A FORMAL SPECIFICATION METHOD FOR PLC-BASED APPLICATIONS

D. Darvas*, E. Blanco Viñuela, CERN, Geneva, Switzerland
I. Majzik, Budapest University of Technology and Economics, Budapest, Hungary

Abstract

The correctness of the software used in control systems has been always a high priority, as a failure can cause serious expenses, injuries or loss of reputation. To improve the quality of these applications, various development and verification methods exist. All of them necessitate a deep understanding of the requirements which can be achieved by a well-adapted formal specification method. In this paper we introduce a state machine and data-flow-based formal specification method tailored to PLC modules. This paper presents the practical benefits and new possibilities of this method, comprising consistency checking, PLC code generation, and checking equivalence between the specification and its previous versions or legacy code. The usage of these techniques can improve the level of understanding of the requirements and increase the confidence in the correctness of the implementation. Furthermore, they can help to apply formal verification techniques by providing formalised requirements.

INTRODUCTION AND MOTIVATION

The complexity of process control systems is steadily increasing, resulting in more and more complex control software. Without an appropriate specification it is increasingly difficult to understand the requirements, hence to develop and maintain the programs.

The motivation of this work originates from CERN (the European Organization for Nuclear Research), where numerous control systems are in use to operate the research facilities. Many of these control systems rely on Programmable Logic Controllers (PLCs). The increasing complexity of the control systems results implies an increasing complexity issue. The non-formal, ad-hoc specification methods in use are less and less able to cope with the complexity of the systems. Therefore we propose a new specification method that aims to handle the increasing complexity of PLC program units.

Obviously, this is not the first work on specification methods for PLCs. *Grafcet* is a standardised specification method [1] based on safe Petri nets. It is convenient to describe finite state machines, but they are not universal enough to be generally used in the development of CERN's control software as the finite state machines are just one part of the final code deployed. *ProcGraph* [2] is one of the recent attempts to improve specifications of PLC programs, but it is on a low abstraction level, too close to the real program code. Other PLC-related specification methods also exist, but none of them seemed to be able to cope with the size and complexity of the PLC programs used at CERN.

Also, there are numerous general-purpose modelling methods that can be used for specifying PLC programs. One of them is the widely-known Simulink Stateflow, that has a complex semantics not adjusted to the needs of the PLC domain, making its usage difficult and potentially error-prone.

The structure of the paper is the following. The next section introduces the main concepts and the structure of the proposed formal specification method. After, the potential benefits of the new method are discussed. Finally, the last section concludes the paper and draws up the future work.

Due to space limitations, the discussion of the syntax and semantics of the specification method is introductory. The reader can find more details in our technical report [3].

MAIN SPECIFICATION CONCEPTS

The goal of our work is to provide a specification method that is a *complete*, *formal* behaviour description of specific functionality which corresponds to one or some PLC components. Therefore it can be used to describe individual PLC components with high precision. Then, these precise descriptions can be naturally composed into complete control systems in the future.

Previous work [4] discussed the requirements towards such a formal PLC specification method. The main requirements can be summarised as follows: (1) providing multiple formalisms adapted to the needs and knowledge of the PLC community, (2) keeping the "core logic" clean by decoupling the input/output-handling, (3) supporting events with proper semantics, and (4) limiting the set of possible errors by limiting the expressivity. The domain-specific requirements are complemented by general requirements, most notably the need for a formal, precise specification formalism, that is also lightweight (can be introduced with small effort) and complete (it describes all required and allowed behaviours). These requirements resulted in PLCspecif, a complete formal specification method for PLC programs.

Structure of the Specification

The main building block of the specification is the *module* that is either a composite module (its behaviour is described by some submodules) or a leaf module (its behaviour is directly described). To provide a specification method that is familiar for the PLC developers, the leaf module descriptions are based on three widely-known formalisms: state machines, data-flow diagrams, and standard PLC timers.

Each module (both composite and leaf modules) is further decomposed into three main parts: (1) input definitions, (2)

* Corresponding author. E-mail: ddarvas@cern.ch, darvas@mit.bme.hu

ISBN 978-3-95450-148-9

core logic, and (3) output definitions. According to the semantics of PLCspecif, these parts are executed sequentially in a loop, following a structure similar to the cyclic execution scheme of the PLCs.

In the *input definitions* part, named expressions can be defined to simplify the specification. For example, if there are three digital inputs from three buttons (`Button1`, `Button2`, `Button3`), but the program provides the same response for pressing any button, writing "`Button1 OR Button2 OR Button3`" in the core logic makes the understanding more difficult. Instead, the user can specify a `ButtonPressed` expression that is defined once and used later in the core logic. This helps to decouple the physical structure (i.e. three digital inputs) from the concepts to be used in the core logic (i.e. a button was pressed).

