

Visual counterexample explanation for model checking with OERITTE

Polina Ovsiannikova^{*†}, Igor Buzhinsky^{*†}, Antti Pakonen[‡], and Valeriy Vyatkin^{*†§}

^{*}Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia

[†]Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland

[§]Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden

[‡]VTT Technical Research Centre of Finland Ltd., Espoo, Finland

Email: polina.ovsiannikova@aalto.fi, igor.buzhinskii@aalto.fi, antti.pakonen@vtt.fi, valeriy.vyatkin@aalto.fi

Abstract—Despite being one of the most reliable approaches for ensuring system correctness, model checking requires auxiliary tools to fully avail. In this work, we tackle the issue of its results being hard to interpret and present OERITTE, a tool for automatic visual counterexample explanation for function block diagrams. To learn what went wrong, the user can inspect a parse tree of the violated LTL formula and a table view of a counterexample, where important variables are highlighted. Then, on the function block diagram of the system under verification, they can receive a visualization of causality relationships between the calculated values of interest and intermediate results or inputs of the function block diagram. Thus, OERITTE serves to decrease formal model and specification debugging efforts along with making model checking more utilizable for complex industrial systems.

Index Terms—user-friendly model checking, counterexample explanation, counterexample visualization

I. INTRODUCTION

One of the biggest advantages of *model checking* [1] is a possibility to ensure that the specification is satisfied in every state of the given formal model. This fact is especially useful when it comes to verification of systems on chips [2], the Internet of things installations [3] and other safety-critical systems. Specifically, in this work, we will emphasize its application for industrial size instrumentation and control (I&C) systems. Model checking has been successfully used for verifying nuclear power plant (NPP) I&C system design in Finland [4], Korea [5], Hungary [6], and at the European Organization for Nuclear Research (CERN) [7]. It was also used to check safety of aircraft flight control [8] systems. The distinctive features of I&C systems with regard to model checking are their complexity, modularity and the need to comply with safety requirements, which means that every failure result should be thoroughly analyzed and fixed.

However, in its initial version, model checking is not user-oriented and its application requires additional knowledge about a system as a whole, as well as time and efforts aimed to localize an error in the model of the system. The tool

OERITTE, presented in this paper, does one step closer to the user-friendly model checking and focuses on explanation of negative verification results.

In this work, we consider I&C modular models specified as *function block diagrams (FBD)*.¹ An FBD is an arrangement of interconnected entities called *function blocks*. Each function block can be assumed to be a deterministic Mealy finite-state machine [10]. We use the synchronous FBD execution semantics: on each discrete step, each block executes once, connections link the outputs of blocks to inputs of other function blocks already on the current time step, and feedback loops are manually broken with delay function blocks to prevent the infinitely fast flow of information. Practically, the FBDs we work with can be constructed with graphical tools such as MODCHK [11] or Simulink Design Verifier [12], but they can also be specified in the languages of model checkers, such as NuSMV [13].

The set of properties to be checked for the system, or its specification, is assumed to be formulated in *linear temporal logic (LTL)* [1]. This logic is a generalization of the propositional Boolean logic over state sequences, where a *state* is an assignment of values to the variables of the checked formal model. In LTL, the following *temporal operators* are used (below, φ_1 and φ_2 are LTL formulas):

- $\mathbf{G} \varphi_1$ (“globally”): φ_1 must be true on the entire trace of the model;
- $\mathbf{F} \varphi_1$ (“finally”): φ_1 must hold eventually;
- $\varphi_1 \mathbf{U} \varphi_2$ (“until”): φ_1 must be true until φ_2 is true, and the latter is required to eventually happen;
- $\mathbf{X} \varphi_1$ (“next”): φ_1 must be true for the next state.

A finite or infinite state sequence is *valid* if it starts in one of the model’s initial states and every pair of adjacent states belongs to the model’s transition relation. An *LTL formula* (also called an LTL property) is assumed to be satisfied for the model when it is satisfied for all its valid state sequences. *Model checking* of an LTL formula constitutes finding whether the formula is satisfied for the model and, if it is not, finding a *counterexample* that demonstrates its violation.

¹The presented contributions are not limited to FBDs as specified in the IEC 61131-3 standard [9]. Instead, we use this term in a more general sense as described in this paragraph and later.

Now, a *counterexample* (or a failure trace) for a property φ in a given formal model is a valid state sequence where φ is not satisfied. In model checking, it is sufficient to consider finite counterexamples and the ones that can be represented as a prefix combined with a loop (so-called lasso-shaped counterexamples).

Counterexamples are usually represented by a table of values and for industrial systems are often incomprehensible [14] without auxiliary tools. Therefore, we outline three challenges that analysts usually face when trying to utilize a failure trace in the verification process:

- 1) For industrial I&C system models, counterexample traces could be quite long (tens of steps) [15] due to the delay elements in the logics, necessary for implementing complex control sequences. In the meantime, each of steps may include dozens of variables where inputs and outputs are usually mixed. Hence, without having a counterexample visualized, it can be difficult to pinpoint on which time step the property fails.
- 2) Specification failures are neither obvious to localize, i.e., a property can be of high complexity, requiring pen and paper to evaluate it and identify the branch that has caused the problem.
- 3) When it comes to faults of the system itself, the analyst has to account for every module to figure out the source of the problem, which requires the analyst to have strong expertise in the formal language used for the system’s implementation.

OERITTE assists the analyst in three ways. First, it visualizes FBD data flow throughout the counterexample. Second, it uses the algorithm from [15] to explain a violated LTL property. Third, it implements a novel algorithm that explains individual assignments in terms of highlighted assignments and connections between them in a given FBD.

The rest of the paper is structured as follows. Section II describes the internal representation of an FBD in OERITTE. The problem of counterexample explanation and algorithm that solves it are given in Sections III and IV. Section V overviews the tool that implements the proposed approach, and Section VI evaluates the approach experimentally. Related research is reviewed and compared with the proposed approach in Section VII. Section VIII concludes the paper.

II. SYSTEM REPRESENTATION AS FBD

In this work, function blocks of an FBD that represents a system under verification can be of two types: basic and complex. A *basic block* is a block corresponding to some atomic operator or a simple function listed in Table I. *Complex blocks* operate with Booleans and integers and can be decomposed into combinations of interconnected function blocks. An FBD, which is a system under verification, is a complex block of the highest hierarchy level, i.e., it is not a part of any other complex block. When we say *blocks*, we consider both types of blocks – basic and complex.

Blocks have input and output interfaces that are comprised of named *gates*. A gate that belongs to the interface of a com-

plex block can have single incoming and multiple outgoing connections. On the other hand, for basic blocks, input gates cannot have outgoing connections, and output gates cannot have incoming ones. We say that blocks A and B are connected if there is a gate in A that has an outgoing connection c and there is a gate in B that has connection c as an incoming one (or vice versa).

