular, this table introduces several requirement types, or patterns, that we use: *invariants*, *request-response* requirements, and requirements specifying *absence of spurious actuation* [28] and *global possibility* of different unit output values. More comments regarding verification outcomes will be given in Section IV-E.

2) DIVISION-PARAMETERIZED REQUIREMENTS

Suppose that f is a request-response or absence of spurious actuation black-box requirement for unit u. Suppose also that f has one or more subformulas $h_1(r), \ldots, h_t(r)$ expressed as functions of division r, $1 \le r \le d(u)$ to which they relate. We would like to parameterize f with the numbers of $j_1, \ldots, j_t$ of divisions for which $h_1(r), \ldots, h_t(r)$ are satisfied. In the simplest cases, for each $1 \le i \le t$, j_i is either 1, resulting in subformulas $h_i(1) \vee \ldots \vee h_i(d(u))$, or $d(u)$, resulting in subformulas $h_i(1) \wedge \ldots \wedge h_i(d(u))$.

As an example, consider the case of $t = 1$ subformula and $h_1(r) = (\texttt{HLEG\_P}_r > 70) \vee (\texttt{NF}_r > 2 \cdot 10^5)$. Requirement 3 from Table 1 requires this subformula to be satisfied for $j_1 = 1$ division. Note that the 1-out-of-4 construction is actually just a disjunction over all the divisions. While this requirement is satisfied even when a failure is assumed (see Section III-D for the details of failure modeling), its version with $j_1 = 2$ (requirement 4) becomes violated in this case. Finally, a similar requirement with $j_1 = 3$ (requirement 5) is violated even when no failures are assumed. By considering such requirement families, it becomes possible to assess the influence of failures in a quantitative way by finding the number j_1 (in our case, $j_1 = 2$) for which requirement outcomes are different depending on the presence of failure. When formulating division-parameterized requirements, we employ user-friendly macros to specify the j_t-out-of-$d(u)$ operation, and the actual Boolean formulas to combine $h_1(r), \ldots, h_t(r)$ are generated automatically.

3) WHITE-BOX REQUIREMENTS

A temporal requirement f is a *white-box requirement* for division i of unit u if points 1–2 from the definition of a black-box requirement are satisfied, but at least one other point is not. For example,

requirement $\mathbf{G}(\neg\texttt{MAN\_RESET}_1 \wedge \texttt{PS.ALU.AND.OUT}_1 \wedge \mathbf{X}(\texttt{MAN\_RESET}_1) \rightarrow \mathbf{X}\neg\texttt{RODS\_DOWN}_1)$ is a white-box requirement for division 1 of PS ALU as it refers to the internal contents of the ALU (output of AND block), and requirement $\mathbf{G}((\mathbf{X}\wedge_{i\in\{1,3,4\}}\texttt{PS.APU.HLEG\_T\_L\_AND\_P\_L}_i) \wedge \neg\texttt{MAN\_RESET}_1 \wedge \neg\texttt{RODS\_DOWN}_1 \rightarrow \mathbf{X}\neg\texttt{RODS\_DOWN}_1)$ is also a white-box requirement for division 1 of the ALU since it treats division 2 differently from divisions 3 and 4 and refers to the outputs of different units (APUs).

White-box requirements may be reasonable, especially when violating points 3–4, to *understand* how particular units behave. In addition, violating points 5–6 is useful to formulate more *detailed* properties specifically for the case of an injected failure—for example, to distinguish inputs from a failing division from others. White-box requirements are not inferior to black-box ones in terms of their analytical power, and the main reason why we treat them separately in this paper is that, while a black-box property is meaningful regardless of whether a failure is assumed in the model, a white-box property usually accounts for only one of these cases.

D. FAILURE MODELING

Instead of specifying a full failure model allowing all the processors and communication links fail (as in [16]), we simplify the model by keeping our focus on verifying *single fault tolerance* in *open-loop models*. Below, we explain this idea in the general form, which is suitable for any modular configuration, and illustrate it on the PS (see Fig. 6):

1) As mentioned in Section III-C, requirements for different units u are formulated separately. In the case of the PS, it is possible to prepare separate configurations for the APU and for the ALU, but in experiments (Section IV-C) and below we only consider the configuration for the ALU due to the one for the APU being very simple from a computational point of view. Requirements for the APU are included into the ALU configuration.
2) For the chosen unit u, it suffices to model only one division (in our example, only the ALU from division 1). If the divisions are identical and their inputs come from the same sources, any verification result for the included (non-failing, see below) instance of u will hold for the redundant (non-failing) instances as well.
3) There is no need to assume hardware failures for the included division of u. The objective is to verify that each *non-failing* division will operate according to the specification. In open-loop analysis, we are not interested in the outputs of the unit instance that can fail, and for the same reason the number of instances of u whose outputs are correct at each time instant is also not important.
4) It is sufficient, overall, to model hardware failures for one division only. If the divisions are identical, we can assign the failures to any single division other than the one that is fully modeled. In Fig. 6, division 2 is selected for failure injection, but divisions 3 and 4 would be equally suitable. In the general case, multiple unit groups may be considered, and their physical isolation may imply that their failures need to be considered independently. Even in this case it may be sufficient to inject failures in a fixed, single division of each unit group (see Section IV-B), although we also permit cases where this is impossible. In this specific case, several formal models are generated with failures injected into different divisions, and all these formal models are model-checked (for each temporal requirement, the results are then aggregated with logical conjunction).

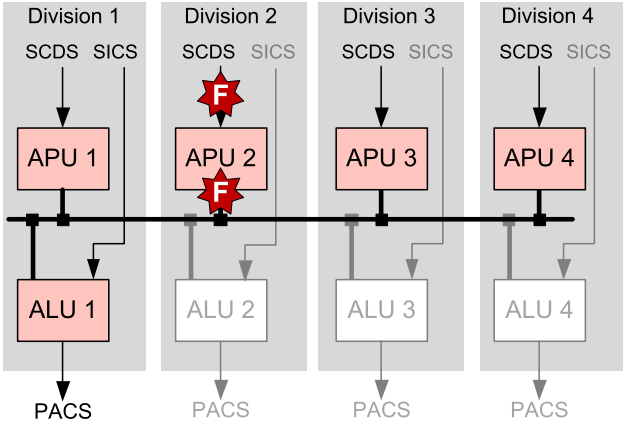

FIGURE 6. Example of failure modeling for the PS. By focusing on single failure scenarios in open loop, the failure model can be made fairly simple.

