


# Formal Verification of PLCs as a Service: A CERN-GSI Safety-Critical Case Study (extended version)

Ignacio D. Lopez-Miguel<sup>1</sup>, Borja Fernández Adiego<sup>2</sup>, Matias Salinas<sup>3</sup>, and  
Christine Betz<sup>3</sup>

<sup>1</sup> TU Wien, Vienna, Austria [ignacio.lopez@tuwien.ac.at](mailto:ignacio.lopez@tuwien.ac.at)

<sup>2</sup> CERN, Beams Department, Geneva, Switzerland

[borja.fernandez.adiego@cern.ch](mailto:borja.fernandez.adiego@cern.ch)

<sup>3</sup> GSI, Darmstadt, Germany [{m.salinas,c.betz}@gsi.de](mailto:{m.salinas,c.betz}@gsi.de)

**Abstract.** The increased technological complexity and demand for software reliability require organizations to formally design and verify their safety-critical programs to minimize systematic failures. Formal methods are recommended by functional safety standards (e.g., by IEC 61511 for the process industry and by the generic IEC 61508) and play a crucial role. Their structured approach reduces ambiguity in system requirements, facilitating early error detection. This paper introduces a formal verification service for PLC (programmable logic controller) programs compliant with functional safety standards, providing external expertise to organizations while eliminating the need for extensive internal training. It offers a cost-effective solution to meet the rising demands for formal verification processes. The approach is extended to include modeling time-dependent, know-how-protected components, enabling formal verification of real safety-critical applications. A case study shows the application of PLC formal verification as a service provided by CERN in a safety-critical installation at the GSI particle accelerator facility.

## 1 Introduction

Formal methods play an essential role in ensuring the reliability, quality, and safety of software systems. They are recommended by industry standards and provide a mathematical approach to software development. One of these standards is DO-178C [25] in the aviation domain, which is accompanied by a guideline on formal methods (DO-333 [26]). The latter enhances the former by explaining how to use formal methods in every stage of the software lifecycle.

Large scientific installations, like particle accelerators, do not have specific standards. However, they tend to apply the generic IEC 61508 [13] and IEC

---

This paper is an extended version of our NFM 2025 paper “Formal Verification of PLCs as a Service: A CERN-GSI Safety-Critical Case Study”. It adds an appendix with the complete modeling of a know-how-protected function and with examples of found discrepancies during verification.

61511 [17] functional safety standards to design, develop, and validate their safety-critical software. IEC 61508 recommends using formal approaches in different parts of the software lifecycle according to the criticality of the component. IEC 61511 recommends the usage of formal methods to specify requirements. ISO 26262 [18] for road vehicles also recommends the use of formal methods for critical components, and IEC 61513 [14] for nuclear power plants emphasize the importance of rigorous development and verification processes to ensure the safety and reliability of safety-critical systems.

All these standards agree that, although formal methods can be expensive, identifying discrepancies between the code and the requirements in the early development stages results in substantial cost reduction in later phases.

However, some organizations might lack the resources to introduce formal methods in their software development process. *Formal verification as a service* addresses this need, offering formal methods expertise. It establishes a win-win situation where organizations benefit from the skills of experts, and service providers improve their tools based on the different case studies. It contributes to quality assurance, enabling organizations to demonstrate to regulatory authorities that exhaustive measures have been taken to ensure safety.

In this paper, we focus on the formal verification of PLC (programmable logic controller) programs as a service. Our contributions are summarized below:

1. We present a collaboration model between the different stakeholders of a PLC project development and the formal verification service providers. It complies with the functional safety standards by ensuring independence and by using formal verification at the early stages of the PLC program lifecycle.
2. We introduce a methodology based on simulation and formal verification to model *know-how-protected functions*, which are proprietary functions whose precise behavior is hidden by the manufacturer (black boxes). They are commonly used in PLC programming, and some include time-dependent components, complicating their modeling. Their exact behavior must be known to formally verify a complete PLC program containing these functions.
3. We show a real case study in which PLCverif [4] was used to verify a safety-critical system containing know-how-protected functions at the particle accelerator installation at GSI Helmholtz Centre for Heavy Ion Research [12].

## 2 Service approach

### 2.1 Collaboration model

Figure 1 depicts the proposed diagram to offer formal verification of PLC programs as a service [21]. It is composed of the following independent teams as recommended by IEC 61511-1 clause 12 [16] of having an independent body in charge of validating the critical software:

- *Requirement engineers*. They are responsible for analyzing the systems and writing their technical requirements using different formalisms. They have

the best knowledge of the actual physical system for which the PLC program is developed, and they know how the system should behave. That is why they write and distribute the requirements to the other teams.

- *PLC program developers.* They follow the requirements handed out by the requirement engineers to implement the PLC program. If the requirements are clear, the interaction with the requirement engineers can be minimal. Their PLC program is then shared with the other two teams.
- *Formal verification engineers.* They ensure that the PLC program behaves exactly as written in the requirements using formal verification. The requirements engineers are informed when a discrepancy between the PLC code and the requirements is found. Then, they work with the developers to solve it.

It is important to highlight the iterative nature of this process. Especially when a discrepancy is found, formal verification engineers need to inform requirement engineers, who will work with the developers to find the root cause of the error. This will lead to updated requirements and/or PLC programs, which are then given again to the formal verification engineers so they can continue their work. This process is repeated until no more discrepancies are found.

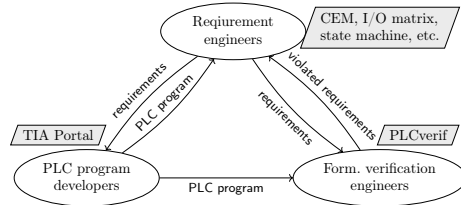
Although this process does not entirely ensure the lack of errors in the code or in the requirements due to the possible bugs in the program verifier, it drastically increases the confidence of the requirement engineers with the PLC program. It also helps them show authorities they made considerable efforts to guarantee the safety of the installation. In fact, formal verification, compared to other methods like testing, can identify more hidden bugs (corner cases).

Another important formal method to mention at this point is the synthesis of PLC programs [34], which would make formal verification redundant since the code would be correct by construction. However, synthesis tools for PLC programs are not widespread in the industry.

In the next two subsections, from Figure 1, we will further explain how the requirement engineers can formalize requirements and how the formal verification engineers verify the given PLC code according to the formalized requirements.

