NASA/TM–20230003752

Automated Verification of Programmable Logic Controller Programs against Structured Natural Language Requirements

Zsófia Ádám
Budapest University of Technology and Economics, Budapest, Hungary

Ignacio D. Lopez-Miguel
TU Wien, Vienna, Austria

Anastasia Mavridou
KBR, NASA Ames Research Center, Moffett Field, USA

Thomas Pressburger
NASA Ames Research Center, Moffett Field, USA

Martin Bes
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Enrique Blanco Vinuela
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Andreas Katis
KBR, NASA Ames Research Center, Moffett Field, USA

Jean-Charles Tournier
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Khanh V. Trihn
KBR, NASA Ames Research Center, Moffett Field, USA

Borja Fernandez Adiego
European Organization for Nuclear Research (CERN), Geneva, Switzerland

March 2023

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Help Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199

NASA/TM–20230003752

Automated Verification of Programmable Logic Controller Programs against Structured Natural Language Requirements

Zsófia Ádám
Budapest University of Technology and Economics, Budapest, Hungary

Ignacio D. Lopez-Miguel
TU Wien, Vienna, Austria

Anastasia Mavridou
KBR, NASA Ames Research Center, Moffett Field, USA

Thomas Pressburger
NASA Ames Research Center, Moffett Field, USA

Martin Bes
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Enrique Blanco Vinuela
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Andreas Katis
KBR, NASA Ames Research Center, Moffett Field, USA

Jean-Charles Tournier
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Khanh V. Trihn
KBR, NASA Ames Research Center, Moffett Field, USA

Borja Fernandez Adiego
European Organization for Nuclear Research (CERN), Geneva, Switzerland

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035

March 2023

This report is available in electronic form at
https://ntrs.nasa.gov/

Executive Summary

PLCverif is an actively developed project at CERN, enabling the formal verification of Programmable Logic Controller (PLC) programs in critical systems. In this paper, we present our work on improving the formal requirements specification experience in PLCverif through the use of natural language. To this end, we integrate NASA's FRET, a formal requirement elicitation and authoring tool, into PLCverif. FRET is used to specify formal requirements in structured natural language, which automatically translates into temporal logic formulae. FRET's output is then directly used by PLCverif for verification purposes. We discuss practical challenges that PLCverif users face when authoring requirements and the FRET features that help alleviate these problems. We present the new requirement formalization workflow and report our experience using it on two critical CERN case studies.

Contents

List of Figures

Chapter 1

Introduction

Over the past few years, formal verification has become a crucial part in the process of software development for critical applications. To this end, CERN's open-source tool PLCverif [4] opened the door for the verification of Programmable Logic Controller (PLC) programs [2, 5, 6] and it has successfully been used to verify several safety-critical applications [7–9].

Given PLC code and a set of requirements formalized in either computation tree logic (CTL) or future-time linear temporal logic (LTL), PLCverif automatically transforms these to an intermediate mathematical model, i.e., control flow automata (CFA) [10]. Once the intermediate model is obtained, PLCverif supports its translation into the input language of various model checkers for verification. Finally, analysis results are presented to the user in a convenient and easy-to-understand format.

The aforementioned process relies on control and safety engineers formalizing requirements. Prior to this work, PLCverif already supported the use of natural language templates, which are a set of pre-made templates with "blanks" where expressions containing variables of the PLC program can be added (e.g., "{1} is always true at the end of the PLC cycle"). During formalization, pattern instantiations are translated to LTL or CTL based on the pre-made templates. However, the expressive power of the existing templates is limited.

In this technical report, we extend our work presented at NFM'23 [11]. We present the integration of NASA Ames' Formal Requirements Elicitation Tool (FRET [12–14]) into PLCverif, which provides a structured natural requirements language with an underlying temporal logic semantics. The integration of FRET within PLCverif helps users express and formalize a greater range of requirements and understand their semantics. The toolchain was successfully used in two critical CERN case studies: a safety program of a superconducting magnet test facility and a module of a process control library.

Zsófia Ádám and Ignacio D. Lopez-Miguel – Work performed while at CERN.

Chapter 2

Background

2.1 Temporal logic

Temporal logic is the most commonly used way to formalize a requirement for model checking. It is a form of logic specifically tailored for statements and reasoning which involves the notion of order in time [15] and it was first introduced by Amir Pnueli [16].

Linear temporal logic (LTL) is a modal logic in which modalities refer to time [16]. It allows the encoding of properties over individual paths. Usually, LTL is an abbreviation for future-time LTL, that is, it will only consider future states. Its syntax allows the use of propositional variables together with the temporal operators `X` (next) and `U` (until). Operators such as `G` (globally/always), and `F` (finally/eventually) can be derived from the former. A future-time LTL formula is valid for a program if all its execution traces starting from an initial state satisfy it.

Analogously, past-time LTL is defined with the temporal operators `Y` (previous) and `S` (since), with which other operators can be derived, such as `H` (historically) and `O` (once). A past-time LTL formula is valid for a program if all its execution traces leading to the current state satisfy it.

Future-time metric (linear) temporal logic (MTL) [17] is an extension of LTL, in which modalities are augmented with time-bounded constraints. Analogously, past-time metric LTL includes time-bounded constraints and it only refers to previous states.

Computation tree logic (CTL), in contrast to LTL, allows the encoding of properties over all possible paths starting from a state. In order to do so, it uses the universal quantifier, `A` (for all paths), and the existential quantifier, `E` (exists one path), over these paths to precede the temporal operators from future-time LTL.

FRET supports future- and past-time LTL, and their metric variants. In contrast, PLCverif supports future-time LTL and CTL.

Although many relevant properties of reactive systems can be specified in both LTL and CTL [18], the logics CTL and LTL are incomparable in their expressiveness. More precisely, there are properties that can be expressed in CTL, but cannot be expressed in LTL, and vice versa [19]. As an example, let us consider the (reachability) CTL formula `AG EF a` for which no equivalent LTL formula exists. Similarly, let us consider the (stability) LTL formula `FG a`, for which no equivalent CTL formula exists.

2.2 PLCverif

PLCverif is a customizable and extensible, plugin-based framework developed at CERN to support formal verification of PLC programs. It translates automatically PLC code and a given set of requirements to the input format of various model checkers, which are then executed. Their output is presented to the user in a convenient, easy-to-understand format. This workflow is shown in Figure 2.1, extracted from [1–3]. It consists of the following parts:

- **PLC code**. This is the code the user wants to verify. It can be written in Structured Text (ST) or Instruction List (IL).
- **Specification**. It is the property the user wants the code to satisfy. This can be given as an assertion directly writing it into the code, or in a non-intrusive way by selecting a requirement pattern from the GUI of PLCverif (Figure 5.3).
  - **Assertion**. If the user wants to verify that a given property is satisfied at a certain point of the code, an assertion can be used. This is written directly into the PLC code by using the keyword for assertions `//#ASSERT`.
  - **Requirement pattern**. There is a set of requirements that are available to the user directly in the GUI. Some of these will reflect the behavior of the system over time. The patterns that can be selected are listed below, where `PLC_END` represents the end of the PLC cycle, `PLC_START` the beginning of the PLC cycle, and `{i}` a blank that has to be filled by the user including expressions containing program variables. Their formulations are written in temporal logic (CTL or LTL).
    * *Implication*. `AG((PLC_END AND ({1})) -> ({2}))`. If {1} is true at the end of the PLC cycle, then {2} should always be true at the end of the same cycle.
    * *Invariant*. `AG(PLC_END -> ({1}))`. {1} is always true at the end of the PLC cycle.
    * *Forbidden state*. `AG(PLC_END -> NOT ({1}))`. {1} is impossible at the end of the PLC cycle.
    * *State change during a cycle*. `AG((PLC_START AND ({1})) -> A[ NOT PLC_END U PLC_END AND ({2}) ])` If {1} is true at the beginning of the PLC cycle, then {2} is always true at the end of the same cycle.
    * *State change between cycles*. `G((PLC_END AND ({1}) AND X( [ NOT PLC_END U (PLC_END AND ({2})) ] )) -> X( [ NOT PLC_END U (PLC_END AND ({3})) ] ))`. If {1} is true at the end of cycle N and {2} is true at the end of cycle N+1, then {3} is always true at the end of cycle N+1.
    * *Reachability*. `EF(PLC_END AND ({1}))`. It is possible to have {1} at the end of a cycle.
    * *Repeatability*. `AG(EF(PLC_END AND ({1})))`. Any time it is possible to have eventually {1} at the end of a cycle.
    * *Leads to*. `NOT (E[(PLC_END -> NOT ({2})) U (PLC_END AND {1})])`. If {1} is true at the end of a cycle, {2} was true at the end of an earlier cycle.
- **Intermediate model**. This is the strategy to model the given code and it is based on a network of automata representing the control flow graph of the PLC program.

