Requirements towards a formal specification language for PLCs

Dániel Darvas*†, István Majzik* and Enrique Blanco Viñuela†
*Budapest University of Technology and Economics, Department of Measurement and Information Systems
Budapest, Hungary, Email: {darvas,majzik}@mit.bme.hu
†European Organization for Nuclear Research (CERN), Engineering Department
Geneva, Switzerland, Email: {ddarvas,eblanco}@cern.ch

***Abstract*—One of the main obstacles of using formal verification for complex PLC (Programmable Logic Controller) programs is the lack of formal requirements. There are no widely used specification methods that could serve as input for formal verification; also that could help the developers to capture the behaviour and handle the complexity of these programs.**

The goal of this research is to bring formal specification closer to the PLC domain in order to help the development, verification and maintenance. This paper aims to briefly overview the particularities of the PLC domain and the state of the art in formal specification. Then it collects the requirements towards a PLC-specific formal specification language based on general works, comparative case studies and own experiences at CERN. Also, it draws up a sketch of a possible specification method that follows the collected requirements.

I. Introduction and Background

Programmable Logic Controllers (PLCs) are robust industrial computers providing standard solutions for control systems. For this discussion, these computers can be treated simply as computers with limited resources and a large number of low-level inputs and outputs. Physical inputs and outputs are represented as input and output variables in the programs. The execution of the user program is mainly cyclic: in each *scan cycle* (1) the input values are sampled and stored in the memory, then (2) the user program is executed, next (3) the computed output values are written to the physical outputs. The values of the input variables and physical outputs are both stable in the memory during the computation phase. The user program can also store states, thus the outputs are depending both on the current and previous input values. The programming languages are defined in the corresponding IEC 61131-3 standard [1].

A. Motivation

The motivation of this work is coming from CERN (European Organization for Nuclear Research) where most research facilities, like the Large Hadron Collider (LHC) rely on a multitude of PLCs. To cope with the complexity and to reduce the maintenance needs, most of the PLC programs at CERN are developed using the UNICOS framework [2]. This framework provides a design methodology, code generation tools and a library of *base objects*. These base objects are not further decomposed in the implementation, but their complexity is still high. Therefore the (re-)use of these base objects needs a precise specification. Besides the usage, as all the systems at CERN are depending on these objects, their modification is a critical task that requires high level of understanding of the underlying logic to avoid the negative collateral effects. These reasons imply a vital need for a suitable specification method that is neither available yet at CERN, nor in the PLC community in general. Previous work aiming to apply formal verification to these base objects [3] also emphasized the need for a good, formal specification method. Currently the main obstacle to use model checking on these PLC programs is the lack of precise, formal requirements to be checked, not the lack of available efficient verification tools.

The rest of the paper is structured as follows. Sect. II overviews the main requirements towards the specification language to be proposed taken into account the peculiarities of the PLC domain and based on the literature overview. Sect. III discusses already existing specification methods related to this work. Sect. IV sketches up the concepts of a new PLC specification language. Finally, Sect. V summarizes the paper.

II. PLC Specification Language Requirements

The goal of this research is to overcome the previously mentioned issues by proposing a suitable PLC specification language. First the domain-specific requirements are summarized, then previous work is discussed to gather requirements towards a good, useful formal specification language.

A. Domain-specific requirements

The motivation of this work is a real insufficiency in the PLC domain, therefore it is indispensable to address the existing problems and particularities. Many specification language-related requirements can be extracted from the previously developed programs and by discussing the problems with the developers. The main domain-specific needs and challenges are summarized below based on the experience gained at CERN.

a) Events: Although the execution of PLC programs is cyclic and not event-triggered, the concept of events still exists (in a latent way). In fact, many Boolean inputs represent events or external actions aiming to modify the internal state of the controller. Events should be treated as "first-class citizens" in the specification.