Furthermore, instead of having a complex failure model for each component of the modular configuration, it suffices to assume that within the division i selected for failure injection, any connection may fail, in which case we replace the correct value with a nondeterministic, arbitrary value from the applicable set of values. This assumption covers the following real-world failures:

1) communication failure between the units (e.g., between the APU and the ALU), between a sensor device or a sensor and a unit (e.g., between NF measurement and the APU, and between a unit and an actuator (e.g., the RODS_DOWN command of the ALU is transmitted incorrectly);
2) hardware failure of a unit—since this corrupts the outputs of the unit, this situation is covered by failure injection to connections.

During the generation of the NuSMV model, to cover each connection, a failure is injected for each input variable of each unit and each output variable of the modular configuration that belong to division i. This means that a usual

connection between modules is broken with a specifically designed basic block `INJECT_FAILURE`. At any cycle, `INJECT_FAILURE` can nondeterministically enter a fault mode and replace the actual signal value with a nondeterministic value. The status of the signal is also nondeterministic, meaning that the failure can be either *self-announcing* (status = fault) or *non-self-announcing*, i.e., not detectable by downstream logic. The actual fault status is separately output by the block, allowing the analyst to observe even non-self-announcing failures.

Rather than analyzing in detail what the possible hardware failure modes are, we allow for all possible combinations of failures, including consequential failures. It might then be up to the analyst to determine if the failure combinations in a counterexample are actually possible. As we allow each signal in the failing division to fail independently and nondeterministically, some of the permitted scenarios can seem unrealistically chaotic, e.g., the different outputs of a failing APU can permanently freeze in a state where some signals fail actively and others passively. Still, such a scenario is also feasible due to single failure, if, e.g., the CPU cooling fails.

E. MODELING COMMUNICATION DELAYS AND ASYNCHRONY

We extend the approach described in Section III-D to account for communication delays and asynchrony. Since each unit operates on its own CPU, modeling delays within block diagrams representing units is unnecessary. A natural approach to asynchrony would involve modeling all the units as different processes with interleaving executions, involving nondeterministic delays. Modeling such an asynchronous behavior in NuSMV can be achieved using the `process` keyword,[7] coupled with `FAIRNESS` [5] declarations to prevent any unit not being scheduled for execution forever. However, this approach results in potentially long counterexamples (e.g., to execute each unit of the simplified PS model from Fig. 6 once, five cycles would be needed) and thus deteriorates model checking performance.

Instead, we exploit the synchronous semantics of NuSMV, allowing all units to be executed simultaneously, but permit communication delays bounded by the number of cycles $d_{\max}$ (specified in the modular configuration for each unit). Delay injection is performed similarly to failure injection, but in this case connections are broken with specifically designed `NONDET_DELAY` basic blocks, and this injection is done for all divisions (not only for the failing one).

`NONDET_DELAY` blocks are implemented in the following way. Assume that $s_1, \ldots, s_k$ are signals whose delays must be synchronized, including fault/validity statuses of these signals. Synchronization is needed only for signals transmitted simultaneously. For an isolated signal (e.g., an input signal to a PS APU), the minimum $k = 2$ is achieved: a signal is paired with its fault/validity status. In contrast, signals from one unit to another (for fixed division indices of both) are grouped, like in the situation of PS ALU in division 1 receiving two paired inputs from PS APU in division 1.

The `NONDET_DELAY` block nondeterministically delays the signals passing through it with bound $d_{\max} = 1$, i.e., by at most one cycle; to implement a multi-cycle delay, several delay modules are connected in a chain of length $d_{\max}$. This possible delay is applied synchronously to all k signals. The presence of the delay is controlled by a Boolean variable `delaying`, whose value is nondeterministic on each cycle. If `delaying` has been true on the previous cycle, the previous values of $s_1, \ldots, s_k$ are returned. Otherwise, their most recent values are returned. However, such an implementation may lead to changes in signal value (e.g., rising and falling edges of Boolean signals) being lost. Hence, the following additional condition restricts the nondeterminism of the module: if `delaying` is true and the value of either of the signals changes between the current and the next cycles, then `delaying` cannot change between these cycles.

[7]This feature of NuSMV is listed as deprecated and is not supported by its successor nuXmv.

IV. EXPERIMENTAL EVALUATION

The proposed approach was evaluated on a case study (Sections IV-A and IV-B) that is a generalization of the PS running example. Block diagrams of units were modeled in MODCHK and exported to NuSMV, modular configurations were prepared in the DSL explained in Section III-A, and functional requirements were translated into LTL, PSL and CTL properties. The performed experiments are explained in Section IV-C, and their results are commented on in Sections IV-D and IV-E.

A. BASIC CASE STUDY: PS ONLY

In this subsection, we restrict our attention to the part of our case study that has already been described in Section II-A—the PS. Failure modeling for the PS has been explained in Section III-D and Fig. 6. According to this explanation, a single modular configuration C_{PS} (Fig. 5) was prepared: it includes four APUs and one ALU (division 1) belonging to a single unit group `PS`, and division 2 is assigned as the failing one. Delays were injected as explained in Section III-E. We found that delaying input signals of APUs with $d_{\max} = 3$ and their output signals with $d_{\max} = 6$ leads to considering all possible processing orders of changed measurement inputs: if such a change happens, the APUs may all become aware of this event on different cycles (assured by APU input delays 0, 1, 2, 3 in different divisions), but the results of processing this change by the APUs may arrive to the ALU in the reverse order (assured by APU output delays 6, 4, 2, 0 in the respective divisions). This means that asynchrony is sufficiently emulated with delay injection.

TABLE 2. Considered modular configurations.