Each of these automata is a Control Flow Automaton (CFA) [10]. Figure 2.2 shows a PLC cycle represented as a CFA, which calls its main function in every cycle (Figure 2.3). This main function expresses as a CFA the code shown in listing 2.1.

Input variables are modeled as non-deterministic since the external system to which the PLC is connected is not modeled, and, hence, these inputs could have any possible value.

- **Formalized requirement**. The property needs to be translated into an LTL or CTL formula if it is needed by the model checker.

- **Reductions**. Once the intermediate model is built and the requirements are formalized, it is possible to search for possible reductions of the intermediate model in order to improve the performance. As an example, please refer to in [20] to see the cone of influence (COI) reduction.

- **Backends**. When all the reductions are done, PLCverif will translate the resulting model into the specific language supported by the model checker, which is selected by the user. Then, the model checker will be executed and will produce an output, which can be a proof that the property holds or a counterexample indicating that there was a violation of the property. The three symbolic model checkers that are used are NuSMV/nuXmv [21], CBMC [22], and Theta [23].

  These general-purpose model checkers are considered to be state-of-the-art due to their performance. For these reasons and for the possibility to translate the intermediate model into their languages, they were selected as the backends for PLCverif [4].

- **Verification report**. The output produced by the model checker is processed and shown to the user in a human-readable reporting the verification result. If there is a violation of the property, it will include a counterexample demonstrating it.

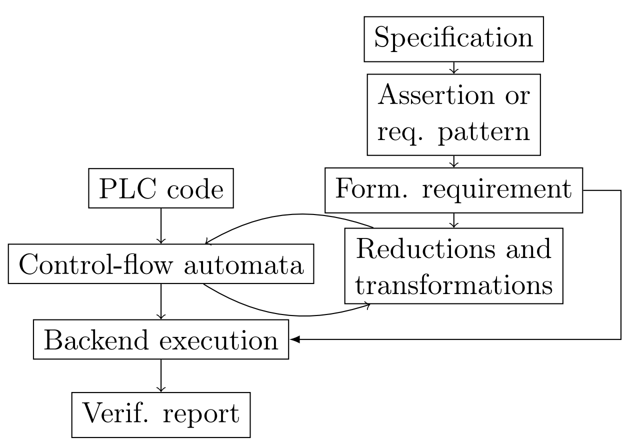


Figure 2.1: Verification process workflow using PLCverif. Given a specification and PLC code, after various intermediate steps, it generates a report specifying if the specification is satisfied or violated. More information can be found in [1–3].

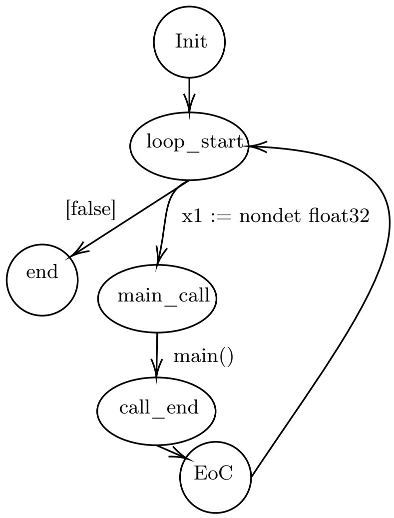


Figure 2.2: CFA–Main PLC cycle.

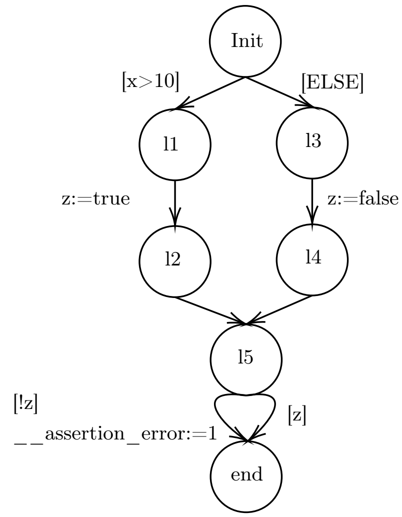


Figure 2.3: CFA–Example.

```
FUNCTION_BLOCK main
	VAR_INPUT
		x : REAL;
	END_VAR
	VAR_OUTPUT
		z : BOOL;
	END_VAR

BEGIN
	IF x>10 THEN
		z := TRUE;
	ELSE
		z := FALSE;
	END_IF;

//#ASSERT z : zTrueAssertion;
END_FUNCTION_BLOCK
```

Listing 2.1: Example of a PLC program in ST.

2.3 FRET

The NASA Formal Requirements Elicitation Tool (FRET) is an open-source framework for creating and understanding formal requirements [12,24]. The core of FRET is defined by its input language, namely FRETish, a structured natural language with strict formal semantics [25]. The core engine is surrounded by features that enable users to efficiently create, manage, understand, and analyze FRETish requirements [14,26]. Example features are the pre-defined FRETish templates, the Glossary component, the requirements simulator LTLSIM and the FRET Analysis Portal. Throughout the years, FRET has been experimentally used to formalize and analyze requirements in several NASA projects [27–31].

A FRETish requirement is described using up to six distinct fields (the * symbol designates mandatory fields): 1) `scope` specifies the time intervals where the requirement is enforced, 2) `condition` is a Boolean expression that triggers the `response` to occur at the time the expression's value becomes true, or is true at the beginning of the scope interval,

3) `component*` is the system component that the requirement is levied upon, 4) `shall*` is used to express that the component's behavior must conform to the requirement, 5) `timing` specifies when the response shall happen, subject to the constraints defined in `scope` and `condition` and 6) `response*` is the Boolean expression that the component's behavior must satisfy.

FRETish provides 8 scopes: *global*, *in*, *before*, *after*, *notin*, *only in*, *only before*, and *only after*. The scope *global* means *always*; the others are with respect to when the system is in a mode or satisfies a Boolean expression. For example, *In mode M* means the requirement is enforced when the system is in mode M, as determined by the Boolean variable M. Also allowed for scope in place of a single Boolean variable is a Boolean expression, except for *in* which in the expression case is written with *while*; e.g., *While vehicle_mode = hover*. In FRETish, the optional condition field is introduced by the words *upon*, *when*, or *if*, which are synonymous in FRETish, or the word *unless*, which is the same as *when !*. FRETish provides 10 timings: *immediately*, *at the next timepoint*, *always*, *eventually*, *never*, *for* N time steps, *within* N time steps, *after* N time steps, *until* *bool_ expr*, and *before* *bool_ expr*. When the scope is omitted it is taken as *global*; when the condition is omitted, it is taken as `true`; when the timing is omitted, it is taken as *eventually*. If we consider the condition being omitted as a separate case, there are $8 \times 2 \times 10 = 160$ possible combinations of $\langle$*scope*, *condition*, *timing*$\rangle$, each formalized as a distinct LTL formula template. The templates are generated by an algorithm that has been formally proven to generate formalizations with the intended semantics [32].

Boolean expressions can use the standard logical connectives (!, &, |) and can involve arithmetic relations (=,!=,<,<=,>,>=) and operators ($+,-,*,/$) over integer and real variables. There are two predefined predicates *preInt* and *preReal* that refer to previous values: the expression $preInt(init, n)$, for integer expression n, returns the value of n at the previous timepoint; if at the beginning of the trace where there is no previous value, then the value of *init* is returned. Currently, FRETish does not allow arbitrary nesting of temporal operators, e.g. "*In mode m, before q the system shall ...*". Timed operators with intermediate bounds are also not currently expressible; e.g., the equivalent of `H[i,j] p`, where `i` $\neq$ `0`.

For each FRETish requirement, FRET generates formulae in a variety of formalisms, including pure future-time LTL and pure past-time LTL, as well as their metric (MTL) variants.

