VTT Technical Research Centre of Finland

Counterexample visualization and explanation for function block diagrams

Pakonen, Antti; Buzhinsky, Igor; Vyatkin, Valeriy

Published in:
Proceedings of 16th International Conference on Industrial Informatics

DOI:
10.1109/INDIN.2018.8472025

Published: 27/09/2018

Document Version
Peer reviewed version

Title	Counterexample visualization and explanation for function block diagrams
Author(s)	Pakonen Antti, Buzhinsky Igor, Vyatkin Valeriy
Citation	In Proceedings of 2018 IEEE 16th International Conference on Industrial Informatics, INDIN 2018. p. 747-753
Date	14.8.2018
URL	DOI not available
Rights	© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Counterexample visualization and explanation for function block diagrams

Antti Pakonen[1], Igor Buzhinsky[2, 3], Valeriy Vyatkin[2, 4]
[1] VTT Technical Research Centre of Finland Ltd., Espoo, Finland
[2] Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
[3] Computer Technology Department, ITMO University, St. Petersburg, Russia
[4] Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden
antti.pakonen@vtt.fi, igor.buzhinskii@aalto.fi, vyatkin@ieee.org

***Abstract*—Model checking is a proven, effective method for verifying instrumentation and control system application logics. If a model of the system being verified does not satisfy a specification, the failure scenario is presented to the user as a counterexample trace. Analysis of the counterexample can be time-consuming if the trace is long, the model is large, or the specification is complex. Spurious counterexamples ("false negatives") often exacerbate the problem. In this paper, we present a method that assists in identifying the root of the failure in both the model and the specification, by animating the model of the function block diagram as well as the LTL property. We also introduce a practical tool for visualizing LTL properties by animation and highlighting of important values based on causality. Using 43 actual design issues identified in practical nuclear industry projects, we then evaluate usefulness of the property visualization and explanation features.**

***Index Terms*—formal verification, model checking, visualization of counterexamples, explanation of counterexamples**

I. Introduction

Thorough verification of industrial instrumentation and control (I&C) system application logics is often paramount, especially if their failure can lead to accidents, hazards to vital infrastructure, or significant financial loss. *Model checking* [1] is a formal, computer-assisted verification method used to determine whether a system model satisfies given specifications. It has been proven [2] effective for the verification of I&C logics expressed using function block diagrams.

One of the definitive advantages [3] of model checking is that the analysis tool (*model checker*) returns a *counterexample trace*, demonstrating—if possible—an execution path of the system model that violates the specification. The counterexample can then reveal design issues in the model (and therefore in the original system). The problem is that the counterexample is not necessarily given in a format that could be called user-friendly, and seldom in a representation that is specifically tailored for the application domain. Finding the root cause of the failure can therefore be time-consuming if the system model is large, the counterexample trace is long, or the specification—typically a temporal logic property [1]—is complex [4].

Of the methods and tools suggested for counterexample visualization and explanation, some aim at finding the root cause of the failure in the model (e.g. as a set of paths), some at finding the failure in the trace (e.g. as a set of variable assignments). In this paper, we address both aspects.

The contribution of the paper is fourfold. First, we introduce a user-friendly method for visualizing counterexamples for function block diagrams, based on animating both the model and the specification. Second, we introduce a practical, application domain independent tool for visualizing LTL [1] properties for counterexamples represented in the NuSMV [5] format. Third, we reveal data of 43 actual design issues identified using model checking in practical customer projects in the nuclear industry. Fourth, we evaluate an improved version of the counterexample explanation algorithm published by Beer et al. [4] using real-world data.

II. Interpreting counterexamples

Due to the usually iterative work process of model checking, analysts constantly have to interpret counterexamples, even if the system design is error-free.

First, analysts often have to process erroneous (or "spurious" [6]) counterexamples—traces that are not valid for the actual, correct system model [7]. Model abstractions are often needed to address the state space explosion problem, but also allow model execution paths that are not relevant for the correct model [3], [6].

Second, spurious counterexamples also emerge if either the model or the specification is simply incorrect (i.e., a "false negative" [1]).[1] Model construction can be (at least partially) automated [8], but property specification is *hard* [9] and error-prone.

Furthermore, the first "correct" counterexample produced by the model checker is not necessarily the most interesting scenario [7], but further analysis can reveal more relevant or critical issues with the model. Re-verifying with a specification that excludes the first trace, we may again increase the number of counterexamples to interpret.

In all, the fact that the work process is usually iterative, also means that it is self-repairing. The downside—having to deal with several counterexamples—emphasizes the need for support techniques and tools [10].

