This document is downloaded from the
VTT Research Information Portal
https://cris.vtt.fi

VTT Technical Research Centre of Finland

Automatic generation of repair suggestions for control logic of I&C systems

Ovsiannikova, Polina; Pakonen, Antti; Vyatkin, Valeriy

Published in:
IECON 2023- 49th Annual Conference of the IEEE Industrial Electronics Society

DOI:
10.1109/IECON51785.2023.10311970

Published: 19/10/2023

Document Version
Peer reviewed version

Link to publication

Please cite the original version:
Ovsiannikova, P., Pakonen, A., & Vyatkin, V. (2023). Automatic generation of repair suggestions for control logic of I&C systems. In *IECON 2023- 49th Annual Conference of the IEEE Industrial Electronics Society* (pp. 1-6). Article 10311970 IEEE Institute of Electrical and Electronic Engineers. https://doi.org/10.1109/IECON51785.2023.10311970

VTT
https://www.vttresearch.com

VTT Technical Research Centre of Finland Ltd
P.O. box 1000
FI-02044 VTT
Finland

By using VTT Research Information Portal you are bound by the following Terms & Conditions.

I have read and I understand the following statement:

This document is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of this document is not permitted, except duplication for research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered for sale.

Download date: 18. Sep. 2024

Automatic generation of repair suggestions for control logic of I&C systems

Polina Ovsiannikova*, Antti Pakonen‡, and Valeriy Vyatkin*§
†Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
§Department of Computer Science, Computer and Space Engineering, Luleå Tekniska Universitet, Sweden
‡VTT Technical Research Centre of Finland Ltd., Espoo, Finland
Email: polina.ovsiannikova@aalto.fi, antti.pakonen@vtt.fi, valeriy.vyatkin@aalto.fi

***Abstract*—We present an approach for suggesting possible repairs for the control logic of I&C systems implemented in the form of function block diagrams (FBDs) during the design phase. Each FBD has a set of functional requirements formulated using linear temporal logic (LTL). To ensure the correctness of the implementation, an FBD is translated into SMV, the language of the NuSMV model checker, which verifies the model against its properties. If a property does not hold, NuSMV generates a counterexample. In previous works, we developed methods on visual counterexample explanation using both, the failing LTL formula and the FBD itself. The current work continues in this direction and utilizes the results of the counterexample explanation to suggest fixes to the FBD considering the failed properties and the whole set of requirements. We propose three strategies for fixes generation and experiment on the examples of the logic from the nuclear domain.**

***Index Terms*—functional requirements, I&C control logic, safety-critical systems, model checking, FBD repair, fix suggestions.**

I. Introduction

Model checking [1] is one of the methods that perform a rigorous check of a formal model of a system with respect to a set of its temporal properties. This kind of check is essential for safety-critical systems such as, for instance, automotive driving systems [2] or nuclear power plants [3]. In Finland, for decades, model checking has been successfully used to verify the control logic of nuclear Instrumentation and Control (I&C) systems [4]. The approach, however, requires strong competence in formal methods and manual efforts in deciphering the counterexamples in case of requirements failures [5]. This motivated a series of works dedicated to the visual counterexample explanation for model checking applied to I&C control logic [5], [6].

In our case, the logic is implemented in the form of Function Block Diagrams (FBDs)[1] and requirements are formulated using Linear Temporal Logic (LTL). To perform verification, we translate an FBD to SMV (the language of NuSMV [7] model checker). We then combine the SMV model with its LTL properties and send them as input to NuSMV. As a result, the model checker produces counterexamples in case some properties fail. The counterexamples, the FBD, and the properties verified are then sent as input to a tool, Oeritte [5], which explains the failure both using the LTL formulas and the FBD provided.

In the previous works [5], [6], we focused on highlighting the erroneous (explanation) paths in FBDs of the control logic of I&C systems. Here, we continue in the direction of making model checking of such FBDs more user-friendly by proposing three strategies for the automatic generation of repair suggestions for such diagrams. Our strategies are based on the results of [5], where we derive the set of inclusion minimal causes (IMC) for the assignments, whose variables belong to the checked LTL formula, and are causes of the requirement failure. Having such a set of causes, we can form the causal tree for the particular assignment as it was done in [5], then, the intuition behind the strategies is that we need to introduce such a minimal change that, when applied to the FBD, the causal tree ceases to exist.