Name	Unit group to check requirements for	Unit groups included into the model	Model size, lines of NuSMV code				State space size[a]	
			No failures		One failure		No failures	One failure
			No delays	With delays	No delays	With delays	No delays[b]	No delays[b]
C_{PS}	PS	PS	855	1109	913	1167	$6.6 \cdot 10^{11}$	$4.0 \cdot 10^{15}$
C_{SAS}	SAS	PS, SAS	1037	1399	1133	1495	$3.3 \cdot 10^{16}$	$2.9 \cdot 10^{35}$
C_{PACS}	PACS	PS, SAS, PACS	1207	1709	1328	1830	$2.1 \cdot 10^{20}$	unknown[c]

[a]Calculated with `print_reachable_states` NuSMV batch command.

[b]Adding delays significantly enlarges the state space, making its size difficult to calculate, thus we only provide information about the no-delay case.

[c]Calculation did not terminate within 48 hours. The actual size must be at least $1.5 \cdot 10^{34}$ due to the added failure injection blocks.

The size and the complexity of the generated PS models are shown in Table 2.

The following functional temporal requirements were prepared for the modular configuration C_{PS} to check the outputs of both APU 1 and ALU 1 (from now on, adding a number i aften the name of a unit or a unit group will mean the reference to division i of this unit or unit group):

- Black-box properties (see Table 1 for examples):
  - one invariant;
  - 27 request-response properties;
  - 5 absence of spurious actuation properties;
  - 2 global possibility properties.
- White-box properties (see Section III-C for examples):
  - 7 request-response properties.

We consider two groups of requirements depending on whether they are prepared for verification with injected delays or not. While *no-delay* requirements rely on instant information flow through the system (e.g., the ALU is required to react to the change of inputs to the APUs on the same cycle), *with-delay* requirements do not, which in practice leads to a higher abundance of unbounded temporal operators **G**, **F** and **O**. For the majority of requirements, we prepared both a no-delay and a with-delay version, and for the remainder the same version was checked regardless of delay injection as adding an explicit instant information flow dependency to these requirements was impossible. In addition, in a couple of cases, a group of several no-delay requirements corresponds to a group of the same number of with-delay requirements, but there is no one-to-one correspondence between requirement pairs.

For some white-box requirements, we also considered different versions depending on whether a failure was assumed in the model, with requirements expecting a failure being weaker. Global possibility requirements were written in CTL, and the rest were formulated in either LTL or PSL. The pattern of division-parameterized requirements was extensively applied especially for request-response black-box properties.

B. EXTENDED CASE STUDY: PS + SAS + PACS

Now we consider the PS in a broader context: the PS operates together with the *Safety Automation System* (SAS), and the outputs of both systems are passed to the *Priority and Actuator Control System* (PACS). The details are again partly based on our own invention (for the SAS, also on [32]). Technically, the PS and the PACS, along with their power supply and other supportive systems, are both needed to perform a safety class 2 reactor trip safety function, while the SAS implements a different, preventive function of *stepwise* reactor trip that belongs to safety class 3 (lower than 2). For this reason, the single failure tolerance criterion will be applied to the PS and the PACS as a whole, and the SAS will be allowed to have an independent failure.

Reactor trip is needed to be actuated when sensor measurements (neutron flux, hot leg temperature and pressure) deviate from their allowed ranges. The SAS is a two-division system consisting of APUs and ALUs (see Fig. 7), just like the PS. The PS and the SAS share neutron flux measurements but have different hot leg temperature measurements. When measurement deviations from their safe ranges are rather small, the SAS performs a preventive, stepwise trip, issuing `RODS_DOWN` signal in several pulses. The SAS is also capable of raising the rods up when the trip criterion is not met. However, if the measurements deviate further, the PS, being of a higher safety class than the SAS, takes over, trips the reactor, and inhibits the SAS from starting the reactor up.

Downstream, the PACS, a four-division system with one function unit in each (without APU/ALU separation), ensures that the PS always has first priority on orders to control rod actuators that are controlled by both the PS and the SAS. The connections between the PS, the SAS and the PACS are shown in Fig. 8.

In our experiments, the PS + SAS + PACS case study is split into three modular configurations C_{PS}, C_{SAS} and C_{PACS} depending on the unit group (`PS`, `SAS` or `PACS`) whose correct responses are verified. The summary of these configurations is shown in Table 2. Configuration C_{PS}, which involves the PS only, has already been considered in Section IV-A. Below, we consider the other two configurations in more detail, reasoning why certain parts of the overall model can be omitted in each case and what failure injection assumptions are sufficient. These reasonings are grounded on the single failure criterion, symmetry and preference of more severe to less severe failures.