Now let us consider the types of basic blocks listed in Table I. The meaning of the blocks in the first three groups is quite straightforward and their interfaces are comprised of two input gates and one output gate. Blocks from the group “Other” require some elaboration:

- With the DELAY basic block, one can construct modules with memory, i.e., if calculations at the next time step require the value that was inferred at one of the previous ones, it is possible to delay it by one cycle and have it as an input in the same complex block in the future. This block has two inputs – one for the variable that should be delayed and one for its default value at the first step. Delay by $N > 1$ cycles can be implemented as a chain of N DELAY blocks.
- The CHOICE basic block imitates a chain of conditional assignments, following the semantics of the case NuSMV operator. It ends with the “else” branch defining the value in case none of the conditions are satisfied. The inputs of this block are conditions and the outputs that correspond to them. The output of CHOICE is the value of the output for the first satisfied condition.
- The output of the COUNT block is the number of input signals that are TRUE at the current step. This block requires that its inputs are Boolean.
- The ASSIGN block implements the identity function: the output is the same as the input.

Connections between basic blocks can be usual and inverted, i.e., the accepting block receives the inverted signal from the output of the producing block. Hence, due to having \wedge , \vee blocks and inversion, the chosen set of basic blocks is sufficient to express any Boolean function. An example of a complex block with its internal structure is provided in Fig. 1.

III. COUNTEREXAMPLE EXPLANATION PROBLEM

Informally, we aim to explain the false outcome of an LTL formula φ on counterexample X of length l to a given FBD D with its set of variables $U = \{u_1, \dots, u_n\}$ using both the values of state variables of the counterexample and the blocks in D . Formally:

Definition III.1 (Assignment). An *assignment* a is a tuple $(u, v_{u,j}, j)$, where $v_{u,j}$ is the value of variable u at discrete

TABLE I
BASIC BLOCKS USED FOR THE MODULE REPRESENTATION.

Logical	$\wedge, \vee, \Leftrightarrow$
Arithmetical	$-, +, \times, \div$
Relation	$>, <, \leq, \geq, =$
Other	DELAY, CHOICE, COUNT, ASSIGN

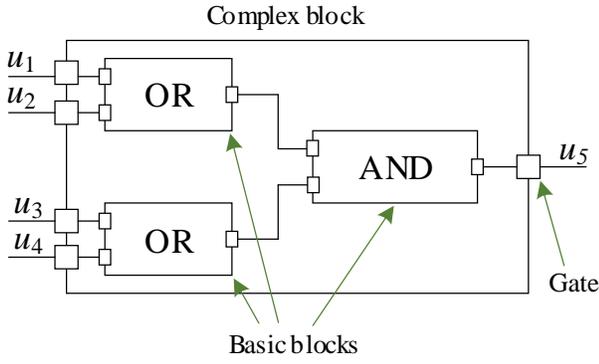


Fig. 1. A complex block that encodes function $u_5 = (u_1 \vee u_2) \wedge (u_3 \vee u_4)$.

time step j . By $v(a)$ we denote the value of this assignment and by $s(a)$ its step. If $u \in U$ is a variable of D then there exists index $i \in [1, n]$ for u , and, to simplify the text, assignments for u instead of $a_{u_i, j}$ are denoted as $a_{i, j}$.

Definition III.2 (Counterexample). A *counterexample* X of length l is a set of assignments of the variables from U for each time step: $X = \{(u_i, v_{i, j}, j) \mid i \in [1, n], j \in [1, l]\}$.

FBD D consists of complex blocks which, in the end, are decomposed into nets of basic blocks. We view basic blocks as sets of symbolic constraints on their input and output variables. These constraints are specified for each instance of each basic block.

Definition III.3 (Basic block constraints). Every basic block B with k input variables and one output variable from Table I except for DELAY is uniquely defined by its set of constraints $C_B = \{v_{1, s} = f_B(v_{2, s}, \dots, v_{k, s}) \mid s \in [1, l]\}$, where s is a step of X , f_B is determined by the type of each basic block, and u_2, \dots, u_k and u_1 are $k - 1$ input and one output variables of a particular instance of B . DELAY corresponds to the following set of constraints: $C_B = \{v_{1, 1} = v_{2, 1}\} \cup \{v_{1, s} = v_{3, s-1} \mid s \in [2, l]\}$, where $v_{2, 1}$ is a default value for the first counterexample step, and u_1, \dots, u_3 are variables of a particular instance of DELAY.

For example, the set of constraints for the AND block is $C_\wedge = \{v_{1, s} = v_{2, s} \wedge v_{3, s} \mid s \in [1, l]\}$. Each constraint in such a set encodes a rule of how the block functions at a particular counterexample step and, therefore, the number of elements in the set equals to the length of a counterexample. Connections also correspond to similar constraints, thus making the constraints of all blocks linked to each other:

Definition III.4 (Connection constraints). Assume that output variable of some block u_1 is connected to input variable u_2 of another block. Then a set of constraints for such connection c is $C_c = \{v_{1, s} = v_{2, s} \mid s \in [1, l]\}$.

Definition III.5 (Complex block constraints). Let B be a complex block with its set of internal blocks M and set of internal connections Σ . Then, a set of constraints for B is

$$C_B = \{C_m \mid m \in M\} \cup \{C_\sigma \mid \sigma \in \Sigma\}.$$

As an FBD is a complex block, the set of constraints C_D is defined for D as well. Definitions III.3, III.4 and III.5 are illustrated in Fig. 2 for a counterexample of length 1. Fig. 2 shows how the set of constraints for the complex block from Fig. 1 can be defined.

Due to the possibility of explaining the outcome of φ through assignments of variables that is present in it [16], [15], we can decompose the process of explaining the outcome of φ to the one of explaining a number of individual assignments in the counterexample. Below, we focus on explaining a single assignment, called an *explanation target*. The explanation target can be represented by an input or output assignment of any block structure: an FBD, a complex block, or a basic block. Initially, explanation targets come from applying the approach [15], but we also allow the situation where the user selects a custom explanation target to focus on a particular part of D , thus allowing more flexibility in explanation.

Definition III.6 (Cause). A set of assignments $C \subseteq X$ is a cause of a target t if there exists such sequence of sets of assignments from X , $Y_0, \dots, Y_m : C = Y_0, t \in Y_m$, where each $Y_{k+1}, k \in [0, m-1]$ extends Y_k with a single assignment $a'_{i, j} \in X$, there exists constraint $c^* \in C_D$ such that the formula

$$c^* \wedge \left(\bigwedge_{a_{i, j} \in Y_k} (v_{i, j} = v(a_{i, j})) \right) \rightarrow (v_{a'_{i, j}} = v(a'_{i, j})) \quad (1)$$

is valid, and $a'_{i, j}$ refers to the output variable of the basic block or connection to which c^* corresponds.