## 2.2 Formal specification

Requirements can be represented using diverse formalisms, which should be simple, clear, and concise for use across the software development lifecycle. The



**Fig. 1.** Diagram with the roles of the collaboration, shared information and used tools.

examples in this section meet these criteria, were applied in the case study (Section 3), and align with functional safety standards. IEC 61511-2 [15] recommends methods like cause-and-effect matrix, state machines, and logic diagrams; IEC 61511-1 [16] provides examples of state machines and logic diagrams.

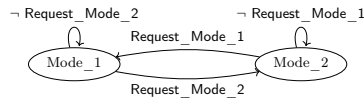
- A *Cause-and-effect-matrix (CEM)* [8] is a tabular representation of Boolean expressions. It is particularly suitable for stateless logic like interlock logic. The example from Table 1 assigns values to the outputs according to  $Out\_1 = (In\_1 \wedge In\_2) \vee (\neg In\_3 \wedge \neg In\_4)$ , and  $Out\_2 = \bigwedge_{i=1}^4 In\_i$ .
- An *input-output matrix (I/O matrix)* gives the conditions to set or reset output variables. One needs to ensure that the inputs are mutually exclusive or to specify output priorities. The I/O matrix from Table 2 shows an example.
- A *state machine* is a graphical representation used to depict the behavior of a system consisting of different states and transitions between them. Figure 2 shows a simple state machine that changes from two modes depending on the requests. For a real example, one can refer to [35, Figure 4.4].

		Outputs	
		Out_1	Out_2
Inputs	In_1	A1	A1
	In_2	A1	A1
	In_3	NA2	A1
	In_4	NA2	A1

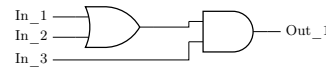
**Table 1.** Example of a CEM.

		Outputs	
		Out_1	Out_2
Inputs	In_1	Reset	Reset
	In_2	Set	Reset
	In_3	Set	Set

**Table 2.** Example of an I/O matrix.



**Fig. 2.** Example of a state machine.

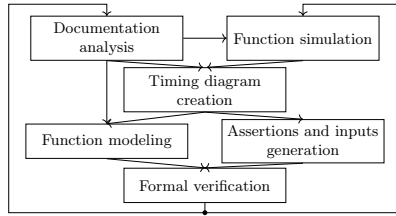


**Fig. 3.** Example of a logic diagram.

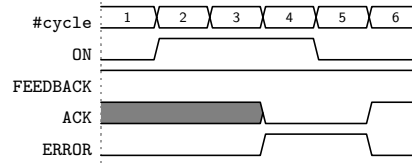
- A *logic diagram* visually represents logical relationships. Figure 3 depicts an example, representing  $Out\_1 = (In\_1 \vee In\_2) \wedge In\_3$ .
- An *assertion* expresses a property of a program at a particular point in the code's execution. Although this is typically used during software development, it can be used to formalize requirements. They are particularly helpful in expressing safety properties, i.e., a state can never be reached.

### 2.3 Formal verification

Formal verification as a service not only verifies the PLC program but also helps to amend errors in the requirements, helping requirement engineers understand the PLC program's behavior better. To formally verify PLC programs after the requirements are formalized, PLCverif [6,22] was used. The reasons for using it are that it is actively developed, has high coverage of PLC languages, uses state-of-the-art model checkers, has been used in real systems, and is partially



**Fig. 4.** Proposed diagram to model a know-how-protected built-in function.



**Fig. 5.** Example of a timing diagram for a simplified version of the FDBACK function.

automated. Other solutions, such as Arcade.PLC [2], MODCHK [24], PLCInspector [33], STBMC [19] or the ones in [23] lack some of these capabilities.

Other papers show how to use PLCverif ([7,9,10,20]), so due to space constraints, we will focus on the modeling of time-dependent know-how-protected functions.

PLC programming platforms such as TIA Portal [27] for Siemens PLCs include know-how-protected functions to simplify some tasks of the PLC developer. These are proprietary functions whose behavior is hidden by the manufacturer. To verify PLC programs that use these functions, it is necessary to understand their behavior precisely. Functions that involve time are particularly challenging, as they require the propagation of signal values across successive PLC cycles.

To model these functions, we propose a method that combines simulations and formal verification (Figure 4). This process produces a model in PLC code of the know-how-protected function using transparent functions and operators that can be used in PLCverif as part of the verification of the whole PLC program.

This process can be considered automata learning since the internal representation of PLCverif uses control flow automata (CFA) [1]. In fact, the conditions of the PLC program and the assignments are translated into transitions and assignments in the CFA. However, since no tools generate PLC code from automata, learning the PLC code directly was deemed more efficient. Furthermore, having the PLC code allowed us to verify it without any extra effort with PLCverif, and to include it directly in the verification of the whole PLC project.

To explain this process (Figure 4), we will use a simplified version of the know-how protected FDBACK function from TIA Portal [30, section 13.3.7]. It checks if the inputs  $ON=0$  and  $FEEDBACK=1$ , and produces an error otherwise. It is used to monitor systems. This example is particularly relevant since it involves time, its documentation is complex, and it was often used in our case study.

- *Documentation analysis.* Our simplified FDBACK function checks whether input  $ON=0$  and input  $FEEDBACK=1$ . Output  $ERROR$  becomes 1 if this does not happen after a given maximum time (e.g., two PLC cycles). Once  $ERROR=1$ , an acknowledgment  $ACK$  is necessary to reset it.
- *Function simulation.* Given the documentation, a simple PLC program is created to simulate the given function (cf. Appendix A.1).

- *Timing diagram creation.* From the simulation of the function and the documentation, we produce a timing diagram. The input variables are changed manually to capture all the possible behaviors from the documentation. Figure 5 exemplifies a timing diagram for this function. Initially,  $ON=0$  and  $FEEDBACK=1$ , thus  $ERROR=0$ . Then  $ON$  turns 1, leading to  $ERROR=1$  after the maximum time (two cycles) is reached. Although  $ON$  becomes 0 again,  $ERROR$  keeps its value until there is an acknowledgment ( $ACK=1$ ) (cf. Appendix A.2).
- *Assertions and inputs generation.* The timing diagram is automatically translated for every cycle into assignments for the inputs and assertions for the outputs. Since  $ACK$  is non-deterministic in the first three cycles, no value is assigned to it. For the first cycle, the assignments are  $FEEDBACK := 1$ ,  $ON := 0$ , and the assertion is  $cycle = 1 \rightarrow \neg ERROR$  (cf. Appendix A.3).
- *Function modeling.* Given the documentation, a simple PLC program is created to simulate the given function (cf. Appendix A.4).
- *Formal verification.* The modeled function is verified with respect to the generated assertions. This ensures that the modeled function behaves as the one in TIA Portal with respect to all the simulated scenarios (cf. Appendix A.5).