[1] It is, of course, theoretically possible to have logic errors in *both* the model *and* the specification in a way that causes the incorrect specification to hold.

For process industry I&C systems, one domain-specific challenge is that the traces are often quite long (see Section V-A). The length is usually caused by the use of timing—i.e., delays—in the logics. Timing, in turn, is needed to carry out complex control sequences, account for process feedback, wait for operator response, etc. [11]. Consequently, analysts must also resort to rather complex formulas when specifying properties related to timing and sequencing [11].

III. Related research

A. Counterexample visualization

The simplest way for a model checker to return the counterexample is to list the values of the model variables for each step of the trace. *Variable tables* (e.g., [12], [13]) are used by tools such as NuSMV or nuXmv [14]. *State diagrams* (e.g., [12], [13], [15], [16]) are used in UPPAAL [17], which also uses them as the modeling language. *Sequence diagrams* (e.g. [12], [18]) are used in Spin [19] and UPPAAL.

In [12], the authors surveyed different visualizations, and the most favored were sequence diagrams (of the type that the survey participants were already familiar with) and "model view", meaning that the trace information was fed back to the modeling tool. Examples of "model view" visualization include [16], [18] and UPPAAL. More domain and application specific visualizations are found in, e.g., [7], [10], [20]–[22].

Specifically for the analysis of function block diagrams, digital timing diagrams are used in [20]. In [22], counterexamples for IEC 61499 control logics are played back using a simulation model of the controlled process. In [7], propagation paths that are calculated to be the cause for a counterexample are visualized by animating a Simulink function block diagram, but the analysis is restricted to Boolean variables.

Among the aforementioned visualization techniques, our approach utilizes variable tables and model view. We also use a novel technique: the values of all subformulas of the temporal formula refuted by the trace are shown for each step of the trace.

B. Automated analysis of counterexamples

The report [23] examined the question of counterexample minimization: leaving only variable assignments which cause the false value of the checked temporal property. The applied solutions to this problem were: (1) using the cone of influence reduction of NuSMV to reduce the number of variables in the counterexample, (2) considering only input variables of each module and non-deterministic variables as the ones entirely determining model behavior; (3) heuristic minimization by means of a random walk, (4) delta debugging, and (5) restricting the number of variables which follow the original counterexample by means of checking additional temporal properties.

A question related to counterexample minimization is the one of counterexample explanation by finding important variable assignments. The work [4] aims to answer the question: "what values on the trace cause it to falsify the specification?", which is done using the theory of causal models. The answer to this question is a set of pairs ⟨variable, trace step⟩, from now on referred to as *atomic causes*. The stated problem is shown to be computationally complex, but a polynomial algorithm is proposed which approximates the sought set of pairs. The idea behind this algorithm is to represent the LTL formula in such a form that its false value is caused exclusively by false values of its subformulas (probably on different steps of the trace). Thus, it is possible to descend from the false values of the entire formula to the values of its atomic propositions which cause the overall formula to be false (here, since this is an approximating algorithm, the word "cause" should be treated informally). The assignment of atomic proposition values produced by the algorithm is sufficient to make the overall formula false, but is not always minimum in this sense. The upgraded version of this algorithm is used in the tool presented in this paper (Section IV-B).

Understanding counterexamples is not limited to the analysis of a particular counterexample returned by a model checker alone. Other related (although not utilized in this paper) approaches involve generating multiple correct and erroneous traces with subsequent analysis of common features within these sets and differences between these sets [3], obtaining shortest counterexamples [24], and finding causes of counterexamples in the form of paths in the block diagram model of the system [7].

IV. Concept and implementation

A. Model animation

MODCHK [25], [26] is a graphical tool developed at VTT for the model checking of I&C application logics specified as function block diagrams. MODCHK allows the user to construct a modular, graphical model of the function block diagram, and specify properties in LTL, CTL [1] or PSL [27]. MODCHK then generates the necessary input files for NuSMV [5], runs NuSMV, and displays the counterexamples by animating the block diagram in the modeling view (see Fig. 1).

Motivated by VTT's customer work [2], MODCHK supports features specific to the nuclear industry. First, modeling is based on a manually constructed library of basic (elementary) function blocks. A lot of work has been done on automatic model generation based on standard languages like the IEC 61131-3 [8], but many major vendors use vendor-specific, non-standard function blocks, instead [25]. Second, the blocks have built-in support for signal *validity* processing. In addition to the actual (binary or analogue) value, each signal carries information about its validity in the form of an additional binary "FAULT" variable, set to either FALSE (no fault) or TRUE (fault). The way how signal validity affects the internal processing of each elementary block is fully user-configurable [26].