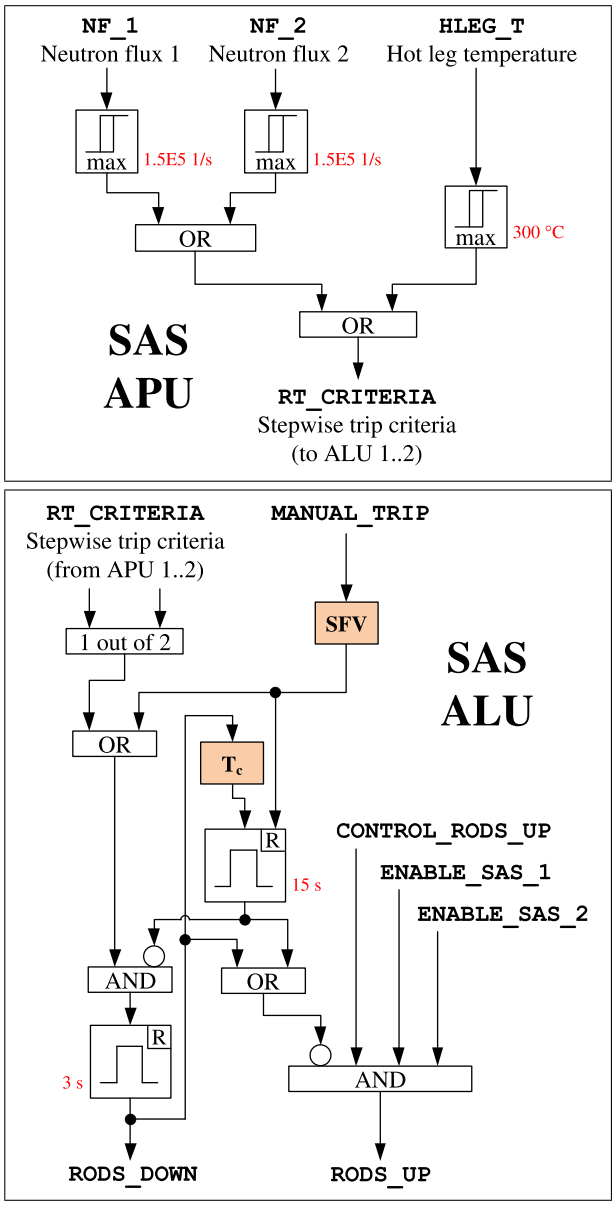

FIGURE 7. SAS APU (top) and ALU (bottom). Function block notations: T_c is a cycle delay, blocks with R (reset signal) in the left upper corner are pulses of the given duration. The rest of notations are given in Fig. 2.

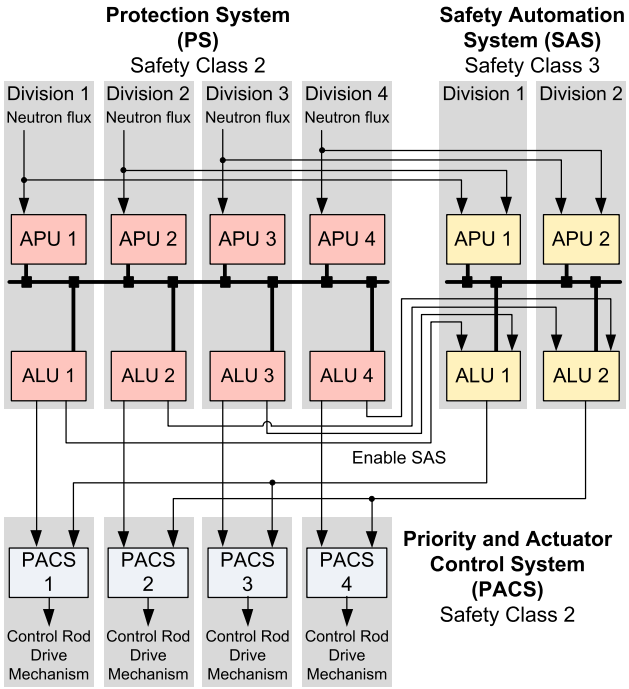

FIGURE 8. Structure of the extended case study. The "Neutron flux" input comes from SCDS; the rest of inputs from SCDS and SICS (see Fig. 1) are not shown for simplicity.

In configuration C_{SAS}, requirements are formulated to check the outputs of SAS APU and SAS ALU. Since the inputs of the SAS depend on the PS, both the SAS and the PS are included as unit groups. Due to the symmetry of PS ALU, SAS APU and SAS ALU with respect to their inputs shown in Fig. 8, verification of SAS ALU 1 and SAS ALU 2 would yield the same results, and thus only SAS ALU 1 is retained (alternatively, we could have retained only SAS ALU 2). Correspondingly, division 2 is selected as the failing one in SAS. The failing division in the PS remains to be determined. SAS ALU 1 receives inputs from PS ALUs 1 and 3 and does not from PS ALUs 2 nor 4, hence a failure in divisions 1 or 3 of the PS may lead to more severe consequences compared to divisions 2 or 4. Among PS 1 and PS 3, we select PS 1 for failure injection since this impairs one of the inputs of SAS APU 1 in addition to the already unreliable SAS APU 2—in the opposite case, the faulty input would go to SAS APU 2.

In configuration C_{PACS}, requirements are formulated to check the single unit of the PACS. Since the inputs of the PACS depend on the PS and the SAS, all these three unit groups are included. Then, since we apply the failure criterion to the PS and the PACS as a functional whole (as explained above), it suffices to assume that the failures in the PS and the PACS co-occur only in the same division. Due to symmetry, it is sufficient to retain only one PACS division. We retain PACS 1, and thus the failing division for both the PACS and the PS must be chosen among 2, 3 or 4. Postponing this decision, we first fix the failing division of the SAS to 1 since SAS ALU 1 is connected to PACS 1 and SAS ALU 2 is not, which means that a failure in division 1 of the SAS may be more severe for PACS 1. What is more, SAS ALU 2 can be omitted entirely. Returning to the failure in the PS, we now see that the influence of the PS on the SAS can be neglected as the only part of SAS connected with the PACS is already failing. PS ALUs 2, 3, 4 are not connected to PACS 1 and hence can be disregarded as well. Nonetheless, non-failing PS ALU 1 is connected to PACS 1. Due to an all-to-all connection and symmetry of PS ALU with respect to its inputs, failures in PS APUs 2, 3, 4 have the same

TABLE 3. Performance of black-box requirements model checking. The percentages of requirements whose verification terminated within the time limit of 10 minutes are shown, and average verification times of these requirements are given in parentheses in seconds. 100% and 0% entries are colored in green and brown respectively. "BDD" stands for the BDD-based model checking algorithm. LTL and PSL requirements, which are processed with the same model checking algorithms, are not distinguished for simplicity of presentation.