This process continues iteratively until no discrepancies are found between the documentation, the timing diagrams, and the PLC model. The modeling of the original  $FDBACK$  function resulted in about 100 lines of code [5, Line 843].

### 3 Case study

The approach presented in this paper was applied to verify the Personnel Access System (PAS) of the FAIR accelerator facility at the *GSI Helmholtzzentrum für Schwerionenforschung* [12]. PAS [11] is a very critical system that prevents personnel from entering areas exposed to particle beams and their radiation. Thus, a failure in the PAS PLC program could have very severe consequences. This PLC program is highly configurable, making exhaustive testing unfeasible due to the enormous number of combinations in the PLC program. Also, it is developed using TIA Portal, hence, it utilizes know-how-protected functions.

Due to CERN’s expertise in the verification of different PLC projects [7,10] and the continuous development of PLCverif, GSI trusted CERN to verify its PAS PLC project. The collaboration was set up as described in Section 2 with three independent teams: (i) Requirement engineers (GSI). (ii) Formal verification engineers (CERN). (iii) Developers (a different team at CERN).

A summary of the results produced by this collaboration is shown below:

- The requirements were formalized according to Section 2.2, leading to a better understanding of the desired program behavior and less ambiguities;
- The PLC program was fully aligned with the formal requirements, amending detected discrepancies (cf. Appendix B for examples of found discrepancies);
- PLCverif was enhanced to support additional know-how-protected functions, including  $FDBACK$ ,  $CTUD$ ,  $ESTOP1$ , and  $FDB\_TIME$  [30]. They are now included in the set of covered functions by PLCverif [4] (delivered together with PLCverif in the `builtin.scl` file [5]) and can be used in future projects.

## 4 Conclusion

The presented approach for formal verification as a service can help to detect errors early, reduce ambiguity, and improve requirements precision. To the best of our knowledge, this is the first time an organization trusted another to formally verify a complete, real-world, safety-critical PLC project (other collaborations like ITER-CERN [10] focused on the verification of specific modules). We hope the presented approach demonstrates that formal methods are feasible, beneficial, and compliant with functional safety standards in safety-critical PLC projects, enabled by organizational collaborations.

As part of our future work, we will seek a more automated process of modeling know-how-protected functions to increase the coverage of PLCverif. We will also work on the automation between requirement specification and verification, ultimately leading to correct-by-construction code generation. This is not a straightforward path due to different challenges, such as a lack of formal tools, legacy systems, established workshops, regulations, and the need for training.

## Acknowledgments



The project leading to this application has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440 and by the WWTF project ICT22-023.

## References

1. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. *International Journal on Software Tools for Technology Transfer* **9**(5-6), 505–525 (Sep 2007). <https://doi.org/10.1007/s10009-007-0044-z>, <https://doi.org/10.1007/s10009-007-0044-z>
2. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: A Verification Platform for Programmable Logic Controllers . In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. pp. 338–341. ASE 2012, ACM (2012), <http://publications.embedded.rwth-aachen.de/file/3w>
3. Blanco Viñuela, E., Darvas, D., Sallai, G.: Testing Solutions for Siemens PLCs Programs Based on PLCSIM Advanced. In: *Proc. ICALEPCS'19*. p. 1108. No. 17 in *International Conference on Accelerator and Large Experimental Physics Control Systems*, JACoW Publishing, Geneva, Switzerland (10 2020). <https://doi.org/10.18429/JACoW-ICALEPCS2019-WEPHA018>
4. CERN: PLCverif (2024), <https://gitlab.com/plcverif-oss>
5. CERN: Built-in functions (2025), <https://gitlab.com/plcverif-oss/plcverif-builtin-projects/-/blob/master/builtin.scl>
6. Darvas, D., Viñuela, E.B., Molnár, V.: PLCverif Re-engineered: An Open Platform for the Formal Analysis of PLC Programs. In: *Proc. ICALEPCS'19*. pp. 21–27. JACoW Publishing, Geneva, Switzerland (08 2020). <https://doi.org/10.18429/JACoW-ICALEPCS2019-MOBPP01>

7. Fernandez Adiego, B., Lopez-Miguel, I.D., Tournier, J.C.: Applying Model Checking to Highly-Configurable Safety Critical Software: The SPS-PPS PLC Program. In: Proc. ICALEPCS'21. JACoW Publishing, Geneva, Switzerland (10 2021). <https://doi.org/10.18429/JACoW-ICALEPCS2021-WEPV042>
8. Fernández Adiego, B., Blanco Viñuela, E., Bonet, M., Charrondiere, M., Hamisch, H., Speroni, R., de Queiroz, M.: Cause-and-Effect Matrix Specifications for Safety Critical Systems at CERN . In: Proc. ICALEPCS'19. JACoW Publishing, Geneva, Switzerland (10 2019). <https://doi.org/10.18429/JACoW-ICALEPCS2019-MOPHA041>
9. Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.C., Bliudze, S., Blech, J., González, V.: Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics* **11**, 1400–1410 (12 2015). <https://doi.org/10.1109/TII.2015.2489184>
10. Fernández Adiego, B., et al.: Applying Model Checking to Critical PLC Applications: An ITER Case Study. In: Proc. ICALEPCS'17. JACoW Publishing, Geneva, Switzerland (1 2018). <https://doi.org/https://doi.org/10.18429/JACoW-ICALEPCS2017-THPHA161>
11. Gaßmann, D., et al.: The Personnel Access System for FAIR. In: 14th International Particle Accelerator Conference, IPAC 2023, Venice, Italy, May 7–12, 2023, Proceedings (2023)
12. GSI Helmholtz Centre for Heavy Ion Research: Research Fields and Experiments (2024), <https://www.gsi.de/en/work/research>, accessed = 2024-11-7
13. International Electrotechnical Commission: IEC 61508:2010 — Functional safety of electrical/electronic/programmable electronic safety-related systems. International standard, International Electrotechnical Commission (2010), <https://webstore.iec.ch/publication/22273>
14. International Electrotechnical Commission: IEC 61513:2011 - Nuclear power plants - Instrumentation and control important to safety - General requirements for systems. International standard, International Electrotechnical Commission (2011), <https://webstore.iec.ch/en/publication/5532>
15. International Electrotechnical Commission: IEC 61511-2:2016 - Functional safety - Safety instrumented systems for the process industry sector - Part 2: Guidelines for the application of IEC 61511-1:2016. International standard, International Electrotechnical Commission (2016), <https://webstore.iec.ch/en/publication/25510>
16. International Electrotechnical Commission: IEC 61511-1:2016+AMD1:2017 CSV - Functional safety - Safety instrumented systems for the process industry sector - Part 1: Framework, definitions, system, hardware and application programming requirements. International standard, International Electrotechnical Commission (2017), <https://webstore.iec.ch/en/publication/61289>
17. International Electrotechnical Commission: IEC 61511:2024 - Functional safety - Safety instrumented systems for the process industry sector. International standard, International Electrotechnical Commission (2024), <https://webstore.iec.ch/en/publication/5527>
18. International Organization for Standardization: ISO 26262-6:2018 — Road vehicles — Functional safety. Part 6: Product development at the software level. International standard, International Organization for Standardization (2018), <https://www.iso.org/standard/68388.html>
19. Lee, J., Kim, S., Bae, K.: Bounded Model Checking of PLC ST Programs using Rewriting Modulo SMT. Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (2022), <https://api.semanticscholar.org/CorpusID:254126390>