In MODCHK, counterexample traces are visualized by animating the original model, i.e., the function block diagram (see Fig. 1). The analyst can play the scenario back and forth, and MODCHK displays each state as if it were a simulator running the logic.

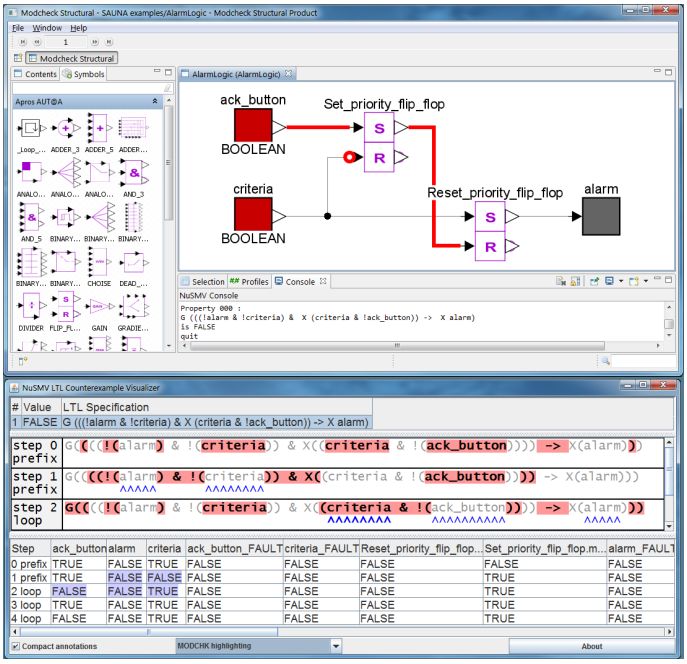

Fig. 1. MODCHK animates the model view, and a prototype tool animates the property.

In the animation, colors, line styles and text monitors are used to show the values of block diagram signals in each trace position (see Fig. 2). For analogue—or, in NuSMV's case, integer—signals, the value is displayed with a number shown at both ends of a signal line. For binary signals, TRUE is shown with a thick, red line, while FALSE is shown as a thin black line. Due to color vision issues, and to emphasize contrast, only one distinctive color—red—is used rather than a mix of different and possibly ambiguous colors. The use of color and thick line for TRUE is inspired by the analogy of logical TRUE (or "1") with voltage and light. If applicable, signal validity is shown with a dashed line for invalid data.

Signal type and status	Visualization in model view	Visualization in property view
binary TRUE, valid		**b_signal**
binary TRUE, invalid		~~**b_signal**~~
binary FALSE, valid		b_signal
binary FALSE, invalid		~~b_signal~~
analogue 20, valid	20 — 20	**(a_signal > 10)**
analogue 0, invalid	0 — 0	(~~a_signal~~ > 10)

Fig. 2. Visualization of both the model and the property is based on different signal line and text styles.

MODCHK can also display timing diagrams for the counterexamples, but the advantages of feeding the trace back to the original function block diagram view are fairly self-evident, and proven by over three years of successful use in practical customer projects [2]. "Model view" was also a favored approach in the expert survey reported in [12].

B. Property animation

Due to the reasons mentioned in Section III, visualization of each step of the counterexample trace in the model may not be sufficient for fast understanding of the essence of the counterexample. Thus, we implemented a cross-platform, open software tool[2] which visualizes LTL counterexamples represented in the NuSMV format and highlights the values of atomic propositions which are important for understanding counterexamples. The tool is based on the polynomial algorithm from [4], which produces an approximation of the minimum set of atomic causes sufficient to make the LTL formula false. We implemented it with several improvements:

1) Atomic propositions are highlighted in the places in the formula *where* they are important: in addition to atomic causes, the tool identifies pairs of the form ⟨concrete instance of the atomic proposition in the formula, trace step⟩, from now on called *enhanced atomic causes*.
2) Processing of liveness properties was improved by considering the counterexample in the lasso-shaped form [1] (i.e. as a prefix and a loop) instead of unwinding the loop of the counterexample several times.
3) Support of past-time LTL operators **Y**, **Z**, **O**, **H** [28] was added.
4) An alternative way of applying the algorithm to aid counterexample understanding was added, which interactively provides the user with the causes (subformula values) of TRUE or FALSE values of any subformula on any step of the counterexample.