As it is shown in Section IV, some of our fixes might be overfitting, meaning that, when the change is applied, the requirements are satisfied, however, the fix is still rejected by the oracle (which, in our case, is represented by the analyst) due to hindering the important features or crossing out parts of functionality as a whole. Meanwhile, such fixes are still useful in the design stage, as they point out the need to rethink the requirements set. Our end goal is not to provide the fixes that the professional would create, but rather localize the problem more evidently and assist the engineer in creating the correct design.

II. Preliminaries

A. Recap of the previous works on causality

In this section, we repeat the main notions from [5], which are necessary for further understanding of the proposed method. Firstly, I&C logic we work with is represented by FBDs, which are sets of interconnected function blocks (FBs). Each FB has input and output interfaces and transforms inputs and internal variables into outputs according to its function. We distinguish FBs of two types, i.e., modular and atomic. Atomic blocks are responsible for undividable pieces of logic, for instance, arithmetic and logic operations or conditional assignments. Modular blocks encapsulate the net of FBs of both types and have a finite number of nested blocks. The FBD itself is a modular FB of the highest level of hierarchy.

This work was supported by the Finnish Research Programme on Nuclear Power Plant Safety 2018-2022 (SAFIR 2022).

[1] "FBD" is also one of the standard programming languages in IEC 61131-3, but in this paper, we use the abbreviation in a broader sense.

The difference between Automatic Program Repair (APR) for programs implemented using traditional programming languages, like C or Java, and for the ones implemented in FBDs is the kind of minimal allowed change. Commonly, APR techniques assume that any line of code can be edited. In our case, the minimal change involves operations with modular blocks. We cannot modify their internal structure, but we can, for example, replace them with other blocks or update the connections.

After an FBD of control logic is designed, we translate it to SMV language, formulate requirements to the logic using, e.g., linear temporal logic (LTL), and perform model checking using NuSMV verifier [7]. If the model fails to satisfy the requirements, NuSMV returns a counterexample, which can be represented as a sequence of model states, where each state is an assignment of all the model variables. We, then, can use a counterexample to explain the failure using only LTL formula [8], or both, the diagram and the formula [5], [6].

The explanation we get in FBD is a full set of inclusion minimal causes (IMCs) for a particular variable value at a particular counterexample step (an assignment). In [5], we provide a full formalization of a notion of IMC, but intuitively an IMC for assignment t is such a minimal set of assignments, that if their values remain the same, while values of other assignments vary, the value of t will not change.

Such an explanation we obtain for all the LTL formula causes that we get using the method from [8]. The result set of IMCs can be represented in the form of a directed causal graph [6], where each node is an assignment and each edge represents a causal dependency, e.g., if there is an edge starting in node n_1 and ending in n_2, it means that n_1 depends on n_2 and n_1 is a *dependable node*.

In a causal graph, we distinguish *causal paths*, which start in the assignment being explained and end in assignments, whose variables belong to the input interface of an FBD or have constant values. The method proposed in this paper deals with such causal paths, changing the dependencies rules in the direction of causal dependencies such that the paths are eliminated.

A causal path can be decomposed into *FB causal path parts* and *connection path parts*. The first represents a part of the path, where the dependable node is an output of some modular FB, and the nodes that it depends on belong to the input interface assignment of the same modular FB. Connection path parts are such two nodes in a causal dependency, whose variables are structurally connected in the diagram and the dependable node are parts of an input interface assignment of some FB.

B. FBDs repair

Since we can view an FBD as a program, where FBs are functions (or methods), and connections are assignments, we analyzed the works from the software repair domain and included the most relevant examples in our literature review.

One of the most comprehensive reviews on APR done is [9]. According to the two main families of automatic repair techniques, defined in [9], our method falls into the category of behavioral offline repair, which means that we modify the program (code) itself, before the execution. Such approaches, we can categorize further. For example, the ones which strive to provide the best fix for any failure [10] or the ones that are very focused on particular code problems [11], [12].

We can also differentiate the methods by the way the fix is obtained. It can be generated using the programming language constructs in a free [13] (any statement can be generated) or defined [10] (e.g., constant values are generated for a particular equation) form, or mined using the existing code examples [14]. In our case, we cannot use the latter, because the domain of programming I&C logic using FBDs is not enough represented on the web and there is a lack of examples that can be openly posted for machine learning techniques. The first category is more applicable, however, it may not always be advisable (or even permissible) to inject new types of FBs, generated from scratch, into an FBD.