Temporal requirements	Model checking algorithm	With proposed simplifications				Without proposed simplifications			
		No failures		One failure		No failures		One failure	
		No delays	With delays	No delays	With delays	No delays	With delays	No delays	With delays
For C_{PS}: 33 LTL, 2 CTL	LTL BMC	100% (5.0)	100% (2.4)	100% (1.5)	100% (1.7)	100% (5.1)	100% (2.4)	100% (5.1)	100% (9.5)
	LTL BDD	100% (4.3)	3% (60.6)	100% (3.1)	3% (141.5)	100% (3.7)	3% (59.9)	48% (170.3)	3% (554.1)
	CTL BDD	100% (0.1)	0%	100% (0.1)	0%	100% (0.1)	0%	100% (3.1)	0%
For C_{SAS}: 12 LTL, 4 CTL	LTL BMC	100% (6.6)	100% (8.0)	100% (0.5)	100% (2.7)	100% (6.4)	100% (8.3)	100% (9.5)	100% (22.5)
	LTL BDD	100% (0.2)	17% (174.7)	50% (219.0)	75% (122.4)	100% (0.2)	17% (174.6)	0%	33% (340.0)
	CTL BDD	100% (0.1)	25% (9.3)	100% (32.2)	50% (11.3)	100% (0.1)	25% (9.2)	50% (456.3)	50% (68.5)
For C_{PACS}: 9 LTL, 4 CTL	LTL BMC	100% (10.1)	100% (18.1)	100% (0.7)	100% (5.6)	100% (10.2)	100% (17.9)	100% (17.9)	100% (89.0)
	LTL BDD	100% (0.3)	0%	56% (294.1)	0%	100% (0.3)	0%	0%	0%
	CTL BDD	100% (0.2)	0%	100% (34.9)	0%	100% (0.2)	0%	0%	0%

consequences for PS ALU 1, which finally means that the choice between these divisions for failure injection is free. We inject a fault into PS 2.

Delay injection for C_{PS} has been explained in Section IV-A. As connections between units become convoluted in C_{SAS} and C_{PACS}, we no longer follow the logic of allowing all processing orders of a change in input measurement by all the units and just delay the inputs of PS and SAS APUs with $d_{\max} = 3$ and inputs of all other units with $d_{\max} = 6$.

Temporal requirements prepared for C_{PS} have been explained in Section IV-A. Requirements for C_{SAS} and C_{PACS} are similar with the following exceptions:

1) for the SAS, white-box requirements never explicitly assume a failure, thus the same requirements are verified for the cases of no injected failure and one failure, like this is always done for black-box requirements;
2) all requirements for the PACS are solely black-box.

C. EXPERIMENTS

Symmetry and determinism of all considered units was checked as described in Section III-B. Since for our case study such a check takes a few seconds per unit (the case study has five different units), below we focus on verification of actual temporal requirements and do not include symmetry and determinism checking time into the reported values.

For each configuration listed in Table 2, four formal models were generated to account for the presence or absence of (1) failure and (2) delay injection. Then, each model was model-checked in NuSMV versus black-box and white-box requirements for the corresponding configuration. White-box requirements were available only for C_{PS} and C_{SAS}.

As the major part of the requirements was formulated in LTL or PSL, we were able to use two model checking algorithms: BDD-based LTL model checking and BMC (the bound $k = 20$ was used). To check global possibility requirements, formulated in CTL, BDD-based CTL model checking was the only possible option. Experiments were performed on Intel Core i7-2670QM CPU with a clock rate of 2.2 GHz. NuSMV was run on a single core with command-line options "`-coi -df -dynamic`" to improve performance. Each temporal requirement was checked with a separate run of NuSMV with a time limit of 10 minutes. For a negative verification outcome, a counterexample was required to be generated to consider this run finished within the time limit.

In addition to experiment groups that have been mentioned above, we compared the proposed techniques to reduce model checking complexity—retaining only a single division of the unit under consideration and fixing failure divisions—with the baseline case by also running experiments without these simplifications. For the case of no injected failures, this only means that some divisions of some units are not excluded from the model. For the case of one injected failure, for each requirement f to be checked, we perform a series of model checking runs with the same time limit until either a negative verification outcome is obtained or all failure combinations have been considered (3 cases of PS failure division in C_{PS}, 4 cases of PS failure division in C_{SAS}, 6 cases of PS and SAS failure divisions in C_{PACS}). The sum of model checking times of this series is considered as the model checking time of f.

D. COMPUTATIONAL RESULTS

The computational results of experiments are given in Tables 3 and 4 for black-box and white-box requirements respectively. The tables report only CPU time (as no parallelism was employed, it approximately corresponds to wall clock time). RAM consumption did not exceed 500 MB in 97.5% of verification runs. Peak RAM consumption was observed for 10 BMC runs that reached $k = 20$ and found no counterexample, requiring 1.0–4.3 GB of RAM and terminating in 25–65 seconds.

TABLE 4. Performance of white-box requirements model checking. The same notations as in Table 3 are used. If there is no corresponding line in Table 3, then this verification case contains only black-box requirements.

Temporal requirements	Model checking algorithm	With proposed simplifications				Without proposed simplifications			
		No failures		One failure		No failures		One failure	
		No delays	With delays	No delays	With delays	No delays	With delays	No delays	With delays
For C_{PS}: 8 LTL	LTL BMC	100% (7.4)	100% (2.3)	100% (3.4)	100% (1.4)	100% (7.8)	100% (2.5)	100% (6.1)	100% (9.9)
	LTL BDD	100% (0.2)	38% (1.0)	100% (0.5)	38% (1.4)	100% (0.2)	38% (1.0)	75% (104.6)	38% (12.5)
For C_{SAS}: 6 LTL	LTL BMC	100% (1.8)	100% (8.3)	100% (0.3)	100% (1.7)	100% (1.8)	100% (8.7)	100% (4.5)	100% (23.2)
	LTL BDD	100% (0.1)	17% (257.4)	50% (199.9)	67% (60.4)	100% (0.1)	17% (280.8)	0%	67% (408.9)

1) PERFORMANCE OF BMC

BMC succeeded in checking all requirements in all configurations within the time limit, with average execution times tending to be larger on more complex modular configurations. The results of BMC will be used for qualitative analysis of verification outcomes in Section IV-E. Note that since BMC is an imprecise approach, it can falsely report a violated requirement to be satisfied if the bound k is too small. There is no simple approach to find the minimum sufficient value of k for an arbitrary model. With $k = 20$, we never observed contradictions between BMC and BDD-based LTL verification, and the maximum length of counterexamples generated by BMC was 14, giving evidence for the sufficiency of $k = 20$ that can result in counterexamples up to 21 cycles long.

