Towards user-friendly model checking of IEC 61499 systems with counterexample explanation

Polina Ovsiannikova*† and Valeriy Vyatkin*†‡
*Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia
†Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
‡Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden
Email: polina.ovsiannikova@aalto.fi, vyatkin@ieee.org

***Abstract*—Distributed automation systems design is getting increasingly popular as the systems become more complex and modular. The IEC 61499 automation architecture is seen as the main enabler of component software design for distributed automation systems. It requires novel approaches to integrated development environments (IDE) supporting the engineering of such component software systems. One important part of automation systems engineering is their verification and validation. The efficacy of verification is largely dependent on how seamlessly this process is integrated with the traditional development tools, such as editors, compilers and debuggers. This work presents a start of such a development, that automatizes the model checking process for IEC 61499 systems and provides a visual explanation of its results in the graphical development environment FBME.**

***Index Terms*—IEC 61499, user-friendly model checking, counterexample explanation, distributed systems**

I. Introduction

IEC 61499 [1] is an international standard for the development of distributed automation and control systems. According to this standard, the controller part of a cyber-physical system is formulated using special elements, function blocks (FB), which, connected to each other, form function block diagrams (FBD)[1]. For distributed systems that are safety-critical, it is especially important to satisfy their predefined requirements as any failure may cost substantial material losses or even human lives. Such systems should be rigorously verified, starting from the formal model level, so that all the possible faults are identified and the logic that causes them is corrected. However, to the best of the authors' knowledge, this whole workflow starting from verification and ending in debugging is not implemented in the existing integrated development environments (IDEs) for ICE 61499 in a user-friendly way.

There exist several approaches to formal verification and one of them that explores the whole model state space is model checking [2]. The formal model of the system together with its specification written in terms of formal languages (e.g., temporal logic) form the input of a verification tool, model checker. In case the system violates its requirements, the model checker produces a counterexample, i.e., a sequence of system states, starting from one of the initials, (or a model trace) where the system property does not hold.

The toughest part comes after the model checking is completed and some violations of the specifications are revealed. In this case, subsequent debugging of the model involves analysis of the counterexample produced by the model checker. First, each system state included in the counterexample is composed of values of all the system variables, and their amount may reach hundreds in industrial-sized systems. Second, the counterexample may contain dozens of states, turning it into a large table of values without information about the model structure, almost impossible to be used in reasoning. The situation gets worse for IEC 61499 systems as prior to verification, they should be converted into formal models, supported by the model checker, which may introduce additional variables and complicate the counterexample decoding. Meanwhile, the initial challenge of the user is not only to verify the system and decode a counterexample but find the root causes of the issue in a most representative way, which is graphical in our case. Therefore, our goal is to develop a method and an algorithm for a visual explanation of errors detected in model checking in terms of IEC 61499.

In this work, we present initial studies on visual counterexample explanation for IEC 61499 systems from [3] which will be implemented in a user-friendly model checking plugin to the existing IDE, FBME [4]. We also enhance the algorithms from [3] and provide a draft of the explanation visualization. The whole toolset to be implemented automatizes the following three steps that are done manually in the current work. First, we convert the system developed in IEC 61499 standard to a formal model suitable for one of the most well-known model checkers, NuSMV [5], with the tool FB2SMV [6]. Then, we use NuSMV to check if the system satisfies its specification provided in form of linear temporal or computation tree logic (LTL and CTL). If the counterexample is produced, we utilize it to infer the influence paths in an FBD that lead to particular values of the variables. The main contribution of the current paper is the method of inferring influence paths in an FBD of a system which then can be visualized in an FBME to facilitate the understanding of the violation, which increases the user-friendliness of model checking.

II. Preliminaries

A. IEC 61499

IEC 61499 defines a design paradigm for distributed automation and control systems. The system here is represented as an FBD, which is formed by a set of interconnected FBs

[1]Hereinafter we consider FBDs developed according to IEC 61499 standard.

that communicate to each other. Each FB has two types of inputs and outputs, i.e., data and event interfaces. Any event input or output can be bound to a subset of data inputs or outputs respectively, which means that the corresponding data will be received and processed or sent only if the particular event fires. Intuitively, any update of the event variable opens the gates for the data connected to it.

