Ovsiannikova, Polina; Buzhinsky, Igor; Pakonen, Antti; Vyatkin, Valeriy

Oeritte : User-Friendly Counterexample Explanation for Model Checking

Published in:
IEEE Access

DOI:
10.1109/ACCESS.2021.3073459

Published: 15/04/2021

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

Please cite the original version:
Ovsiannikova, P., Buzhinsky, I., Pakonen, A., & Vyatkin, V. (2021). Oeritte : User-Friendly Counterexample Explanation for Model Checking. *IEEE Access*, *9*, 61383-61397. Article 9405616.
https://doi.org/10.1109/ACCESS.2021.3073459

IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 10, 2021, accepted April 6, 2021, date of publication April 15, 2021, date of current version April 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3073459

Oeritte: User-Friendly Counterexample Explanation for Model Checking

POLINA OVSIANNIKOVA[1,2], IGOR BUZHINSKY[1,2], ANTTI PAKONEN[3], AND VALERIY VYATKIN[1,2,4], (Member, IEEE)

[1]Computer Technologies Laboratory, ITMO University, 197101 Saint Petersburg, Russia
[2]Department of Electrical Engineering and Automation, Aalto University, 02150 Espoo, Finland
[3]VTT Technical Research Centre of Finland Ltd., 02044 Espoo, Finland
[4]Department of Computer Science, Computer and Space Engineering, Luleå Tekniska Universitet, 971 87 Luleå, Sweden

Corresponding author: Polina Ovsiannikova (polina.ovsiannikova@aalto.fi)

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

ABSTRACT Thorough verification is a part of the design process of instrumentation and control systems if they must comply with crucial safety requirements. Model checking can be applied to the formal model of such a system to reason about its correctness based on the specification provided. When a violation occurs, the model checking tool outputs the proof of the violation in the form of a failure trace, which represents a state sequence of system model transitions where the requirement does not hold. This sequence, however, even for modular systems, is a mere table of values. Due to the lack of any insights into the inner model processes and structures that caused a problem, the debugging process of the formal model becomes time and effort consuming. The tool presented in this paper, Oeritte, is aimed at assisting the analyst in this challenge. It implements a method for automatic visual counterexample explanation which includes reasoning both over the falsified LTL formula and over the NuSMV function block diagram of the formal model of the system. The tool is applied to an industrial-sized safety control system of a nuclear power plant.

INDEX TERMS Counterexample explanation, counterexample visualization, function block diagram, NuSMV, user-friendly model checking.

I. INTRODUCTION

One of the most reliable approaches to ensure the correctness of an instrumentation and control (I&C) system is a formal verification technique called model checking [1]. It is applied in avionics [2], [3], automotive driving industry [4]–[6] and for verification of I&C systems of nuclear power plants [7]–[10]. Even though model checking is able to scrutinize the whole state space of a system model in search for deviations, several disadvantages separate it from being spread ubiquitously. The first challenge is related to formal model inference. Verification results are valuable only if the right formalism is chosen for the system domain and the model's behavior corresponds to the behavior of the original system in time of model checking [11]. Then, exploring all model's behaviors may be computationally demanding. This issue is considered in [12] and algorithms to reduce the computational complexity are being developed [13]–[15].

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

Another downside of the approach, which we tackle in this paper, is the time and effort consuming process of errors localization in the model being verified.

The overall model checking process consists of three stages: (1) a formal model of a system is created, (2) the formal model together with the temporal logic specification is sent as an input to a verification tool, such as NuSMV [16] or SPIN [17], (3) the tool informs the user whether the specification is satisfied. If it is not, the tool produces a *counterexample*, which is a sequence of the states of the model where the specification does not hold. More precisely, each element of this sequence comprises the values of the model variables. However, counterexamples do not show the structure of the system or the internal dependencies of its variables. Therefore, especially for I&C systems, which tend to be modular and complex, the analyst gets the daunting task of determining the cause of the problem.

This article extends the work [18], where we presented Oeritte, a tool for visual counterexample explanation. Oeritte takes a modular formal model of a system in the NuSMV format together with its violated requirements and provides a graphical automatic counterexample explanation using both the violated property and the formal model. Among all, the current paper complements the previous work with discussion on causality, a formal proof of the fact that the algorithm solves the formulated problem, and an industrial-sized case study.

The rest of the paper is structured as follows. Preliminaries are given in Section II. Section III provides the formulation of the notion of a cause and the problem of counterexample explanation. The algorithm that solves the latter is given in Section IV. Section V overviews the tool that implements the proposed approach, and Section VI evaluates the approach experimentally using an industrial-sized formal model. Then, the related research is reviewed in Section VII and Section VIII discusses the results. Section IX concludes the paper and overviews the future work directions.

II. PRELIMINARIES

A. FUNCTION BLOCK DIAGRAMS

In this paper, by a model, we mean a *function block diagram* (FBD).[1] Essentially, an FBD is a set of interconnected *function blocks*, where each function block infers the values of its output variables by performing a particular transformation on its input and internal variable values. Each function block in an FBD is an instance of a *function block type*, which can be viewed as a Mealy machine [20] that defines such a transformation.

We consider our models to have discrete time, and on each time instant the variables of all the included function blocks are assigned new values. The current *time step*, which is an integer, is not directly available in the model as a variable, but will be useful to reason about execution sequences of the model. The execution semantics is synchronous: the output values of each block are functions of its input values from the current or the previous time step (custom initialization may be applied on the first time step, and input variables without incoming connections are assigned the same default values at all time steps) and the signals are propagated through connections instantly. Delay function blocks help to prevent infinitely fast information flow in FBDs with feedback loops.

To design an FBD one might use such graphical tools as MODCHK [21] or Simulink Design Verifier [22], but they can also be encoded textually, e.g., with languages of model checkers, such as NuSMV [16]. Fig. 1 shows a simple example of an FBD.

The basic concept of an *assignment* is required to start a formal description of an FBD D with its set of variables $U = \{u_1, \ldots, u_n\}$.

Definition 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 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 an index $i \in [1, n]$ for u, and we denote the assignment of $u = u_i$ at time step j as $a_{i,j}$.

[1] FBDs are one of the graphical programming languages officially supported by IEC 61131-3 [19], but the contributions of this work are not limited to FBDs as specified in this standard. We use them in a more general sense as described in this subsection.

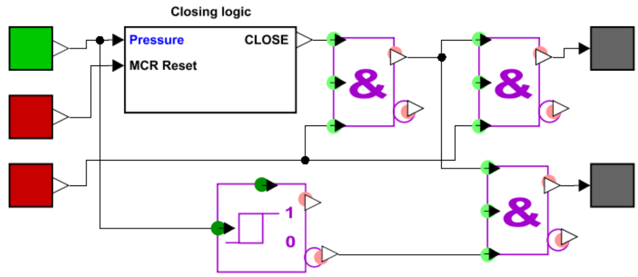


FIGURE 1. An example of an FBD implemented in MODCHK. Solid squares on the left and right sides of the diagram stand for its input and output variables.

Information between two variables is transmitted through a connection.

Definition 2 (Connection): A connection c is a tuple $(u_i, u_j, N), i, j \in [1, n]$, which is defined by two different variables of the same type and an indicator of connection inversion N. Connection c can be also represented with a set of *connection constraints* $C_c = \{v_{i,s} = v_{j,s} \mid s \in [1, l]\}$, if $N = 0$, or $\{v_{i,s} = \neg v_{j,s} \mid s \in [1, l]\}$, if $N = 1$ and the variables are Boolean, where s is a counterexample step.

The connections between the variables are directed, i.e., the information can only flow from outputs of some blocks to inputs of some other blocks. Multiple outgoing connections are allowed but multiple incoming connections are not.

The fundamental notion for FBD is a *block* that is defined by its type that determines its set of constraints over the variables of the block and its interface. Formally:

Definition 3 (Counterexample): A *counterexample* X of length l is a set of assignments of the variables from U for each time step j: $X = \{(u_i, v_{i,j}, j) \mid i \in [1, n], j \in [1, l]\}$.

Essentially, a counterexample is a sequence of model states. As we said earlier, an FBD has discrete time and updates all its variables (its state) exactly once during each time step. Therefore, we can say that each state of the counterexample represents the values of all the model's variables at a particular time step.

Definition 4 (Constraint): A *constraint* over variable $u_i \in U$ and a set of variables $\{u'_1, \ldots, u'_k\} \subseteq U$ defined on a time step s of X is a Boolean expression $v_{i,s} = f(v_{1,s}, \ldots, v_{k,s})$, where f is a function.

Definition 5 (Block type): A *block type* is a tuple (I, O, C_B), where I and O are the sets of input and output variables that form *a block interface*, and C_B is a set of constraints over the values of the variables in O with regard to the values of the variables in I, defined for a range of counterexample steps $[1, l]$.