Some special inputs called *event inputs* can also be defined. An event input is an expression with Boolean type that has a priority assigned. We call an event input *enabled* if its expression is evaluated to true. In each module only the enabled event input with the highest priority can *trigger*.

The input definitions are followed by the *core logic description* part. It can be described using various formalisms, that are introduced later in this paper.

The *output definitions* part is responsible to assign values to the output variables, based on the input values and on the core logic (e.g. the current state of a state machine). This helps to keep the core logic clean. Including the output variable assignments in a state machine (as entry/exit actions or as transition actions) might make the state machine difficult to overview, therefore it is error-prone.

Optionally, the output definitions part can be followed by *invariant properties*. These are additional requirements and assumptions identified during the specification phase that are not obviously described by the core logic, but have to be satisfied by the module.

Expression Descriptions

The input or output definitions may contain complex expressions. While the arithmetic form is suitable to describe simple expressions (e.g. "`a OR b`"), it does not scale up well. PLCspecif supports the usage of other expression description methods: AND/OR tables and switch-case tables.

AND/OR tables were introduced in the RSML formalism [5], but not widely used since. In an AND/OR table each column represents a case, that is true if in all the rows the value of the expression in the row header equals to the value in the corresponding cell of the case. The whole expression is the OR-connection of the defined cases. The symbol "·" marks that the value of the variable is not taken into account ("don't care"). Figure 1(a) shows an example, representing the `a AND NOT b AND (c OR NOT d)` expression.

Switch-case tables —as their name suggests— are based on similar principles as the case constructs of various programming languages. Figure 1(b) shows an example, representing `_Value`, limited by lower limit `PMin` and upper limit `PMax`.

	Case 1	Case 2
a	true	true
b	false	false
c	true	·
d	·	false

(a) AND/OR table

	_Value < PMin	_Value > PMax	result
Case 1	true	·	PMin
Case 2	false	true	PMax
Case 3	false	false	_Value

(b) Switch-case table

Figure 1: Tabular expression description examples.

Core Logic Descriptions

One single formalism cannot conveniently fit the different types of modules (with state-based, data-flow-oriented and time-dependent behaviour). Therefore we introduced three different types of core logic descriptions.

State Machine. A *state machine module* is composed of hierarchical *states* and *transitions*. A state can be *basic* or *composite* (grouping several basic states together). A transition can go from any state to a basic state[1]. It can have a Boolean expression as condition: the transition can fire only if the expression is evaluated to true. There are two kinds of transitions: *event-triggered* and *non-event-triggered*. A transition is enabled, if its source state is currently active, the condition of the transition is evaluated to true, and if it is event-triggered, the connected event triggers. When an enabled transition *fires*, the current active state of the state machine is changed to the target state of the transition. A transition firing cannot cause any other effects (e.g. it cannot provoke actions or do variable assignments).

The brief semantics of the state machine is the following. First, *all* enabled non-event-triggered transitions are exhaustively fired. Then the triggering event input (i.e. enabled event input with the highest priority) is selected and if there is any enabled transition triggered to this event input, this transition will fire. Finally, all enabled non-event-triggered transitions are exhaustively fired again. Note that in each execution cycle at most one event-triggered transition can fire per state machine module.

In addition, state machines can be extended by deep history states, similarly to UML State Machines.

Input-output Connection Module. State machines are suitable for modules that are stateful and the state to be stored can conveniently represented by a handful of states in the specification. However, if the state can only be described by some integers or real numbers (e.g. storing previous measurements or value requests), state machines are inappropriate and we suggest the usage of *input-output connection modules*. The idea of the input-output connection module is inspired by Function Block Diagrams (FBDs) [6] and similar data-flow-like formalisms. It graphically defines how the outputs of the module should be assigned based on the current inputs and outputs from the previous cycles.

This module description consists of *pins* representing input and output values, and *edges* representing data con-

[1] Transitions going to composite states are not allowed as they could make the semantics of a state machine more difficult to understand.

ISBN 978-3-95450-148-9

nections between pins. Furthermore, it can contain blocks, representing common functions (e.g. logic operations, arithmetic operations, selection), user-defined functions, and platform functions.

Figure 2 shows a simple example. Here, `ValueOutput` keeps *its previous value* if the Boolean input `Sample` is false. If `Sample`=true, the new value of `ValueOutput` will be $-1 \times$ `ValueRequest`.