An exemplary screen of the tool is given in Fig. 3, where three main panels are visible. On top, the list of all properties loaded from the results of executing NuSMV is displayed. Only properties whose values are FALSE are available for examination (since these properties are the ones for which NuSMV has provided a counterexample trace).

The middle panel displays the selected LTL formula for each step of the counterexample. The coloring used here is based on analogy with MODCHK model animation (see Fig. 2), but the colors and styles should also be user-configurable. Two types of annotations are visible:

1) Subformula values are shown using colors. TRUE values are shown in bold black font on red background, and FALSE values are shown in gray font on white background. For operators, coloring is only applied to the symbol of the operator (e.g. "−>", "F") and the corresponding parentheses.
2) Whether a particular atomic proposition is an enhanced atomic cause is shown below them with blue "^" characters.

[2]https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer

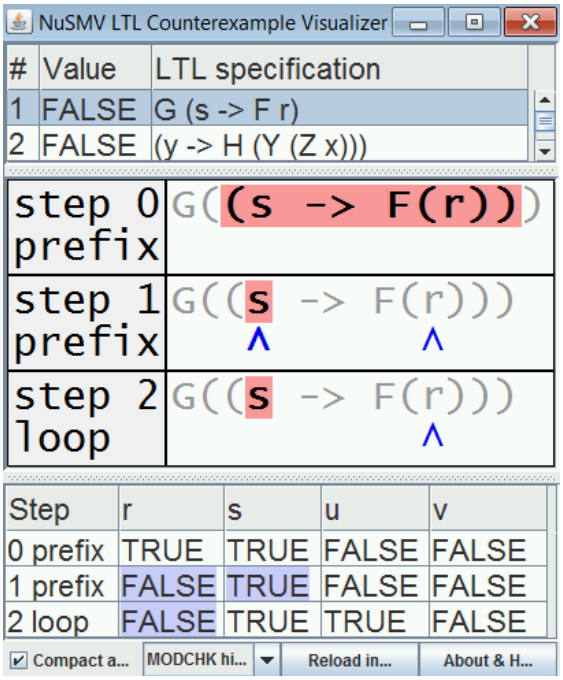

Fig. 3. Counterexample visualization tool (compact annotation mode).

Fig. 4 shows the alternative (full) mode of the middle panel. Its difference from the one explained above is the annotation of truth values of LTL subformulas *below* the property. First, this emphasizes the scope of each operator inside the formula. Second, clicking on a value annotation below the formula will highlight this annotation and the *immediate cause* of the corresponding subformula value. For example, on Fig. 4 the false value of the **G** operator on step 0 is explained by the false value of its argument on step 1. Such annotations allow the user to interactively see the causes of a particular subformula value (starting from the FALSE value of the entire formula). Applying this procedure iteratively, it is possible to descend to enhanced atomic causes, and thus understand *why* these values cause the FALSE value of the formula. In addition, enhanced atomic causes are indicated with blue background.

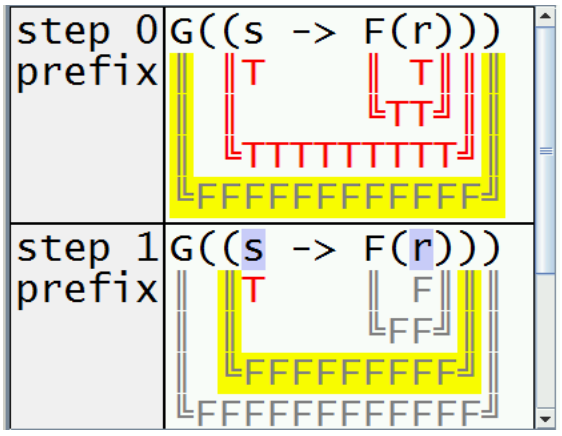

Fig. 4. Full annotation mode of the counterexample visualization tool.

In the bottom panel (Fig. 3), a table of values of all variables present in the counterexample (not limited to the values present in the LTL formula) is provided. Atomic causes (variable values) which correspond to enhanced atomic causes highlighted in the middle panel are marked with blue background.