Chapter 3

Integrating FRET into PLCverif

FRET's user interface was designed with usability in mind; engineers with varying levels of experience in formal methods can express requirements using FRETish, a restricted natural language plus standard Boolean/arithmetic expressions with precise, unambiguous meaning. For a FRETish requirement, FRET produces textual and diagrammatic explanations of its exact meaning and temporal logic formalizations in Metric LTL. FRET also supports interactive simulation of the generated logical formulae to increase confidence that they capture the intended semantics.

3.1 Limitations of Requirement Formulation in PLCverif

As already anticipated in the introduction, PLCverif *patterns* have significant limitations:

1. they offer a *limited set of 8 pre-made templates* only,
2. they do not offer any tool for *validation* of complex requirements; i.e., methods for checking if the created requirement is the same as the intended one.

FRET is able to improve on the current limitations the following ways:

Expressive Power. Users are able to formulate requirements in FRETish as constrained and unambiguous sentences, which are then automatically transformed to LTL expressions.

Validation. FRET has a built-in simulator, allowing the user to check different temporal variable valuations. Furthermore, FRET generates a textual and diagrammatic description of the requirement to further help precise understanding.

3.2 Realization of the New Workflow with FRET

The bold parts of Figure 3.1 show how FRET fits into the workflow of PLCverif. If "Edit in FRET" is chosen, FRET is opened and the user can work in the requirement editor. Once the requirement is written in FRETish and formalized, it is then sent back to PLCverif.

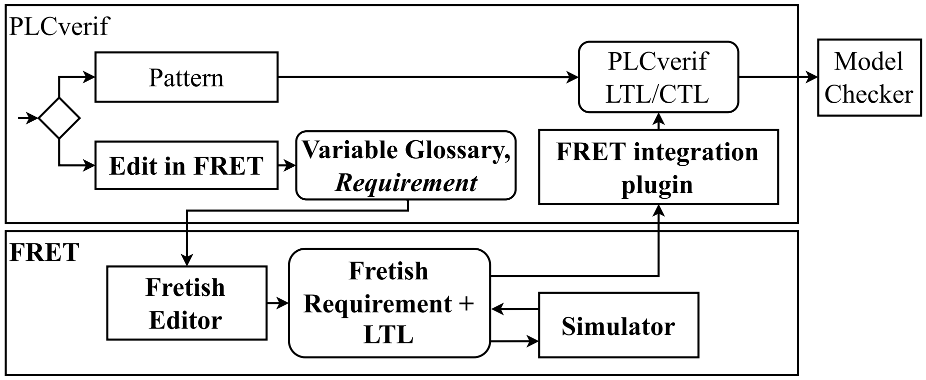


Figure 3.1: Requirement formalization workflow in PLCverif. Features integrated into PLCverif in this work are shown in bold.

The External Mode of FRET. The feature set of FRET covers much more than what PLCverif could currently utilize (e.g., requirement hierarchies). PLCverif handles each requirement in a separate verification case, thus only the following features are utilized: the requirement editor, the formalization component and the requirements simulator. To facilitate integration with external tools including PLCverif, we developed a Node.js script for running the aforementioned features as a standalone tool. This new mode also implements the ability to import a variable glossary into the requirement editor of FRET.

Variable Glossary. PLCverif extracts the list of variable names and types by parsing the PLC code. Now the resulting *variable glossary* can also be exported to a JSON file for FRET to use. This enables features such as *autocompletion* of variable names in the FRET requirement editor. It also facilitates the process of creating formalized properties that can be directly used by PLCverif for verification as these requirement variables directly match PLC code variables.

Since variable names used in PLC code may include other characters besides alphanumeric, we extended the FRET editor to support identifiers with periods, percents, or double-quoted identifiers that can contain any special character.

The supported data types differ between the two tools (e.g., PLC programs might use arrays, while FRET only has scalar types). Assigning a data type to a variable is not mandatory in FRET and in this work it is only utilized by the simulator to show what possible values a variable can be assigned. Thus implementing a best-effort mapping proved to be adequate (e.g., sending each array element as a separate variable).

Preparing FRET for PLCs. The working principle of a PLC is described by the so-called *PLC Scan Cycle*, consisting of three main steps: (1) read sensor values, (2) execute the PLC program and (3) write the actuator values. PLC requirements must be able to express cycles, e.g. "next cycle" or "at the end of the cycle".

To enable this, FRET's formalization algorithm was extended to express new built-in predicates `nextOcc(p,q)` and `prevOcc(p,q)`, and FRETish was extended with the phrases *"at the next/previous occurrence of `p`, `q`"* meaning: at the next (previous) time point where `p` holds (if there is such), `q` also holds. These are expanded into the following LTL formulae, where L (R) is the formula that specifies the left (right) endpoint of the scope interval:

- Future for nextOcc

```
($R$ | (X (((!p & !$R$) U p) => ((!p & !$R$) U (p & q)))))
```

- Past for prevOcc

```
($L$ | (Y (((!$L$ & !p) S p) => ((!$L$ & !p) S (p & q)))))
```

Based on our experience, many PLC program requirements are checked at the end of PLC cycles. This is the most critical moment since the calculated values are sent to the actuators of the system. In the FRET PLC requirements, a variable called `PLC_END` can be used to express this exact moment.

In addition, FRET allows the creation of templates to help users write common requirements. For this work, we have created new FRET templates for PLCverif (e.g., *In PLC_END* *[COMPONENT]* shall *always* *satisfy* *[RESPONSE]*).

Chapter 4

Case studies

The integration has been used in two CERN critical systems. We introduce these projects and some selected properties here. The first case study utilizes the validation capabilities, while the second employs the improved expressiveness.

4.1 Safety PLC program

The safety PLC program of the *SM18 Cluster F*, a superconducting magnet test facility at CERN, is meant to protect the personnel and the equipment of this installation.

This PLC program, written in Ladder Diagram (LD) [33], contains 31 Function_Blocks, the corresponding 31 instance Data_Blocks and 2 Functions. Regarding the number of variables, it contains 13 INT variables and 157 BOOL variables. The exported code from SIMATIC Step7 [1] (former version of Siemens TIA Portal) contains around 2800 lines of code. Figure 4.1a shows a small part of this safety program and figure 4.1b the exported code that is imported in PLCverif.

The *SM18 cluster F* is dedicated to test the new superconducting magnet technology for the High Luminosity Large Hadron Collider (HL-LHC) [34], an upgrade of the existing LHC particle accelerator. Its main risks are related to the cryogenic system and the powerful power converters up to 20.000 Amps.

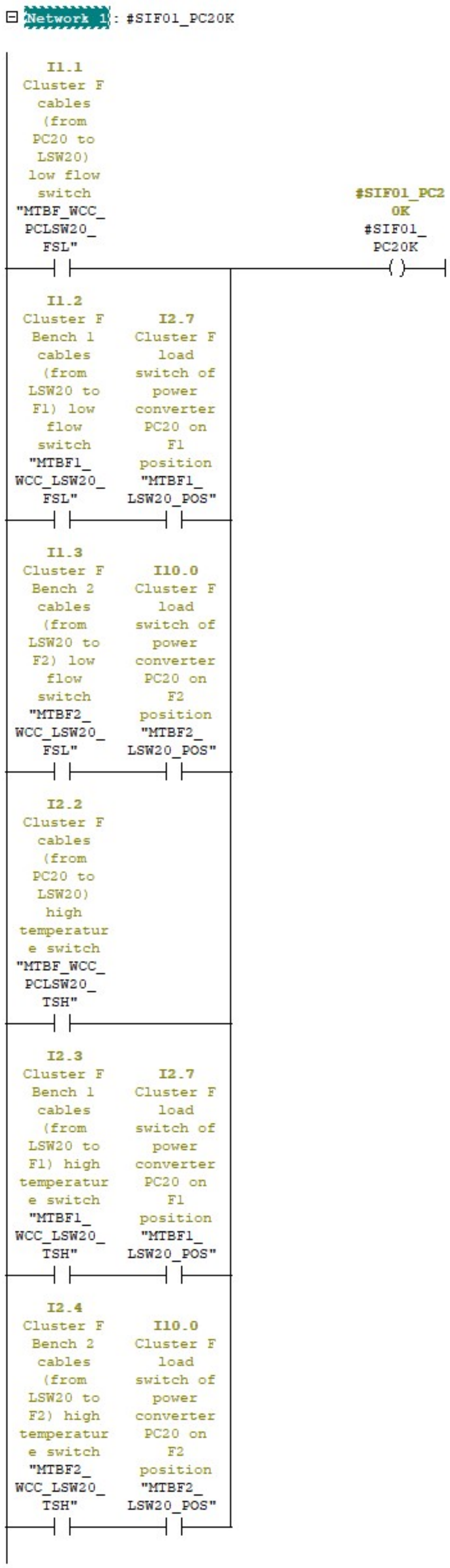
Error Property. The property of this case study corresponds to the expected logic of one of the safety functions (*SIF01_PC20K*):

if at the end of the PLC cycle (PLC_END) the flow (*_FSL) or thermo (*_TSH) switches monitoring the cooling system detect a low flow or a high temperature, and the power converter (PC20k) is connected to the magnet (*_LSW20_POS), then the safety function should shut down the power converter (SIF01_PC20K).