b) Event semantics: It is also important to adopt a semantics that is appropriate for the PLC domain. Due to the cyclic behaviour, we cannot have the assumption usual in event-triggered systems that all previous events are fully handled before a new event is triggered. As in a PLC multiple

events can happen simultaneously, the priority of events has to be defined. If multiple *contradictory* events happen, the one with the highest priority should suppress the events with lower priorities, but several *independent* events can trigger in the same cycle.

c) Clean core logic: PLC programs work directly with physical input and output signals, therefore a significant part of the programs has to perform *input and output handling*. While this task is unavoidable, decoupling the I/O-handling helps to focus on the core logic. As the PLCs have limited resources, the developers try to minimize the number of variables, but this approach is not needed to be followed in the specification. In the specification, it is important to use "concepts" rather than expressions, i.e. it is better to *define internal variables* (variables for specification purposes only) defined by expressions on the input variables, and to use these internal variables in the definition of the logic.

d) Hierarchical, modular structure: To support reuse and the abstract design of specification, the method should provide a hierarchical, modular structure. Hierarchy and modularity helps to "divide and conquer", to have a simple logic in the leaf modules (i.e. modules that are not further decomposed), and to avoid duplicated specification of same submodules.

e) Multiple formalisms: The leaf modules of the hierarchical structure should be specified by a specification language adapted to the behaviour of the current module, thus multiple module definition formalisms are needed. Among others, control-oriented, data-processing-oriented, as well as timing-related behaviours shall be considered. State machines and logic circuits are widely used in the development and specification of PLC program modules, but these can only be considered as "explanatory doodling", not as precise specification, as their formal semantics are not defined. New, formal languages should be proposed that are adapted to the specialities of the domain and based on the existing knowledge and practice of the developer community.

f) Limited expressivity: Rich languages can provide rich features, but also more space for problems and they can require longer training period. The expressivity of a PLC specification language should be restricted. In some cases, some of the restrictions can be explicitly relaxed, but then the verification of these parts should be carried out with special attention.

g) Time-dependent behaviour: As PLCs can have time-dependent behaviour, the proposed formalism should support timed models. This behaviour is generally captured by timers in PLCs. Three kinds of timers are defined in the corresponding standard [1]: TP (signal pulsing), TON (on delay), TOFF (off delay). Each has a different semi-formal semantics. These standardised timers should be part of the language as modules, because they are widely used and well-known by the automation engineers.

B. Requirements from the Literature

Besides the experience from practice, previous work in the literature can point to necessary requirements and best practices of developing a new, domain-specific specification language. In this part some general work are summarized.

In the first place, a (formal) specification method should satisfy obvious general requirements, e.g. it has to be correct, unambiguous, consistent, verifiable [4].

In 2000, van Lamsweerde published a survey [5] on existing formal specification methods and a roadmap for the future. The conclusion of the paper is that "formal specification techniques suffer a number of weaknesses". Such weaknesses are e.g. the (1) limited scope (i.e. the specification can only capture a part of the system), (2) poor separation of concerns (i.e. the intended properties, the environmental assumptions and the properties of the application domain overlap), (3) too low-level ontologies, (4) high cost, and (5) poor tool support. The author states that future formal specification methods should be *lightweight* (i.e. not requiring deep formal methods expertise), at least partially *domain-specific*, structured, multi-paradigm and multiformat (i.e. integrating multiple languages and letting the specifier use the best for the current needs, thus for each subsystem the most appropriate language shall be chosen, or different languages might be necessary for specifying functional and extra-functional requirements).

Knight et al. approached the question "Why formal methods are not used widely?" more practically. In [6] they designed an evaluation framework to assess formal specification methods. Furthermore, they selected three specification languages (Z, PVS, Statecharts) and applied them for a subsystem of a nuclear reactor. Then, the specifications were assessed by developers not expert in formal methods and by nuclear engineers. After a short training period, the general idea and the main advantages of formal specification were welcomed and understood. From the point of view of the nuclear engineers, Z and PVS were too complex, but Statecharts were claimed to be effective for communication and easy to learn, although difficult to search and navigate. The authors emphasize that often overseen features are also recommended to help the industrial usage. These comprise the support for documentation and readability, e.g. by including free-text annotations connected to the elements of the specification.