There exist FB of three kinds: basic, composite and service interface. The kind of block defines its function, architecture and logic. A basic FB is the fundamental element of IEC 61499 architecture. Its logic is defined by an execution control chart (ECC), which is, essentially, a Moore automaton and consists of states, transitions and actions. Its transitions are guarded by the conditions on input events and variables of the FB along with its internal variables. ECC actions may be of two types, i.e., emitting an output event or an algorithm execution, where the algorithms are formulated using structured text, a programming language of IEC 61131-3 [7]. Whenever any of incoming transitions of a state is executed, the corresponding algorithms are run and the events are fired. Next, composite FBs encapsulate nets of interconnected basic FBs and provide an interface for the inputs and outputs of the internal diagram. The function of the composite FB is defined by data and event flows in its internal diagram. The last type, a service interface FB are not considered in this work. The final FBD is assembled using the available FBs.

B. Checking and debugging of IEC 61499 applications

There exists a good amount of research on the application of different formal verification techniques to IEC 61499 programs. Usually, the authors divide them into dynamic and static [8] or online and offline [9] verification. While the first group of methods aims to observe the system during its operational state (or simulation) and inform the analyst if the system's behavior violates the specification here and now, the second group is responsible for pre-running check.

In this paper, we deal with pre-operational checks and address static verification. There exist various modeling approaches of IEC 61499 applications that aid in subsequent verification. For example, the authors of [10] design IEC 61499 systems using UML and generate test cases for functional and non-functional requirements for the application. Their final FBD is inferred automatically from the UML model. On the other hand, a more thorough way to analyze the model state space, model checking, is used in the number of works [9], [11], [12], [13], [14]. They propose modeling systems using SIGNAL formalisms [9], transition systems [14], translating ECC charts to timed CTL [11] and model checking them in SIGNAL, SMV-verifiers or UPPAAL [15] correspondingly. Approaches [12] and [13] translate the programs into Prolog code or signal-net systems in order to perform model checking.

As we can see, the work on verification of IEC 61499 systems and on their model checking, in particular, has already been done. Among the mentioned works, probably, the most user-friendly approach to verification is suggested in [13], where the failure trace can be visualized in the signal signal-net system that was verified and in the original Net Condition/Event System. Our approach differs from the mentioned works, by making a step further in user-friendly verification, that involves not only displaying the counterexample in the FBD but a visual guidance along the paths that influenced values of variables they choose and make the verification process seamless.

III. Inference of an influence graph

In this section, we present the general method of counterexample explanation using an example, skipping the formal definitions. We work with FBDs developed according to the IEC 61499 standard and call them *systems*. We extend their full sets of variables with the events of their nested FBs and ECC states of the basic FBs treating them as Boolean variables. This means, that when an event fires or an ECC state becomes active their corresponding variables become `true`. Thus, we have three kinds of variables: events, ECC states and model variables. Assume that the system was converted into the format of a model checker, verified and the specification was violated.

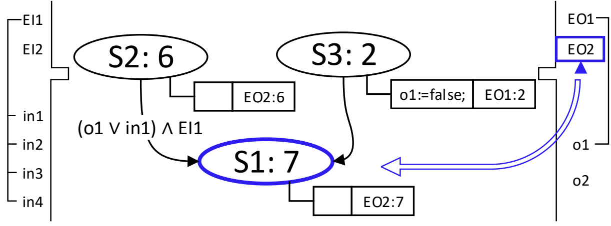
Now, a counterexample produced by a model checker does not include information about the system structure, and restoration of the "story" taken place in the system requires a considerable amount of time and effort. Such "story" exists for each element (or variable) of the system, i.e., an event, a variable or an active ECC state, at each counterexample step and essentially is a directed graph of changes of system variables in the past in relation to the chosen variable at the chose step. In this graph, each node is *an assignment* a, which is a tuple $(u(a), s(a), v(a))$, where $v(a)$ is a value of a particular variable $(u(a))$ at a particular counterexample step $(s(a))$. We call such a graph *an influence graph* and build it for an assignment chosen by the user, which we call *explanation target*, as follows.