Figure 4.2 shows how the property is expressed in FRETish and in LTL.

Validation. Before verifying this property with PLCverif, the user should make sure that the formalized property behaves as expected. The main challenge is the number of operators and parentheses, making manual validation difficult. FRET's simulator aids by allowing the user to check any temporal valuation and whether it satisfies the property or not as shown in Figure 4.2c.

[1] https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/software/step7-tia-portal.html



(a) Part of the LD safety program.

```
FUNCTION_BLOCK "SIF01"
TITLE=
VERSION : 0.1


VAR_OUTPUT
  SIF01_PC20K : BOOL ;
  SIF01_PC15K : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =#SIF01_PC20K

      ON    "MTBF_WCC_PCLSW20_FSL";
      O     ;
      AN    "MTBF1_WCC_LSW20_FSL";
      A     "MTBF1_LSW20_POS";
      O     ;
      AN    "MTBF2_WCC_LSW20_FSL";
      A     "MTBF2_LSW20_POS";
      ON    "MTBF_WCC_PCLSW20_TSH";
      O     ;
      AN    "MTBF1_WCC_LSW20_TSH";
      A     "MTBF1_LSW20_POS";
      O     ;
      AN    "MTBF2_WCC_LSW20_TSH";
      A     "MTBF2_LSW20_POS";
      =     #SIF01_PC20K;
NETWORK
TITLE =#SIF01_PC15K

      ON    "MTBF_WCC_PCINV15_FSL";
      O     ;
      AN    "MTBF2_WCC_LSW15_FSL";
      A     "MTBF_PC15F_RELAY_ST";
      ON    "MTBF_WCC_PCINV15_TSH";
      O     ;
      AN    "MTBF2_WCC_LSW15_TSH";
      A     "MTBF_PC15F_RELAY_ST";
      O     ;
      A     "MTBF_LSW15_POS_E2";
      A     "MTBF_PC15F_RELAY_ST";
      =     #SIF01_PC15K;
END_FUNCTION_BLOCK
```

(b) Exported safety program in STL language.

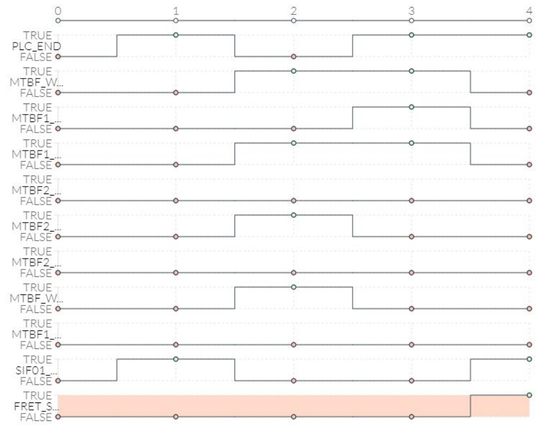
Figure 4.1: Safety program from the case study 1.

```
F_MAIN shall always satisfy if (PLC_END & (!MTBF_WCC_PCLSW20_FSL | !MTBF_WCC_PCLSW20_TSH |
     (MTBF1_LSW20_POS & (!MTBF1_WCC_LSW20_FSL | !MTBF1_WCC_LSW20_TSH)) | (MTBF2_LSW20_POS &
     (!MTBF2_WCC_LSW20_FSL | !MTBF2_WCC_LSW20_TSH)))) then SIF01_DB.SIF01_PC20K
```

(a) Property of SIF01_PC20K in FRETISH.

```
G ((PLC_END and ((((!
  MTBF_WCC_PCLSW20_FSL) or
  (! MTBF_WCC_PCLSW20_TSH))
  or (MTBF1_LSW20_POS and
  ((! MTBF1_WCC_LSW20_FSL)
  or (! MTBF1_WCC_LSW20_TSH)
  ))) or (MTBF2_LSW20_POS and
  ((!MTBF2_WCC_LSW20_FSL) or (!
  MTBF2_WCC_LSW20_TSH)))))
  -> SIF01_DB.SIF01_PC20K)
```

(b) Generated LTL property by FRET.



(c) Future LTL simulation

Figure 4.2: Property of the SIF01_PC20K safety function.

4.2 Standard PLC program

This case study is concerned with UNICOS [35], a CERN framework for the development of hundreds of industrial control systems. The selected program library is called the *OnOff* object. Its purpose is to control physical equipment driven by digital signals, which can be composed of different types of devices. This makes its PLC program highly configurable and its associated logic complex.

This PLC program is written in SCL (Structured Control Language), the Siemens PLC language equivalent to ST (Structured Text) from the IEC 61131 standard [33]. This program contains 1 main Function_Block, its corresponding instance Data_Block, 4 Functions, and 1 global Data_Block. It also contains 1 TON (Timer On Delay) and 2 TP (Pulse Timer) instances. Regarding the number of variables, it contains 9 WORD variables, 11 INT variables, 36 REAL variables, 8 TIME variables, 1 DINT variable, 6 BYTE variables, and 141 BOOL variables. The program has been written using the Siemens TIA Portal programming environment [2] and it contains around 1080 lines of code. Fig 4.3 shows a small part of this standard program.

[2]https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html

```
(* MODE MANAGER *)

IF NOT (#HLD AND #PHLD) THEN

(* Forced Mode *)
    IF (#AuMoSt_aux OR #MMoSt_aux OR #SoftLDSt_aux) AND
        #E_MFoMoR AND NOT(#AuIhFoMo) THEN

        #AuMoSt_aux   := FALSE;
        #MMoSt_aux    := FALSE;
        #FoMoSt_aux   := TRUE;
        #SoftLDSt_aux := FALSE;

(* Software Local Mode *)
    ELSIF (#AuMoSt_aux OR #MMoSt_aux) AND #E_MSoftLDR AND NOT #AuIhFoMo THEN

        #AuMoSt_aux  := FALSE;
        #MMoSt_aux   := FALSE;
        #FoMoSt_aux  := FALSE;
        #SoftLDSt_aux:= TRUE;

(* Manual Mode *)
    ELSIF (#AuMoSt_aux OR #FoMoSt_aux OR #SoftLDSt_aux) AND
           #E_MMMoR AND NOT(#AuIhMMo) THEN

        #AuMoSt_aux   := FALSE;
        #MMoSt_aux    := TRUE;
        #FoMoSt_aux   := FALSE;
        #SoftLDSt_aux := FALSE;

(* Auto Mode *)
    ELSIF (#MMoSt_aux AND (#E_MAuMoR OR #E_AuAuMoR )) OR
          (#FoMoSt_aux AND #E_MAuMoR) OR
          (#SoftLDSt_aux AND #E_MAuMoR) OR
          (#MMoSt_aux AND #AuIhMMo) OR
          (#FoMoSt_aux AND #AuIhFoMo)OR
          (#SoftLDSt_aux AND #AuIhFoMo) OR
           NOT(#AuMoSt_aux OR #MMoSt_aux OR #FoMoSt_aux OR #SoftLDSt_aux) THEN

        #AuMoSt_aux   := TRUE;
        #MMoSt_aux    := FALSE;
        #FoMoSt_aux   := FALSE;
        #SoftLDSt_aux := FALSE;
    END_IF;

    (* Status setting *)
    #LDSt     := FALSE;
    #AuMoSt   := #AuMoSt_aux;
    #MMoSt    := #MMoSt_aux;
    #FoMoSt   := #FoMoSt_aux;
    #SoftLDSt := #SoftLDSt_aux;
 ELSE
    (* Local Drive Mode *)
    #AuMoSt  := FALSE;
    #MMoSt   := FALSE;
    #FoMoSt  := FALSE;
    #LDSt    := TRUE;
        #SoftLDSt:= FALSE;
 END_IF;
```

Figure 4.3: Small piece of the SCL program of case study 2.