With the proposed simplifications enabled, BMC always requires less time in the case of an injected failure. This is explained by a higher abundance of violated temporal requirements and the sufficiency of shorter counterexamples. Since BMC works by iteratively increasing counterexample length, this on average leads to its faster termination. Without proposed simplifications, the foregoing observation does not hold due to running BMC several times per requirement.

2) PERFORMANCE OF BDD-BASED MODEL CHECKING

BDD-based LTL and CTL model checking is often unable to handle the case of injected delays, and in some cases fails to terminate within the time limit even without delays. Speaking of CTL verification, properties were checked for no-delay cases, but many with-delay cases remained unsolved. No imprecise approach like BMC exists for CTL model checking, and hence we were unable to obtain verification outcomes for these properties by other means. An additional attempt to verify the same properties with an increased time limit of one hour also failed.

3) PERFORMANCE OF MODELING SIMPLIFICATIONS

In the case of no injected failures, differences between the experiments with enabled and disabled simplifications, which in this case only means excluding unused divisions of some units from the overall model, are negligible. This result is explained by the use of cone of influence (COI) reduction during verification, which simplifies the verified model on its own, retaining only the parts influencing the checked temporal property. Thus, this simplification is achievable due to particular divisions not being referred to in temporal properties, which is a part of the proposed approach (Section III-C), and explicitly excluding unused model parts does not speed up the computations.

In the case of one injected failure, the value of the simplifications is more visible as they allow performing only one verification run instead of several. As a result, not only verification time decreases but also more verification runs can terminate in a reasonable time.

E. ANALYSIS OF VERIFICATION OUTCOMES

As our case study is partly fictitious, we did not intend to reveal previously unknown I&C issues nor to rigorously verify its fault tolerance. Yet, temporal requirements that we checked demonstrate how failures affect verification outcomes. Some examples of these outcomes are given in Table 1. Below, we comment on each type of verified requirements separately:

- Invariants are the simplest temporal properties of the form $\mathbf{G}f$, where f is a formula over state variables of the model without temporal operators. In our case study, invariants are used to check the consistency of outputs of individual units, which in our case is always guaranteed by the unit regardless of its possible inputs. Hence, these invariants are insensitive to both failures and delays.
- Request-response and absence of spurious actuation requirements are typically the properties that are most sensitive to failure injection, and in addition two versions of such properties depending on the presence of delays are usually possible. Requirements 3–5 from Table 1 exemplify a family of division-parameterized requirements whose verification outcomes, when taken together, give a detailed picture of the influence of a failure. In a fault-free case, since requirement 4 is satisfied and requirement 5 is not, the safety function will not be activated unless the precondition (excessive measurements) is satisfied in two divisions. With a failure, since requirement 3 is satisfied and requirement 4 is not, excessive measurements in a single division may be sufficient for activation. Proving that these are necessarily

measurements in a non-failing division 1, 3 or 4 (that is, a proper reason to activate the safety function) requires a more detailed white-box property $\mathbf{G}(\texttt{RODS\_DOWN}_1 \rightarrow \mathbf{O}(\wedge_{i=1,3,4}((\texttt{HLEG\_P}_i > 70) \vee (\texttt{NF}_i > 2 \cdot 10^5)))$, whose outcome is true.
- Global possibility of different unit output values was proved for all no-delay cases, including the ones with failures, but many with-delay cases remained unsolved.

Finally, we comment on the need of multiple I&C systems in one model for accurate verification outcomes: this limits behavior scenarios to more realistic ones that are still manifold enough to cover the single failure criterion. Note that the satisfaction of temporal requirements behaves differently depending on requirement types when new (upstream) I&C systems are added:

- Since LTL property satisfaction for a model is defined as its satisfaction for all paths in this model, a violated LTL or PSL property may become satisfied, but not otherwise. For division-parameterized requirements, adding upstream unit groups would refine the number of divisions needed to refute the requirement.
- Global possibility CTL requirements behave oppositely: in some states, certain variable values may become unreachable, violating a previously satisfied property. This enables a potential discovery of new system issues.
- CTL requirements of other kinds may behave differently, but they are not considered in our case study.

V. RELATED WORK

A number of studies address failure tolerance of safety-critical systems with formal methods, especially with model checking. A common problem with many of previously proposed methods is that the system model has to be kept very abstract, or the state space becomes too large [16], and/or the analysis cannot be performed in a reasonable time [15]. Instead of detailed system design, the models are based on ''specified behavior'' [11], ''functional model'' [14], or other abstractions or simplifications.

When revealed, a typical model scale is 10^6 states [11], [13], [33]. In [34], the authors use a model with 10^9 states (but due to the necessary iterations, entire verification effort still takes days). Meanwhile, the complexity of the detailed design (no-fault) models VTT is verifying is on a wholly different level, often with 10^{20}, sometimes even 10^{30} reachable states. On the other hand, we note that when symbolic model checkers [21], [35] are used for verification, as in our case, the impact of the state space size on the computational complexity of verification is indirect, and model checking performance remains quite unpredictable given the model to be verified. Still, according to [35], abstractions should be used to reduce the state space.

In [12], real-time model checker Uppaal [36] is used to verify the fault tolerance of a 3-redundant aerospace system. Possible failures are specified in detail as extensions of timed automata describing fault-free components. The complexity of the case study considered in [12] is difficult to compare with ours due to the model being based on explicit states rather than function block design. The authors report no computational problems, although Uppaal is able to verify only a limited subset of CTL.

In [10], faults are added to a model of an aircraft wheel brake system with two redundant pistons, which is then verified with SCADE design verifier. In [11], fault variables are introduced in a state model of a 2-redundant spacecraft controller system, which is then verified with explicit-state model checker Spin [37]. In [9], NuSMV is used to verify whether an avionic altitude switch with three altimeters can tolerate measurement errors. In [13], the application is a 2-redundant railway interlocking system, and in [15], it is an FPGA logic. Finally, in [14] failure mechanisms are added to a NuSMV model of an automotive brake-by-wire system based on *failure modes and effects analysis* (FMEA) [38].