Intuitively, in every set Y_k from the definition above there exists a cause of the new assignment that is added to Y_k to obtain Y_{k+1} and at some extension step $q < m$, t should be added to get Y_{q+1} .

This definition can also be explained in terms of logical inference. Suppose that each statement is an assignment. Then the definition says that it is possible to infer t given a set of statements C if the allowed rules are limited to using input-output dependencies of each individual basic block or connection in the direction of the information flow.

Definition III.7 (Inclusion-minimal cause). $C \subseteq X$ is an *inclusion-minimal cause* of t if C is a cause of t and there is no $C' \subset C$ that is a cause of t .

Having these definitions, we say that to *explain the target* (or to find a cause of the target) means to find the union of its inclusion-minimal causes.

Here, it is worth mentioning the commonly accepted notion of causality from [17]. This work defines *actual* but-for causes of φ under some contingency in the model represented by structural equations. Meanwhile, our causes are not necessarily counterfactual. In other words, we deal with general causation, considering a set of assignments as a cause of t if it is sufficient to infer t in the given FBD.

As an example, consider the basic block AND from Fig. 2 and a counterexample of length 1. Assume that the explanation target is $t = (u_{14}, 0, 1)$, and $v_{12,1} = 1, v_{13,1} = 0$ (we denote logical values TRUE and FALSE as 1 and 0 respectively). To find out if any of input variables $U = \{u_{12}, u_{13}\}$ of AND are included in a cause of t , we, first, substitute c^* in (1) with $v_{14,1} = v_{12,1} \wedge v_{13,1}$. Then, as soon as U and t belong to the same basic block without delay, the only one constraint is required to infer the cause, hence, the length of the sequence of sets from Definition III.6 is two, where the first one is a cause. Now, we rewrite (1) as

$$(v_{14,1} = v_{12,1} \wedge v_{13,1}) \wedge \left(\bigwedge_{a_{i,1} \in C} (v_{i,1} = v(a_{i,1})) \right) \rightarrow (v_{14,1} = 0), \quad (2)$$

where i in the middle part is an index of the variable from U .

Having (2), the next step is to pick such assignments for C so that the relation (2) is valid and C is inclusion-minimal. In this example, there exists one such set of assignments $C = \{(u_{13}, 0, 1)\}$.

With the set of assignments U that can be potentially but not necessary added to Y_0 from Definition III.6 in (1), it is possible to set an *explanation scope*. In the previous example, the scope was defined by the input assignments of AND at step 1. Alternatively, if we explain t using input assignments of both OR blocks at the same step, constraints for all basic blocks shown in Fig. 2 and two constraints for the connections $\{v_{10,1} = v_{12,1}, v_{11,1} = v_{13,1}\}$ will be used in the extension procedure. Assume $v_{8,1} = 0$ and $v_{9,1} = 0$. Then the chosen scope produces the following inclusion-minimal cause: $C = \{(u_8, 0, 1), (u_9, 0, 1)\}$.

IV. ASSIGNMENT EXPLANATION ALGORITHM

The problem stated in Section III assumes that among all system assignments an inclusion-minimal cause of an explanation target should be found. To do this, first, we define a global explanation scope as the union of all input assignments of the FBD that the explanation target belongs to and assignments inside the FBD that have names of the gates which do not have incoming connections. Next, for any complex block it is a dubious help to see how, e.g., its output depends on its inputs,

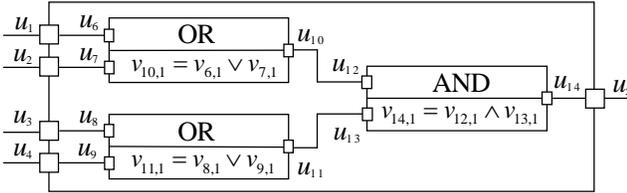


Fig. 2. Complex block B from Fig. 1 with constrains of its internal blocks for the first counterexample step defined. The full set of constraints for B for the first counterexample step is represented by the union of constraints for the depicted basic blocks and the set of connection constraints $C_c = \{v_{6,1} = v_{1,1}, v_{7,1} = v_{2,1}, v_{8,1} = v_{3,1}, v_{9,1} = v_{4,1}, v_{12,1} = v_{10,1}, v_{13,1} = v_{11,1}, v_{5,1} = v_{14,1}\}$, where $v_{i,j}$ is a value of variable u_i at counterexample step j , $i \in [1, |U|]$, $j \in [1, l]$.

Algorithm 1: Assignment explanation algorithm
explain.

Data: FBD D , counterexample X , explanation target $t \in X$

Result: set C – the union of all inclusion-minimal causes of t in D

```

1 if  $t$  corresponds to an input variable of  $D$  or a constant
  block input then
2   return  $\{t\}$  // this is a terminating cause
3 else if  $t$  is an input variable of a basic block in  $D$  then
4   /* follow the connection and add it to the tree */
5    $t' \leftarrow$  the assignment of the output variable at the
6   opposite end of the connection where  $t$  is located
7    $C_0 \leftarrow$  explain( $D, X, t'$ )
8   return  $C_0 \cup \{t\}$ 
9 else //  $t$  is an output variable of some basic block
10   $t_1, \dots, t_m \leftarrow$  causes found for the current basic
11  block type according to Table II
12   $C \leftarrow \{t\}$ 
13  /* recursively explain the assignments of the local
14  cause */
15  for  $i = 1$  to  $m$  do
16     $C_0 \leftarrow$  explain( $D, X, t_i$ )
17     $C \leftarrow C \cup C_0$ 
18  return  $C$ 

```

the analyst usually wants to know why such dependency takes place. Hence, in the explanation result we also include inclusion-minimal causes for every nested explanation scope if they exist for such scope. Thirdly, sometimes (for complex blocks) there can be more than one inclusion-minimal cause and it is the user who chooses the one of their interest, thus, we need to discover the union of all such causes.

A. FBD preprocessing

While complex blocks of the lowest hierarchy level in the FBD can be decomposed into basic blocks, this is not done in the original FBD. Thus, the values of the internal variables of such complex blocks are missing in a counterexample, although they are required for the explanation procedure. To obtain an extended counterexample, before running the algorithm for target t on FBD D , decomposition is performed automatically, and the full set of constraints for each of mentioned complex blocks is added to the full constraint set of the diagram C_D and the values of new variables are calculated for each counterexample step $s \in [1, s(t)]$.

This stage also provides a way to ensure that the complex block is parsed correctly, as otherwise, after execution, its output variable values could differ from the ones stated in the counterexample.

B. Recursive explanation

The algorithm is provided in Alg. 1 and is illustrated in Fig. 3, where the problem is to explain why output variable

u_5 of the complex block is FALSE at counterexample step s .