Error Property. The property presented here is related to the transitions between two operation modes shown in Figure 4.4: (1) *Auto mode*, where the *OnOff* object is driven by the control logic of a higher object of the hierarchy of the program, and (2) *Manual mode*, where the operator drives the object.

The property extracted from the specification is: When the OnOff object is in Manual mode (MMoSt) and the control logic requests to move to Auto mode (AuAuMoR) at any point in the PLC cycle, the OnOff object should move to the Auto mode (AuMoSt). Figure 4.5 shows the property in FRETISH and the LTL formula.

Thanks to the new *"at the next occurrence of"* phrase, we can verify that this property is satisfied at the end of the current cycle (which is more precise and strict), as shown in

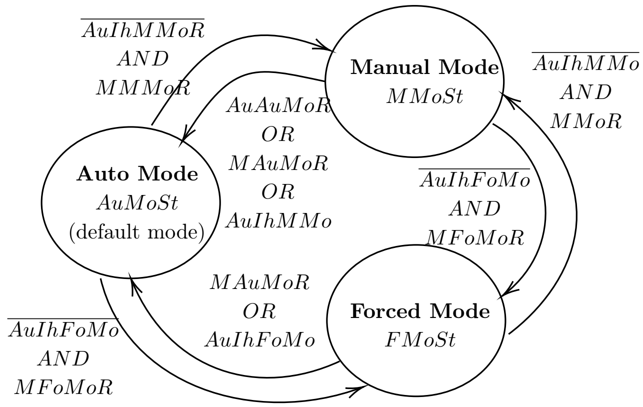


Figure 4.4: OnOff operation modes specification.

```
When (MMoSt & AuAuMoR) the
   CPC_FB_OnOff shall eventually
   satisfy AuMoSt & PLC_END
```

(a) Property expressed in FRETish.

```
(G (((!(MMoSt and AuAuMoR)) and (X (MMoSt
  and AuAuMoR))) -> (X(F(AuMoSt and
  PLC_END))))) and ((MMoSt and
  AuAuMoR) -> (F(AuMoSt and PLC_END)))
```

(b) Generated LTL property by FRET

Figure 4.5: PLCverif property for the *OnOff* object.

```
the CPC_FB_OnOff shall always satisfy
   if (MMoSt & AuAuMoR & !PLC_END)
   then at the next occurrence of
   PLC_END, AuMoSt
```

(a) Property expressed in FRETish.

```
G (((MMoSt and AuAuMoR) and (! PLC_END)) ->
  (X ((((! PLC_END) ) U PLC_END) ->
  (((! PLC_END)) U (PLC_END and AuMoSt)))))
```

(b) Generated LTL property by FRET

Figure 4.6: PLCverif property for the *OnOff* object using the `nextOcc` predicate

Figure 4.6.

These properties can not be expressed with the current PLCverif patterns, but they are expressible with the restricted natural language of FRET. Now PLCverif users can express a large variety of requirements in a natural language and validate these in the simulator.

Chapter 5

A Closer Look at FRET and PLCverif

In this chapter, we show the workflow of our tool chain in practice. Figure 5.1 represents the high-level workflow of the tools: PLCverif lets the user freely edit the PLC code and the configuration. The configuration includes the requirement, which now can be formulated in FRET. The user can open up FRET from PLCverif and edit the requirement there as many times as they want, utilizing all the earlier described functionalities of FRET. When the user is satisfied with the configurations, the verification can be executed with a click of a button.

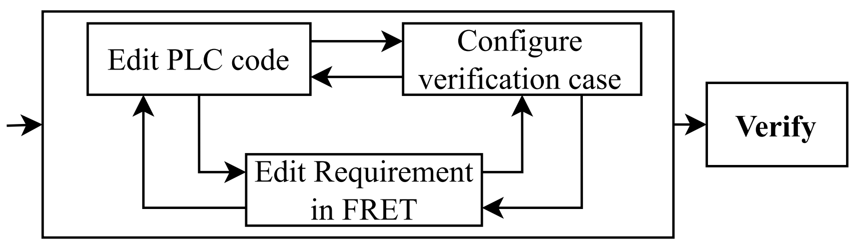


Figure 5.1: PLCverif lets the user freely edit the PLC code and the verification configuration, including the requirement.

5.1 Starting out in PLCverif

The screenshots shown in this section are taken from the OnOff case study and show the functionality of the graphical user interface of PLCverif.

The main window includes the PLC code editor, which is shown in Figure 5.2. In this example, the relevant code of the OnOff project for the first property (see Figure 4.2) of the case study is shown in this editor. The project structure is displayed on the side, in which all the necessary files for this project appear.

The verification cases, such as the one shown in Figure 5.3, also appear in the project structure with the extension *.vc3*. The configuration options of the verification case file include for example the verification backend, metadata and also the requirement.

With the presented integration with FRET the user is now able to select the option *Edit in FRET* while formalizing a new requirement. Once this option is selected, FRET pops up and the window from Figure 5.5 appears. After the user is done editing the requirement, they save it, FRET closes and the requirement written in FRETISH and its corresponding temporal logic (TL) formula appear in PLCverif.

At this point, the user is able to click on *Verify!* in order to launch the verification case. They can also still modify any of the so far mentioned options or open up FRET again if they would like to modify the requirement.

Once PLCverif is done running the verification case, it reports the result of it as shown in Figure 5.4. In this case, since the property is not satisfied, PLCverif returns a counterexample. This means that if the input variables of the OnOff object are set to the values from this counterexample, the property is violated.

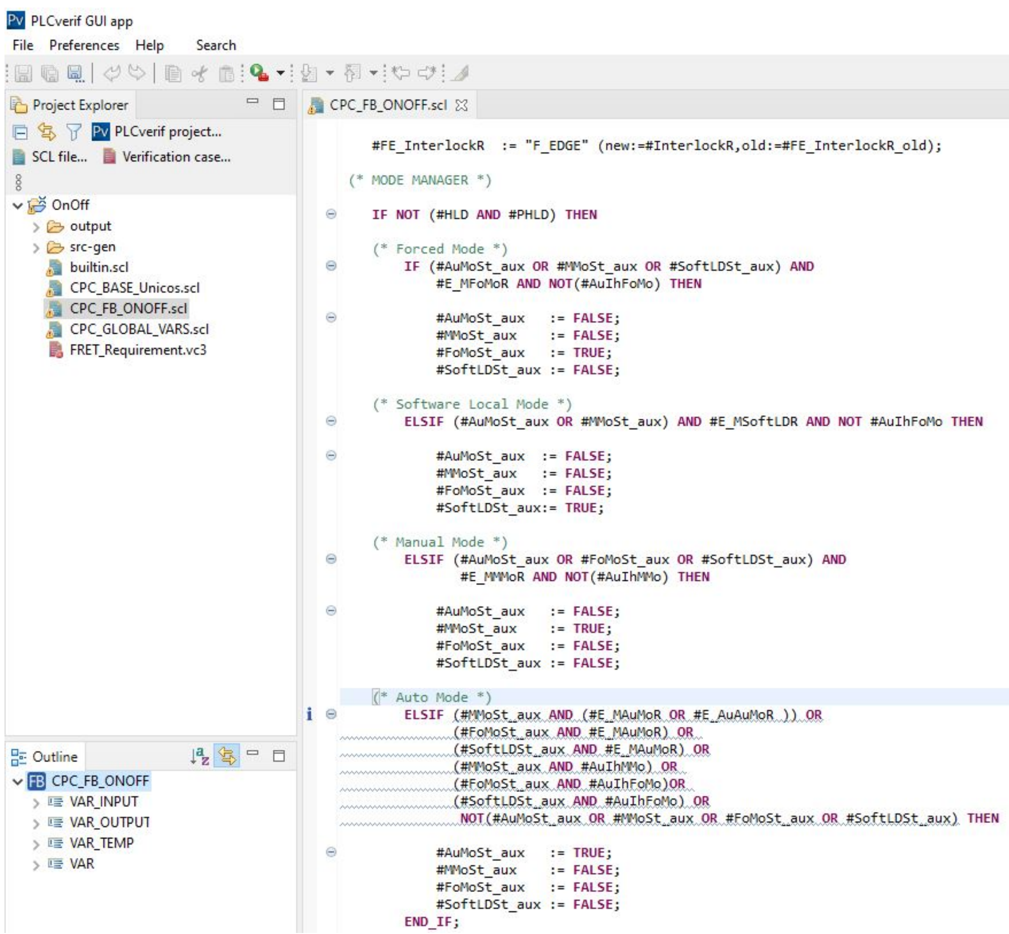


Figure 5.2: PLCverif *GUI*. It shows the relevant PLC code from the OnOff case study. The project structure details the different files that composes this project, as well as the verification cases (*.vc3*).