20. Lopez-Miguel, I.D., Fernández Adiego, B., Ghawash, F., Blanco Viñuela, E.: Verification of neural networks meets plc code: An lhc cooling tower control system at cern. In: Iliadis, L., Maglogiannis, I., Alonso, S., Jayne, C., Pimenidis, E. (eds.) *Engineering Applications of Neural Networks*. pp. 420–432. Springer Nature Switzerland, Cham (2023)
21. Lopez-Miguel, I.D., Fernández Adiego, B., Blanco Viñuela, E., Salinas, M., Betz, C.: Working Together for Safer Systems: A Collaboration Model for Verification of PLC Code. In: *Proc. ICALEPCS'23*. JACoW Publishing, Geneva, Switzerland (09 2022), <https://icalepcs2023.vrws.de/html/auth0780.htm>
22. Lopez-Miguel, I.D., Tournier, J.C., Fernández Adiego, B.: PLCverif: Status of a Formal Verification Tool for Programmable Logic Controller. In: *Proc. ICALEPCS'21*. JACoW Publishing, Geneva, Switzerland (03 2022). <https://doi.org/10.18429/JACoW-ICALEPCS2021-MOPV042>
23. Ovatman, T., Aral, A., Polat, D., Ünver, A.O.: An Overview of Model Checking Practices on Verification of PLC Software. *Softw. Syst. Model.* **15**(4), 937–960 (Oct 2016). <https://doi.org/10.1007/s10270-014-0448-7>, <https://doi.org/10.1007/s10270-014-0448-7>
24. Pakonen, Antti and Tahvonen, Topi and Hartikainen, Markus and Pihlanko, Mikko: 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* (2017), <https://cris.vtt.fi/en/publications/practical-applications-of-model-checking-in-the-finnish-nuclear-i>
25. Radio Technical Commission for Aeronautics: DO-178C - Software Considerations in Airborne Systems and Equipment Certification. International standard, Radio Technical Commission for Aeronautics (2012), <https://my.rtca.org/productdetails?id=a1B36000001IcmqEAC>
26. Radio Technical Commission for Aeronautics: DO-333 - Formal Methods Supplement to DO-178C and DO-278A. International standard, Radio Technical Commission for Aeronautics (2012), <https://my.rtca.org/productdetails?id=a1B36000001IcfeEAC>
27. Siemens: Totally Integrated Automation Portal (2023), <https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html>, accessed on 06/12/2023
28. Siemens: Engineering Tools. S7-PLCSIM V16 online help. Siemens, [https://support.industry.siemens.com/cs/attachments/109773149/S7-PLCSIMenUS\\_en-US.pdf](https://support.industry.siemens.com/cs/attachments/109773149/S7-PLCSIMenUS_en-US.pdf)
29. Siemens: Function Manual. SIMATIC S7-1500. S7-PLCSIM Advanced. Siemens, [https://support.industry.siemens.com/cs/attachments/109826194/s7-plcsim\\_advanced\\_function\\_manual\\_en-US.pdf](https://support.industry.siemens.com/cs/attachments/109826194/s7-plcsim_advanced_function_manual_en-US.pdf)
30. Siemens: Industrial Software SIMATIC Safety - Configuring and Programming. SIMATIC, [https://support.industry.siemens.com/cs/attachments/54110126/ProgFAIenUS\\_en-US.pdf](https://support.industry.siemens.com/cs/attachments/54110126/ProgFAIenUS_en-US.pdf)
31. Siemens: OpennessScripter: Detailed Documentation. Siemens, [https://support.industry.siemens.com/cs/attachments/109742322/109742322\\_OpennessScripter\\_DOC\\_V117\\_en.pdf](https://support.industry.siemens.com/cs/attachments/109742322/109742322_OpennessScripter_DOC_V117_en.pdf)
32. Siemens: TIA Portal Openness: API for automation of engineering workflows. Siemens, [https://support.industry.siemens.com/cs/attachments/109798533/TIAPortalOpennessenUS\\_en-US.pdf](https://support.industry.siemens.com/cs/attachments/109798533/TIAPortalOpennessenUS_en-US.pdf)
33. Xiong, J., Zhu, G., Huang, Y., Shi, J.: A User-Friendly Verification Approach for IEC 61131-3 PLC Programs. *Electronics* **9**(4) (2020). <https://doi.org/10.3390/electronics9040572>, <https://www.mdpi.com/2079-9292/9/4/572>

34. Yoo, J., Cha, S., Kim, C.H., Song, D.Y.: Synthesis of fbd-based plc design from nuscr formal specification. *Reliability Engineering & System Safety* **87**(2), 287–294 (2005). <https://doi.org/https://doi.org/10.1016/j.res.2004.05.005>, <https://www.sciencedirect.com/science/article/pii/S0951832004001279>
35. Ádám, Z., et al.: Automated Verification of Programmable Logic Controller Programs against Structured Natural Language Requirements. Tech. rep., NASA Ames Research Center (03 2023), <https://ntrs.nasa.gov/citations/20230003752>

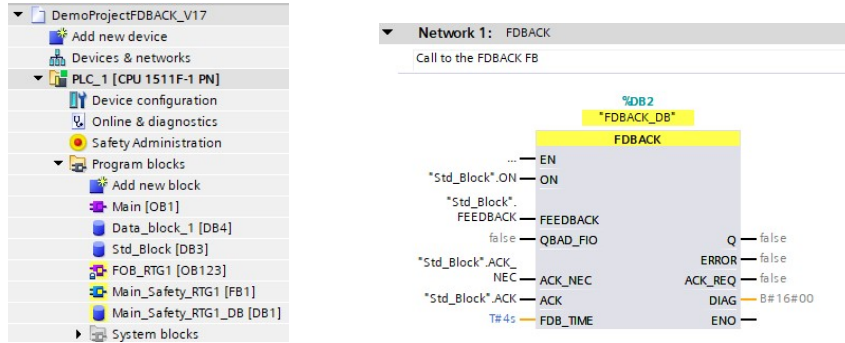
## Appendix

### A Modelling of the FDBACK function

In this appendix, we will show in detail an example of modeling a *know-how protected* function. We will use the same example as in the main text, i.e., the FDBACK function. For the steps of the process where we need to interact with TIA portal (function simulation and timing diagram creation), we will use the original FDBACK function. For the other parts, we will use the simplified version that we presented in the section 2.3 to simplify the explanation.