There are also approaches that do not aim to change the functionality implemented in the program code, rather they work with modules contracts [15] or produce a sequence of repairing actions as in [11], where the repair plan is generated to fix a UML diagram, having the sequence of actions recorded during its development. In our approach, we can say that we generate input contracts for the FBs along the causal paths. Such contracts are boolean formulas that negate (or require to be different) the values of inputs, on which the output depends in the current scenario. The difference is that we do not change the contracts themselves, rather we modify the assignment of input values or the information flow that precedes the assignment of input values so that the new input values obtained after execution of the modified FBD satisfy the contracts for the counterexample scenario being considered.

Some approaches, as [16] do not focus on eliminating every error in all the scenarios, rather they aim to increase the number of non-failing traces. In our case, we still consider more useful a fix that makes the system satisfy all its requirements, however, if this is impossible, we can show the fixes candidates to the analyst. As the authors of [16] do, we also consider assisting the analyst in the creation of successful design as our main goal. This is similar to [17], where the simple APR approaches are implemented as a plugin to IDE, so that the programmer gets the repair suggestions during writing the code and thus can notice in real-time if the error is being made. Their empirical study shows that such an approach is viewed as helpful among developers.

Error-eliminating techniques that involve FBDs are rare. Despite the fact that we consider behavioral online repair in this paper, [18] draws particular interest as it proposes the design approach for self-diagnosis and self-healing in IEC 61131 programs inspired by biological processes.

III. Generation of Repair Suggestions

In this paper, we consider the same scenario as in [5], i.e., the I&C control logic implemented as NuSMV formal model

was verified against an LTL formula, and the counterexample was obtained. The counterexample and the system were opened with Oeritte and the causes of the LTL formula were discovered. Then, the analyst chooses the assignment they want to find an explanation for and aims to find the fixes so that the chosen assignment takes the correct value in the scenario defined by a counterexample. In addition, we assume having a library of FBs that can be used in an FBD and all the requirements for the logic formulated in LTL formalism.

We do not consider the addition of an arbitrary, automatically generated FB type as a practically applicable repair suggestion. Such a fix would change the programming language, not just the problematic FBD. In select domains like the nuclear industry, such modifications are not even permitted—I&C FBDs are built using predefined, non-standard, and vendor-specific [19] FBs from pretested libraries. Therefore, when searching for a possible fix, we cannot aim at changing the internal logic of library FBs or creating new FBs with the patching logic. This restricts the set of possible changes we can introduce to the FBD, which are:

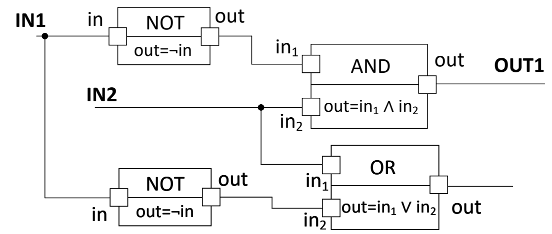
- to add/remove FB to/from an FBD,
- to add/remove a connection to/from an FBD,
- to add/remove negations to/from the variables of input interfaces of FBs in an FBD,
- to change values of the constants.

Having this set, we developed three strategies for generating repair suggestions. The first one is based on injecting a new FB to the erroneous FBD, the second assumes changing the information flow by redirecting the existing connections, and the last one combines the two aforementioned.

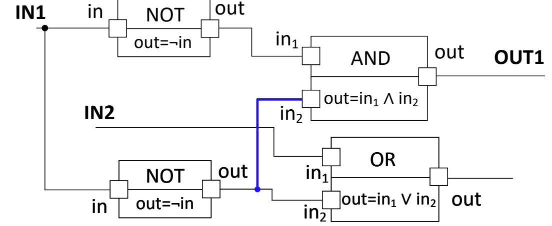
The core idea of all the strategies is to modify the information flow in a way that a causal graph cannot be formed in the same execution scenario, i.e., the same sequence of values provided to the input variables. We define *a repair candidate* as such a set of possible changes that, when applied, prevent the chosen variable (explanation target) from taking its value from the counterexample step of interest, while preserving the values of the remaining variables from the LTL formula, whose failure is being analyzed. Now, if we have a set of changes that includes adding of connection A and removing the connection B, while both connections end in the same variable, such a repair candidate can be denoted as a *replacement* of connection B with connection A.