V. Experimental results

A. Test data

Since 2008, VTT has applied model checking in practical customer projects in the Finnish nuclear industry, and verified I&C application logics related to the Olkiluoto 3 EPR new-build, the I&C renewal of the Loviisa 1&2 VVER-440, and the early, architecture level design of the planned Hanhikivi 1 AES-2006 nuclear power plants [2]. In these projects, VTT has identified 43 design issues.[3]

A short, generalized description of each issue, along with the failed property type is shown in Table I. In the properties, a reoccurring (19%) pattern $(\mathbf{G}p \rightarrow (property))$ is used to exclude execution paths related to irrelevant model states ($\neg p$), or to look for alternate—perhaps more serious—counterexamples by filtering out the first one (where $\neg p$ occurs). For further discussion on the property types, see [11].

From Table I, we can also see that the average length of the trace is 13.8, with eight traces (19%) having 20 steps or more. The length of the trace can depend on the model (how, e.g., time is modeled) or the type of the specification. The average number of variables in the NuSMV model is 262, with the highest numbers over 700. The LTL properties can refer to as many as 12 different model variables (two being the most common number, and less than five in 72% of the properties).

B. Experimental setup

23 of the 43 issues were detected before 2014, when the MODCHK tool was put to use, based on manually constructed NuSMV models. For the other 20 traces, the model could directly be animated using MODCHK.

For all the related models, we collected the counterexample traces as output by NuSMV. Since the property visualization tool can only process LTL, PSL [27] properties were rewritten as nested LTL expressions. The traces were then fed into the property visualization tool. The output of the tool was evaluated by a VTT expert already familiar with the majority of the originally identified design issues, having been involved in each of the customer projects from which the data were collected.

C. Results

The features of important value highlighting (or "explanation") and LTL property animation were considered separately. The evaluation grades are listed in Table I. The usefulness of each feature in the context of each design issue was rated on a scale of four grades:

[3]The issues are about a single system not fulfilling a stated property in some scenario, regardless of how unlikely that scenario is. The safety relevance of the issues is not considered here, and may be insignificant or purely theoretical.