#### A.1 Function simulation

We created a simple TIA Portal project containing only the function we want to model. Figure 6 shows the small project structure that was used to simulate the original FDBACK function. The inputs and outputs of the FDBACK function (called by the `Main_Safety_RTG1` Function Block) are shown in Figure 7 and can be enabled for simulation.



**Fig. 6.** Project structure in TIA Portal to simulate FDBACK function. **Fig. 7.** FDBACK function interface call in the TIA Portal program.

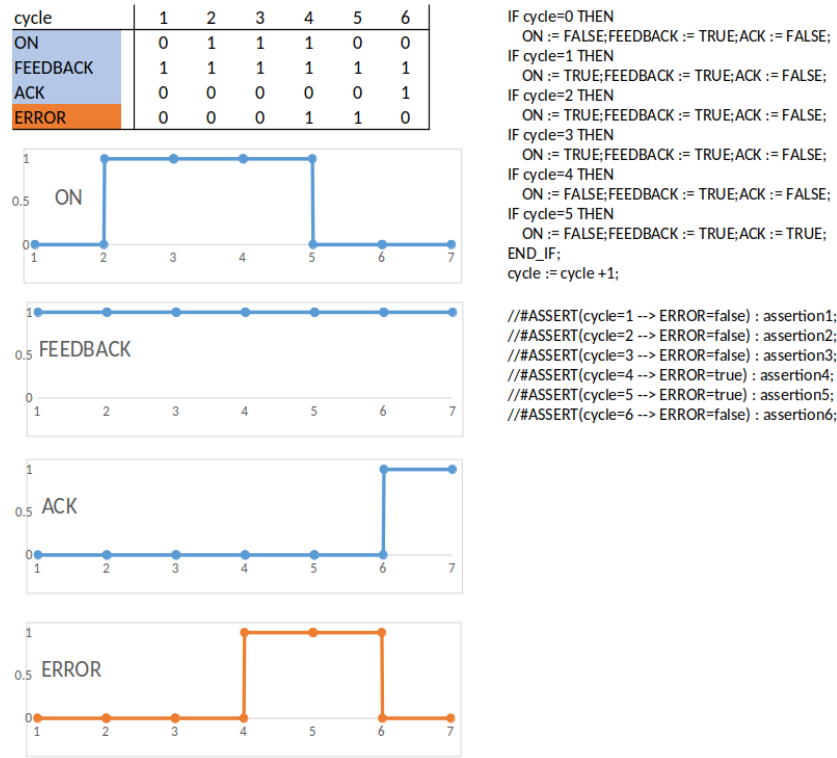
#### A.2 Timing diagram creation

For this project, we used the PLCSIM simulator [28] provided by Siemens, which is integrated into TIA portal. To simulate the FDBACK function, we manually forced its inputs according to what we wanted to check from the documentation. Once the inputs were set, the outputs were observed. Note that safety inputs (in yellow in Figures 8 to 10) cannot be forced on the simulator. For this reason, we used a standard Data Block (DB) called `std_Block`, and we assigned the variables of this DB to the safety inputs of the FDBACK function. By forcing the `std_Block` variables, we can change the input values of the FDBACK function.

Since PLCSIM does not provide a timing diagram as such, from the manual simulations, a timing diagram was created manually (cf. table from Figure 11).







**Fig. 11.** Screenshot of the spreadsheet to generate the code to verify the model of the simplified know-how protected **FEEDBACK** function.

The generated code is the one used in Listing 1.2 to verify the function. For the modeling of the original FDBACK function, six spreadsheets with 25 cycles each were used.

Although it could be possible to have a unique assertion as the one shown below for each timing diagram, it is preferable to split it into smaller assertions, as in the PLC code from Listing 1.2, to be able to find the root cause of discrepancies faster.

$$\begin{aligned}
 & (cycle = 1 \rightarrow \neg ERROR) \wedge (cycle = 2 \rightarrow \neg ERROR) \wedge (cycle = 3 \rightarrow \neg ERROR) \wedge \\
 & (cycle = 4 \rightarrow ERROR) \wedge (cycle = 5 \rightarrow ERROR) \wedge (cycle = 6 \rightarrow \neg ERROR)
 \end{aligned}$$

#### A.4 Function modeling

The model in PLC code of the simplified know-how-protected FDBACK function can be seen in Listing 1.1. This model was done manually according to the documentation and simulations. One can see that even though the requirement looks simple, its implementation is not trivial due to the timing aspect. Furthermore,

this is just a simplified version of the real one, whose model in PLC code was implemented in about 100 lines of code [5, Line 843].

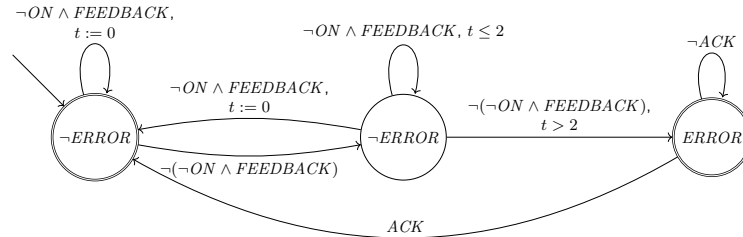
---

```

1 FUNCTION_BLOCK FDBACK_simplified
2   VAR_INPUT
3     ON : BOOL;
4     FEEDBACK : BOOL;
5     ACK : BOOL;
6   END_VAR
7   VAR
8     // Elapsed Time (ET) variable is a Timer On Delay.
9     // It sets ET.Q to true after a given time
10    ET : TON;
11  END_VAR
12
13  VAR_OUTPUT
14    ERROR : BOOL := FALSE;
15  END_VAR
16
17 BEGIN
18  IF ERROR THEN // manual acknowledgement
19    IF ACK THEN
20      ET(IN := FALSE); // reset timer
21      ERROR := FALSE; // reset the error
22    END_IF;
23  ELSIF NOT (NOT ON AND FEEDBACK) THEN
24    // start timer (ET is the Elapsed Time function)
25    ET(IN := TRUE, PT := 2*200); // 2 cycles (each safety cycle is 200ms)
26    IF ET.Q THEN // if waiting time is over
27      ERROR := TRUE;
28    END_IF;
29  ELSE
30    ET(IN := FALSE); // reset timer
31  END_IF;
32
33 END_FUNCTION_BLOCK
    
```

---

**Listing 1.1.** PLC code modeling the simplified *know-how protected FDBACK* function.



**Fig. 12.** Timed automaton representing the simplified FDBACK function.

As mentioned in section 2.3, the problem of modeling know-how-protected functions can be understood in terms of automata learning. For the simplified FDBACK function, the corresponding timed automaton that can be extracted from the PLC code is shown in Figure 12.