After strategies produce a set of repair candidates according to their algorithms, the inference paths for the variables from all the formulas are translated to NuSMV code and verified against all the model requirements. Now, *a fix* is such a repair candidate that does not lead to new counterexamples for the same or other LTL formulas in the full set of specifications. In case, one or more properties failed and the explanation of one or more LTL causes includes variables that belong to the changed part, the repair candidate is discarded, otherwise, the repair candidate is accepted and the algorithm restarts to search for the fix to eliminate the new failure scenario.

Below we describe in detail the strategies for the discovery of repair candidates. Each strategy assumes that we have an FBD, a failed LTL formula with its counterexample, an explanation graph for the assignment of interest, and, for the second and third strategies, a list of all the FBs that can be added to the FBD.



(a) Initial FBD containing an error.



(b) The proposed fix (new connection is highlighted in bold blue.

Fig. 1. Example of the fix found by running the connection replacement strategy, assuming that the LTL property verified was `G (¬IN1 ∧ ¬IN2 = OUT1)`.

A. Connections replacement strategy

This strategy does not require a library of FBs, instead, it tries to fix the information flow by adjusting the existing connections.

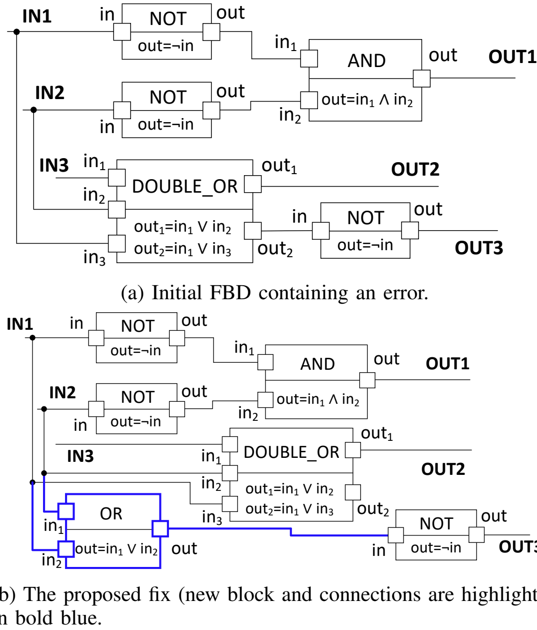
This strategy works with connection causal path parts. Here, the idea is that we pick a connection and define the desired value for the dependable node (e.g., if the current value is `true`, the desired value is `false`). Then we search for the output variables of FBs in the current FBD, whose values are equal to the desired value at the step of the dependable node, and try to replace the existing incoming connection to the variable of the dependable node with the connection starting in the new found variable. To check whether the replacement of the old connection with the new one is a repair candidate, we execute the FBs of the FBD that influence the variables of the LTL formula in the new setting. If variables at all the steps that do not contain an error remain and the step with the error is fixed, then the connection replacement is accepted as a repair candidate.

The extension of this strategy aims not only to replace existing ones but create new connections for the input variables whose set of incoming connections is empty, i.e., they are assigned some constant value at every counterexample step, and their assignments are included in the FB causal path part. We call such assignments *constant causes*. To find a suitable connection for a constant cause input, we search for all assignments of the output variables such that, if connected to the constant causes inputs, would assign a value different from the constant value at the counterexample step of the constant cause node. On its own, this extension can unluckily find the

repair candidate because commonly, the engineers might put wrong connections but do not miss connections altogether. However, it avails when applied as a part of the combined strategy (Section III-C).

B. FBs injection strategy

This strategy requires the library of FBs that can be included in the FBD. It seeks to eliminate a causal dependency from the explanation graph by creating a "detour" in the assignment dependencies along its paths. The strategy is applied to the parts of the causal path which represent the dependency of the assignment of some output of a modular block on a subset of the assignments of its inputs (FBs causal path parts). In this case, we search for a new FB from the library that can substitute the existing FB. If the subset of inputs included in the considered path part corresponds to a full set of input variables and the block has a single output variable, the exiting block is replaced with the found block, otherwise, the connections to the inputs of interest are redirected to the inputs of the new FB (Figure 2). The same applies to the output variables.



(a) Initial FBD containing an error.

(b) The proposed fix (new block and connections are highlighted in bold blue.