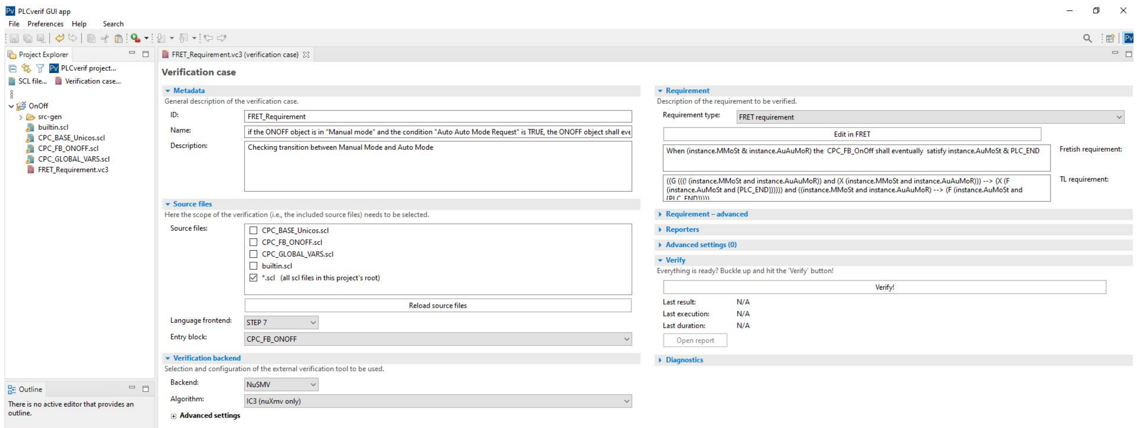


Figure 5.3: PLCverif *verification case* editor. The user can now select *Edit in FRET* in order to formalize the requirement. In this case, FRET is launched and the window from 5.5 appears. Once this new window is closed, the generated specification in FRETish and in temporal logic (TL) appears in PLCverif. The user can then verify the property.

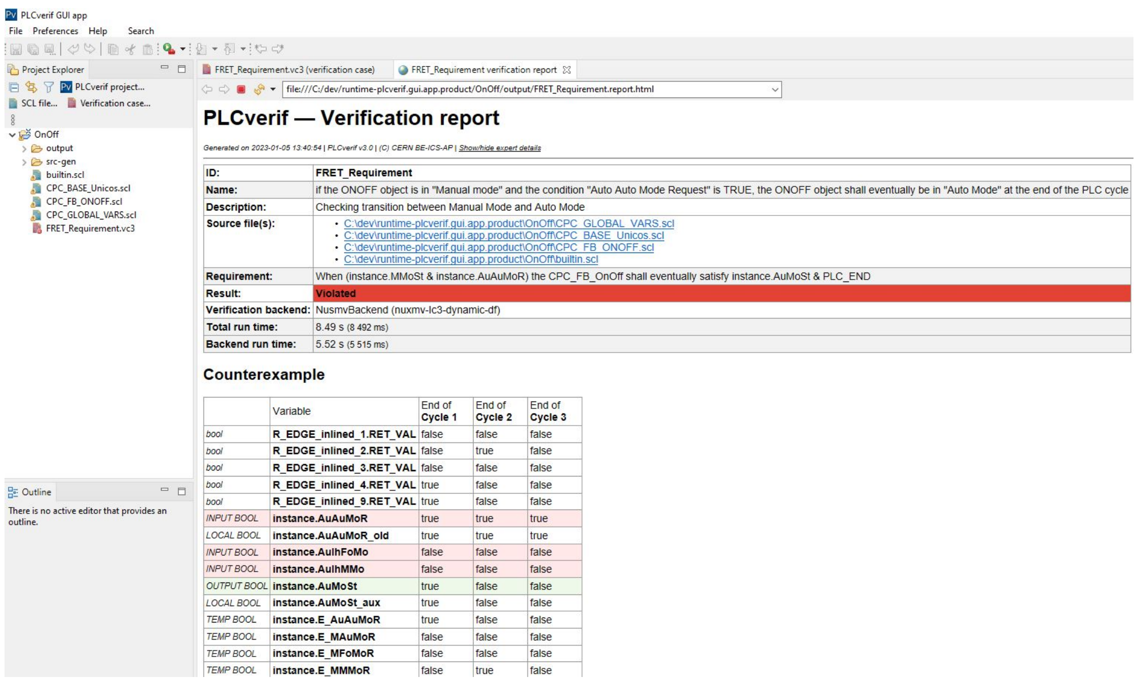


Figure 5.4: PLCverif *verification report*. Since the property is violated, i.e., the PLC code from the OnOff object does not satisfy the property, PLCverif returns a counterexample. If the input variables of the OnOff object are set to the values from this counterexample, the property is violated.

5.2 Editing the Requirement in FRET

Similarly to the last section, the following screenshots represent different steps of creating the requirement with FRET for the first property of the OnOff case study. Formulating, editing and validating a requirement in FRET has its own workflow by itself, which is introduced in this section.

Frst, Figure 5.5 shows the FRET *Update Requirement* dialog in which the user writes in FRETISH the property to be verified. The dialog is split into two parts - the editor on the left and the *Assistant*, *Templates*, *Glossary* tabs on the right.

On the left part, free text can be optionally entered in Rationale and Comments. Next, a FRETISH requirement can be written using the six sequential fields: scope, condition, component, shall, timing, and response. As the requirement is typed in, the FRET editor dynamically colors the text corresponding to the fields. Parser feedback appears in red at the bottom.

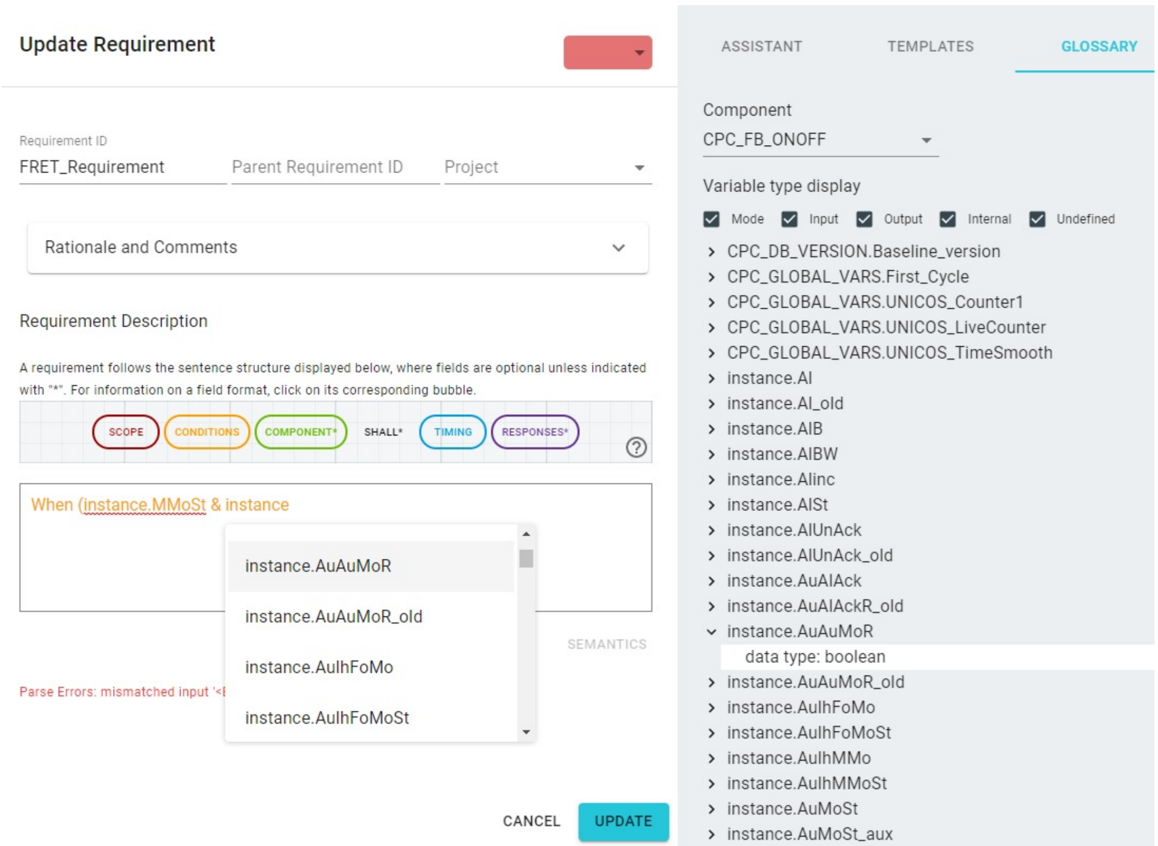


Figure 5.5: FRET *Update Requirement* editor after launching it from PLCverif. All the variables from the PLC program are available in FRET. The user can easily write the requirements by making use of the autocomplete functionality.