A possible reason why Statecharts [7] are often welcomed by the non-computer engineers can be the fact that it was not developed in a purely academic environment, but in strong collaboration with avionics engineers, taking their habits and knowledge into account [8].

A good example for Statechart-based languages is the RSML formalism. It was designed to help the specification process of the TCAS II avionics system. In [9] the authors discuss some lessons learned, like simplicity and readability are "extremely important" [9]. Based on the experience from RSML, a new language (SpecTRM-RL) was created, but both seem not to be used widely. Also, the provided solutions do not fit to the PLC domain and using only a Statechart-like formalism does not provide a convenient method for specifying PLC programs, e.g. behaviour of modules with many numeric state variables are difficult to be captured.

A recent work on the topic collects requirements towards a model-based requirements engineering tool for embedded systems [10]. In their survey the highest ranked requirements were the support for various different representations, document generation, expression of non-functional requirements, and for maintaining traceability links; not formal verification

or automated code generation.

The literature overview briefly summarized above pointed out that a good specification should be *lightweight* [5], [6]. This also implies the need for *domain-specificity*. The specification should be adapted to the concrete (sub)system being specified, therefore a "*portfolio of languages*" is needed, of which the most appropriate can be chosen for each submodule [5]. Similarly, the functional and extra-functional properties need different languages.

III. Existing, Related Specification Methods

The motivation for a new specification language is coming from CERN, but we believe that the problem is more general. There is no widely-accepted, formal or semi-formal specification method for PLC programs. The IEC 60848 standard [11] defines a specification language based on finite state machines called *Grafcet*. This language can be seen as a safe Petri net extended with guards and variable assignments. Although in some cases Grafcet can capture the behaviour of some submodules, it is not applicable generally. Also, the semantics of the language is complex and sometimes not intuitive, furthermore it is confusing that a standardised PLC implementation language, the SFC uses similar syntax with different semantics [12].

Other work also targeted the formal specification of PLC programs. In [13] the authors describe a specification method called *ST-LTL* based on LTL (Linear Temporal Logic). In fact, this is a formal language that can be easily checked on the implementation. On the other hand, the formalism is far from the automation engineer's general knowledge, also it can be difficult to scale with the growing size and complexity, furthermore it is not obvious to see if the specification is inconsistent or incomplete.

The formal specification methods are not widespread in the PLC domain yet, but many formalisms are widely used in computer engineering, e.g. the B Method, the VDM-SL, the Z notation, or temporal logics. An obvious solution would be to use one of them. However, it is not really a suitable solution, as these methods are typically (1) too difficult to be used for engineers not trained specially in formal verification, and (2) the usage is even more difficult in the case of PLC programs, as the methods are not adapted specifically to the PLC semantics, even the general properties of the PLCs (e.g. cyclic execution) have to be explicitly defined, therefore it is difficult to address the domain-specific requirements discussed in Section II-A.

A good example for formal specification from the industry is the SCADE Suite by Esterel Technologies/ANSYS. It provides a model-based environment to support the development of critical embedded software, including an automated code generator compliant to various industrial standards. However, even if its cyclic computation model is close to the PLC computation model, this toolset does not provide solutions specifically for IEC 61131-compliant PLCs.

The B, Grafcet, SCADE, Spec-TRM, ST-LTL, VDM-SL, and Z methods are compared in the Table I based on some of the main requirements discussed in Section II-A and II-B. As can be seen, none of the tools/methods provide a solution that is adapted to the PLC domain, satisfies most of the domain-specific and the general requirements at the same time. Therefore we think that none of these methods could be integrated to a PLC development process with low cost.