## A.5 Formal verification

In order to verify that the model is working as expected, the PLC code shown in Listing 1.2 was used. It basically consists of a set of consecutive calls to the model of the function `FDBACK_simplified.FDBACK_simplified_inst()`. Each call represents a PLC cycle. For each cycle, the input variables are set to the values according to the timing diagram (see Figure 5). If a variable is not set, its value is non-deterministic, as with `ACK`. At the end of each cycle, it is checked if the output variable `ERROR` has the same value as in the timing diagram.

---

```

1 DATA_BLOCK "FDBACK_simplified_inst" FDBACK_simplified
2 BEGIN
3 END_DATA_BLOCK
4
5 FUNCTION_BLOCK call_FDBACK_simplified
6     VAR
7         cycle : INT := 1;
8     END_VAR
9 BEGIN
10
11 // setting the input variables according to the timing diagram
12 // in the cycles where ACK is not set, its value is non-deterministic
13 IF cycle=1 THEN
14     "FDBACK_simplified_inst".FEEDBACK:= TRUE;
15     "FDBACK_simplified_inst".ON := FALSE;
16 ELSIF cycle=2 THEN
17     "FDBACK_simplified_inst".FEEDBACK := TRUE;
18     "FDBACK_simplified_inst".ON := TRUE;
19 ELSIF cycle=3 THEN
20     "FDBACK_simplified_inst".FEEDBACK := TRUE;
21     "FDBACK_simplified_inst".ON := TRUE;
22 ELSIF cycle=4 THEN
23     "FDBACK_simplified_inst".FEEDBACK := TRUE;
24     "FDBACK_simplified_inst".ON := TRUE;
25     "FDBACK_simplified_inst".ACK := FALSE;
26 ELSIF cycle=5 THEN
27     "FDBACK_simplified_inst".FEEDBACK := TRUE;
28     "FDBACK_simplified_inst".ON := FALSE;
29     "FDBACK_simplified_inst".ACK := FALSE;
30 ELSIF cycle=6 THEN
31     "FDBACK_simplified_inst".FEEDBACK := TRUE;
32     "FDBACK_simplified_inst".ON := FALSE;
33     "FDBACK_simplified_inst".ACK := TRUE;

```

```

34 END_IF;
35
36 // check with assertions that the output variables have the same
37 // values than in the timing diagram
38
39 FDBACK_simplified."FDBACK_simplified_inst"() ;
40
41 // #ASSERT(cycle=1 --> ("FDBACK_simplified_inst".ERROR = FALSE)) :
42     assertion1;
43 // #ASSERT(cycle=2 --> ("FDBACK_simplified_inst".ERROR = FALSE)) :
44     assertion2;
45 // #ASSERT(cycle=3 --> ("FDBACK_simplified_inst".ERROR = FALSE)) :
46     assertion3;
47 // #ASSERT(cycle=4 --> ("FDBACK_simplified_inst".ERROR = TRUE)) :
48     assertion4;
49 // #ASSERT(cycle=5 --> ("FDBACK_simplified_inst".ERROR = TRUE)) :
50     assertion5;
51 // #ASSERT(cycle=6 --> ("FDBACK_simplified_inst".ERROR = FALSE)) :
52     assertion6;
53
54 cycle := cycle + 1;
55
56 END_FUNCTION_BLOCK

```

**Listing 1.2.** PLC code used to verify the simplified *know-how protected* FDBACK function.

As already mentioned, PLCverif was used to verify the PLC code modeling know-how protected functions. This process was straightforward once the code was generated. However, it is essential to highlight how the time was treated with PLCverif. Since the time of the PLC cycle is not relevant for verification purposes – what is important is when the output of the timer is activated – a fixed time of the PLC cycle was fixed. In this case, we used the usual safety time of the PLC cycle of  $T\_CYCLE = 100\text{ms}$ . This can be seen in Figure 13 from the verification case of PLCverif.

Furthermore, it is important to allow as many cycles as needed to be able to trigger the output of the timer. That is, if the output is triggered after  $T$  milliseconds of being started, then we should have at least  $c_c = \text{int}(T/T\_CYCLE) + 1$  cycles. We should have also at least the number of cycles that we have in the timing diagram  $c_t$ . Thus, the number of loop unwinding for a bounded model checker like CBMC should be  $\max\{c_c, c_t\}$ . In our case,  $c_c = 2$  (200ms/100ms) and  $c_t = 6$ , thus we set it to 6. This can be seen in Figure 14 from the verification case of PLCverif.

Once the verification case is executed and we get that all assertions are satisfied, as shown in Figure 15, the modeling process is finished. If PLCverif reports a violation of an assertion, it will also give us a counterexample. Then, we need to investigate where the error is coming from and amend it so that the model aligns with the timing diagram.

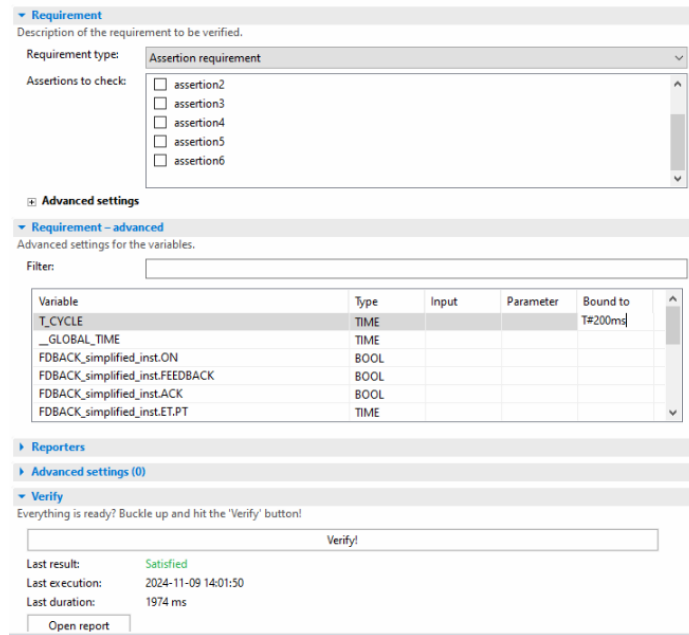


Fig. 13. Verification case of PLCverif. The time of the cycle is set to 100ms.