Starting in the explanation target, we traverse the FBD backwards structurally and according to counterexample steps numbers. Each kind of a system variable has its own algorithm that returns child nodes to be added to the result graph and which is schematically depicted in Figure 1.

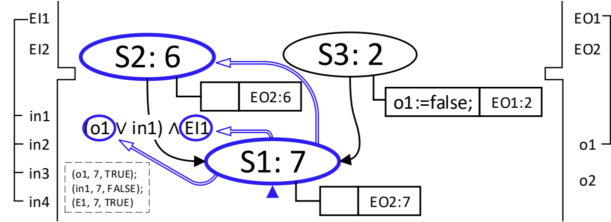
We infer child nodes for an event a_e only if it has fired. In this case, two scenarios are possible. First, if $u(a_e)$ is an output of composite FB or input of any FB, then, following incoming connections, we take an event that fired, which is the closest in past according to counterexample step numbers (Figure 1a). Otherwise, the child is the ECC state that emitted a_e.

Children of an ECC state a_s (Figure 1b) are the states that are reachable following incoming transitions of a_s in case they are unconditional. If there are incoming transitions that have guards, we take the one that was active at $s(a_s)$ and search for important assignments that caused the predicate to be `true`. Hereinafter, the latter can be done with one of the algorithms developed in our previous works [16] or [17].

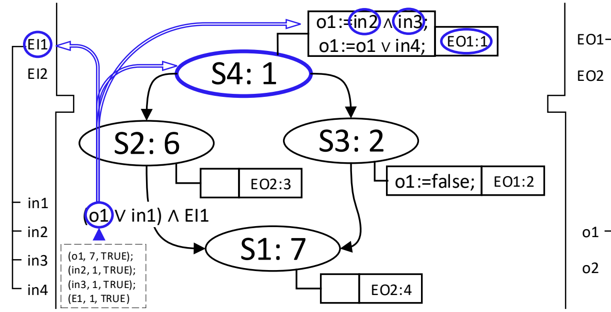
To find child nodes for a variable a_v, we check the structural position of $u(a_v)$ in D. If it is an output of a composite FB, we follow the incoming connection and pick the assignment of



(a) Explanation for an event variable. Assume that event EO2 fired at counterexample step 7.



(b) Explanation for an ECC state variable.



(c) Explanation for a model variable.

Fig. 1: Conceptual models showing explanation rules for each of the variables kinds in a part of IEC 61499 basic block. Blue triangles mark the explanation target, empty blue arrows direct the attention of the reader to the child nodes of the explanation target in the influence graph, which are marked with bold blue ovals. Notation "`v : n`" means that variable `v` is `true` at counterexample step `n`. In dashed rectangles we provide assignments of important variables.

the found variable at step $s(a_v)$. In case $u(a_v)$ is an input of any FB in D, first, we take assignments of the closest events in the past that fired when $u(a_v)$ changed its value. Then, we add to this set the assignment of the output variable connected to $u(a_v)$ from the step when the events fired. In the last case, when $u(a_v)$ is an output of a basic FB (Figure 1c), first, we look for a counterexample step when the value of the variable was changed and take assignments of the output events that fired at the found step and which are connected to the variable. Then, we look for ECC states that change the value of $u(a_v)$ in their algorithms and pick the one closest in the past to the step when the value was changed. This state is added to the final result together with the explanation of the right part of the assignment expression.

IV. Case Study

A. System model

We checked the applicability of the algorithms described in Section III for debugging by applying them manually on the model similar to a drilling station with distributed control that

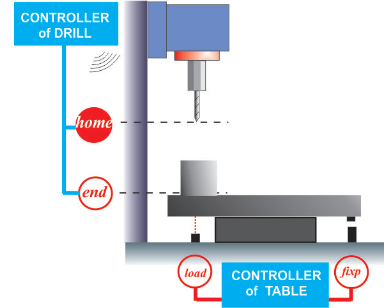


Fig. 2: A schematic view of a drilling station with rotating table feed.