TABLE I. Comparison of available specification methods

		B / Z	Grafcet	SCADE	Spec-TRM	ST-LTL	VDM-SL
General req.	Precise meaning	+	+	+	+	+	+
	Lightweight	–	+	+	+	–	–
	Specific for the PLC domain	–	+	–	–	+	–
	Support for documentation	–	–	+	+	–	–[1]
	Available tool support	+	+	+	–	–	+
DSL req.	II-A a) Events	–[1]	+[2]	+	+[2]	–	–[1]
	II-A b) Event priorities, suppressions	–	–	–	–	–	–
	II-A c) Clean core logic	–	–	+[2]	–	–	–
	II-A d) Hierarchical, modular structure	+	+[2]	+	+	+[2]	–
	II-A e) Multi-formalism	–	–	+	–	–	–
	II-A g) Time-handling	–	+	+	+[2]	+[2]	–[1]

[1] Not supported in the original formalism, but supported in one of its extensions.
[2] With restrictions.

IV. Main Concepts of the Proposed Solution

Based on the requirements and experiences discussed before, a first version of a specification language suitable for PLC programs is defined here. Only a short sketch of the method is provided, not an extensive description. The starting point of the proposed solution is the Statechart formalism, as it is intuitive and generally close to the (automation) engineers. However, it has to be extended and altered, for the following reasons.

1) The Statechart formalism is not a complete specification method, it just captures the behaviour of a part of the system. It was intended to be embedded in a broader framework [8].
2) The common semantics definitions do not fit entirely to the PLC properties. For example, the "events" are represented by input variables and the program execution is cyclic, therefore it cannot be assumed that only one event will be processed at a time, as it is assumed for example in the semantics of the UML (Unified Modeling Language) State Machines.
3) Not every stateful behaviour can be captured efficiently by state machines, e.g. modules with integer or floating-point state variables. This is common in process control programs, thus a Statechart-based formalism should be accompanied by other languages.

A. Structure (High-level Syntax)

The base element of the proposed solution is the *module*. A module is either a *composite module* that is refined by submodules, a *leaf module* capturing the behaviour of a part of the system, or an *alternative module* that helps to choose between multiple implementations (e.g. timed or non-timed implementation) based on some parameters.

In every module it is possible to define internal variables based on the input variables and outputs based on the current state of the module. In leaf modules events can also be defined. This decoupling of I/O-handling and computation helps to keep the *core logic* as small as possible. The scope of the internal

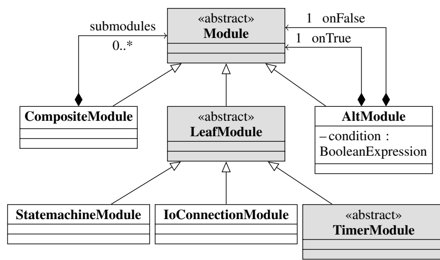

Fig. 1. Metamodel of the module structure

variables is by default restricted to the defining module and its descendants, but this restriction can be explicitly relaxed. The core logic of the leaf modules can be defined using one of the three defined formalisms, adapted to the logic it is implementing. The first possibility is to use a well-defined, restricted *statechart* describing the logic that allows to use hierarchical states (without regions), transitions (with guards or external triggers but without variable assignments, actions and internal events), and history nodes.

If it is not efficient or not possible to use statecharts, because for example the module is using integer state variables, an *input-output connection module* can be used. The representation contains all input variables that can be used in the computation and all output variables that have to be assigned by the given module. The logic description consists of defining different stateless computation blocks and the directed connections between the variables and block inputs and outputs. In this way the assignment rule of each output variables can be given graphically. Furthermore, it is easy to enforce that each output variable should be assigned exactly once in each cycle. The module structure explained above is summarized by its metamodel in Fig. 1.