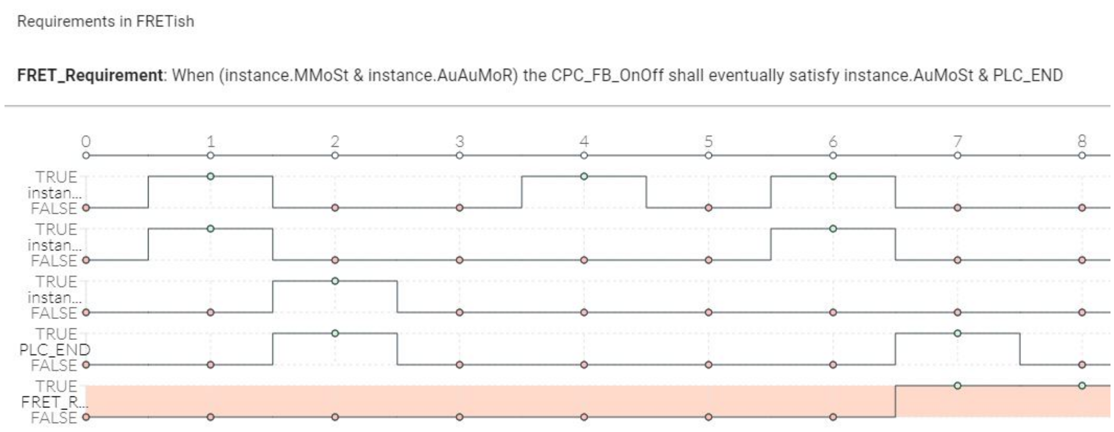


Figure 5.6: FRET simulator. After writing the requirement, the user can test its right behavior by simulating different cases.

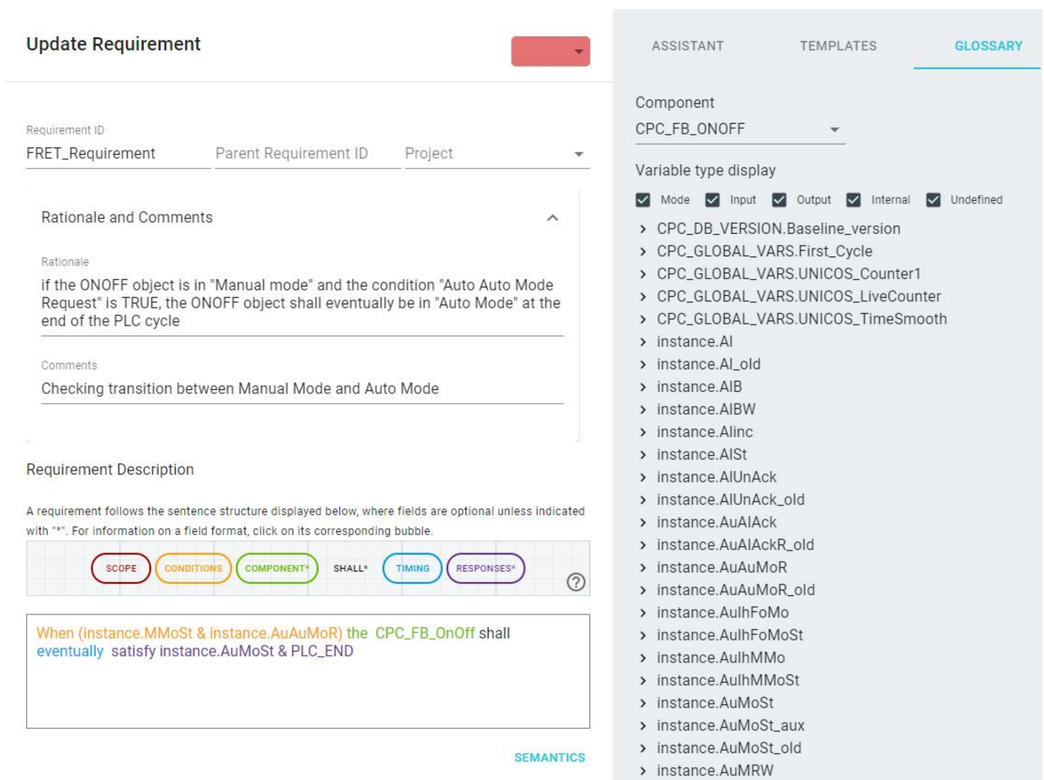


Figure 5.7: FRET *Update Requirement* editor when modifying an existing requirement. The user can modify an existing requirement. Everything that was created is now also available.

On the right side, in the *Glossary* tab, we can see the variable glossary imported from PLCverif. These variables can be autocompleted in the requirement editor (as shown in Figure 5.5). Once the user is done writing the requirement, they can test different cases using the FRET simulator (Figure 5.6). Given a FRETish requirement, the visualizer shows temporal traces of each of the variables involved, as well as the valuation of the requirement for each point in time. The user is able to change the values of signals and observe the consequences on the resulting valuation, i.e., the requirement is satisfied or violated.

After closing FRET, the property is exported to PLCverif. Afterwards, if the user wants to modify an existing FRET requirement, they can open FRET from PLCverif again and this time the *Update Requirement* editor will open as shown in Figure 5.7.

Chapter 6

Conclusion

The creation of requirements in FRET fits well into the verification workflow and improves both usability and expressiveness, as shown by the case studies. PLCverif [4] and FRET [12] are both open source and available in their respective repositories.

Plans for improvements include the support of verification of time properties in PLCverif and the support for different type widths in the FRET simulator (*e.g. 16/32 bit integers*).

To the best of our knowledge, this is the first attempt to specify formal requirements using a structured natural language for PLC program verification.

Bibliography

1. Darvas, D.; Majzik, I.; and Blanco Viñuela, E.: *Formal Verification of Safety PLC Based Control Software*, Springer, vol. 9681 of *Lecture Notes in Computer Science*. 2016, pp. 508–522. URL `http://link.springer.com/chapter/10.1007%2F978-3-319-33693-0_32`.

2. Viñuela, E. B.; Darvas, D.; and Molnár, V.: PLCverif Re-engineered: An Open Platform for the Formal Analysis of PLC Programs. *Proc. ICALEPCS'19*, no. 17 in International Conference on Accelerator and Large Experimental Physics Control Systems, JACoW Publishing, Geneva, Switzerland, 08 2020, pp. 21–27. URL `https://jacow.org/icalepcs2019/papers/mobpp01.pdf`, https://doi.org/10.18429/JACoW-ICALEPCS2019-MOBPP01.

3. Lopez-Miguel, I. D.; Adiego, B. F.; Tournier, J.-C.; Viñuela, E. B.; and Rodriguez-Aguilar, J. A.: Simplification of Numeric Variables for PLC Model Checking. *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 10–20. URL `https://doi.org/10.1145/3487212.3487334`.

4. PLCverif: A tool to verify PLC programs based on model checking techniques. URL `https://gitlab.com/plcverif-oss`.

5. Darvas, D.; Fernández Adiego, B.; and Blanco Viñuela, E.: PLCverif: A tool to verify PLC programs based on model checking techniques. *Proc. ICALEPCS'15*, 10 2015.