Recalling that an FBD itself is a complex block of complex blocks, to explain its output assignment, we need to find the output gate connected to the gate of the output of interest in the nested complex or basic block (Fig. 3, iteration 1). Then, if the found gate belongs to a complex block, the output assignment of such a block is explained through the underlying net of blocks, whereas to explain an output of a basic block, rules from Table II are utilized. As a result, we have a set of input assignments that are sufficient to make explained basic block output have its particular assignment – an inclusion-minimal cause (Fig. 3, iteration 2). If the obtained inputs have incoming connections, we continue the explanation procedure recursively in the same way; intermediate results from each step are added to the overall result set. After the algorithm terminates, the result composed of all inclusion-minimal causes in global and all the nested explanation scopes is obtained (Fig. 3, iteration N) and its graphical visualization described in Section V-E.

The time and memory complexity of the algorithm is $O(n \cdot s(t))$, where n is the number of variables in the FBD (including ones that belong to internal basic blocks). These estimates can be achieved if the result of each call of `explain` is memorized and not recomputed.

Statement IV.1. Alg. 1 finds the union of all inclusion-minimal causes of t .

For brevity, we leave this statement without a formal proof and only give an intuitive explanation why it is satisfied. The algorithm performs a backward (in terms of the information flow in the FBD) cone-of-influence analysis, seeking for all assignments that could possibly be the cause of t according to Definition III.6. Note that this definition requires that any cause must be sufficient to reach the target by inferring new assignments only in the direction of the information flow, which means that a search against this flow could reach all these causes. Moreover, the rules in Table II were specifically chosen to return the union of inclusion-minimal causes for an output of an FBD composed of an isolated basic block. For arbitrary FBD, Statement IV.1 could be proven by induction.

C. Discussion

The algorithm finds the union of all inclusion-minimal causes for the target within the global and all the nested explanation scopes. Whereas our definition does not consider the system as a whole at every extension step, there may exist such combinations of constraints that generally restrict combinations of values of input assignments of basic blocks. For instance, consider Fig. 4, where signals merge in a common ancestor if traversing backwards from $a_{2,s}$.

Another case is an explanation result of basic block `COUNT` that includes all the input assignments and is inclusion-minimal. It follows that further explanation results will include causes for all of the input assignments of `COUNT` which may be irrelevant with regard to the system structure. Assume that the output of this block equals 4 and is connected to

an input of “ \leq ” block, where it is compared to 6. Having this context in mind, we know that inputs of our interest are FALSE because two more TRUE signals are required to change the “ \leq ” output. This example shows that without the knowledge of how the result of this block is used further, it is hard to say should we consider its TRUE or FALSE signals in the explanation. Nevertheless, assignments that are not included into the result of the algorithm are insufficient in the target explanation, therefore, the user still gets information that significantly reduces an FBD area being analyzed.

V. IMPLEMENTATION

The implementation of the algorithm described in Section IV was incorporated into the tool OERITTE with the user interface developed to aid the analyst in the debugging process.

A. Input data

The tool accepts a NuSMV model, an LTL formula and a counterexample for the provided formula on the provided model as input. A restricted, but, nonetheless, already usable according to our practical experience, subset of NuSMV and LTL is supported. Below are the main limitations:

- The main module of the NuSMV model is restricted to declarations of input variables and nested modules.
- In other modules, each internal variable must be declared with `init` and `next` operators. These assignments must be deterministic (set notation $\{ . . . \}$ is disallowed). `INIT` and `TRANS` declarations are not allowed.
- `DEFINE` declarations are not allowed to use the `next` operator.
- Only Boolean and integer scalar types are supported.
- Inputs of the NuSMV modules should be annotated with their types in form “`varName : type`”, where `type` is `boolean` for Boolean and any integer interval in form `start . . end` for integer, e.g., `0 . . 100`.
- In LTL formulas, bounded operators (e.g., `G[0, 3]`) and past time operators (e.g., `H`) are not supported.

TABLE II
BASIC BLOCK EXPLANATION RULES OF FINDING LOCAL INCLUSION-MINIMAL CAUSES. ASSUME THAT THE REQUEST IS THE EXPLANATION TARGET REPRESENTED BY THE TUPLE (u, v, s) , WHERE v IS THE VALUE OF u AT STEP s , AND A SET OF ASSIGNMENTS REPRESENTING A CAUSE IS RETURNED.

Block	Rule
Logical AND	If v is TRUE, then return all the block input assignments for step s , else return only input assignments that are FALSE at s .
Logical OR	If v is FALSE, then return all the block input assignments for s , else return only inputs that are TRUE at s .
CHOICE	Return all the condition assignments prior to and including the one that is satisfied at s and its output assignment.
DELAY	Return input assignment from step $s - 1$.
Others	Return all the input assignments for step s .

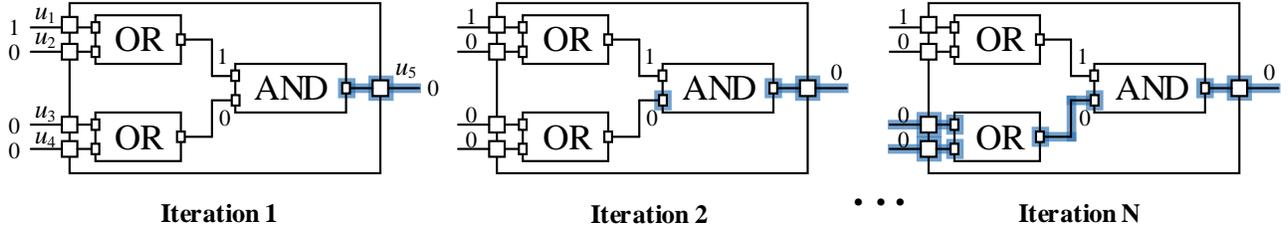


Fig. 3. Illustration of the assignment explanation process. Digits above the connections show values of the transmitted signals.

B. Encoding NuSMV modules as complex blocks

The aforementioned determinism assumption is required to represent NuSMV modules as complex blocks since our basic blocks are purely deterministic. The input variables of the complex block correspond to input variables of the module, and the output variables correspond to its internal variables and `DEFINE` declarations (the absence of `next` operators inside them allows treating these declarations as if they were internal variables). Logical and arithmetic NuSMV operations are directly transformed to basic blocks listed in Table I. To handle delays introduced with the `next` operator, we create a delayed version of each input variable by passing it through a `DELAY` block. Each output variable is then wired to a `CHOICE`, which, depending on whether this is the first cycle, outputs the `init` or the `next` expression for this variable: `init` expressions always use undelayed variables, while `next` expressions may use both undelayed and delayed ones.