It is necessary to give multiple possible syntax not only for describing the core logic, but also for describing the input and output definitions. In simple cases, ordinary Boolean expressions can be used. In more complex cases, the AND/OR-tables (like the ones defined in [9]) are more efficient and less error-prone.

In the proposed solution, every element can be annotated by the user to make easier the understanding for example by providing important explanations. The annotations can also help to use the specification (or an artefact generated from the specification) as documentation for maintenance and reuse.

B. Execution (Informal Semantics)

The execution of each module consists of 4 phases. It is started by (1) computing the defined input expressions and the (2) enabled events. Next, the (3) logic of the module is executed. The execution of each module is finished by (4) computing the values of the defined output variables.

The execution of the logic depends on the applied formalism in the case of leaf modules. For composite modules, this means the *sequential* execution of the submodules, in a predefined order. The execution of an alternative module consists of executing the module corresponding to the evaluated condition value. In a statechart module, the execution means the exhaustive firing of all enabled transitions that are not triggered by any event. Then at most one event is selected that is enabled and not suppressed by any higher priority event, and at most one transition fires that is triggered to this selected event. After that all enabled non-triggered transition fires. The execution of an input-output connection module is the iterative evaluation of all defined signals starting from the input variables and the previous values of output variables, until the new values can be assigned to the outputs.

V. SUMMARY AND FUTURE WORK

This paper described a first step towards a new formal specification language for complex PLC programs. The goal of this research is to provide a specification method tailored to the PLC domain. This paper sketched up the main concepts of a potential specification language based on (1) the general requirements towards formal specification methods, (2) the specialities of the PLC domain, and (3) the experienced challenges and difficulties.

Future work includes the design and description of formal syntax and semantics of the language. This should be followed by providing tool support to construct the specification, to verify its consistency and to check the conformance of PLC programs in regard to the specified behaviour. After an experimental phase and collecting feedback from its users the specification method is intended to be introduced in the PLC development workflow of CERN.

REFERENCES

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

[2] E. Blanco Viñuela *et al.*, "UNICOS evolution: CPC version 6," in *Proc. of the 12th Int'l Conf. on Accelerator & Large Experimental Physics Control Systems*, 2011, pp. 786–789.

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

[4] *IEEE Std 830-1998 Standard*, IEEE Computer Society Std., 1998.

[5] A. van Lamsweerde, "Formal specification: A roadmap," in *Proc. of the Conf. on The Future of Software Engineering*. ACM, 2000, pp. 147–159.

[6] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano, "Why are formal methods not used more widely?" in *4th NASA Langley Formal Methods Workshop*, 1997, pp. 1–12.

[7] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[8] D. Harel, "Statecharts in the making: A personal account," in *Proc. of the Third ACM SIGPLAN Conf. on History of Programming Languages*. ACM, 2007, pp. 5–1–5–43.

[9] M. Heimdahl, N. Leveson, and J. Reese, "Experiences from specifying the TCAS II requirements using RSML," in *Proc. of the 17th AIAA/IEEE/SAE Digital Avionics Systems Conf.*, vol. 1, 1998, pp. C43/1–C43/8.

[10] S. Teufl, M. Khalil, and D. Mou, "Requirements for a model-based requirements engineering tool for embedded systems: Systematic literature review and survey," fortiss GmbH, White Paper, 2013.

[11] *IEC 60848:2013 – GRAFCET specification language for sequential function charts*, International Electrotechnical Commission Std., 2013.

[12] J. Provost, J.-M. Roussel, and J.-M. Faure, "A formal semantics for Grafcet specifications," in *IEEE Conf. on Automation Science and Engineering*, 2011, pp. 488–494.

[13] O. Ljungkrantz, K. Åkesson, M. Fabian, and C. Yuan, "A formal specification language for PLC-based control logic," in *Proc. of the 8th IEEE Int'l Conf. on Industrial Informatics*, 2010, pp. 1067–1072.