Other ways of combining FMEA and model checking have also been proposed. In [34] and [39], FMEA is supported by using model checking to identify the system-level consequences of component failures. In [33], probabilistic model checking is used to identify which components contribute most to system-level failures.

To the best of our knowledge, the only work from the nuclear I&C domain that analyzes failure tolerance with model checking is [16], where a complex failure model is proposed that specifies concrete failures in each device. The analyst can specify the number of single failures and CCFs and then verify whether a success criterion is reached for each of five considered accident scenarios. The case study in [16] includes seven four-redundant I&C systems in the overall NuSMV model (some of its parts are abstracted away when they are not needed). While such a model may appear to be larger than we have, the units considered in [16] were significantly simpler computationally, being only composed of blocks without memory, signals were not paired with their validity statuses, and communication delays were not modeled. For such a case study, it was only possible to verify invariants with k-induction [40]; BDD-based model checking was computationally infeasible and LTL BMC was not tried. The author attributes the computational complexity of model checking to the used failure model, which is precisely what has been refined in our work.

VI. CONCLUSION

In this paper, a novel verification approach has been proposed to address verification of I&C systems that achieve failure tolerance with redundancy. The approach was demonstrated to work on nuclear I&C systems, but its core assumption of single failure tolerance is rather general, making the approach potentially useful in other safety-critical fields, such as the

aerospace [12] and the automotive [14] ones. While previous attempts (e.g., [15], [16]) at introducing hardware failure modes to I&C application logic model checking have resulted in prohibitively complex models and limited analysis capabilities, our approach enables formal model generation for redundant I&C systems with a simplified failure model and also simplifies subsequent verification with a number of abstractions. The approach is coupled with the idea of formulating temporal requirements that permits such abstractions without the change of verification results.

Model checking experiments with the generated models show that LTL, PSL and CTL properties can usually be verified with BDD-based (exact) model checking, although not always if multiple safety functions are considered within one model. A more serious obstruction for exact model checking is the inclusion of communication delays and asynchrony. On the other hand, both the cases of multiple safety functions and delays are well-handled with BMC, although at the expense of coverage of possible scenarios. Model checking of temporal properties other than invariants has not been previously possible for nuclear I&C system models of similar complexity.

Apart from efficient failure modeling, the proposed approach is capable of handling the interaction of multiple interconnected subsystems. According to [28], in 17% of the I&C design issues previously identified by VTT, the fault scenario requires several I&C systems to interact. We evaluated our approach using a model of a semi-fictitious nuclear reactor protection system, aiming at the complexity of real-world detailed designs. At the moment, our case study includes three I&C systems. In the future, the failure modeling approach should be further experimented with on an industrial scale as the success of verification for our case study suggests that it can be expanded. VTT's customer projects could provide the opportunity.

Our study has more limitations that may be addressed in future work. First, our model of delays (Section III-E) is suitable to emulate asynchrony but is insufficient to verify real-time requirements since discrete time steps are not mapped to any real time intervals. We are studying the possibility of applying timed automata verification to solve this problem. For example, it may be performed in HyCOMP [41] or Uppaal [12], [36], [42].

Second, failure localization in particular divisions of unit groups (that is, assignment of failing divisions in modular configurations) is currently done manually. As this procedure is done accounting for identity of units in different divisions and the formalized notion of symmetry, it may be automated.

Finally, the proposed approach is not fully integrated with the graphical tool MODCHK that we use. While systems like the ones generated by the developed model generation tool can be visually designed in MODCHK as well, mimicking the structure of the generated models (e.g., we have a visual model of PS + SAS + PACS without failures and delays), neither automatic import from the generated NuSMV models nor generation of MODCHK models directly is available.

ACKNOWLEDGMENT

The authors would like to thank M. Johansson, K. Wahlström, M. Halinen and N. Lahtinen of STUK for valuable discussions and comments, and P. Ovsiannikova, J. Lahtinen and the anonymous reviewers for giving advice to enhance the manuscript.

REFERENCES

[1] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[2] J. Rushby, "Theorem proving for verification," in *Summer School on Modeling and Verification of Parallel Processes*. Berlin, Germany: Springer, 2000, pp. 39–57.

[3] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, no. 1, pp. 3–25, 2009.

[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[5] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[6] A. Pakonen, J. Valkonen, S. Matinaho, and M. Hartikainen, "Model checking for licensing support in the Finnish nuclear industry," in *Proc. Int. Symp. Future Nucl. Power Plants (ISOFIC)*, 2014, pp. 1–9.

[7] A. Pakonen, T. Tahvonen, M. Hartikainen, and M. Pihlanko, "Practical applications of model checking in the Finnish nuclear industry," in *Proc. 10th Int. Top. Meeting Nucl. Plant Instrum., Control Human Mach. Interface Technol. (NPIC HMIT)*, 2017, pp. 1342–1352.

[8] Areva, *U.S. EPR Protection System Technical Report, Revision 4*, document ANP-10309NP, 2012. [Online]. Available: https://www.nrc.gov/docs/ML1216/ML121660317.html

[9] M. P. E. Heimdahl, Y. Choi, and M. W. Whalen, "Deviation analysis: A new use of model checking," *Automated Softw. Eng.*, vol. 12, no. 3, pp. 321–347, Jul. 2005.

[10] A. Joshi and M. P. E. Heimdahl, "Model-based safety analysis of Simulink models using SCADE Design Verifier," in *Proc. Comput. Saf., Rel., Secur.*, R. Winther, B. A. Gran, and G. Dahll, Eds. Berlin, Germany: Springer, 2005, pp. 122–135.

[11] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann, "Validating requirements for fault tolerant systems using model checking," in *Proc. IEEE Int. Symp. Requirements Eng. (RE)*, Apr. 1998, pp. 4–13.

[12] M. Zhang, Z. Liu, C. Morisset, and A. P. Ravn, "Design and verification of fault-tolerant components," in *Methods, Models Tools for Fault Tolerance*, M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, Eds. Berlin, Germany: Springer, 2009, pp. 57–84.