Definition 6 (Block instance): A *block instance* (or *block*) of type T with name N is a tuple (T, N).

Block type determines C_B together with two sets I and O, while names, given to the blocks, differentiate their instances of the same types. We say that two blocks are connected if an output variable of one of them is connected to an input variable of the other. There are two kinds of blocks, *atomic* and *modular*, which differ in their set of constraints. Table 1 overviews the set of atomic block types that are used in the current work.

TABLE 1. Atomic blocks used in the current paper for the construction of an FBD.

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

Definition 7 (Atomic block constraints): Every atomic block B from Table 1 except for `DELAY` with k input variables and one output variable 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 f_B is determined by the type of each atomic 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 $u_1, .., u_3$ are variables of a particular instance of `DELAY`.

Thus, intuitively, every atomic block corresponds to an atomic operator or a simple function. 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 the length of a counterexample. Available logical operators together with connections inversion allow the formulation of any Boolean function.

Below we describe simple functions from the group "other" from Table 1.

- `DELAY` allows the implementation of feedback loops. At the first execution cycle, some default value is assigned to the output of this block, then, every subsequent execution makes the output take the value of the input variable from the previous step. Therefore, `DELAY` has two input variables (one for a predefined default value and one for the current input signal) and one output.
- With `CHOICE` it is possible to implement a cascade "if" assignment for a variable. It has one output variable, an input variable for every clause and another input for every value that should be assigned to the output if the corresponding clause is satisfied.
- `COUNT` takes several Boolean signals and outputs a total number of ones that are `TRUE` at the current step.
- The `ASSIGN` block implements the identity function: the output is the same as the input.

FBD D consists of interconnected *modular blocks*, which are decomposed into nets of blocks of both kinds, thus allowing implementation of more sophisticated calculations.

Definition 8 (Modular block constraints): Let B be a modular block with its set of internal blocks M and set of internal connections Σ. Then, a set of constraints for B is $C_B = \{C_m \mid m \in M\} \cup \{C_\sigma \mid \sigma \in \Sigma\}$.

In I&C systems design, one may use libraries with basic blocks that are not decomposable, e.g., flip-flops, logical operators with more than two arguments, etc. Such basic blocks in our case are represented as modular blocks with atomic blocks constituting their internal net. An FBD D itself is a special modular block of the highest level of the hierarchy, that may contain both modular and atomic blocks, therefore, the set of constraints C_D is defined for D as well.

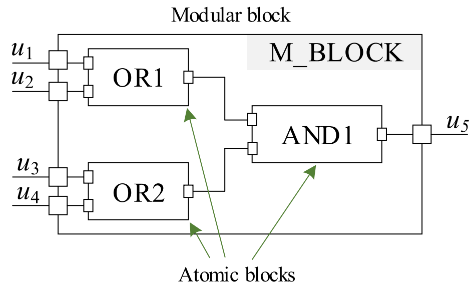


FIGURE 2. A modular block with name `M_BLOCK`, input interface $I = \{u_1, u_2, u_3, u_4\}$ and output interface $O = \{u_5\}$ that encodes function $u_5 = (u_1 \vee u_2) \wedge (u_3 \vee u_4)$. It includes three interconnected atomic blocks with names `OR1`, `OR2`, `AND1`.

Definitions 6 and 5 are shown in Fig. 2. Fig. 3 illustrates how a set of constraints can be defined for a block from Fig. 2 for a counterexample of length 1.

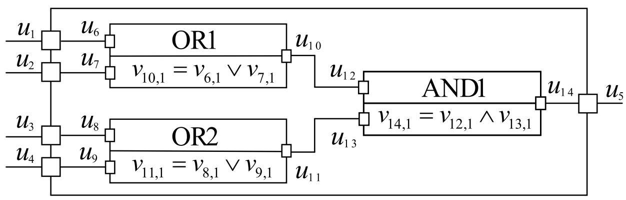


FIGURE 3. Modular block B from Fig. 2 with the constrains of its internal blocks for the first counterexample step defined. The full set of constraints for B for the first counterexample step is represented by the union of constraints for the depicted atomic blocks and the set of connection constraints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

B. LINEAR TEMPORAL LOGIC

Boolean logic provides a set of operators sufficient to formulate propositions about the single model *state*, or variable values of the model at a particular time step. While this is enough for Boolean circuits, dynamic systems tend to evolve through time and their variable valuations may depend on the previous model state(s). Temporal logics allow such time specifications over state sequences of a model (or model traces). Here, we consider the requirements formulated with *linear temporal logic* (LTL), which extends Boolean logic with a set of temporal operators. Below, we list the examples of the most used ones, supposing φ_1 and φ_2 to be the LTL formulas:

- **G** φ_1 ("globally"): φ_1 must be true on the entire trace of the model;
- **F** φ_1 ("finally"): φ_1 must hold eventually;
- φ_1 **U** φ_2 ("until"): φ_1 must be true until φ_2 is true, and the latter is required to eventually happen;
- **X** φ_1 ("next"): φ_1 must be true for the next state.

A *valid state sequence* of a model starts in one of the model initial states and its every pair of adjacent states belongs to the transition relation of the model. An LTL formula is satisfied for the model if 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* (or a failure trace) that demonstrates its violation.

We consider linear counterexamples, represented by valid state sequences of the model, which most of the model checkers produce as an output. Such counterexamples may take finite or lasso-shaped form, where, in the latter, a failure trace consists of a finite prefix and a loop.

Models of industrial systems contain dozens of variables, nested modules and complex dependencies [23]. In this case, being a mere table of values, counterexamples offer a limited help in localizing the issues, consuming time and resources for their decoding. Having an FBD as a model, visualization techniques especially avail dealing with such an issue, therefore, the current work focuses on providing a tool for visual counterexample explanation on an FBD of a system.

III. COUNTEREXAMPLE EXPLANATION

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 $U = \{u_1, \ldots, u_n\}$ using both the values of state variables of the counterexample and the blocks in D.

Due to the possibility of explaining the outcome of φ through the assignments of its variables that is present in it [23], [24], 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 modular block, or an atomic block. Initially, explanation targets come from applying the approach in [23], 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 9 (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, $Y_0, \ldots, Y_m : C = Y_0, t \in Y_m$, 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

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

is valid, and $a'_{i,j}$ refers to the output variable of the atomic 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 atomic block or connection in the direction of the information flow.

Definition 10 (Inclusion-minimal cause): $C \subseteq X$ is an *inclusion-minimal cause* (IMC) 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 IMCs. This is due to the following points: (1) sometimes, in the context of the current counterexample, outputs of some blocks have several different IMCs, that should be displayed, i.e., the result of disjunction of two `true` variables has two IMCs, (2) the IMCs may be composed not only of input assignments of D but also of its internal assignments and we claim that showing such internal IMCs provides additional assistance to the user.

As an example, consider the atomic block `AND` from Fig. 3 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 atomic block without delay, the only one constraint is required to infer the cause, hence, the length of the sequence of sets from Definition 9 is two, where the first one is a cause. Now, we rewrite (1) as

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

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 necessarily added to Y_0 from Definition 9 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 atomic blocks shown in Fig. 3 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

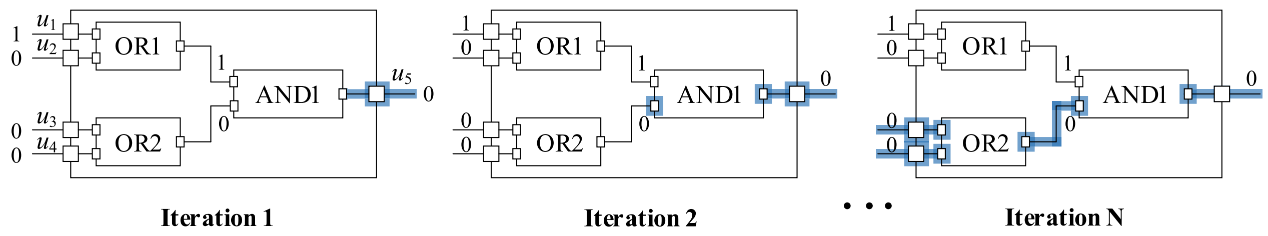


FIGURE 4. Illustration of the assignment explanation process. Digits above the connections show values of the transmitted signals. The explanation process results in a subset of variables of the system which are "responsible" for the value being explained.

$v_{9,1} = 0$. Then the chosen scope produces the following IMC: $C = \{(u_8, 0, 1), (u_9, 0, 1)\}$.

IV. ASSIGNMENT EXPLANATION ALGORITHM

The problem, stated in Section III, assumes that among all system assignments a union of IMCs 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 variables which do not have incoming connections. Next, for any modular block, it is a dubious help to see how, e.g., its output depends on its inputs, the analyst usually wants to know why such dependency takes place. Hence, in the explanation result, we also include IMCs for every nested explanation scope if they exist for such a scope. Thirdly, sometimes (for modular blocks) there can be more than one IMC and it is the user who chooses the one of their interest, thus, we need to discover the union of all such causes.