TABLE I
EVALUATION OF PROPERTY EXPLANATION AND VISUALIZATION FEATURES BASED ON INDUSTRY PROJECT COUNTEREXAMPLES

#	Generalized description of issue	Failed property type	Trace		Variables		Evaluated usefulness	
			length	fails at	model	prop.	explanation	animation
1	Spurious actuation due to short signal pulses	$\mathbf{G}(p \rightarrow q)$	18	18	399	3	good	great
2	Spurious actuation due to delay element	$\mathbf{G}p \rightarrow \mathbf{G}(q \rightarrow \mathbf{O}r)$	16	2	773	9	excellent	excellent
3	Spurious actuation due to memory element	$\mathbf{G}p \rightarrow \mathbf{G}(q \rightarrow \mathbf{O}r)$	25	11	779	10	excellent	excellent
4	Reset signal takes no effect	$\mathbf{G}(p \rightarrow q)$	7	2	146	2	good	good
5	Signal remains set due to maintenance action	$\mathbf{G}(p \rightarrow q)$	11	4	171	3	great	great
6	Signal is set due to maintenance action	$\mathbf{G}p \rightarrow \mathbf{G}q$	5	0	171	2	good	good
7	Spurious actuation due to operator actions	$\mathbf{G}p \rightarrow \mathbf{G}\neg q$	17	1	645	5	great	great
8	Spurious commands on system startup	$\mathbf{G}(p \rightarrow \mathbf{O}q)$	19	6	346	2	good	great
9	Overtly long actuation signal	**never** {SERE}[a]	69	31	410	1	good	good
10	Overtly long actuation signal	**never** {SERE}	19	13	300	1	good	good
11	Conflicting actuation commands	**never** {SERE}	8	3	168	2	good	good
12	Conflicting commands due to invalid data	$\mathbf{G}(p \rightarrow q)$	5	2	146	11	good	good
13	Conflicting commands due to conflicting input	$\mathbf{G}\neg p$	5	1	146	12	good	good
14	A functional requirement is incorrect	$\mathbf{G}(p \rightarrow q)$	19[b]	1	338	4	good	good
15	Spurious actuation	$\mathbf{G}(p \rightarrow \mathbf{O}q)$	5[b]	2	86	9	excellent	great
16	Permanent inhibition of safety commands	**always** {SERE}|->{SERE}!	8	5	62	3	great	great
17	Signals blocked due to delay logic	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	9[a]	3	294	2	good	good
18	Signals blocked due to delay logic	$\mathbf{G}p \rightarrow \mathbf{G}(q \rightarrow \mathbf{F}r)$	44	–[c]	105	5	good	good
19	Operator can perform forbidden action	$\mathbf{G}p \rightarrow \mathbf{G}(q \wedge \mathbf{X}r \rightarrow \mathbf{X}s)$	23	10	109	4	good	good
20	Incorrect operational state is selected	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	7	6	290	8	good	good
21	Conflicting internal variables are set	$\mathbf{G}p \rightarrow \mathbf{G}\neg q$	23	7	289	4	good	good
22	Redundant systems inhibited at the same time	$\mathbf{G}\neg p$	6	2	777	4	good	good
23	Low priority signal overrules high priority signal	$\mathbf{G}(p \rightarrow q)$	22	2	151	4	good	good
24	Safety function inhibited due to maintenance	$\mathbf{G}p \rightarrow \mathbf{G}(q \rightarrow r)$	6	1	13	4	great	great
25	Actuator function inhibited due to maintenance	$\mathbf{G}(p \rightarrow q)$	4	1	185	5	good	good
26	Low priority signal overrules high priority signal	**always** {SERE}|->{SERE}!	16	3	94	5	good	good
27	Safety function inhibited after system startup	$\mathbf{G}(p \rightarrow (q \vee \mathbf{O}r))$	5	1	148	3	good	good
28	Redundant systems inhibited at the same time	$\mathbf{G}\neg p$	10	4	135	2	good	good
29	Both redundant computers inhibited	$\mathbf{G}\neg p$	7	1	246	2	good	good
30	Test mode initiation logic fails	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	10	5	135	3	good	good
31	Test mode inhibition logic fails	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	8	3	246	4	good	great
32	No operational mode is selected	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	11	2	300	4	good	good
33	Conflicting operational modes are selected	**never** {SERE}	8	2	273	2	good	good
34	Alarm can be acknowledged before it occurs	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	7	3	275	3	great	excellent
35	Actuation command inhibited on system startup	$\mathbf{G}(p \rightarrow q)$	16	2	344	4	good	great
36	A functional requirement is incorrect	$\mathbf{G}(p \rightarrow q)$	9	0	399	3	good	great
37	Short actuation command bursts occur	$\mathbf{G}(p \wedge \mathbf{X}q \rightarrow \mathbf{X}r)$	11	8	16	3	good	great
38	Safety command inhibited on system startup	$\mathbf{G}(p \rightarrow \mathbf{O}q)$	4	0	416	3	great	great
39	Rapidly fluctuating actuation commands sent	$\mathbf{G}\neg(p \wedge \mathbf{X}q \wedge \mathbf{XX}r)$	7	4	31	2	good	good
40	Rapidly fluctuating actuation commands sent	**never** {SERE}	33	16	142	8	good	good
41	Safety command inhibited after a delay	**always** {SERE}|->{SERE}!	22	12	91	2	good	good
42	Invalid data inhibits another function	$\mathbf{G}(p \rightarrow q)$	4	1	512	5	good	good
43	Signal validity processing logic fails	$\mathbf{G}(p \rightarrow q)$	4	1	154	2	good	good

[a] SERE (Sequential Regular Expression) style of PSL is used to describe multi-cycle behavior. [27]
[b] Analysis was based on bounded model checking, which restricts the length of the counterexample.
[c] A counterexample for a liveness property contains a scenario where the desired response never occurs.

1) "Fair" meant that the feature provided no specific added value for interpretation.
2) "Good" meant that the feature accurately pointed out the first failure of the property formula on the trace, as well as which of the (potentially) several conditions failed.
3) "Great" meant that the result provides clues that lead to the model variable(s) that served as the root of the issue.
4) "Excellent" meant that the result directly pointed to the model variable(s) that served as the root of the issue.

Notably, in Table I, there are zero "fair" grades, meaning that (1) both features were deemed useful for each of the cases, and (2) the causality method suggested by Beer et al. [4] perfectly accurately pointed to the first failure of each property. Given the trace lengths indicated by the data, such indication is obviously useful. The failure positions for each trace are indicated in the "fails at" column of Table I.

The most useful results often coincide with the use of the past operator $\mathbf{O}$. These are examples of issues where the time between the cause and the effect may be long due to delay from, e.g., feedback from the controlled process [11].

D. Example

As a practical example, let us consider the design behind issue 34 in Table I.[4] Here, we present a masked and simplified

[4] Unfortunately, it is very hard to present the most promising results due to confidentiality, since the failure in the logic cannot be presented without describing, e.g., vendor-specific, proprietary function blocks. Also, the usefulness of the results for the given example is also more prominent in the original context.

version of the original logic, only containing the function blocks needed to reproduce the issue. The original model has an I/O number of 22 and consists of 65 function blocks, resulting in 275 NuSMV model variables.