Fig. 2. Example of the fix found by running the block injection strategy, assuming that the LTL property verified was `G (OUT1 = OUT3)`. The proposed repair candidate includes adding the FB and the connections in bold blue and removing the connection between `DOUBLE_OR.out2` and `NOT3.in`.

The new block must comply with the following rules. First, it must contain the dependency of its output variable on the same or larger number of its inputs of the same type as in the path part. Then, if executed following the same counterexample scenario, it should produce a value other than in the counterexample at the time step of the dependable node. Lastly, after the new FB is injected when executing the FBD following the scenario from the counterexample, the values of the variables from the formula at each step should remain the same except for the value of the assignment being explained (the LTL cause).

The second and the third conditions are ensured by the isolated execution of the candidate FB and comparing its output with the desired one and by the selective execution of the diagram, i.e, running the FBs that are present in the information flows that leads to the assignments whose variables are present in the LTL formula.

C. Combined strategy

The combined strategy requires the library of FBs. It can be applied when, for instance, no fixes were found using both previous strategies. The strategy inserts "dummy" blocks along the causal paths and tries to connect them in a way the formula is satisfied in the current execution scenario. We call them "dummy" because they do not change the behavior of the system, but give us the possibility to apply the previous two strategies using the new block and see whether the created piece of logic completes the FBD and eliminates the error.

The algorithm for finding the repair candidate is the following. First, the connection causal path part is chosen, let us assume that it embodies the structural connection C_1 from variable out_1 to variable in_1. Then, we choose the new FB to be inserted into the diagram, where at least one output variable (assume, it is out_2) is of the same type as in_1 and generally depends on at least two input variables, where at least one is of the same type as out_1. Assume we found such a set of input variables of a new block and it includes in_2. Then, we add the connection from out_1 to in_2 and assign constant values to the remaining input variables of the new block.

Now, we determine if such constant assignment is suitable. For this, we execute the FBD with the new block starting with the first counterexample step, and, if the values of out_2 at each counterexample step are the same as values of out_1, we keep the constants assignment, otherwise, we provide different values for the constants and execute the new FB again. When the new FB and the constants assignment is chosen, we remove C_1 and add a connection from out_2 to in_1. All these changes (adding the new FB, adding two new connections, and removing C_1) are added to the set of changes of the repair candidate F_1. Then we, apply the extended connection replacement strategy to connect the input variables of the new block with output variables in a way that all counterexample steps will now satisfy the property (Section III-A) and obtain F_2. The result candidate F is a union of the changes from F_1 and F_2.

This strategy can be applied as fully automatic or guided by the user input, which can significantly reduce computation time. For example, if the value of out_1 at the step of interest is `true`, one good candidate is a binary block AND with one of its inputs set to `true`. If the value of out_1 is `false`, it will be convenient to use a binary block OR with one input set to `false`. The user can also choose the number of variables that should influence the variable at the current step.

IV. Experimental evaluation

To demonstrate the approach and the strategies, we implemented the algorithms within the tool Oeritte and used the real-world design issues reported in [4] as test cases. We found fixes for three issues and revealed the necessity for a more fundamental requirement and logic redesign for one issue.

A. Example 1

First, let us try to address issue number 6 from [4]. Here, the idea is that two inhibit signals (`INH1` and `INH2`) cannot be set to `true` at the same time. However, this happens if both `SWT` signals are switched to `true` simultaneously.

The LTL formula that shows the failure is: `G ¬(INH1 ∧ INH2)`, the explanation was inferred for `INH2 = true` at the first counterexample step. The method generated three precise fixes based on the connection replacement strategy. In each case, the fix breaks the symmetry in the feedback loops, by rerouting signals related to `INH1` straight to the `AND4` block, so that `DELAY1` is bypassed (Figure 3).

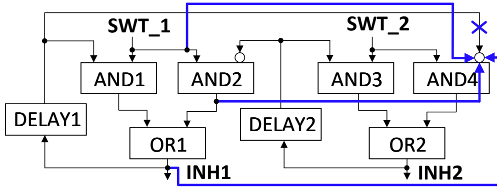


Fig. 3. All connection fixes found by the connection replacement strategy for Example 1. New connections proposed are highlighted in bold blue. Note, that each fix is represented by one highlighted connection and the removal of the connection with a blue cross over it.