A. RECURSIVE EXPLANATION

The algorithm is provided in Alg. 1 and is illustrated in Fig. 4, where the problem is to explain why output variable u_5 of the modular block is `FALSE` at counterexample step s.

TABLE 2. Atomic block explanation rules of finding local IMCs. 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.

Recalling that an FBD itself is a modular block of modular blocks, to explain its output assignment, we need to find the output variable connected to the variable of the output of interest in the nested modular or atomic block (Fig. 4, iteration 1). Then, if the found variable belongs to a modular block, the output assignment of such a block is explained through the underlying net of blocks, whereas to explain an output of an atomic block, the rules from Table 2 are utilized. As a result, we have a set of input assignments that are sufficient to make explained atomic block output have its particular assignment – an IMC (Fig. 4, 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 IMCs in global and all the nested explanation scopes is obtained (Fig. 4, iteration N), its graphical visualization described in Section V.

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 atomic blocks). These estimates can be achieved if the result of each call of `explain` is memorized and not recomputed.

Algorithm 1: Assignment Explanation Algorithm `explain`.

Data: FBD D, counterexample X, explanation target $t \in X$
Result: set C – the union of all IMCs of t in D

if t corresponds to an input variable of D or a constant block input **then**
  return $\{t\}$ `/* this is a terminating cause */`
else if t is an input variable of an atomic block in D **then**
  `/* follow the connection and add it to the tree */`
  $t' \leftarrow$ the assignment of the output variable at the opposite end of the connection where t is located
  return `explain`$(D, X, t') \cup \{t\}$
else
  `/* t is an output variable of some atomic block */`
  $t_1, \ldots, t_m \leftarrow$ causes found for the current atomic block type according to Table 2
  $C \leftarrow \{t\}$
  `/* recursively explain the assignments of the local cause */`
  for $i = 1$ **to** m **do**
    $C \leftarrow C \cup$ `explain`(D, X, t_i)
  return C

Theorem 1: Alg. 1 finds the union of all IMCs of t.

The algorithm performs a backward (in terms of the information flow in the FBD) cone-of-influence analysis, seeking for all assignments that could be the cause of t according to Definition 9. 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 2 were specifically chosen to return the union of IMCs for an output of an FBD composed of an isolated atomic block. The formal proof is provided in Appendix A.

V. IMPLEMENTATION

The implementation of the algorithm described in Section IV was incorporated into the tool Oeritte[2] 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 $\{\ldots\}$ 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 the form "`varName : type`", where `type` is `boolean` for Boolean and any integer interval in the 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.

B. ENCODING NuSMV MODULES AS MODULAR BLOCKS

The aforementioned determinism assumption is required to represent NuSMV modules as modular blocks since our atomic blocks are purely deterministic. The input variables of the modular 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 into atomic blocks listed in Table 1. 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. FBD PREPROCESSING

Modular blocks in an FBD parsed from NuSMV code are decomposed into nets of interconnected atomic blocks that do not appear in the original model and, therefore, a counterexample lacks values of such atomic blocks variables. Nevertheless, these values are required for the explanation procedure. To obtain an extended counterexample, before running the algorithm for target t on FBD, the full set of constraints for each of the mentioned modular blocks is added to the full constraint set of the FBD 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 modular block is parsed correctly, as otherwise, after execution, its output variable values may differ from the ones stated in the counterexample.

D. MAIN WINDOW OVERVIEW

Graphical user interface of Oeritte is presented in Fig. 5. Two tabs Project and Workspace (Fig. 5a) separate the overall project settings from the working environment. Fig. 5 shows the contents of Workspace tab. Here, two interactive areas represent a counterexample as a table of values (Fig. 5b) and as a list of steps (Fig. 5c). A click on the item of the latter list evaluates the FBD in diagram (Fig. 5d) (hereinafter, the *diagram*) and LTL formula tree view (Fig. 5i) according to the step chosen. For it, all system variables are assigned with values defined by the counterexample step, hence, all the nodes in the LTL formula tree are calculated and all the system modules are executed.

Both atomic and modular blocks in the diagram have the same appearance (Fig. 6). 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 (a round pin (Fig. 6d) means input negation). A tooltip with the variable name appears when the cursor hovers over any of the pins. Blocks with the single input or output pin on the left and right sides of the diagram represent an interface of the current diagram. Lines connecting module inputs and outputs correspond to connections between the variables. Input and output variable values of the block for the chosen step are placed near the corresponding connecting points of these lines. If the diagram contains modular blocks, it is possible to open their internal nets in separate tabs (Fig. 5e). Names of the tabs show paths of such modular blocks in the original model and each diagram may be scaled with buttons (Fig. 5f).

The area to the left of the diagram provides information about the LTL formula being analyzed. Its string form resides in combobox (Fig. 5g) (and the type of the user interface control tells us that it is possible to dynamically switch between several formulas), tabs (Fig. 5h) show its evaluation for the

[2]https://github.com/ShakeAnApple/cxbacktracker/

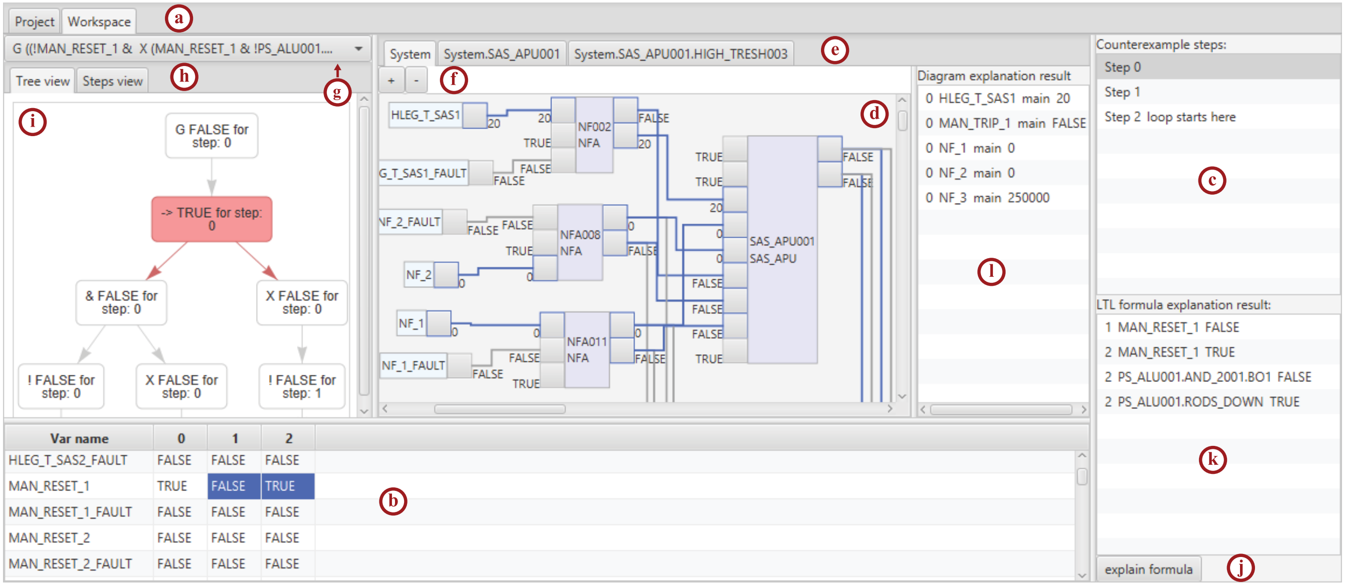


FIGURE 5. Main view of the tool in the explanation mode. Blue lines correspond to the connections between the variables whose assignments are included into the union of IMCs of the explanation target.

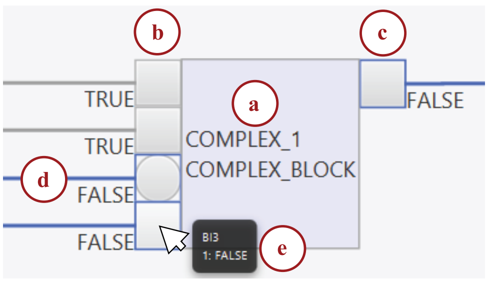


FIGURE 6. Visual representation of a modular block in the explanation mode. Connections between the variables (pins) and pins themselves, which assignments are included in a cause are highlighted with blue. Tooltip (e) shows that the value of variable `BI3` at step 1, which is `false`, is a part of the explanation.