C. Main window overview

The main view of OERITTE² is presented in Fig. 5. Here two areas are used to depict an LTL formula tree (Fig. 5a) and the visualized version of an FBD (Fig. 5b) (hereinafter, the *diagram*). The table below them (Fig. 5c) shows the values of all the system variables for every counterexample step, and the discovered failure causes for the diagram and the LTL formula can be found in lists (Fig. 5d) and (Fig. 5e) correspondingly. Panel (Fig. 5f), containing the trace steps sequence, is clickable and utilized for navigation: once an element is activated, both the diagram and the formula tree are evaluated according to the chosen step, so that all variables in both diagrams are assigned with values defined by the counterexample step, hence, all the

²<https://github.com/ShakeAnApple/cxbacktracker/>

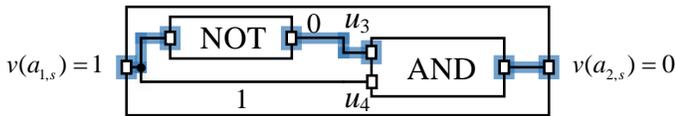


Fig. 4. Illustration of a non-intuitive algorithm result at step s due to gates u_3 and u_4 having a common ancestor u_1 . The path of an explanation process for $a_{2,s}$ is highlighted with bold blue. Here block `NOT` inverts the signal from gate u_1 , thus, block `AND` computes the expression $v(a_{2,s}) = v(a_{1,s}) \wedge \neg v(a_{1,s})$, which is always false. The algorithm will result in the path in bold blue and $\{(u_1, 1, s)\}$ will be the inclusion-minimal cause of u_2 .

nodes in the LTL formula tree are calculated and all the system modules are executed.

OERITTE has two groups of features: LTL formula explanation and diagram interpretation. For the first, we implemented the cause identification algorithm from [15]. The LTL formula parse tree (see Fig. 5a) is located on the leftmost panel of the main view. Depending on the calculation result of the branch, the nodes of the tree are colored in red, green and grey for `FALSE`, `TRUE` and an arithmetic result respectively. The “explain formula” button forces the causes of the formula value at the current step to appear in the list on the right and be colored in the value table. By default, the formula is explained for the first step (step 0 in the tool) with the first diagram evaluation.

D. Complex block

In Fig. 6, an example of such visual representation of a NuSMV module is given. This block is a part of a diagram in the explanation mode. Each block has a *name* and a *type* (Fig. 6a). Two sets of pins on the left (Fig. 6b) and right (Fig. 6c) sides are the block’s inputs and outputs that together form its interface. If the diagram is not in the explanation mode, hovering the pin will trigger a tooltip with the variable name appearing. Blocks with the single input or output pin on the left and right sides of the diagram represent a *system interface* – input and output variables of the whole system. Lines connecting module inputs and outputs correspond to connections between the blocks. Output variable values are shown near the output connecting points of these lines, whereas input variable values of the blocks are placed near the input connecting points. Input negation is represented as a circle instead of a square (Fig. 6d).

E. Explanation mode

Visualization of the assignment explanation algorithm result is shown in Fig. 5 and is done as follows. To switch to the explanation mode, one needs to click the desired pin at the desired step or explanation target on the diagram view, which triggers the aforementioned backward explanation process that results in a set of assignments belonging to a calculated cause. To display such a set on the diagram view, we hide the time dimension and highlight edges that connect output and input pins from the common set of causes with blue. At the same time, if some variable is a cause at several time steps, the pin representing this variable is provided with a tooltip where

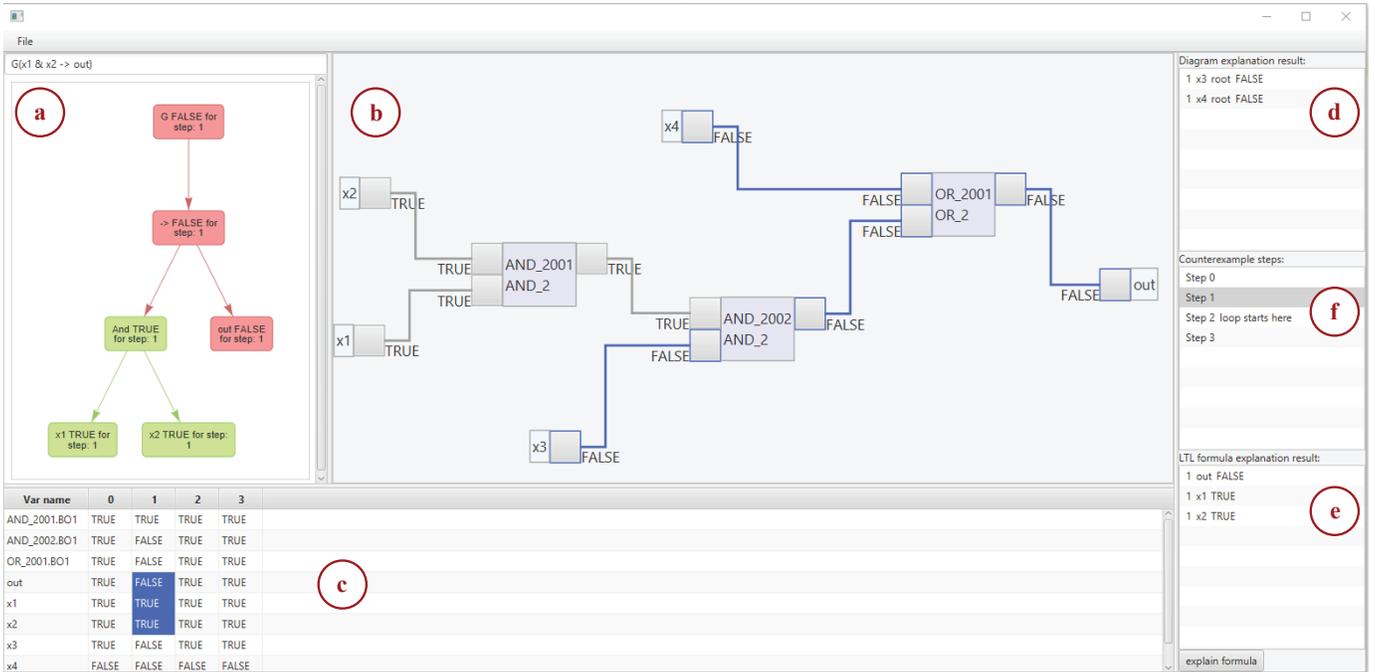


Fig. 5. Main view of the tool in the explanation mode.

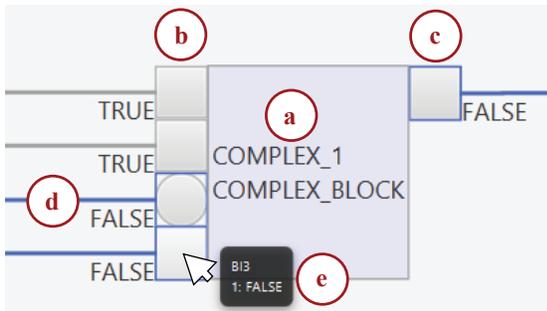


Fig. 6. Visual representation of a complex block in the explanation mode.

all its values included in the cause are displayed in form “stepNumber:value” (Fig. 6e). Together with graphical visualization, list (Fig. 5c) shows *terminating assignments*, i.e., assignments, whose gates do not have incoming connections and belong to the input interface of the system. They are displayed in form “stepNumber varName blockName value”. This is the output of our interest, which includes evaluation paths in the system that influenced the chosen assignment to have its value. We argue that showing such paths and not only the assignments of an inclusion-minimal cause provides useful visual information to the analyst.