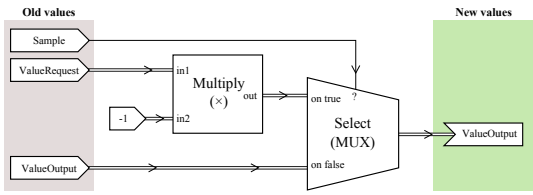


Figure 2: Input-output connection core logic example.

PLC Timers. The state machines and input-output connection modules of PLCspecif do not contain timed behaviours to keep them simple. However, it is crucial to be able to define time-related operations. State machines are often extended by clock variables to describe time, but this method is error-prone, also it does not fit to the existing knowledge of the target group. Instead, we propose to use PLC timers defined in IEC 61131-3 [6] (TP, TON, TOFF). Their semantics is well-known by the developers and they can use these timers confidently.

Example. Figure 3 shows the specification of a simple state machine module. The described component is a combination of a flip-flop and a multiplexer. If the module is enabled, its `Value` output is the `ValueReq` input, limited by `PMin` and `PMax`. The module can be made enabled by having a true signal in one of the `EnableReq` inputs. If there is a rising edge on the `DisableReq` input, the module will be disabled, and in this state the `Value` output will be 0. Disabling the module has priority over enabling it. The module keeps its state if no enable or disable request is received.

In the example one can observe the structure and general elements of the specification, the decoupled input/output handling, and the different ways of specifying expressions. To help the understanding, each part of the specification can be annotated by textual descriptions, however we omitted them in this example to reduce its size.

BENEFITS

This section summarises the different benefits of using a formal specification method tailored to the PLC domain, such as PLCspecif.

Improved Understanding

Software of control systems, especially the ones evolving for long time can become complex. Proper documentation and specification is primordial to be able to fully understand such PLC program. This understanding is necessary both for reusing and maintaining the components.

ExampleModule

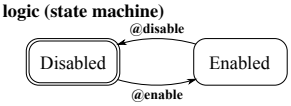
Assigned inputs:
- ValueReq : INT16
- EnableReq_fromLogic : BOOL
- EnableReq_fromScada : BOOL
- EnableReq_fromField : BOOL
- DisableReq : BOOL
- PMin : INT16 param
- PMax : INT16 param

Assigned outputs:
- Value : INT16
- Status : BOOL

Input definitions: — (none)

Event definitions:
- @disable ⇐ `rising_edge`(DisableReq) (`pri`=1)
- @enable ⇐ EnableReq_fromLogic OR EnableReq_fromScada OR EnableReq_fromField (`pri`=2)

Core logic (state machine)



Output definitions:
- _Value =

ValueReq < PMin	ValueReq > PMax	result
T	·	PMin
F	T	PMax
F	F	ValueReq

- Value =

in_state(Enabled)	result
T	_Value
F	0

- Status = `in_state`(Enabled)

Invariant properties:
- ALWAYS *PMin* ≤ *Value* ≤ *PMax* ASSUMING *PMin* ≤ *PMax*

Figure 3: Example module specification.

A simple, restricted, domain-specific formal specification method that matches target platform (PLC) as PLCspecif can help the users to express their requirements and intents confidently. Decoupling the input/output handling also helps to have a clean, easy-to-understand specification, where the core logic can be simple. PLCspecif provides a simple specification method with simple semantics, especially suitable for the needs of the PLC-based process control domain. For example, handling rising and falling edges is a base feature of PLCspecif, as it is an often used feature in PLC programs. Its manual implementation in similar formalisms usually involves concurrency, that can make the behaviour of the module less intuitive and less understandable.

Consistency Checking and Formal Verification

A specification with formal semantics may not only improve the understanding of the specifiers and developers, but it opens the door for formal verification as well. Different requirements can be checked on the specification directly using formal techniques, such as model checking: *general properties* (e.g. the model does not contain deadlock, livelock or nondeterminism) and *application-specific properties* (expressing additional requirements and assumptions). These properties can be checked in the specification phase on the formal specification, as well as later on the implementation. For example, in Figure 3, a safety property expresses the assumption that the *Value* output will be between the two given limits: "ALWAYS `PMin` ≤ `Value` ≤ `PMax`" assuming that the `PMin` value is not above `PMax`. However, using formal verification it can be shown in the specification phase that

ISBN 978-3-95450-148-9

the property is violated if `PMin` > 0. The given counterexample demonstrates that if the module is in disabled state, the `Value` is set to zero, even if this is out of the `PMin`..`PMax` range.

The verification is made possible by the definition of formal semantics for PLCspecif, defined as a transformation to a lower-level formalism: timed automaton. Timed automaton have a well-defined semantics, also this representation can be directly the input of a suitable model checker tool.