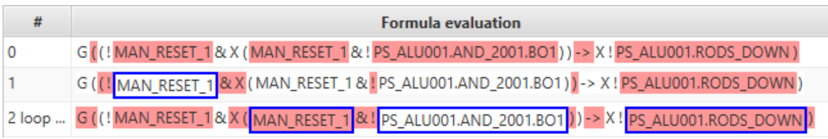
provided counterexample in the parse tree view (Fig. 5i) and step-wise (Fig. 7a). Depending on the calculation result of the branch, the nodes of the tree are colored in red, white and grey for `true`, `false` and an arithmetic result respectively.

Oeritte incorporates two kinds of explanation techniques: the cause identification algorithm from [23] for LTL formulae failures and individual assignment explanation from Section IV for the diagram. The LTL formula explanation process may be initialized with button (Fig. 5j) for the step chosen in list (Fig. 5c) and it result would appear in panel (Fig. 5k), table (Fig. 5b) and LTL steps view (Fig. 7a). By default, the formula is explained for the first step (step 0 in the tool) with the first diagram evaluation. For the individual assignment explanation, panel (Fig. 5l) shows the union of minimal causes of a target in the scope of input variables of the diagram in the current tab. The result, which includes all the minimal causes for the target, is depicted in the diagram in the form of highlighted variables and connections in between.

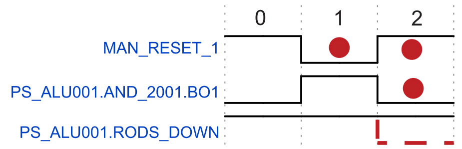
E. TYPICAL WORKFLOW

Assume the analyst has several LTL formulas failed for some modular NuSMV model and both, the formulae and the model, meet the requirements on input data (Section V-A). The first step now is to open Oeritte and provide it with the counterexamples, the formulae (standard NuSMV counterexample output is acceptable) and the model of the system. After input data is loaded and the analyst selected the formula to work with in the combobox (Fig. 5g), a click on any of the steps triggers (1) the explanation of the LTL formula for the first step (step 0 in the tool), and (2) a diagram of the provided system and LTL formula tree evaluation. An explanation of the LTL formula failure, or its cause, which is, essentially, a set of assignments, is highlighted in blue in table Fig. 5b, in steps view of LTL formula (Fig. 7a) and is textually represented in panel (Fig. 5k) in the form “< `step_number` > < `var_name` > < `var_value` >”.

If LTL explanation is not enough to grasp the idea behind the failure, a click on any of the highlighted or provided textually assignments triggers the backward explanation process in the block diagram that results in a union of IMCs, which, in the end, is a set of assignments. To display such a set in 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 obtains a tooltip where all its values included in the causes are displayed in the form “< `step_number` >:< `value` >” (Fig. 6e). Together with graphical visualization, list (Fig. 5l) shows *terminating assignments*, i.e., assignments, whose variables do not have incoming connections and belong to the input interface of the model. They are displayed in the form “< `step_number` >



(a) LTL formula steps view in OERITTE. Colors of variables and operators (red for `true` and white for `false`) correspond to their valuations at the steps outlined on the left. The assignments included into a cause of the formula failure have bold blue border around their names.



(b) Timing diagram representation of the counterexample for variables included in the verified LTL property. The dashed red line shows the expected value of PS_ALU001.RODS_DOWN, red circles indicate assignments included into a cause of the formula failure.

FIGURE 7. Visual LTL formula explanation in Oeritte and with timing diagram. In the text, MAN_RESET_1 is replaced with MAN_RESET and PS_ALU001.RODS_DOWN with RODS_DOWN.

< var_name > < block_name > < value >". This is the output of our interest, which includes evaluation paths in the model that influenced the chosen assignment to have its value. We argue that showing such paths and not only the assignments of an IMC provides useful visual information to the analyst. In case, the assignment belongs to the nested block that is not visible, a tab for its parent block will be opened automatically.

It is also possible to get the explanation on the diagram for an assignment that is not included in the LTL formula explanation result. For it, one should simply choose the step in Fig. 5c and define an explanation target by clicking on the pin with the desired variable name in Fig. 5d.

VI. CASE STUDY

As in [25], we demonstrate our method and the tool using a fictitious FBD implementation [25] of the U.S. EPR protection system [26], [27] encoded in NuSMV. On the top layer of hierarchy, it consists of acquisition and processing units (APUs) and actuation logic units (ALUs), combinations of which form two fault-tolerant subsystems: protection system (PS) and safety automation system (SAS). Based on signals from PS and SAS, priority and actuator control system (PACS) drives the control rods. Fig. 8 in [25] shows the full structure of the case study, with a note that process automation system (PAS) was replaced with external inputs.

The NuSMV file encoding such a system contains approximately 650 lines of code describing 20 different function block types and 32 function block instances. We see that the model of the system violates the LTL property $\mathbf{G}\ (\neg \texttt{MAN\_RESET} \land \mathbf{X}(\texttt{MAN\_RESET} \land \varphi) \rightarrow \mathbf{X}\,\neg(\texttt{PS\_ALU001.RODS\_DOWN}))$, where $\varphi = \neg\texttt{PS\_ALU001.AND2001.BO1}$ stands for the safety criterion. This property tells that the rods down command shall be deactivated when the safety criterion is satisfied, and the operator issues a manual reset. For the simplicity of reading, further, we will denote variable PS_ALU001.RODS_DOWN as RODS_DOWN. The counterexample for such a case consists of 3 steps including the values of all 375 model variables. The situation gets even harder when we notice that the safety criterion is not a mere function of inputs and outputs of the whole diagram, but the output of the block nested in one of the modular blocks inside the model.

The first step of our analysis is to see why the LTL formula itself has failed. The steps view of the LTL formula (Fig. 7a) reveals the situation where the rising edge of MAN_RESET had no influence on the commands to the rods despite that the safety criterion allowed it (Fig. 7b visualizes the counterexample with timing diagram). The causes of the failure here are shown with blue boxes around the names of the variables. At steps 1 and 2, MAN_RESET has values `false` and `true` correspondingly. In these circumstances, if the criterion is satisfied, then the formula requires RODS_DOWN to be `false` at step 2. However, it is not the case and boxes around PS_ALU001.AND_2001.BO1 and RODS_DOWN draw out attention to this fact. This fact is a clue, but, unfortunately, tells little about the processes taking place in the system, so we switch to the diagram by clicking on RODS_DOWN in Fig. 7a at step 2.

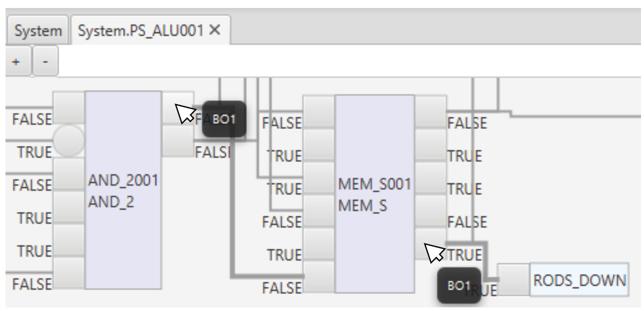


FIGURE 8. The connection between the criterion variable AND_2001.BO1 and an input of the block MEM_S001, whose output is connected to RODS_DOWN. Bold lines here and further are not related to the explanation mode and mean that the corresponding connections were selected by a mouse click.

Many connections get highlighted and we double click PS_ALU001 to learn if the problem lies in its internal composition. Now we can click on the criterion in Fig. 7a at step 2 and clearly see that the criterion variable transmits its value to block MEM_S001 that, in turn, communicates its output variable value to PS_ALU001 output, RODS_DOWN (Fig. 8)

The brief check of what influences the criterion shows that at all the counterexample steps it is set based on variable values from the current step (Fig. 9). The MAN_RESET signal is set externally, hence, we move to the inference analysis of

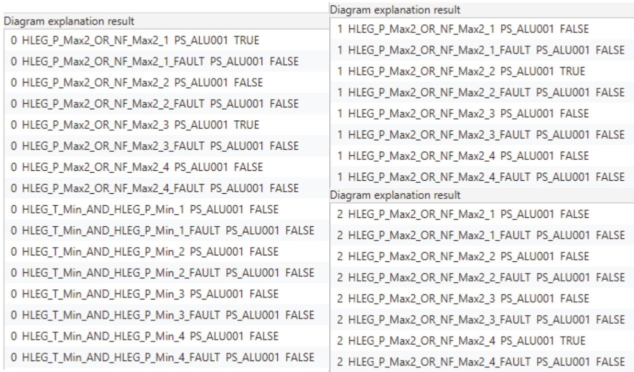


FIGURE 9. Explanations of `AND_2001.BO1` at steps 0, 1, 2 show that the criterion depends only on the assignments at the current step. To save space, three screenshots of the explanation results are shown in one picture.

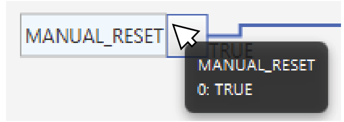


FIGURE 10. The part of the explanation of `RODS_DOWN` at step 2 shows its independence of `MANUAL_RESET` at step 2.