[13] C. Bernardeschi, A. Fantechi, and S. Gnesi, "Model checking fault tolerant systems," *Softw. Test., Verification Rel.*, vol. 12, no. 4, pp. 251–275, Dec. 2002.

[14] S. Sharvia and Y. Papadopoulos, "Integrating model checking with HiP-HOPS in model-based safety analysis," *Rel. Eng. Syst. Saf.*, vol. 135, pp. 64–80, Mar. 2015.

[15] R. Leveugle, "A new approach for early dependability evaluation based on formal property checking and controlled mutations," in *Proc. 11th IEEE Int. On-Line Test. Symp.*, Jul. 2005, pp. 260–265.

[16] J. Lahtinen, "Hardware failure modelling methodology for model checking," VTT Tech. Res. Centre Finland, Espoo, Finland, Tech. Rep. VTT-R-00213-14, 2014.

[17] STUK. (2013). *YVL B.1 Safety Design of a Nuclear Power Plant*. [Online]. Available: https://www.stuklex.fi/en/ohje/YVLB-1

[18] Areva. (2013) *U.S. EPR Final Safety Analysis Report*. [Online]. Available: https://www.nrc.gov/reactors/new-reactors/design-cert/epr/reports.html

[19] A. Pakonen and I. Buzhinsky, "Verification of fault tolerant safety I&C systems using model checking," in *Proc. 20th IEEE Int. Conf. Ind. Technol. (ICIT)*, Feb. 2019, pp. 969–974.

[20] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: A new symbolic model checker," *Int. J. Softw. Tools Technol. Transf.*, vol. 2, no. 4, pp. 410–425, 2000.
[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," *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.
[22] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Softw. Syst. Model.*, vol. 15, no. 4, pp. 937–960, Oct. 2016.
[23] B. F. Adiego, D. Darvas, E. B. Viñuela, J. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Trans. Ind. Informat.*, vol. 11, no. 6, pp. 1400–1410, Dec. 2015.
[24] S. Preuße, H. Lapp, and H.-M. Hanisch, "Closed-loop system modeling, validation, and verification," in *Proc. 17th IEEE Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2012, pp. 1–8.
[25] J. Machado, B. Denis, and J.-J. Lesage, "Formal verification of industrial controllers: With or without a plant model?" in *Proc. 7th Portuguese Conf. Autom. Control (CONTROLO)*, 2006, pp. 341–346.
[26] I. Buzhinsky, A. Pakonen, and V. Vyatkin, "Explicit-state and symbolic model checking of nuclear I&C systems: A comparison," in *Proc. 43rd Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Oct./Nov. 2017, pp. 5439–5446.
[27] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *Proc. 18th IEEE Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2013, pp. 1–6.
[28] A. Pakonen and K. Björkman, "Model checking as a protective method against spurious actuation of industrial control systems," in *Proc. 27th Eur. Saf. Rel. Conf. (ESREL)*, 2017, pp. 3189–3196.
[29] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems—A Cyber–Physical Systems Approach*. Cambridge, MA, USA: MIT Press, 2011.
[30] S. A. Edwards and E. A. Lee, "The semantics and execution of a synchronous block-diagram language," *Sci. Comput. Program.*, vol. 48, no. 1, pp. 21–42, 2003.
[31] Rolls-Royce, "Spinline, a Rolls-Royce modular I&C digital platform dedicated to nuclear safety," Rolls-Royce, Meylan, France, Tech. Rep. 0004/TS/12, 2012.
[32] K. Björkman, J. Frits, J. Valkonen, K. Heljanko, and I. Niemelä, "Model-based analysis of a stepwise shutdown logic," VTT Tech. Res. Centre Finland, Espoo, Finland, Tech. Rep. 115, 2009.
[33] H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue, "Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples," in *Proc. 6th Int. Conf. Quant. Eval. Syst. (QEST)*, Sep. 2009, pp. 299–308.
[34] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay, "Experience with fault injection experiments for FMEA," *Softw., Pract. Exper.*, vol. 41, no. 11, pp. 1233–1258, Jan. 2011.
[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics*. Berlin, Germany: Springer, 2001, pp. 176–194.
[36] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Proc. 3rd Int. Conf. Quant. Eval. Syst. (QEST)*, Sep. 2006, pp. 125–126.
[37] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.
[38] *Failure Modes and Effects Analysis (FMEA and FMECA)*, IEC Standard 60 812:2018, 2018.
[39] V. Molnár and I. Majzik, "Model checking-based software-FMEA: Assessment of fault tolerance and error detection mechanisms," *Periodica Polytechnica Elect. Eng. Comput. Sci.*, vol. 61, no. 2, pp. 132–150, 2017.
[40] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using *k*-induction," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2011, pp. 351–368.
[41] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "HyComp: An SMT-based model checker for hybrid systems," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2015, pp. 52–67.
[42] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Rel. Eng. Syst. Saf.*, vol. 105, pp. 104–113, Sep. 2012.

IGOR BUZHINSKY was born in 1992. He received the B.Sc. and M.Sc. degrees in applied mathematics and computer science from ITMO University, St. Petersburg, Russia, in 2013 and 2015, respectively, the M.Sc. degree in software engineering and service design from the University of Jyväskylä, Jyväskylä, Finland, in 2015, and the D.Sc. (Tech.) degree from Aalto University, Espoo, Finland, in 2019.

He is currently a Postdoctoral Researcher at Aalto University, and also a Software Engineer with ITMO University. His research interests include formal verification and synthesis of finite-state models, practical application of model checking for safety assessment of industrial automation systems, and machine learning.

ANTTI PAKONEN was born in 1979. He received the M.Sc. (Tech.) degree from the Helsinki University of Technology, Espoo, Finland, majoring in I&C systems, in 2004.

He is currently a Senior Scientist and a Project Manager of the VTT Technical Research Centre of Finland Ltd., Espoo, where he has been employed since 2002. His research interests include I&C software engineering, I&C architecture evaluation, practical application of model checking in industrial applications, and knowledge management.