6. Lopez-Miguel, I. D.; Tournier, J.-C.; and Fernández Adiego, B.: PLCverif: Status of a Formal Verification Tool for Programmable Logic Controller. *Proc. ICALEPCS'21*, 03 2022.

7. Fernández Adiego, B.; and Blanco Viñuela, E.: Applying model checking to critical PLC applications : An ITER case study. *Proc. ICALEPCS'17*, 10 2017.

8. Fernández Adiego, B.; Darvas, D.; Blanco Viñuela, E.; Tournier, J.-C.; Bliudze, S.; Blech, J.; and González, V.: Applying Model Checking to Industrial-Sized PLC Programs. *IEEE Transactions on Industrial Informatics*, vol. 11, 12 2015, pp. 1400–1410.

9. Fernández Adiego, B.; Lopez-Miguel, I. D.; Tournier, J.-C.; Blanco Viñuela, E.; Ladzinski, T.; and Havart, F.: Applying Model Checking to Highly-Configurable Safety Critical Software: The SPS-PPS PLC Program. *Proc. ICALEPCS'21*, 03 2022.

10. Beyer, D.; Henzinger, T. A.; Jhala, R.; and Majumdar, R.: The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, Sept. 2007, pp. 505–525. URL `https://doi.org/10.1007/s10009-007-0044-z`.

11. Ádám, Z.; Lopez-Miguel, I. D.; Mavridou, A.; Pressburger, T.; Bęś, M.; Blanco Viñuela, E.; Katis, A.; Tournier, J.-C.; Trinh, K. V.; and Fernández Adiego, B.: From Natural Language Requirements to the Verification of Programmable Logic Controllers: Integrating FRET into PLCverif. *NASA Formal Methods*, K. Y. Rozier and S. Chaudhuri, eds., Springer Nature Switzerland, Cham, 2023, pp. 353–360. URL `http://doi.org/10.1007/978-3-031-33170-1_21`.

12. FRET: Formal requirements elicitation tool. URL `https://github.com/NASA-SW-VnV/fret`.

13. Mavridou, A.; Bourbouh, H.; Giannakopoulou, D.; Pressburger, T.; Hejase, M.; Garoche, P.-L.; and Schumann, J.: The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 300–310.

14. Katis, A.; Mavridou, A.; Giannakopoulou, D.; Pressburger, T.; and Schumann, J.: Capture, Analyze, Diagnose: Realizability Checking Of Requirements in FRET. *Computer Aided Verification*, S. Shoham and Y. Vizel, eds., Springer International Publishing, Cham, 2022, pp. 490–504.

15. Bérard, B.; Bidoit, M.; Finkel, A.; Laroussinie, F.; Petit, A.; Petrucci, L.; Schnoebelen, P.; and Mckenzie, P.: *Systems and Software Verification*.

16. Pnueli, A.: The Temporal Logic of Programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, IEEE Computer Society, USA, 1977, p. 46–57. URL `https://doi.org/10.1109/SFCS.1977.32`.

17. Alur, R.; and Henzinger, T.: Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, vol. 104, no. 1, 1993, pp. 35–77. URL `https://www.sciencedirect.com/science/article/pii/S0890540183710254`.

18. Fernández Adiego, B.: Bringing automated formal verification to PLC program development.

19. Baier, C.; and Katoen, J.-P.: *Principles of model checking*. MIT press, 2008.

20. Darvas, D.; Fernández Adiego, B.; Vörös, A.; Bartha, T.; Blanco Viñuela, E.; and Suárez, V.: Formal Verification of Complex Properties on PLC Programs. vol. 8461, 06 2014, pp. 284–299.

21. Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; and Tonetta, S.: The nuXmv Symbolic Model Checker. *CAV*, 2014, pp. 334–342.

22. Clarke, E.; Kroening, D.; and Lerda, F.: A Tool for Checking ANSI-C Programs. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, K. Jensen and A. Podelski, eds., Springer, vol. 2988 of *Lecture Notes in Computer Science*, 2004, pp. 168–176.

23. Tóth, T.; Hajdu, A.; Vörös, A.; Micskei, Z.; and Majzik, I.: Theta: a Framework for Abstraction Refinement-Based Model Checking. *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, D. Stewart and G. Weissenbacher, eds., 2017, pp. 176–179.

24. Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; Rhein, J.; Schumann, J.; and Shi, N.: Formal Requirements Elicitation with FRET. *REFSQ Tools*, 2020.

25. Giannakopoulou, D.; Pressburger, T.; Mavridou, A.; and Schumann, J.: Automated formalization of structured natural language requirements. *Information and Software Technology*, vol. 137, 2021, p. 106590.

26. Mavridou, A.; Katis, A.; Giannakopoulou, D.; Kooi, D.; Pressburger, T.; and Whalen, M. W.: From Partial to Global Assume-Guarantee Contracts: Compositional Realizability Analysis in FRET. *Formal Methods*, M. Huisman, C. Păsăreanu, and N. Zhan, eds., Springer International Publishing, Cham, 2021, pp. 503–523.

27. Bourbouh, H.; Farrell, M.; Mavridou, A.; Sljivo, I.; Brat, G.; Dennis, L. A.; and Fisher, M.: Integrating Formal Verification and Assurance: An Inspection Rover Case Study. *NFM*, A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, eds., 2021.

28. Dutle, A.; Muñoz, C.; Conrad, E.; Goodloe, A.; Perez, I.; Balachandran, S.; Giannakopoulou, D.; Mavridou, A.; and Pressburger, T.: From Requirements to Autonomous Flight: An Overview of the Monitoring ICAROUS Project. *FMAS'2020*, 2020, pp. 23–30.

29. Mavridou, A.; Bourbouh, H.; Giannakopoulou, D.; Pressburger, T.; Hejase, M.; Garoche, P.-L.; and Schumann, J.: The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. *RE*, 2020.

30. Perez, I.; Mavridou, A.; Pressburger, T.; Goodloe, A.; and Giannakopoulou, D.: Automated Translation of Natural Language Requirements to Runtime Monitors. *TACAS*, 2022.

31. Pressburger, T.; Katis, A.; Dutle, A.; and Mavridou, A.: Authoring, Analyzing, and Monitoring Requirements for a Lift-Plus-Cruise Aircraft. *Requirements Engineering: Foundation for Software Quality: 29th International Working Conference, REFSQ 2023, Barcelona, Spain, April 17–20, 2023, Proceedings*, Springer, 2023, pp. 295–308.

32. Conrad, E.; Titolo, L.; Giannakopoulou, D.; Pressburger, T.; and Dutle, A.: A compositional proof framework for FRETish requirements. *CPP*, 2022.

33. Programmable controllers - Part 3: Programming languages. International standard, International Electrotechnical Commission, 2013. URL `https://webstore.iec.ch/publication/4552`.

34. High Luminosity Large Hadron Collider webpage. URL `https://home.cern/science/accelerators/high-luminosity-lhc`.

35. UNICOS webpage. URL `https://unicos.web.cern.ch/`.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*
01-03-2023

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Automated Verification of Programmable Logic Controller Programs against Structured Natural Language Requirements

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Zsófia Ádám, Ignacio D. Lopez-Miguel, Anastasia Mavridou, Thomas Pressburger, Martin Bes, Enrique Blanco Vinuela, Andreas Katis, Jean-Charles Tournier, Khanh V. Trihn, Borja Fernandez Adiego

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Ames Research Center, Moffett Field, CA 94035

8. PERFORMING ORGANIZATION REPORT NUMBER
L–

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2023–

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 59
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
PLCverif is an actively developed project at CERN, enabling the formal verification of Programmable Logic Controller (PLC) programs in critical systems. In this paper, we present our work on improving the formal requirements specification experience in PLCverif through the use of natural language. To this end, we integrate NASA's FRET, a formal requirement elicitation and authoring tool, into PLCverif. FRET is used to specify formal requirements in structured natural language, which automatically translates into temporal logic formulae. FRET's output is then directly used by PLCverif for verification purposes. We discuss practical challenges that PLCverif users face when authoring requirements and the FRET features that help alleviate these problems. We present the new requirement formalization workflow and report our experience using it on two critical CERN case studies.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
U	U	U	UU		STI Information Desk (email: help@sti.nasa.gov)

19b. TELEPHONE NUMBER ***(Include area code)***
(757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18