`RODS_DOWN` and see that it has a range of causes at different time steps.

The first thing we learn here is that for some reason, this signal at step 2 does not depend on `MAN_RESET` at the same step, while the property tells the opposite (Fig. 10). The following move is to decode the reason. At step 0 we see a straightforward dependence between the unsatisfied criterion, and the rods sent down, despite active `MAN_RESET` (Fig. 11a shows that `MEM_S001.BO1` here depends only on its input `BI1`). At the next step criterion allowed lifting the rods, however, we see the result of the block `OR_2001` from step 0, which is set to `true` by `MAN_RESET`, influencing `RODS_DOWN` to be active (Fig. 11b). We do not pay attention to other highlighted signals as they correspond to fault propagation and are always set to `false`. At step 2 `MEM_S001.BO1` again depends on inputs at the previous step, moreover, the chain origins in the satisfying criterion that now prevents resetting the rods down command despite the external command (Fig. 11c).

This reasoning brings us to the conclusion that, in our scenario, the satisfied criterion or active `MAN_RESET` signal cause `MEM_S001.BO1` being active. Moreover, the unsatisfied criterion at the current step sends the rods down immediately. In our counterexample, first, both the criterion was unsatisfied and `MAN_RESET` was set to `true`, and then the criterion was satisfied all the time, thus `MEM_S001.BO1` was locked in its active state.

VII. RELATED RESEARCH

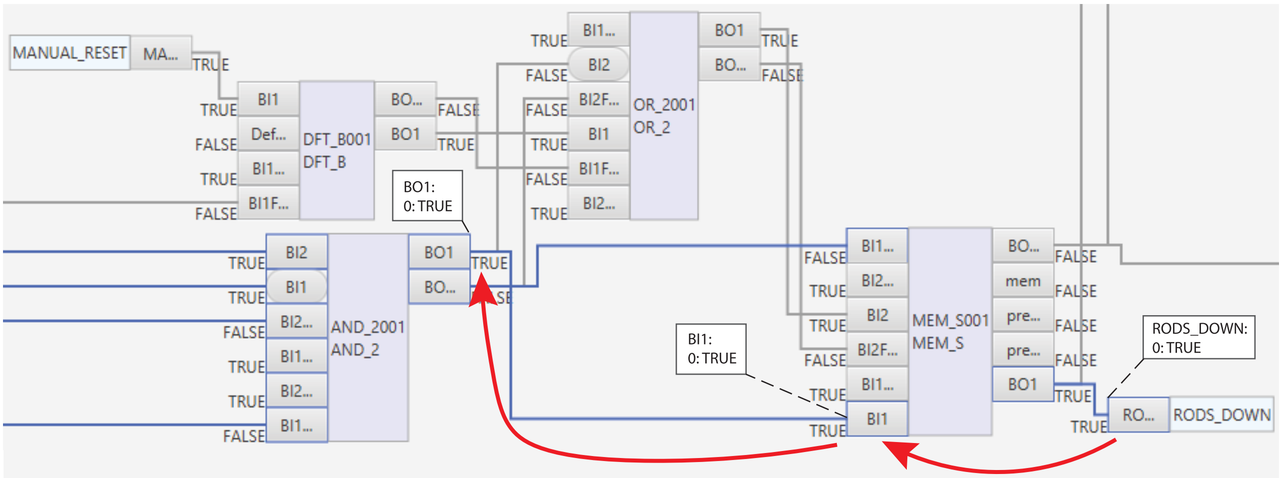
To the best of our knowledge, counterexample visualization was proposed in [28] in the tools VEDA and ViVe, and is one of the features of MODCHK [21] and Simulink Design Verifier [22]. While MODCHK is a graphical front-end for NuSMV and animates an FBD directly according to a given failure trace, Simulink Design Verifier generates a test case out of a counterexample obtained. Timing diagrams are used in [8]. Arguably, the most utilizable format for displaying a counterexample "model view" utilized in [29] and in [30] for simulation of IEC 61499 models. Counterexample visualization is the first step to user-friendly model checking, however, the mentioned works do not assist in discovering model deviations.

We can explain a counterexample from the verified property point of view, in an FBD or use a synergy of both. Only a few [23], [24], [31] deal with the property alone. Work [31] formulates both systems and specifications using predicate logic and considers only one-step counterexamples, [24] presents the approach explaining LTL formulae, which [23] complements with past-time LTL operators explanation and provides an open-source tool with a graphical user interface that highlights global causes of the main formula and local ones of sub-formulas. Giving an idea of what might go wrong in a property valuation, the core reason for the failure typically stays hidden inside the system and refers to the values of the variables missing in the formula.

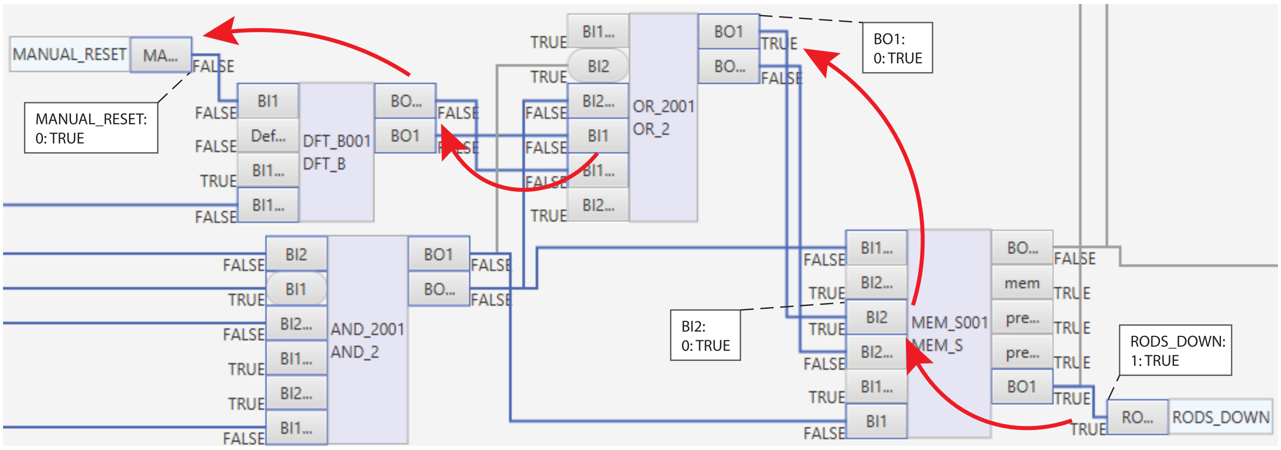
A verified model of a system is used in counterexample analysis in [32]–[35]. These approaches require multiple runs of a model checker to obtain more failure traces or additional good ones. [36] requires a single counterexample, although, here, the counterexample is a sequence of executed program statements, which contradicts our definition of a counterexample. Our method stands out by requiring a single failure trace and providing a visual explanation, crucial for I&C systems developed in the form of FBD.

Perhaps the most outstanding approach that inspired the current work is [37]. It requires a single counterexample and explains the failure in an FBD. The explanation here is inferred based both on the model structure and a property verified. However, only STANCE models are supported and the safety specification is formulated with the use of STANCE constructs. A so-called observer then monitors its satisfaction and outputs the variable value at a particular step to be explained. For this to happen, all the execution paths starting in model initial states must obtain activation condition formulae.

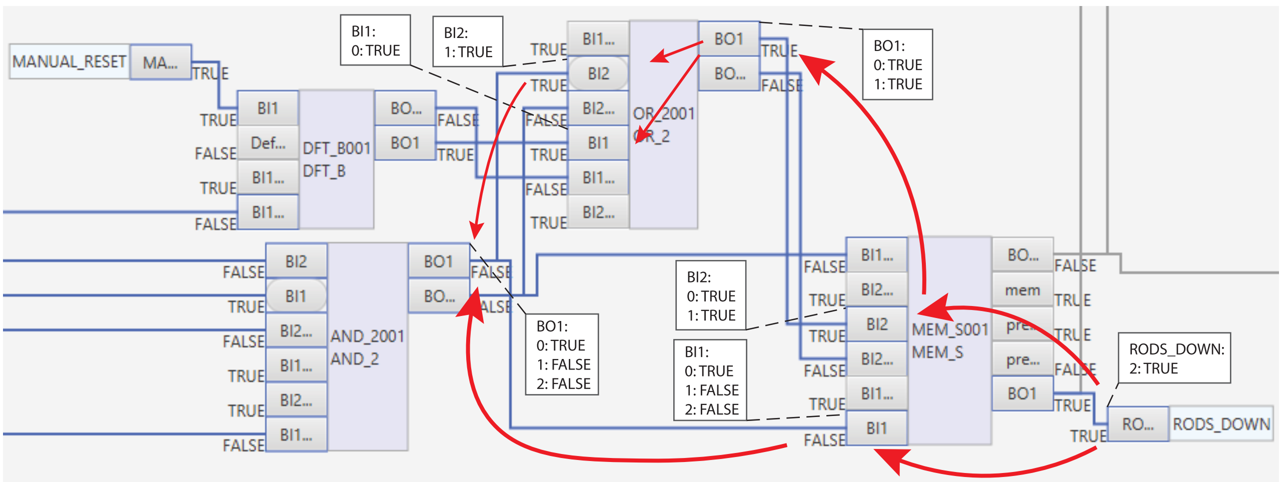
In our work we combine the ideas from [37] and [23] as follows. We directly implement [23] that allows us to obtain both safety and liveness property explanations with respect to the formula. Then, we let the user choose the variables and the time steps to be visually explained in an FBD and navigate such an explanation. Therefore, despite steps towards more user-friendly counterexample visualization and explanation have already been made, Oeritte is the only tool that combines explanation techniques into a consistent infrastructure.