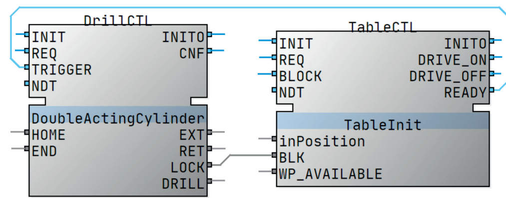
was used in a case study for verification tool VEDA [18]. The system consists of a drill and a table that rotates under the drill delivering workpieces to the position of the drill. The drill can be switched on only when the table is in a fixed position and there is a workpiece below, which is detected by the sensor. After the drilling is finished, the table drives the workpiece away. We implemented the system in IEC 61499 and tested it on a real system in a laboratory using NxtStudio development environment and Iceblock PLCs.

The control of the system consists of two basic FBs: drill and table controllers. The overall FBD of the system is formed by the interconnected controllers and FBs representing sensors, carriage and drilling mechanisms. To save space, in Fig. 3 we provide a screenshot only of interaction between two mentioned controllers and parts of their ECCs, which are relevant for our case study.

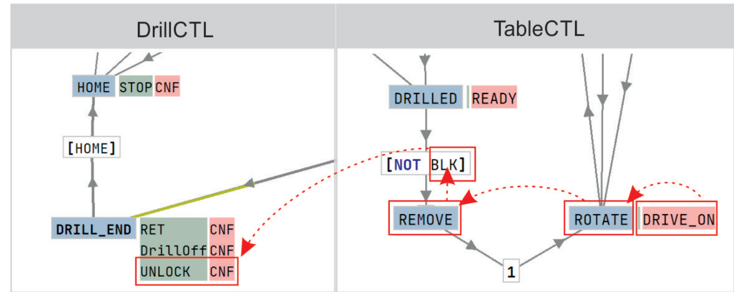
B. System verification

One of the main safety properties that the system described above should satisfy is that the detail being processed must not be moved to another position until the drilling is over and the drill is lifted. A failure in this specification leads to the detail malfunctioning. Such property may be represented in LTL form as follows: $\mathbf{G}\,\neg(DrillCTL.RET \wedge TableCTL.DRIVE_ON)$. As shown in Fig. 3b, `DrillCTL.RET` does not mean that the drill is returned to its initial position and one more transition guarded by `HOME` is required to make sure that the drill is lifted.

To check this specification, first, we used FB2SMV to generate the SMV model of our system. Then the obtained model together with its LTL property was sent as input to the NuSMV model checker, which showed that the requirement does not hold and generated a counterexample. The length of the latter was 520 states, each of which contained 167 variables. To localize the failure we used an LTL counterexample visualizer from [16], and found out that the failure occurs between steps 381 (where `DrillCTL.RET` becomes `true`) and 401 (where the `TableCTL.DRIVE_ON` event is emitted). Now, to learn why `TableCTL.DRIVE_ON` had been emitted before `DrillCTL` was in its `HOME` state, we start building the influence graph in `TableCTL.DRIVE_ON` at step 401. This assignment was influenced by active ECC state `ROTATE` at step 399 that, in turn, was obtained as a result of an unconditional transition from state `REMOVE` at step 395. Here we see that the model



(a) Interfaces of the drill and the table controllers. The rest of the FBD is omitted to save space.



(b) Parts of the controllers ECC showing the process of the fault localization. Red bold rectangles highlight the elements, whose assignments are nodes of an graph of influences with the root in DRIVE_ON which is `true` at counterexample step 401. Red dashed arrows imitate edges of this graph.