A formal specification can also be the source of automated test generation, thus to use a variety of verification methods.

Code Generation

A complete, formal specification contains the behaviour description that is required to implement the PLC module. Based on this knowledge, a developer can manually create the code, or this process can be automated. The basis of the code generation is the formal semantics of PLCspecif. The code generation method is constructed as a systematic representation of the structures of the timed automata describing the formal semantics in a PLC programming language. This way the generated code can be correct by construction.

While the specification contains the behaviour of the module, the code generator shall be configurable to select implementation alternatives in order to provide a code expected by the user that is easily understandable. In case of PLCspecif, the user has influence on the structure of the code (which submodules should be extracted into functions or function blocks, how to represent state machines, etc.). Based on the specification and the configuration of the way of implementation, it is possible to generate code for PLCs. Currently, our proof-of-concept code generator supports the Siemens implementation of the IEC 61131-3 standard ST (Structured Text) language [6], the Structured Control Language (SCL).

A common counterargument against PLC code generation is the fact that most code generators provide complex, unmaintainable code, while it is required to have full understanding of the generated PLC code. By careful design and configurable structure PLCspecif allows to generate readable code, following the structure of the specification.

Equivalence and Conformance Checking

Equivalence and conformance checking is the process of comparing the behaviour of models with formal semantics. It allows to compare two models against each other by checking for example if they provide the same outputs for the same input sequences. This can show the equivalence of two modules with different internal structure. Besides this use case, conformance checking can be applied to check implementation against specification (e.g. if a manual implementation is conformant to a specification, or if a specification properly captures the behaviour of a legacy implementation).

Different equivalence and conformance relations have been defined with different sensitivity. This way the user can focus on the differences that are relevant from the point of view of the correctness of the PLC program.

These equivalence/conformance relations can be checked on the timed automata constructed for the formal semantics description, or on execution traces. Comparison between specification and implementation is allowed by previously designed methods [7] for building models of PLC code.

Examples and Experiences

As an initial validation of the specification method, we mention our experiment that targeted one of the base objects used in CERN's UNICOS framework [8]. The PLCspecif specification was made on the basis of understanding the behaviour (by having discussions with the developers) of the OnOff object, that is the UNICOS representation of a binary state equipment (e.g. valve, heater, pump). By being able to capture and clarify the intended behaviour of a PLC code of 600 lines, we demonstrated that PLCspecif can scale up to the size of real PLC program components. Code generation was also performed and the produced code was tested in the test bench used for UNICOS modules.

CONCLUSIONS AND FUTURE WORK

In this paper we have introduced PLCspecif, a formal behaviour specification method for PLC programs. Using such a method can provide benefits in many ways: it improves understanding, it can be a basis of verification and automated implementation. Our plans for future work includes the definition of extensions that address the integration of modules, to support the specification of the software of complex control systems.

REFERENCES

[1] *IEC 60848:2013 GRAFCET specification language for sequential function charts*, IEC Std., 2013.

[2] T. Lukman, G. Godena, J. Gray, M. Heričko, and S. Strmčnik, "Model-driven engineering of process control software – beyond device-centric abstractions," *Control Engineering Practice*, vol. 21, no. 8, pp. 1078–1096, 2013.

[3] D. Darvas, E. Blanco, and I. Majzik, "Syntax and semantics of PLCspecif," CERN, Report EDMS 1523877, 2015, `https://edms.cern.ch/file/1523877/`.

[4] D. Darvas, I. Majzik, and E. Blanco, "Requirements towards a formal specification language for PLCs," in *Proc. of the 22th PhD Mini-Symposium*. BUTE DMIS, 2015, pp. 18–21.

[5] N. Leveson, M. Heimdahl, H. Hildreth, J. Reese, and R. Ortega, "Experiences using statecharts for a system requirements specification," in *Proc. of the Sixth Int. Workshop on Software Specification and Design*. IEEE, 1991, pp. 31–41.

[6] *IEC 61131-3:2013 Programmable controllers—Part 3: Programming languages*, IEC Std., 2013.

[7] B. Fernández, D. Darvas, J.-C. Tournier, E. Blanco, and V. M. González, "Bringing automated model checking to PLC program development – A CERN case study," in *12th Int. Workshop on Discrete Event Systems*. IFAC, 2014, pp. 394–399.

[8] E. Blanco *et al.*, "UNICOS evolution: CPC version 6," in *Proc. of the 13th Int. Conf. on Accelerator & Large Experimental Physics Control Systems*, 2011, pp. 786–789.

ISBN 978-3-95450-148-9