VI. CASE STUDY

The main feature of OERITTE is a diagram view where it is possible to visually navigate through the causes of the assignment of interest to find out the roots of the system malfunction. That is why the tool should be evaluated by the analyst interacting with it to discover a problem in the real system. As the latter, we use the mode selection logic

introduced in [4]. It contains an actual design issue, revealed by using model checking in a practical nuclear industry project.

The logic is schematically represented in Fig. 7 and is used to select one of two operational modes, *mode_a* or *mode_b*. The operator can select the mode using the *set_a* or *set_b* command. In addition, if *mode_a* is active, and the signal *c* then changes to true, the mode is automatically switched to *mode_b*. The processing order for the feedback loop is specified using a cycle delay block.

One functional requirement for the logic is that the two modes shall not both be active at the same time. To accommodate for the cycle delay, the analyst specifies the LTL property $\mathbf{G}\neg((\text{mode_a}\wedge\text{mode_b})\wedge\mathbf{X}(\text{mode_a}\wedge\text{mode_b}))$, i.e., *mode_a* and *mode_b* shall not both be on for two consecutive cycles. This specification is violated and NuSMV generates a counterexample. The result of the explanation of the LTL formula is that both *mode_a* and *mode_b* are TRUE at the steps 2 and 3.

As this explanation does not contain information about the reasons for such system behavior, the analyst considers the diagram. Having both modes active on the time steps 2 and 3, the analyst clicks on the *mode_b* output at the time step 3, and the explanation for that value is shown. Almost all diagram connections are highlighted, but, by hovering the mouse pointer over the pins, the analyst notices that the second flip-flop was set once at step 1 and never reset (Fig. 8a). The analyst then notices in the “diagram explanation result” panel that the *set_a* command on time step 2 does *not* have an effect, when, in fact, it should reset the *mode_b* flip-flop a cycle later. Then, the question is why the second flip-flop did

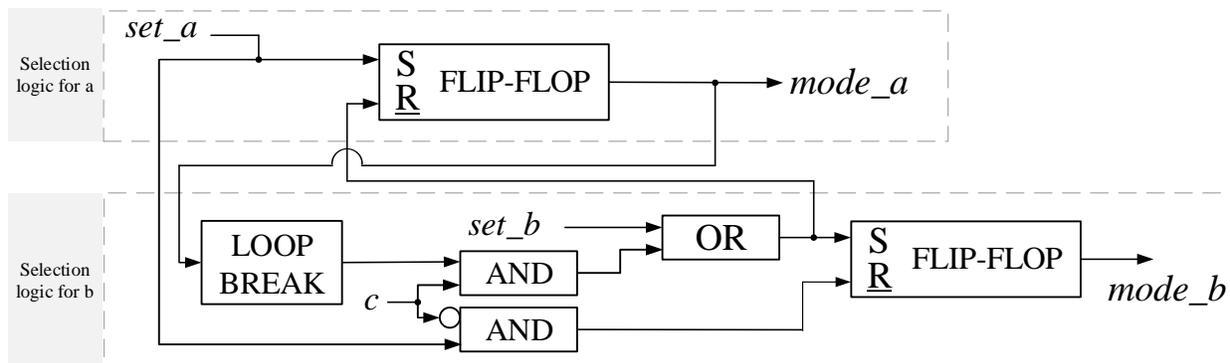


Fig. 7. Case study: mode selection logic [4]. *Flip-flop* blocks produce TRUE if $S \wedge \neg R$ and FALSE otherwise. If both S and R are FALSE then the value from the previous time step is used as output. A *loop breaker* works as a signal delay so that its output at the step N equals to its input at the step N-1 and the predefined value for the step 0, which is FALSE in our case. A *circle* means that a block consumes the inverted input.

not reset its output signal at the steps 2 and 3. Clicks on its reset input at steps 2 and 3 outlines the path to c (Fig. 8b) and to set_a (Fig. 8c) correspondingly, therefore, the analyst is shown that signal c is the cause at time step 2, and inactive set_a command is the cause at step 3.

Combined, these results help the analyst realize that the issue is about an extremely short set_a signal pulse arriving at a very specific time – one cycle before the signal c resets. (As explained in [4], the issue may also be counter-intuitive to the designer because invoking set_a at a time when c is active is not necessarily feasible.)

VII. RELATED RESEARCH

There exist several approaches to counterexample explanation. Some works [18], [16], [15] focus on the failed property itself and, among mentioned, [15] is the most advanced one, which was directly implemented in our work in the LTL formula explanation part. It generalizes an algorithm from [16] that explains a false outcome of LTL formula f on a given counterexample. This is done by recursively finding assignments that are sufficient to cause the value of a subformula of f on a given counterexample step, starting from the entire f on the first step.

However, the problem of the research direction of these approaches is that it is only possible to get an explanation with the variables that are actually included in the LTL formula. Unfortunately, these values may be influenced by others and the real cause of the violation of f may be still hidden (as it was shown in Section VI), requiring the analyst to manually examine the formal model.

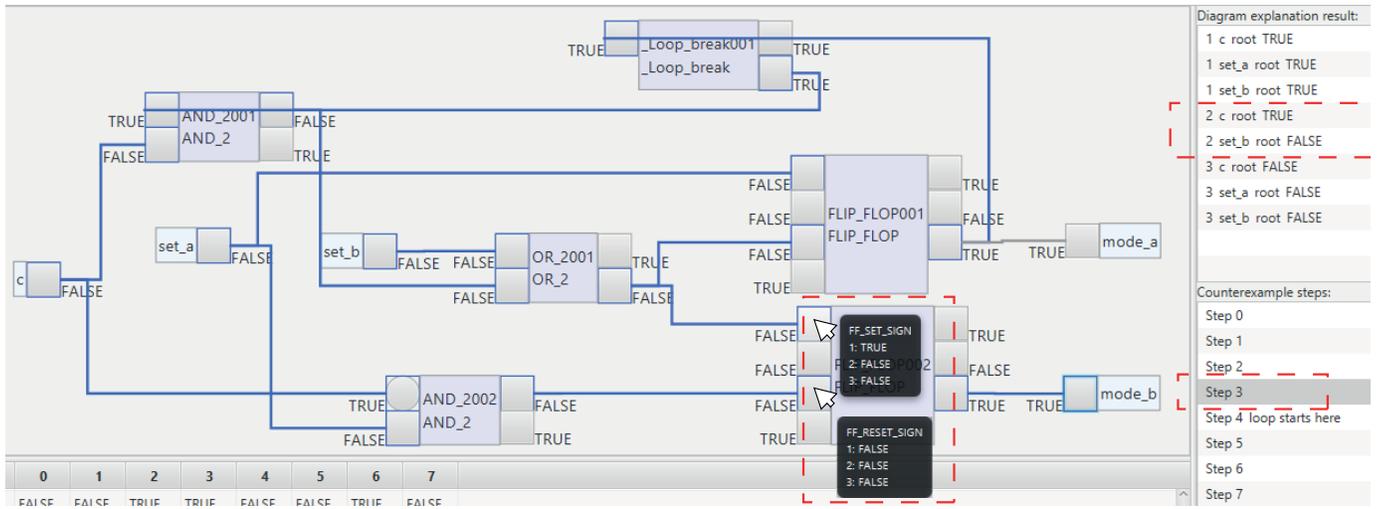
Direct analysis of a system under verification is considered in [19], [20], [21], [22]. These approaches require more than one system trace and, therefore, additional executions of supporting programs to acquire necessary counterexamples or traces that correspond to successful model runs. By contrast, in [23] a single failure trace is enough to reason about the causes of the error, although here a counterexample is represented by a sequence of executions of program statements and such sequence is unknown in our scenario. Compared to the works above, our approach needs only one counterexample