B. Example 2

Second, let us try to address issue 7 from [4]. In this example, `START` should be active when two or more measurements are above 10, and `STOP` when two or more measurements are below 7—a majority vote in case of single measurement failure. When we get as inputs a set of measurements, half of which fit the first criteria and half comply with the second one, both signals `START` and `STOP` become active, which must be avoided.

The failing LTL property is: `G ¬(START ∧ STOP)`, the explanation was inferred for `STOP = true` at the third counterexample step. For this example, we received all three types of fixes, two of them (found using connection replacement and combined strategies) are depicted in Figure 4. The third fix is discovered by the block injection strategy, which assumes replacing `MAX` block with `MIN` block. The `AND` block injection, in particular, calls attention to the fact that the priority between `START` and `STOP` needs to be specified (for the unlikely event of conflicting inputs).

C. Example 3

Third, let us try to address issue number 2 from [4]. The intended functionality is that when `ACT_A` is set, there is a 5-second time window, during which the command `SET_B` can

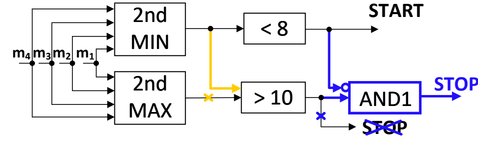


Fig. 4. The connection fix found by the connection replacement strategy for Example 2 (in bold orange) and the fix found by a combined strategy (in bold blue) that includes adding a block `AND` and connecting its remaining input to output `START`.

set `ACT_B` (and reset `ACT_A`). However, if `SET_B` is not set during the expected time window, `ACT_A` and `ACT_B` cannot change their values further.

The failing property is: `G ACT_A → X X X X((¬SET_A ∧ SET_B) → ¬ACT_A)`, the explanation was inferred for `ACT_A = true` at the 9th counterexample step. For this scenario, we obtained three connection replacement fixes (Figure 5) and one block injection fix. The latter assumes replacing block `AND1` with block `RS` from the library, with the output of `PULSE1` being redirected to the `RESET` input of `RS` block and `SET_B` to `SET`.

Of the fixes, connecting `SET_B` to the `RESET` variables maintains the intention that `ACT_B` can only be set in the 5-second time window, but allows resetting `ACT_A` at any time. Removing `PULSE1` fixes the freezing issue, but also disables the timing logic. Connecting `DELAY1` to the `RESET` variable is equivalent to removing the `SR` block altogether, which means removing the memorization of `ACT_A` from the logic. Each fix raises questions about the original requirements being incomplete or contradictory, and the likely cause of the issue.

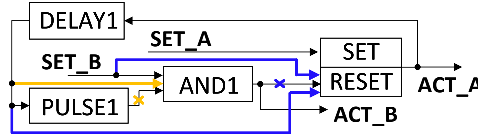


Fig. 5. All connection fixes found by the connection replacement strategy for Example 3. New connections proposed are highlighted in bold blue and orange. Note, that each fix is represented by one highlighted connection and by one cross that indicates the removal of an old connection of the same color.

D. Example 4

Finally, let us try to address issue number 3 from [4]. This example demonstrates the case when not only the logic but the requirements should be redesigned. Here, `ACT` is supposed to be `true` if `VAR` drops below 10 and then turn to `false` if `VAR` proceeds to drop under 8. The property reads that after `VAR` falls below 10, we expect `ACT` to be `true` at the next step. However, inputs that are generated by NuSMV are random, which leads to the case, when `VAR` is 10 at one step and 7 at the next step, thus, `ACT` is never set to `true`. In the real scenario, however, the value of the variable would not fall that sharply and we would obtain inputs within the mentioned interval. The approach cannot generate fixes in this

case, meaning that the analyst should consider reformulating the requirements and the logic.

V. Discussion

All three strategies were found useful in proposing fixes. Together with the counterexample explanation technique from the previous works, they effectively highlight the problem parts. An interesting observation is that all of the repair candidates generated for Example 1 were accepted as fixes after the changed FBD was successfully verified against all its properties. Also, all the repair candidates except for one were accepted as fixes for Example 3. For the second example, in turn, several erroneous candidates were generated, however, the perfect fix was still found from the first iteration of the algorithm. One explanation for this is that the failure trace for Example 2 was the shortest, and, if we think of each counterexample step as a local test case, the smallest number of test cases naturally leads to a higher number of false positive repair candidates.