The simplified logic (see Fig. 5) is used to ensure that an operator acknowledges an alarm. When *criteria* turns true, *alarm* is set, and memorized in a flip-flop switch until the operator uses *ack_button*. The *ack_button* signal, is turn, is memorized in another flip-flop switch (so that *alarm* will remain off after acknowledgment), until *criteria* turns false.

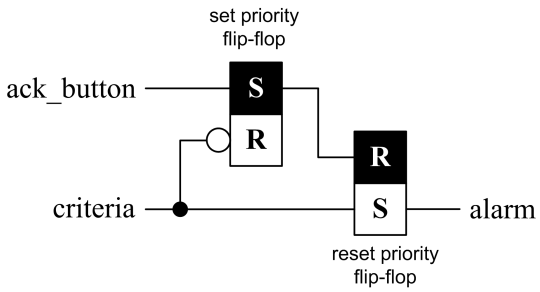

Fig. 5. Exemplar alarm logic.

A simple LTL property about *criteria* resulting in *alarm* would read: $\mathbf{G}(criteria \rightarrow alarm)$, which is obviously not true, since *ack_button* can reset *alarm* even if *criteria* is true. Instead, the analyst specifies: $\mathbf{G}((\neg alarm \wedge \neg criteria) \wedge \mathbf{X}(criteria \wedge \neg ack\_button) \rightarrow \mathbf{X}\ alarm)$. In other words, if both the the *alarm* and *criteria* are first false, and *criteria* then becomes true, *alarm* shall be true unless *ack_button* is also true.

Verification with MODCHK produces a counterexample trace, presented using a timing diagram in Fig. 6.

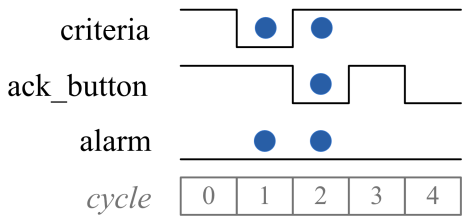

Fig. 6. Counterexample trace for the alarm logic, with important value highlighting.

Fig. 1 shows a screen capture where MODCHK is displaying step 1 of the trace, and the property animation tool shows steps 0 to 2. These views help the analyst see that:

1) On step 1, it is apparent from the MODCHK view that *ack_button* is true even if *alarm* is (or has not) been true. The property tool highlights that *alarm* and *criteria* are both false.
2) On step 2, the property tool highlights that *criteria* has now turned true, while *ack_button* has turned false, but the *alarm* is still false.

The overall **G** property is true from step 2 on. The expressed implication is not true at step 1, but due to the **X** operators, the cause is really only apparent on the next step. Highlighting values on steps 1 and 2 helps the analyst see that the failure point is the transition between these two states, as *ack_button* and *criteria* simultaneously change their values.

Based on the visualizations, it is easier for the analyst to notice that the somewhat unlikely scenario is about (1) the operator acting spuriously and (2) the flip-flops ending up in an unintended state due to two signal changes on the exact same processing cycle. The lowering edge of the *ack_button* signal effectively masks the *criteria* signal, and the operator can "preemptively" reset an *alarm* that is only about to occur.

VI. Conclusion and Future Work

The data we have collected of actual nuclear industry model checking projects proves several points. In terms of counterexamples, we can see that (1) the traces are often very long, (2) the models are often large, and (3) the properties are sometimes complex. All these factors make pinpointing the root cause of the failure in either the model or the specification extra challenging. As the data shows, the failures can occur in differing positions on the trace, even at the very end.

In this paper, we introduced a counterexample visualization tool that supplements the animation of the system model with the concurrent animation of the verified *property*. Furthermore, the tool also employs a causality based approach for highlighting signal values that are important for failure of the property.

We evaluated the property visualization tool against the industry data. The primary observation is that for every one of the 43 tested scenarios, the tool accurately indicated the exact point of the first failure, and, for properties with several conditions, accurately pointed out the condition that failed. We could also identify some cases where either the animation of the property or the highlighting of the important value (or both) clearly helped the analyst see the root cause of the failure. Identifying the root cause was especially helpful in cases where the distance between the cause and the point where the failure becomes apparent was long (in terms of trace positions).

In many cases, the set of variable assignments produced by the tool is the minimum one required to make the analyzed LTL formula false. However, a notable limitation of the tool is that it cannot highlight variables not included in the formula, leading to the explanations given by the tool being not the most useful ones in some situations. For some of the test cases, we later modified the original property to include a key variable we knew to be important, and then got even better results. One idea of automatic identification of counterexample causes which are not included in the formula is to explain the found atomic causes using paths in the MODCHK model, applying the ideas of [24].