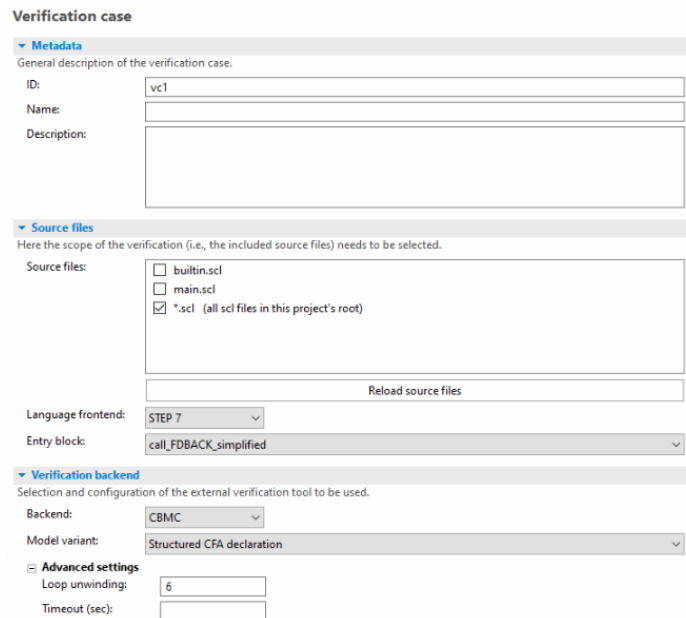



Fig. 14. Verification case of PLCverif. The loop unwinding is set to 6 to cover all the cycles in the timing diagram.

## PLCverif — Verification report



Generated on 2024-11-09 13:52:16 | PLCverif v3.0 | (C) CERN BE-ICS-AP | [Show/hide expert details](#)

<b>ID:</b>	vc1
<b>Name:</b>	
<b>Description:</b>	
<b>Source file(s):</b>	<ul style="list-style-type: none"> <li>• <a href="#">D:\CERN\cern.plcverif.gui.product-win32.win32.x86_64\workspace\FEEDBACK\builtin.scf</a></li> <li>• <a href="#">D:\CERN\cern.plcverif.gui.product-win32.win32.x86_64\workspace\FEEDBACK\main.scf</a></li> </ul>
<b>Requirement:</b>	All the assertions are to be checked.
<b>Result:</b>	<b>Satisfied</b>
<b>Verification backend:</b>	CbmcBackend (CFD_STRUCTURED-UW6)
<b>Total run time:</b>	1923 ms
<b>Backend run time:</b>	1899 ms

### Warnings and errors

Please take the following warnings and errors into account when considering the above verification result.

- **CBMC reported that the given requirement is satisfied. However, take into account that CBMC performs bounded model checking with a limited bound, thus it may not find violations which require many execution cycles to reach.** (CBMC output parsing)

### Diagnosis

No diagnosis is available.

[Show/hide more details](#)

Fig. 15. Verification report of PLCverif. All the assertions are satisfied.

## B Examples of found discrepancies

This section summarizes the discrepancies found during the verification of different PLC projects by CERN, not only during the verification of the GSI project. The examples are simplified to show where the problem lies more easily. Most of the discrepancies can be classified into the following three buckets:

1. *Incomplete requirements.* This is the most common type of discrepancy found. The implementation works as expected by the requirement engineers, but the requirements have not been formalized correctly.
2. *Bugs in the PLC program.* It happens when the requirement is correct, but the implementation has an error.
3. *Minor documentation errors.* These are simple errors that are easily fixed. A reader can understand the requirements without any issues. For example, a misspelled variable would be part of this type of error.

In the next subsection B.1, we will give some examples of incomplete requirements. However, we will not extend the other two types of discrepancies since they are self-explanatory.

Other types of problems can also be found during the formal verification of a PLC project. A recurrent one that appears in the early stages of the collaboration is *what* to specify and *how* to do it formally. Furthermore, simple things like the exact software used are sometimes not specified, as well as what happens if there are *hardware failures*.

## B.1 Incomplete requirements

The examples shown in this subsection are not exhaustive but include the majority of the most important discrepancies found. We will cover situations related to (i) priorities, (ii) incomplete diagrams and tables, (iii) and lack of explanations. We will show incomplete requirements and propose solutions to complete them.

**Priorities.** Although it is not ideal, different requirements often express conditions for the same output variables. If no priorities are set, this can lead to ambiguities. As an example, let us take the following requirements:

- ( $R_1$ ) If `v1_up`  $\rightarrow$  set `v_out`.
- ( $R_2$ ) If `v1_down`  $\rightarrow$  reset `v_out`

Listing 1.3 shows an example of how this requirement can be implemented. In this case,  $R_2$  has a higher priority than  $R_1$  since if `v1_down` is true, then `v_out` will be true no matter the value of `v1_up`. What is executed later has a higher priority. However, another implementation could change the order of the IF statements, leading to  $R_1$  having a higher priority than  $R_2$ . This ambiguity can be solved by

1. telling which of the two requirements has a higher priority,
2. if `v1_up` and `v1_down` cannot be true simultaneously (e.g., physical constraints), stating this fact,
3. adding all the necessary variables in each requirement:

- ( $R'_1$ ) If `v1_up` and not `v1_down`  $\rightarrow$  set `v_out`.
- ( $R'_2$ ) If `v1_down` and not `v1_up`  $\rightarrow$  reset `v_out`.

---

```

1 FUNCTION_BLOCK req_priorities
2   VAR_INPUT
3     v1_up : BOOL;
4     v1_down : BOOL;
5   END_VAR
6   VAR_OUTPUT
7     out : BOOL;
8   END_VAR
9 BEGIN
10  IF v1_up THEN
11    v_out := TRUE;
12  END_IF;
13  IF v1_down THEN
14    v_out := FALSE;
15  END_IF;
16 END_FUNCTION_BLOCK

```

---

**Listing 1.3.** PLC code implementing a solution for ambiguous requirements where the priorities are not set.

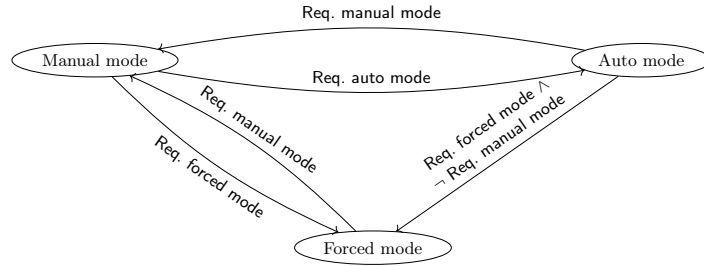


Fig. 16. Example of an incomplete state machine.