(a) Explanation of RODS_DOWN at step 0. The violated criterion at the current step immediately moves the rods down.

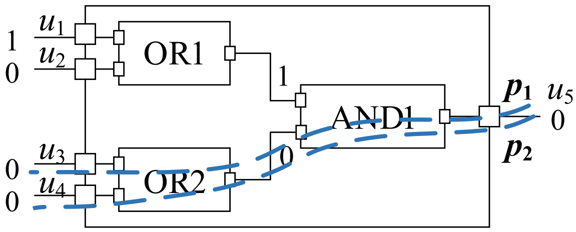


(b) Explanation of RODS_DOWN at step 1. The dependence between RODS_DOWN and MANUAL_RESET is shown.



(c) Explanation of RODS_DOWN at step 2. The picture shows that both `false` output of AND_2001 (or, in other words, the satisfied safety criterion) and MANUAL_RESET being `true` make MEM_S001.BI2 to be `true`. Also, we see that, at the current step, the dependence chain originates at step 0, which means that the current value of RODS_DOWN was not changed from the moment it was first set due to the shown combinations of AND_2001.BO1 current value and OR 2001.BO1 previous value.

FIGURE 11. Counterexample explanation for the property failure of the EPR protection system. For more clarity, we replaced original black tooltips with white boxes with exactly the same content and put starting letters of the names of the variables into the pins. Red curved arrows show the direction of the analyst's attention. All the screenshots are taken from the diagram under the `System.PS_ALU001` tab.



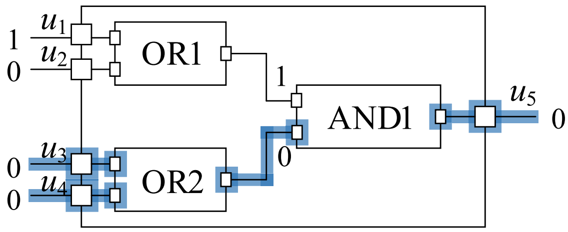


FIGURE 12. Visual comparison of the explanation result from [37] (on the left) and ours (on the right). Here, according to [37], two activation paths, p_1 and p_2 form a cause of the value of u_5 and this cause constitutes the explanation. In our case, the explanation is comprised of the assignments included into IMCs of u_5, starting from local explanation scope (the output of `AND1`) and finishing with the ones belonging to the input interface of the modular block (u_3, u_4).

A. THE CLOSEST RELATED APPROACH

Compared to [37], we outline the differences in the theoretical approach to the definition of a cause, in the algorithm and in its implementation.

Our theoretical problem statement suggests that we aim to find a union of IMCs for the explanation target. In an FBD, we consider a set of assignments as an IMC if such a set is obtained in a finite number of refinements of a set, first comprising only of the explanation target, where each refinement replaces one of the assignments from the set with its local IMC. By contrast, [37] considers a set of active propagation paths as a cause and a set of disjoint causes as an explanation. Following this definition, the same assignment might be included in the explanation more than once in case the paths constituting a cause converge (e.g., a cause of false outcome of $\vee$). Also, unlike [37] our approach to defining a cause does not require an additional formulation of activation paths. Our approaches are compared in Fig. 12

When it comes to the algorithm, the main difference between the one from [37] and ours lies in their outcome. While a set of the paths consists of independent entities that can be shown separately, our outcome is essentially an influence tree, where every included node is explained through its children.

This is clearly shown in our implementation, Oeritte. In the diagram explanation mode, tooltips attached to highlighted pins show the assignments constituting IMCs. Also, we highlight the connections between intermediate targets and their local causes. Another advantage of our implementation is the integration of LTL formula explanation into the tool, which helps to pinpoint the assignments to explain, no matter if it was liveness or safety property verified.

VIII. DISCUSSION

A. APPLICABILITY SCOPE

Any issue detected using Oeritte (or NuSMV) holds for the formal model of the system. The model is an abstraction, and does not necessarily include all relevant aspects of the real-world system and its actual environment. The model can also be simply incorrect. In order to verify that the issue is also relevant for the actual system, the analyst can try to reproduce the scenario using the hardware implementation, a simulation model, or by manually reviewing the available design documentation.

The algorithm provided in Section IV is defined for discrete-time models, whose state evolves through time and may be dependent on previous executions. In general, it is utilizable for the explanation of finite computation results, obtained within a finite number of algorithmic steps, even if these results are not produced by an FBD. For example, instead of an FBD, a computation graph of an algorithm written in an imperative programming language may be passed as an input to an accordingly adjusted version of Alg. 1. In this case, the main task is to create such a graphical representation of the explanation that will benefit the analyst. If the implementation supports manual identification of the explanation scope, then all the assignments bounded by this scope should be known before running the algorithm.

B. CAUSALITY

The idea of causality is discussed in a variety of philosophical treatises of past and present. One of the earliest definitions was given by Aristotle [38], who distinguished four forms of causality, i.e., material, formal, efficient and final. For us, the biggest interest lies in the first three, which refer to (1) the whole characterized by its constituents, (2) the reversed relation, where the choice of the details is explained by the principles of the system obtained, and (3) an outcome being the result of the preceding sequence of changes.

In modern science, one of the most commonly accepted approaches to discover event dependencies is counterfactual causation. The idea had been evolving since the 70s [39] and, fundamentally, means that the cause is a difference that makes the current world have the effects observed. In other words, event A is a cause of event B if unless A happened, B would not have happened. However, due to a horde of examples where the theory application resulted in non-intuitive outputs (for instance, the rock-throwing example from [40]), Lewis reworked his theory in 2000 [41], adjusting it to be able to deal with transitivity, preemption and other issues. Nevertheless, we will not go deeper into this approach as our definition is not based on counterfactual causality. For example, consider atomic block AND with two inputs (u_1, u_2) and one output (u_3). If its output is `true`, then, from a counterfactual point of view, it happens because both inputs are `true`.

In case of `false` output with two `false` inputs, however, none of the inputs is individually a cause, because, following the definition, changing only one input from `false` to `true` would not influence the result. But this may appear counter-intuitive, since even a single false input leads to false output, hence, according to our definition, there are two IMCs of $u_3 = 0$: $C = \{(u_1, 0, t)\}$ and $C = \{(u_2, 0, t)\}$. Fundamentally, while counterfactual definitions seek to find the knowledge (preferably, minimal in some sense) of the state of the system such that the negation of this knowledge is sufficient to make the explanation target `false`, our approach seeks the knowledge that is sufficient to conclude that the explanation target is `true`.

Another work [40] defines *actual* but-for causes of φ under some contingency in the model represented by structural equations [42]. It also suggests amendments to the commonly discussed problems in causal relations but it is based on the counterfactual theory, while, as mentioned above, our causes are not necessarily counterfactual.

In other words, we see our causality connected closer to the first three types of causes formulated by Aristotle and call it *general*. We deduce the minimal set of assignments sufficient to infer t in the context of a given FBD, with respect to the process taken place in the system and shown by a counterexample. However, our definition does not consider the system as a whole at every extension step, meanwhile, there may exist such combinations of constraints that generally restrict the ranges of output assignments of atomic blocks. For instance, consider Fig. 13, where signals merge in a common ancestor if traversing backwards from $a_{2,s}$, hence eliminating any scenario where the output of block AND is `true`. By our definition, $\{a_{1,s}\}$ is always an IMC of $a_{2,s}$, which may sound counter-intuitive as, in this FBD, $a_{2,s}$ can always be concluded regardless of any other knowledge of $a_{2,s}$. The following version of the definition of a cause is based on the whole set of constraints of a system model.