For the further development of the visualization tools, we can also identify the following needs:

- MODCHK should also highlight the important values (i.e., the signals in the function block diagram) when animating the model view.

- In our past work, we have found PSL a very useful for expressing process industry specific properties [11], so PSL property animation should also be supported.

There is no reason not to provide the analyst with a whole range of different tools. In addition to the visualizations demonstrated in this paper, timing diagrams are obviously useful, and, in fact, are often used by VTT experts in design issue reporting.

ACKNOWLEDGMENT

This work has been funded by the Finnish Research Programme on Nuclear Power Plant Safety 2015-2018 (SAFIR 2018), and by the Ministry of Education and Science of the Russian Federation, project RFMEFI58716X0032.

We wish to thank VTT's clients in the nuclear industry for permitting the use of highly confidential customer project data for our research.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[2] 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 2017)*. American Nuclear Society, 2017, pp. 1342–1352.

[3] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Model Checking Software: 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings*. Springer Berlin Heidelberg, 2003, pp. 121–136.

[4] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," *Formal Methods in System Design*, vol. 40, no. 1, pp. 20–40, 2012.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Springer, 2002.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*. Springer Berlin Heidelberg, 2000, pp. 154–169.

[7] 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*, Nov 2010, pp. 74–83.

[8] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Software & Systems Modeling*, vol. 15, no. 4, pp. 937–960, Oct 2016.

[9] G. J. Holzmann, "The logic of bugs," in *10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. ACM, 2002, pp. 81–87.

[10] M. L. Bolton and E. J. Bass, "Using task analytic models to visualize model checker counterexamples," in *2010 IEEE International Conference on Systems, Man and Cybernetics*, Oct 2010, pp. 2069–2074.

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

[12] 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, Oct 2006.

[13] S. Jeong, J. Yoo, and S. Cha, "VIS analyzer: A visual assistant for VIS verification and analysis," in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, May 2010, pp. 250–254.

[14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification: 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Springer International Publishing, 2014, pp. 334–342.

[15] H. Aljazzar and S. Leue, "Debugging of dependability models using interactive visualization of counterexamples," in *2008 5th International Conference on Quantitative Evaluation of Systems*, 2008, pp. 189–198.

[16] H. Goldsby, B. H. C. Cheng, S. Konrad, and S. Kamdoum, "A visualization framework for the modeling and formal analysis of high assurance systems," in *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings*. Springer Berlin Heidelberg, 2006, pp. 707–721.

[17] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006)*, Sept 2006, pp. 125–126.

[18] A. Campetelli, F. Hölzl, and P. Neubeck, "User-friendly model checking integration in model-based development," in *24th International Conference on Computer Applications in Industry and Engineering (CAINE 2011)*. The International Society for Computers and Their Applications, 2011.

[19] M. J. Hornos and J. C. Augusto. (2012) Installation process and main functionalities of the Spin model checker. [Online]. Available: http://hdl.handle.net/10481/19601

[20] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, and P. Seong, "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, pp. 171–188, 2010.

[21] J. C. Campos and M. D. Harrison, "Interaction engineering using the IVY tool," in *1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '09. ACM, 2009, pp. 35–44.

[22] S. Patil, V. Vyatkin, and C. Pang, "Counterexample-guided simulation framework for formal verification of flexible automation systems," in *IEEE 13th International Conference on Industrial Informatics (INDIN 2015)*, July 2015, pp. 1192–1197.

[23] J. Lahtinen, T. Launiainen, K. Heljanko, and J. Ropponen, "Model checking methodology for large systems, faults and asynchronous behaviour — SARANA 2011 work report," VTT, VTT Technology 12, 2012.

[24] V. Schuppan and A. Biere, "Shortest counterexamples for symbolic model checking of LTL with past," in *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, UK, April 4-8, 2005. Proceedings*, ser. LNCS, vol. 3440. Springer Berlin Heidelberg, 2005, pp. 493–509.

[25] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *18th IEEE Conference on Emerging Technologies & Factory Automation (ETFA 2013)*. IEEE, 2013, pp. 1–6.

[26] 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.

[27] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. Springer US, 2006.

[28] M. Benedetti and A. Cimatti, "Bounded model checking for past LTL," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, ser. LNCS, vol. 2619. Springer, 2003, pp. 18–33.