from a model checker to explain the failure. Despite being often developed in form of modular diagrams, I&C systems require a more sophisticated visual approach for localizing failures which is provided by OERITTE.

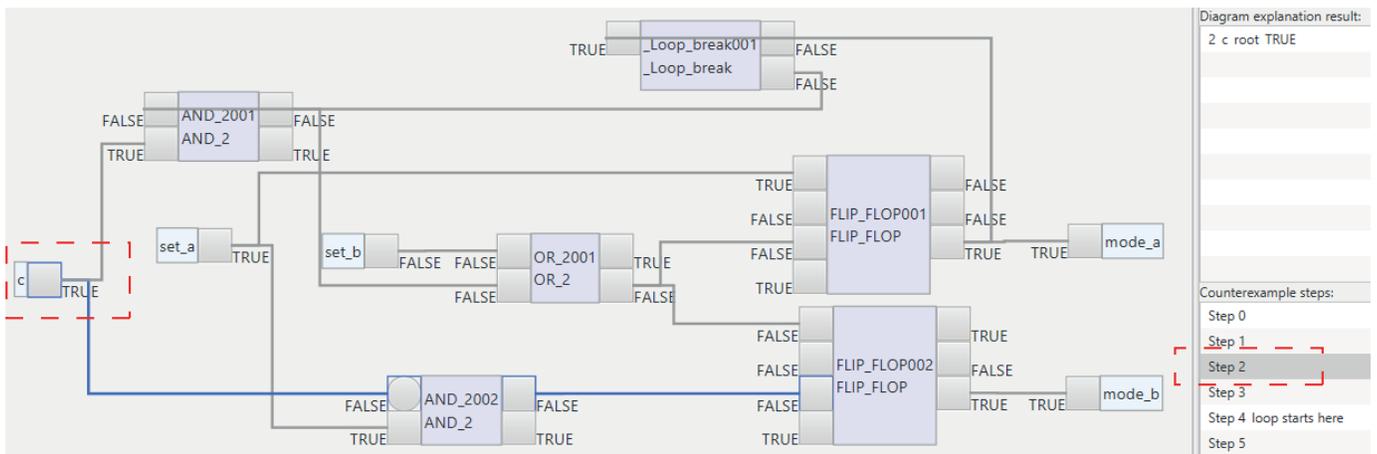
Works [5], [24], [25] are related to counterexample visualization. In [5] counterexamples are visualized using timing diagrams, however, “model view” is arguably [24] the most usable representation. MODCHK [11] is a graphical front-end for NuSMV that visualizes the counterexamples by animating the function block diagram. For use in nuclear applications, MODCHK also supports vendor-specific, non-standard function block types. Similarly to MODCHK, in [25], counterexamples for IEC 61499 block diagrams are shown using a simulation model of the controlled process. Another graphical tool is the Simulink Design Verifier [12], which generates a simulation test case from the counterexample. Being able to visualize, the aforementioned tools do not support explanation of specifications failures.

A viewpoint that considers a structural approach to counterexample explanation and that is the closest to ours is taken in [26], where the considered problem is to explain an output of an observer (synchronous program connected to the system under verification), which is a single value. However, here only a limited set of basic blocks is supported, the approach is applied only to STANCE models and lacks flexibility in explanation scope determination. Another difference is that definitions in [26] are formulated over the paths in the system, where all those starting in system inputs require to obtain an activation condition formulae before the explanation starts. As a result, sets of paths representing a cause of the observer output is calculated.

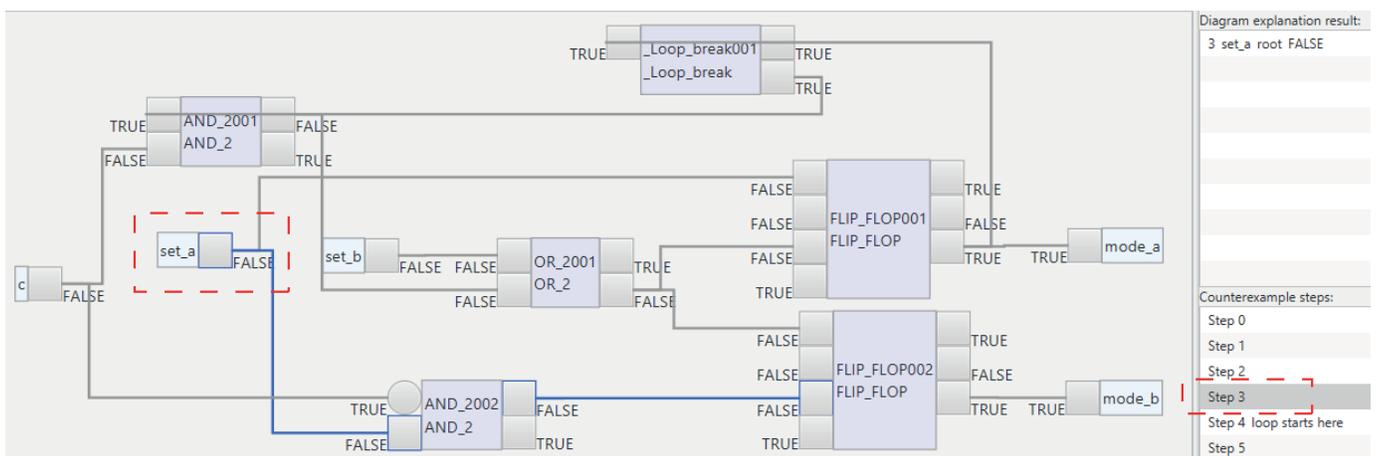
In our work, we synthesize the ideas of the works [16], [15], [26] in the following form: we borrow the idea of explaining an LTL formula with particular values of variables in it, but also let the user get explanations for each such particular values. We solve the comprehensibility problem by presenting only one set of paths and letting the user visually navigate through the FBD and the counterexample to get more focused explanations. Therefore, despite steps towards more user-



(a) Explanation for the `mode_b` output for the third step. To save space, in this figure we combined two screenshots with different tooltips being shown.