This makes us optimistically view the scaling of this method to more complex scenarios. First, usually, complex logics contain more FBs with memory, hence, we can get longer counterexamples, which hypothetically means less model checking as fewer repair candidates are obtained. Second, the causal paths are longer, so there is more space for experiments for all the strategies and a higher chance of finding the fixes.

The second interesting point we discovered is that even if the method does not generate fixes or generates fixes that, when applied, prevent an FBD from performing its function, it can be a signal for the analyst to reconsider their requirements as in Example 4. Another type of issue that can be discovered is lacking constraints on inputs which are impossible in the real-world environment but can be generated during the open-loop model checking (e.g., the logic and error from Example 2). These scenarios can be discovered using both the explanation and repair suggestion techniques of Oeritte. When the fix cannot be generated, the analyst can follow the explanations for different assignments and determine the missing requirements or constraints on inputs.

VI. Conclusion and future work

Based on the case studies we can conclude that the method effectively provides solutions to the cases when the engineer misplaced the connection, used the wrong FB similar to the right one, or did not implement part of logic (e.g., prioritization from Example 2).

The approach will not provide fixes if the complex restructuring is required, however, it will assist the analyst throughout a redesign process. One of the directions for future work is to integrate such automatic suggestions of fixes into the IDEs used for FBD development. Having such a plugin will allow a verification-driven development. Several repairs generation strategies can be run in parallel without slowing down the user experience.

Another room for improvement is the combined strategy. As it was stated, the strategy in its fully automatic version requires computation resources in order to pick the right FB from the library, and the block is defined as suitable based on multiple executions. It is worth investigating if it is possible to formulate a SAT problem in a way that multiple executions could be avoided. Another direction of improving this strategy is to accept user input when choosing the "dummy" block or the number of inputs it should have.

References

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

[2] V. Todorov, F. Boulanger, and S. Taha, "Formal verification of automotive embedded software," in *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, 2018, pp. 84–87.

[3] E. Németh and T. Bartha, "Formal verification of safety functions by reinterpretation of functional block based specifications," in *International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*. Springer, 2008, pp. 199–214.

[4] A. Pakonen, "Oops! Examples of I&C design issues detected with model checking," in *International Symposium on Future I&C for Nuclear Power Plants (ISOFIC 2021)*, 2021.

[5] P. Ovsiannikova, I. Buzhinsky, A. Pakonen, and V. Vyatkin, "Oeritte: User-friendly counterexample explanation for model checking," *IEEE Access*, vol. 9, pp. 61 383–61 397, 2021.

[6] P. Ovsiannikova, A. Pakonen, and V. Vyatkin, "Change-based causes in counterexample explanation for model checking," in *The 47th Annual Conference of the IEEE Industrial Electronics Society (IECON 2021)*, 2021.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource tool for symbolic model checking," in *International Conference on Computer Aided Verification (CAV)*. Springer, 2002, pp. 359–364.

[8] A. Pakonen, I. Buzhinsky, and V. Vyatkin, "Counterexample visualization and explanation for function block diagrams," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 2018, pp. 747–753.

[9] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[10] T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin, "Automatic program repair using formal verification and expression templates," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2019, pp. 70–91.

[11] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

[12] S. R. L. Marcote and M. Monperrus, "Automatic repair of infinite loops," *arXiv preprint arXiv:1504.05078*, 2015.

[13] M. Nica, S. Nica, and F. Wotawa, "On the use of mutations and testing for debugging," *Software: practice and experience*, vol. 43, no. 9, pp. 1121–1142, 2013.

[14] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 307–318.

[15] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, 2002, pp. 746–757.

[16] F. Logozzo and T. Ball, "Modular and verified automatic program repair," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 133–146, 2012.

[17] D. Campos, A. Restivo, H. S. Ferreira, and A. Ramos, "Automatic program repair as semantic suggestions: An empirical study," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 217–228.

[18] S. S. Khairullah and C. R. Elks, "Self-repairing hardware architecture for safety-critical cyber-physical-systems," *IET Cyber-Physical Systems: Theory & Applications*, vol. 5, no. 1, pp. 92–99, 2020.

[19] A. Pakonen, I. Buzhinsky, and K. Björkman, "Model checking reveals design issues leading to spurious actuation of nuclear instrumentation and control systems," *Reliability Engineering & System Safety*, vol. 205, p. 107237, 2021.