Fig. 3: Communication between the drill and the table controllers for the system from Fig. 2 implemented in FBME: interfaces and parts of the ECC views.

occurred in this state because BLK at step 385 became `false`, which happened due to action UNLOCK in the state DRILL_END of the drill controller that set its output LOCK to `false` at step 385.

This deduction shows us the scenario where the detail is moved away from the drill while the drill is still not lifted and even though it is stopped, this situation remains critical.

V. Conclusions and Future Work

Thorough verification is a crucial stage of the design process of industrial automation systems. This paper gives a start to our work on user-friendly model checking for IEC 61499 applications and shows the feasibility of using additional verification techniques for agile automation systems. Our goal is to provide analysts with the appropriate tools that will allow them to benefit from model checking without having actual knowledge in formal methods. We begin with the visualisation of the influence graph in an FBD and its draft is shown in Figure 3b. The case study explains how the graph is inferred and illustrates the usefulness of such a toolset, revealing the failure scenario which is hard to obtain and notice during the simulation or conventional testing.

One of the main goals of our future work is to automatize the manual process we performed in our case study and integrate it into a novel IDE for IEC 61499, FBME [4]. The first steps towards this have already been done. We developed a debugging panel where a counterexample for the opened FBD can be uploaded and visualized in the diagram. Our future directions include improving the models, generated by FB2SMV to enable automatic verification of systems with integer variables and checking open-loop scenarios along with user-friendly graph of influences visualization.

Acknowledgments

This work was supported, in part, by the H2020 project 1-SWARM co-funded by the European Commission (grant agreement: 871743) and by the Government of the Russian Federation under Grant 08-08.

References

[1] V. Vyatkin, *IEC 61499 function blocks for embedded and distributed control systems design*. ISA-Instrumentation, Systems, and Automation Society and O3neida, 2012.
[2] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.
[3] M. Tereshchuk, "An algorithm and a tool for visualization of the causes of a cyber-physical system specification violation," Bachelor's Thesis, ITMO University, Saint Petersburg, Russia, 2020.
[4] JetBrains, "Jetbrains/fbme," 2020. [Online]. Available: https://github.com/JetBrains/fbme
[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.
[6] (2021) FB2SMV: IEC 61499 function blocks xml code to smv converter. [Online]. Available: https://github.com/dmitrydrozdov/fb2smv
[7] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 1995.
[8] J. O. Blech, P. Lindgren, D. Pereira, V. Vyatkin, and A. Zoitl, "A comparison of formal verification approaches for IEC 61499," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–4.
[9] C. Schnakenbourg, J. Faure, and J. Lesage, "Towards IEC 61499 function blocks diagrams verification," in *IEEE International Conference on Systems, Man and Cybernetics*, vol. 3, 2002, pp. 6 pp. vol.3–.
[10] T. Hussain and G. Frey, "Uml-based development process for IEC 61499 with automatic test-case generation," in *2006 IEEE Conference on Emerging Technologies and Factory Automation*, 2006, pp. 1277–1284.
[11] B. Glatz, F. Cleary, M. Horauer, H. Schuster, and P. Balog, "Complementing testing of iec61499 function blocks with model-checking," in *2016 12th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, 2016, pp. 1–7.
[12] V. Dubinin, V. Vyatkin, and H. Hanisch, "Modelling and verification of IEC 61499 applications using prolog," in *2006 IEEE Conference on Emerging Technologies and Factory Automation*, 2006, pp. 774–781.
[13] V. Vyatkin and H. M. Hanisch, "Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems," in *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.01TH8597)*, vol. 2, 2001, pp. 113–118 vol.2.
[14] V. Dubinin, V. Vyatkin, and A. Shalyto, "Formal modeling and verification of IEC 61499 function blocks on the basis of transition systems," in *2016 International Siberian Conference on Control and Communications (SIBCON)*, May 2016, pp. 1–4.
[15] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," 2006.
[16] 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.
[17] 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.
[18] V. Vyatkin and H.-M. Hanisch, "Verification of distributed control systems in intelligent manufacturing," *Journal of Intelligent Manufacturing*, vol. 14, no. 1, pp. 123–136, 2003.