**Incomplete diagram.** Let us take the state machine from Figure 16. It represents how a system can change its operation mode by requesting it. From **manual mode**, it is possible to transition to **auto mode** and to **forced mode**. However, it is not specified what happens when the corresponding requests to transition to **auto mode** and to **forced mode** are both true simultaneously. This is also the case for the transitions between **auto mode** to **manual mode** and **forced mode**.

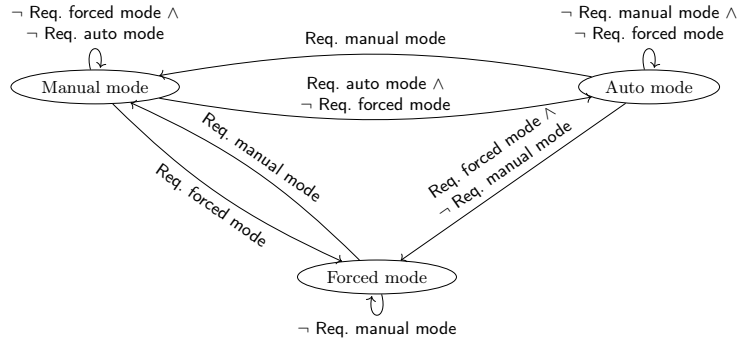


Fig. 17. Example of how the incomplete state machine from Figure 16 can be fixed.

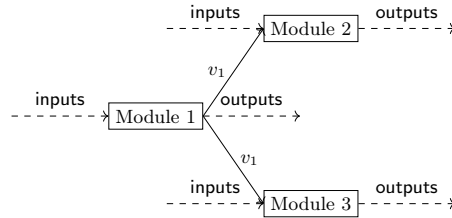
In order to fix this situation, one needs to specify all the necessary conditions for each guard so that only one transition is activated at a time. Figure 17 shows a possible fix to the previous ambiguous state machine. Now, if the system is in **manual mode** and there is a simultaneous request to transition both to **auto mode** and to **forced mode**, the system will transition to **forced mode**.

**Lack of explanation.** In some cases, we experienced situations where there was a lack of explanation about certain requirements.

- **Global/Shared/Input-Output variables.** When a project comprises different modules, some variables flow from one module to another. They are part of the output variables in one module and of the inputs in other modules. Since the requirements are usually formalized per module, these variables can be treated as inputs in some modules. However, they are not free inputs for

those modules in the sense that they can only take a limited set of possible values given by the output of the other module. This fact is usually not stated, leading to the violation of properties with values for those variables that are not possible. A possible way to formally verify these situations is by using assume-guarantees (possible with PLCverif) or contracts.

Figure 18 shows an example in which the variable  $v_1$  is an output of module 1 and an input of modules 2 and 3. In this case,  $v_1$  cannot take any value and is limited to the possible values produced by module 1. Therefore, if a requirement for module 2 or 3 includes this variable, it might be violated with a value for  $v_1$  that can never happen. Requirement engineers might have already in mind that the value for that variable is limited to a specific range given by module 1 but might not have specified it when writing the requirements for modules 2 and 3.



**Fig. 18.** Example of modules in which a variable  $v_1$  is the output from one of them and the input for the other two.

As an example, let us take the code for module 1 and module 2 from Listing 1.4 and the following requirement:

( $R_3$ ) Always, at the end of the execution of module 2,  $v\_2=TRUE$ .

---

```

1 FUNCTION_BLOCK module_1
2   VAR_OUTPUT
3     v_1 : BOOL;
4   END_VAR
5 BEGIN
6   v_1 := FALSE;
7 END_FUNCTION_BLOCK
8
9 FUNCTION_BLOCK module_2
10  VAR_INPUT
11    v_1 : BOOL;
12  END_VAR
13  VAR_OUTPUT
14    v_2 : BOOL;
15  END_VAR
16 BEGIN
17    v_2 := NOT v_1;
18 END_FUNCTION_BLOCK

```

---

**Listing 1.4.** Module 1.

If we only verify module 2 for every possible value of  $v_1$ , we would get the counterexample  $\{v_1=TRUE, v_2=FALSE\}$ . However,  $v_1$  only takes the value **FALSE** at the end of module 1, which is the input of module 2. Therefore, this counterexample is not real.

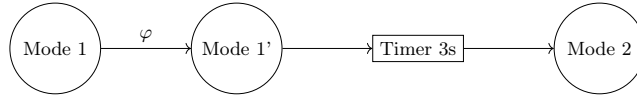
An option to specify this requirement is shown below. We have the assumption from  $R_4$  and the conditional requirement  $R'_3$  based on this assumption. Now, no counterexamples would be found.

- ( $R_4$ ) Always, at the end of the execution of module 1,  $v_1=FALSE$ .
- ( $R'_3$ ) Given that  $v_1=FALSE$  at the beginning of module 2, always, at the end of the execution of module 2,  $v_2=TRUE$ .

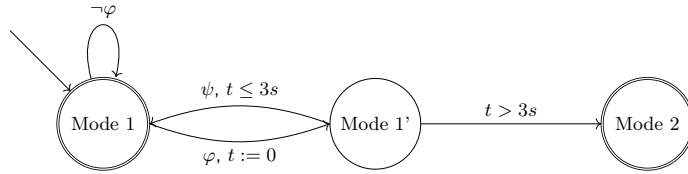
Nevertheless, it is important to note that modules are recommended to be robust to any possible input values. It can happen that, due to, e.g., hardware failures, variables take other values that were not supposed to take.

- **Timers.** When time is involved in the system, formalizing requirements becomes harder and more error-prone. In this case, every step needs to be formalized, such as when timers are activated, what happens before reaching the total time, what happens afterward, how it is reset, etc.

Figure 19 shows a state machine in which it is possible to transition from **mode 1** to **mode 2** if  $\varphi$  is true after a certain amount of time. However, it is not specified if and how the timer is reset and what happens if  $\varphi$  is not true. On the other hand, in Figure 20, we have created a timed automaton specifying how the timer (clock) works. It can also be reset if  $\psi$  is true.



**Fig. 19.** Example of diagram with ambiguous timer.



**Fig. 20.** Example of timed automaton representing the use of a timer.