(b) Explanation for the reset input of the second flip-flop block for the *second* step. `c` is shown to be the cause of this signal not activated.



(c) Explanation for the reset input of the second flip-flop block for the *third* step. `set_a` is shown to be the cause of this signal not activated.

Fig. 8. Counterexample explanation for the case study logic. The points of the analyst's attention are highlighted with red dashed rectangles and are not displayed by the tool.

friendly counterexample visualization and explanation have already been made, OERITTE is the only tool that combines explanation techniques into a consistent infrastructure.

VIII. CONCLUSION

In this paper, we have presented a novel counterexample explanation algorithm and an open-source tool, OERITTE, which implements it together with a known LTL formula explanation algorithm [15] and offers graphical backward counterexample analysis. Inspired by works [16], [26], the tool provides methods and visual elements supporting explanations in terms of both the LTL formula and the model (FBD) in a form of paths from causes to the target values that they explain. It is worth mentioning that our approach can be used to explain the results of finite computations even when they are produced by models which are not FBDs. The elements of the user interface of OERITTE and their functionality address the challenges mentioned in the introduction. The variables of the diagram (Fig. 6b, Fig. 6c) and the LTL formula tree (Fig. 5a) are evaluated according to a chosen counterexample step, which addresses the first challenge. The second challenge is covered by the possibility to retrieve causes of the failure using only the formula structure, where the LTL formula tree (Fig. 5a), the button “explain formula” and highlighted values (Fig. 5c) help with visualization. Finally, the diagram (Fig. 5b) combined with the presented method of individual assignment explanation assists in the analysis of the system as a whole.

As shown by the case study, the current version of OERITTE already avails, moreover, it has potential in reaching a wider audience in the future. Our future work includes broadening the scope of supported models and developing the counterexample explanation methods for timed and probabilistic automata. Another enhancement that would enlarge the range of models and specifications to be analyzed is relaxing input data restrictions, i.e., introducing support for bounded and past time LTL operators as well as removing necessity to specify the data types in a NuSMV model.

ACKNOWLEDGMENTS

This work was supported, in part, by the Finnish Research Programme on Nuclear Power Plant Safety 2018-2022 (SAFIR 2022) and by the Government of the Russian Federation under Grant 08-08.

REFERENCES

- [1] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, Cambridge, Massachusetts, 1999.
- [2] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frollov, E. Reeber *et al.*, “Replacing testing with formal verification in Intel[®] CoreTM i7 processor execution engine validation,” in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 414–429.
- [3] K. Nakahori and S. Yamaguchi, “A support tool to design IoT services with NuSMV,” in *2017 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2017, pp. 80–83.
- [4] A. Pakonen, T. Tahvonen, M. Hartikainen, and M. Pihlanko, “Practical applications of model checking in the Finnish nuclear industry,” in *10th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human Machine Interface Technologies (NPIC & HMIT)*. American Nuclear Society, 2017, pp. 1342–1352.
- [5] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, P. Seong *et al.*, “FBDVerifier: Interactive and visual analysis of counter-example in formal verification of function block diagram,” *Journal of Research and Practice in Information Technology*, vol. 42, no. 3, p. 171, 2010.
- [6] E. Németh and T. Bartha, “Formal verification of safety functions by reinterpretation of functional block based specifications,” in *International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*. Springer, 2008, pp. 199–214.
- [7] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, “Applying model checking to industrial-sized PLC programs,” *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.
- [8] G. E. Gelman, K. M. Feigh, and J. Rushby, “Example of a complementary use of model checking and agent-based simulation,” in *2013 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 2013, pp. 900–905.
- [9] International Electrotechnical Commission, *International Standard IEC 61131-3:2013: Programmable Controllers. Part 3: Programming Languages*. IEC, 2013.
- [10] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2016.
- [11] A. Pakonen and K. Björkman, “Model checking as a protective method against spurious actuation of industrial control systems,” in *27th European Safety and Reliability Conference (ESREL 2017)*. Taylor & Francis Group, London, UK, 2017, pp. 3189–3196.
- [12] “Simulink design verifier,” <https://www.mathworks.com/products/simulink-design-verifier.html>, last accessed 16 Sep 2020.
- [13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “NuSMV 2: An OpenSource tool for symbolic model checking,” in *International Conference on Computer Aided Verification (CAV)*. Springer, 2002, pp. 359–364.
- [14] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, P. Seong *et al.*, “FBDVerifier: Interactive and visual analysis of counter-example in formal verification of function block diagram,” *Journal of Research and Practice in Information Technology*, vol. 42, no. 3, p. 171, 2010.
- [15] A. Pakonen, I. Buzhinsky, and V. Vyatkin, “Counterexample visualization and explanation for function block diagrams,” in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 2018, pp. 747–753.
- [16] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Treffer, “Explaining counterexamples using causality,” *Formal Methods in System Design*, vol. 40, no. 1, pp. 20–40, 2012.
- [17] J. Y. Halpern, “A modification of the Halpern-Pearl definition of causality,” *arXiv preprint arXiv:1505.00162*, 2015.
- [18] A. Ek, “Explanation of counterexamples in the context of formal verification,” B.S. Thesis, Uppsala University, Department of Information Technology, 2016.
- [19] A. Groce and W. Visser, “What went wrong: Explaining counterexamples,” in *International SPIN Workshop on Model Checking of Software*. Springer, 2003, pp. 121–136.
- [20] A. Groce, D. Kroening, and F. Lerda, “Understanding counterexamples with explain,” in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 453–456.
- [21] S. Leue and M. T. Befrouei, “Counterexample explanation by anomaly detection,” in *International SPIN Workshop on Model Checking of Software*. Springer, 2012, pp. 24–42.
- [22] F. Leitner-Fischer and S. Leue, “Causality checking for complex system models,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 248–267.
- [23] C. Wang, Z. Yang, F. Ivančić, and A. Gupta, “Whodunit? causal analysis for counterexamples,” in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2006, pp. 82–95.
- [24] K. Loer and M. D. Harrison, “An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation,” *Automated Software Engineering*, vol. 13, no. 4, pp. 469–496, 2006.
- [25] S. Patil, V. Vyatkin, and C. Pang, “Counterexample-guided simulation framework for formal verification of flexible automation systems,” in

2015 *IEEE 13th International Conference on Industrial Informatics (INDIN)*. IEEE, 2015, pp. 1192–1197.

- [26] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, “Paths to property violation: A structural approach for analyzing counter-examples,” in *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. IEEE, 2010, pp. 74–83.