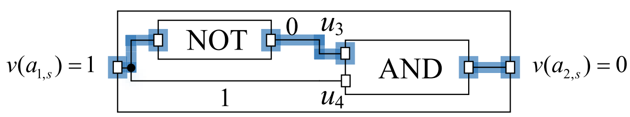


FIGURE 13. **Illustration of a non-intuitive algorithm result at step s due to variables u_3 and u_4 having a common ancestor u_1. The path of an explanation process for $a_{2,s}$ is highlighted with bold blue. Here block NOT inverts the signal from variable u_1, thus, block AND computes the expression $v(a_{2,s}) = v(a_{1,s}) \wedge \neg v(a_{1,s})$, 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.**

Definition 11 (Flow-independent cause): Consider FBD D, its set of constraints C_D, a counterexample X, a set of assignments $C \subseteq X$ from this counterexample, and an explanation target $t \in X$. Then C is a *flow-independent cause* of t iff the formula

$$\left(\bigwedge_{r \in C_D} r\right) \wedge \left(\bigwedge_{a_{i,j} \in C} (v_{i,j} = v(a_{i,j}))\right) \rightarrow (v_{t,t_j} = v(t))$$

is valid.

Definition 12 (Minimal flow-independent cause): C is a *minimal flow-independent cause* (MFIC), if there is no $C' \subset C$ that satisfies the relation above.

Essentially, Definition 12 means that a set of assignments C is an MFIC if it corresponds to a minimal set of assignments that should be fixed in an FBD to keep the value of the target unchanged even in case other variables values vary. Still, Fig. 14 shows that the new definition brings up another issue. The set of constraints of an FBDs does not encode the direction of the information flow. This fact allows the assignments that the value of the target does not depend on to be included in MFIC.

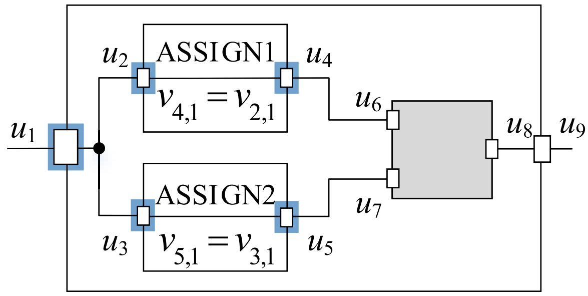


FIGURE 14. **Causes that are intuitively redundant but allowed by Definition 12. The subset of constraints for this block is $C_D = \{v_{4,1} = v_{2,1}, v_{5,1} = v_{3,1}, v_{1,1} = v_{2,1}, v_{1,1} = v_{3,1}\}$, which means that the values of all the variables $u_1, \ldots, u_5$ are equal at the current counterexample step. Following Definition 12, the minimal MFIC of any of the assignments of these variables is a singleton that includes any of them, i.e., $\{a_{1,1}\}$, $\{a_{2,1}\}$, $\{a_{3,1}\}$, $\{a_{4,1}\}$, $\{a_{5,1}\}$ (highlighted with bold blue).**

Recalling that our Definition 9 filters away assignments not required for the calculation of the target, we can combine it with Definition 12 that considers the diagram as a whole and, as a result, get the definition of a combined cause that gets rid of both problems mentioned.

Definition 13 (Combined cause): Assume C' and C'' are the unions of all MFICs and IMCs correspondingly. Then, a set of assignments C is a *combined cause* if $C = C' \cap C''$.

The algorithm that finds combined causes and its implementation in the graphical user interface is a part of the ongoing and future work.

IX. CONCLUSION AND FUTURE WORK

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 [23] and offers graphical backward counterexample analysis.

Inspired by works [24], [37], the tool provides methods and visual elements supporting explanations in terms of both the LTL formula and the model (FBD) in the form of paths from causes to the target values that they explain. The new part of a user interface – LTL formula steps view, originally implemented in [23] – increases the comprehensibility of a cause of the LTL formula failure. The counterexample explanation functionality of the tool might be scaled for FBDs encoded with any language by implementing a parser from the source language into the program and counterexample representations, whose implementations are provided in the tool's repository.

The elements of the user interface of Oeritte and their functionality address the challenges of counterexample visualization, as well as LTL formulae and a counterexample explanation. 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 model as a whole.

An industrial-sized case study proves that Oeritte assists in counterexample explanation for models of complex systems, saving time and efforts of analysts.

Intuitively, our algorithm builds a tree of logical inference with the root in the explanation target and the leaves in input assignments of the opened diagram. This bounds the search area and, as shown in the case study, sufficiently reduces the time spent on understanding the issue. On the other hand, there is still room for making the results more precise. For instance, consider a counterexample where the formula might have been true unless the last state triggered its failure. Here, the way inference paths were modified since the previous step and why this change took place might play a key role in the explanation process. In another scenario especially applicable to models of complex systems, numerous assignments from different steps in scattered diagram areas influence the target. Having a single counterexample, we could calculate a set of changes (minimal or not) in variable values that may indirectly indicate the cause of the problem. Adding to this method a possibility for the user to fix the assignments, so that the changes for them are not suggested, will contribute to narrowing down the search.

One of the branches of our future work includes the theoretical formulation of the aforementioned diagram search space reduction ideas, development and implementation of supporting graphical user interface concepts. Another point of enhancement is the tool itself. We will continue improving the user interface (especially its diagram area) and eliminating the input data restrictions. One of the most challenging milestones in explanation enhancement is to show the complex inference paths clearly, visually separating minimal causes and adding the time dimension.

APPENDIX A
PROOF OF THEOREM 1

Definition 14 (Computation graph): A *computation graph* of an FBD is a directed graph whose vertices correspond to assignments. Due to the determinism assumption of atomic blocks, these assignments can be expressed as functions of some other assignments. These dependencies correspond to the arcs of the computation graph. The graph is also acyclic as feedback loops constituting of assignments from a single time step are forbidden in considered FBDs.

Definition 15 (Causal path): In the computation graph of FBD D for counterexample X, a *causal path* $p = a_1\, a_2 \ldots a_m$, represented by a sequence of assignments a_i, where $i \in [1, m]$ and $a_m = t$, is a directed path from an assignment a_1 to the explanation target such that each vertex $a_j, j \in [1, m-1]$, belongs to the local IMC of a_{j+1}.

Proof: Suppose that C_t is the union of IMCs of t. As Alg. 1 explores exactly all causal paths (this is due to line 7), it remains to prove that assignment a belongs to some IMC iff a belongs to some causal path. For it, we need to prove the following two statements:

1) If there exists a causal path p from a to t, then a belongs to some $C \subseteq C_t$.
2) If $a \in C$, where $C \subseteq C_t$, then there exists a causal path from a to t.

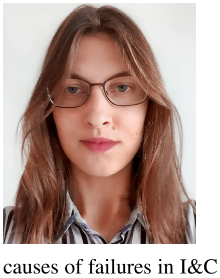
By C^*_a we denote a local IMC of a. To prove the first statement, we take $C = \bigcup_{a' \in p, a' \neq a} C^*_{a'}$, i.e., the union of local IMCs of the assignments constituting p. Suppose that $C \nsubseteq C_t$. Then it is possible to remove some assignment from C and it would remain a cause of t. But removing some assignment will make it impossible to deduce its parent according to Definition 9 (since all the children form a local IMC). Without this parent, it becomes impossible to deduce the parent of this parent. Applying this consideration a finite number of times will make us conclude that it is impossible to deduce t. Thus, $C \subseteq C_t$. Contradiction.

To prove the second statement, we will find how to construct such a path for each computation graph. We can select C to be an IMC of t. For C, there is a sequence of expanding sets that eventually reaches t. Some of these extensions introduce some assignments $a^1_1, \ldots, a^1_k$, to which there is an arc from a (there is at least one such assignment, otherwise removing a would retain C a cause of t and C would not be IMC). If for all $a^1_1, \ldots, a^1_k$ there are no arcs to assignments used in the deduction for C, then, as before, we could have removed a from C. Thus, there is at least one assignment $a^1_{i_1}$ for which such an arc exists. Repeating the same consideration for the parents of $a^1_{i_1}$, we get that there is some assignment $a^2_{i_2}$ to which there is an arc from $a^1_{i_1}$. There exists k such that by repeating the same consideration k times, we will always end up finding an arc leading to t. Thus, we have found a causal path $p = a\, a^1_{i_1} \ldots a^k_{i_k}\, t$. □

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[2] G. E. Gelman, K. M. Feigh, and J. Rushby, "Example of a complementary use of model checking and agent-based simulation," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 2013, pp. 900–905.
[3] H. Wang, D. Zhong, and T. Zhao, "Avionics system failure analysis and verification based on model checking," *Eng. Failure Anal.*, vol. 105, pp. 373–385, Nov. 2019.
[4] V. Todorov, F. Boulanger, and S. Taha, "Formal verification of automotive embedded software," in *Proc. 6th Conf. Formal Methods Softw. Eng.*, 2018, pp. 84–87.

[5] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikučionis, and P. Olsen, "Formal analysis and testing of real-time automotive systems using UPPAAL tools," in *Proc. Int. Workshop Formal Methods Ind. Crit. Syst.* Cham, Switzerland: Springer, 2015, pp. 47–61.
[6] P. Filipovikj, N. Mahmud, R. Marinescu, C. Seceleanu, O. Ljungkrantz, and H. Lönn, "Simulink to UPPAAL statistical model checker: Analyzing automotive industrial systems," in *Proc. Int. Symp. Formal Methods*. Cham, Switzerland: Springer, 2016, pp. 748–756.
[7] A. Pakonen, I. Buzhinsky, and K. Björkman, "Model checking reveals design issues leading to spurious actuation of nuclear instrumentation and control systems," *Rel. Eng. Syst. Saf.*, vol. 205, Jan. 2021, Art. no. 107237.
[8] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, and P. Seong, "FBD-Verifier: Interactive and visual analysis of counter-example in formal verification of function block diagram," *J. Res. Pract. Inf. Technol.*, vol. 42, no. 3, p. 171, 2010.
[9] E. Németh and T. Bartha, "Formal verification of safety functions by reinterpretation of functional block based specifications," in *Proc. Int. Workshop Formal Methods Ind. Crit. Syst. (FMICS)*. Berlin, Germany: Springer, 2008, pp. 199–214.
[10] 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 Trans. Ind. Informat.*, vol. 11, no. 6, pp. 1400–1410, Dec. 2015.
[11] L. C. Cordeiro, E. B. de Lima Filho, and I. V. Bessa, "Survey on automated symbolic verification and its application for synthesising cyber-physical systems," *IET Cyber-Phys. Syst., Theory Appl.*, vol. 5, no. 1, pp. 1–24, Mar. 2020.
[12] I. Buzhinsky and A. Pakonen, "Symmetry breaking in model checking of fault-tolerant nuclear instrumentation and control systems," *IEEE Access*, vol. 8, pp. 197684–197694, Oct. 2020.
[13] A. Cimatti and A. Griggio, "Software model checking via IC_3," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2012, pp. 277–293.
[14] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 121–125, 2003.
[15] 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.
[16] 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 *Proc. Int. Conf. Comput. Aided Verification (CAV)*. Berlin, Germany: Springer, 2002, pp. 359–364.
[17] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.
[18] P. Ovsiannikova, I. Buzhinskyt, A. Pakonen, and V. Vyatkin, "Visual counterexample explanation for model checking with OERITTE," in *Proc. 25th Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, 2020, pp. 1–10.
[19] *Programmable Controllers. Part 3: Programming Languages*, International Standard IEC 61131-3:2013, International Electrotechnical Commission, 2013.
[20] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Cambridge, MA, USA: MIT Press, 2016.
[21] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *Proc. IEEE 18th Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2013, pp. 1–6.
[22] *Simulink Design Verifier*. Accessed: Nov. 26, 2019. [Online]. Available: https://www.mathworks.com/products/simulink-design-verifier.html
[23] A. Pakonen, I. Buzhinsky, and V. Vyatkin, "Counterexample visualization and explanation for function block diagrams," in *Proc. IEEE 16th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2018, pp. 747–753.
[24] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," *Formal Methods Syst. Des.*, vol. 40, no. 1, pp. 20–40, Feb. 2012.
[25] I. Buzhinsky and A. Pakonen, "Model-checking detailed fault-tolerant nuclear power plant safety functions," *IEEE Access*, vol. 7, pp. 162139–162156, 2019.
[26] "U.S. EPR protection system," AREVA NP, Paris, France, Tech. Rep. ANP-10309NP, Revision 4, 2012. [Online]. Available: https://www.nrc.gov/docs/ML1216/ML121660317.html
[27] (2013). *U.S. EPR Final Safety Analysis Report*. [Online]. Available: https://www.nrc.gov/reactors/new-reactors/design-cert/epr/reports.html
[28] V. Vyatkin and H.-M. Hanisch, "Verification of distributed control systems in intelligent manufacturing," *J. Intell. Manuf.*, vol. 14, no. 1, pp. 123–136, 2003.
[29] K. Loer and M. D. Harrison, "An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation," *Autom. Softw. Eng.*, vol. 13, no. 4, pp. 469–496, Oct. 2006.
[30] S. Patil, V. Vyatkin, and C. Pang, "Counterexample-guided simulation framework for formal verification of flexible automation systems," in *Proc. IEEE 13th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2015, pp. 1192–1197.
[31] A. Ek, "Explanation of counterexamples in the context of formal verification," M.S. thesis, Dept. Inf. Technol., Uppsala Univ., Uppsala, Sweden, 2016.
[32] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Proc. Int. SPIN Workshop Model Checking Software*. Berlin, Germany: Springer, 2003, pp. 121–136.
[33] A. Groce, D. Kroening, and F. Lerda, "Understanding counterexamples with explain," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2004, pp. 453–456.
[34] S. Leue and M. T. Befrouei, "Counterexample explanation by anomaly detection," in *Proc. Int. SPIN Workshop Model Checking Softw.* Berlin, Germany: Springer, 2012, pp. 24–42.
[35] F. Leitner-Fischer and S. Leue, "Causality checking for complex system models," in *Proc. Int. Workshop Verification, Model Checking, Abstract Interpretation*. Berlin, Germany: Springer, 2013, pp. 248–267.
[36] C. Wang, Z. Yang, F. Ivančić, and A. Gupta, "Whodunit? Causal analysis for counterexamples," in *Proc. Int. Symp. Autom. Technol. Verification Anal.* Berlin, Germany: Springer, 2006, pp. 82–95.
[37] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, "Paths to property violation: A structural approach for analyzing counter-examples," in *Proc. IEEE 12th Int. Symp. High Assurance Syst. Eng.*, Nov. 2010, pp. 74–83.
[38] A. Falcon, "Aristotle on causality," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Stanford, CA, USA: Stanford Univ., 2019.
[39] D. Lewis, "Counterfactuals and comparative possibility," *J. Phil. Log.*, vol. 2, no. 4, pp. 418–446, 1973.
[40] J. Y. Halpern, "A modification of the Halpern–Pearl definition of causality," in *Proc. 24th Int. Conf. Artif. Intell. (IJCAI)*. Palo Alto, CA, USA: AAAI Press, 2015, pp. 3022–3033.
[41] D. Lewis, "Causation as influence," *J. Philos.*, vol. 97, no. 4, pp. 182–197, 2000.
[42] J. Pearl, *Models, Reasoning and Inference*. Cambridge, U.K.: Cambridge Univ. Press, 2000.



POLINA OVSIANNIKOVA was born in 1994. She received the B.Sc. degree in software engineering and the M.Sc. degree in applied mathematics and computer science from ITMO University, Saint Petersburg, Russia. She is currently pursuing the double Ph.D. degree with Aalto University, Espoo, Finland, and ITMO University.

Her research interests include formal verification and its industrial applicability, user-friendly model checking, methods for identification of causes of failures in I&C systems, and automation technologies for vertical farming.

IGOR BUZHINSKY was born in 1992. He received the B.Sc. and M.Sc. degrees in applied mathematics and computer science from ITMO University, Saint Petersburg, Russia, in 2013 and 2015, respectively, the second 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 a former Research Fellow with ITMO University and a former Postdoctoral Researcher with Aalto University. His research interests include formal verification, synthesis of finite-state models, and reliability of deep learning.

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

He is currently a Senior Scientist and a Project Manager with 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.

VALERIY VYATKIN (Member, IEEE) received the Ph.D. and Dr.Sc. degrees in applied computer science from Taganrog Radio Engineering Institute, Taganrog, Russia, in 1992 and 1999, respectively, the Dr.Eng. degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1999, and the Habilitation degree from the Ministry of Science and Technology of Sachsen-Anhalt, in 2002.

He was a Visiting Scholar with Cambridge University, Cambridge, U.K., and had permanent appointments with The University of Auckland, New Zealand, Martin Luther University, Germany, and in Japan and Russia. He is currently on joint appointment as the Chair of Dependable Computations and Communications, Luleå University of Technology, Luleå, Sweden, and a Professor of Information Technology in Automation, Aalto University, Finland. He is also the Co-Director of the International Research Laboratory Computer Technologies, ITMO University, Saint-Petersburg, Russia. His research interests include dependable distributed automation and industrial informatics, software engineering for industrial automation systems, artificial intelligence, distributed architectures, and multi-agent systems in various industries: smart grid, material handling, building management systems, data centers, and reconfigurable manufacturing.

Dr. Vyatkin was a recipient of the Andrew P. Sage Award for the Best IEEE Transactions Paper in 2012. He is the Chair of IEEE Industrial Electronics Society (IES) Technical Committee on Industrial Informatics.