Well-Formedness and Invariant Checking of PLCspecif Specifications

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), Beams Department
Geneva, Switzerland, Email: {ddarvas,eblanco}@cern.ch

***Abstract*—Developers of industrial control systems constantly quest for quality in order to improve availability and safety. Some of the threats to quality are the development errors due to incorrect, ambiguous or misunderstood requirements. Formal specification methods may reduce the number of such issues by having unambiguous, mathematically sound semantics, which also allows the development of precise analysis methods. In this paper we present two of the analysis methods developed for PLCspecif, our formal specification language targeting PLC modules: well-formedness analysis and invariant analysis.**

I. Introduction and Background

Industrial control systems (ICS) operate a wide variety of plants: e.g. chemical, pharmaceutical, or water treatment plants [1]. Typically these systems rely on programmable logic controllers (PLCs) which are robust industrial machines, programmed in variants of the languages defined in IEC 61131-3 [2]. As we rely more and more on such systems, their safety and availability is a priority. While these systems are often not safety-critical as there are dedicated systems to ensure the safety, an outage might cause significant economic loss.

The development errors of the PLC software are threats to availability and safety. Many of them are caused by ambiguous, imprecise, or incorrect requirements. Formal specifications provide ways to improve the quality of the specification both by providing restricted and precise languages (compared to natural languages) and by having unambiguous semantics, thus allowing the usage of automated analysis methods which can reveal potential problems, such as contradictions, impossible cases, etc.

The usage of formal specification in the ICS domain is not wide yet, as the existing formal specification languages are not well-suited for the target audience. The existing and widely-known, general-purpose formal specification methods are not adapted to the ICS domain, therefore their usage necessitates deep knowledge and excessive effort. Such effort can only be justified in case of highly critical systems.

The authors analysed the specialities of the ICS domain [3] and proposed PLCspecif, a formal language to specify the behaviour of PLC software modules [4]. We claim that the specification of PLC software modules (either isolated safety logics or reusable objects) are the first targets for formal specification, because their specification provides a good effort–benefit ratio. However, a formal specification method does not guarantee by itself that the specified behaviour is well-formed, correct and matches the intentions. In this paper we target these challenges by providing two analysis methods for PLCspecif.

The rest of the paper is structured as follows. Section II briefly overviews the related work. Section III introduces the key concepts of PLCspecif, our formal specification language for PLC software modules. Next, two analysis methods are discussed: Section IV shows the well-formedness checking of PLCspecif specifications, then Section V presents invariant checking. Finally, Section VI concludes the paper.

II. Related Work

Neither formal specification, nor static analysis is a widely-used technique in the field of PLC software development.

The static analysis of PLC programs was targeted in [5], [6]. Some commercial tools are available as well, such as *PLC Checker*[1] or *logi.LINT*[2]. Although static code analysis may point out mistakes or code smells without requiring excessive effort, due to the wide variety of PLC languages and the different needs in the various application domains, the usage of static analysis tools is not general yet.

Formal specification is even less wide-spread in the PLC software development domain. As discussed earlier, the usage of general-purpose specification languages need great effort, therefore they are only applied in highly critical systems. There were several attempts to provide formal or semi-formal specification methods, directly targeting PLC programs. *ST-LTL* [7] is a variant of the LTL formalism, specifically targeting PLC programs. As this method is rather simple, there is no need for static analysis methods.

ProcGraph [8], [9] is a semi-formal specification method based on state machines. According to the published examples, the specifications may become large and complex. The specifications often include PLC code snippets, this also increases the complexity and the difficulty of understanding and therefore the specification might be error-prone. However, static analysis or invariant checking methods are not mentioned as part of the proposed approach [9].

NuSCR [10] is a formal specification method included in the NuSEE environment for specification and verification of

[1] http://www.itris-automation.com/plc-checker/
[2] http://www.logicals.com/en/add-on-products/151-logi-lint

PLC-based safety-critical systems. Supposedly NuSRS, the requirement specification editor based on the NuSCR approach included in NuSEE contains certain well-formedness checks, but static analysis and invariant checking is not explicitly mentioned in [10]. The NuSDS tool that is to be used in the design phase contains certain consistency checks, but the details are not discussed.

III. The PLCspecif Specification Language

The PLCspecif language is designed to provide an easy-to-use, practice-oriented, yet formal specification of PLC software module behaviours. The behaviour description is complete (the description does not contain abstract parts left for later decision), this is why we emphasise that PLC software modules are targeted, where such complete specification is feasible.

In [3] the authors reviewed the literature and the real needs experienced in the PLC-based ICS development processes at the European Organization for Nuclear Research (CERN). Such requirements towards the formalism are domain-specificity, appropriate event semantics, support for a variety of formalisms and decoupling of input/output handling (pre- and post-processing) from the core logic definition [3].

Based on these identified needs, PLCspecif was designed to be a hierarchical, module-based specification language. Each module is either a composite module (whose behaviour is described by submodules) or a leaf module (describing a certain part of the global behaviour). Each module has several parts: input and event definitions, core logic definition, and output definitions. The core logic of the leaf modules can be described using one of the three defined formalisms: state machines, input-output connection diagrams (a data-flow description formalism adapted to PLC programs) or PLC timers. These formalisms have domain-specific, special, unified semantics which allow to mix multiple formalisms in the same specification and to use the most appropriate formalism for each part of the described behaviour.

The semantics of PLCspecif is described formally. For this we have defined a simple, low-level timed automata (TA) formalism, and a precise mapping of PLCspecif specifications to this TA formalism. TA was chosen to be the underlying formalism for semantics definition to facilitate the usage of formal verification directly on the specification.

Due to space restrictions we omit the detailed discussion of the syntax and semantics of PLCspecif. The reader find the formal definition of PLCspecif in our report [11].

The formal specification defines the desired behaviour in an unambiguous way. However, besides the clean description, various methods were developed to make PLCspecif more useful in practice. A *code generation* method was designed [12] that constructs PLC code with a behaviour that corresponds to its specification. In case of legacy or safety systems this method might not be appropriate or applicable, thus a method is required to show the correspondence of an existing PLC code to a specification. For this reason, a *conformance checking* method was designed too [13]. It checks the conformance between the implementation and the specification using model checkers, with configurable level of conformity.

TABLE I
Categories of Well-Formedness Rules

	Category	# Rules
(1)	Field value uniqueness	3
(2)	Object-local checks	12
(3)	Reference restrictions	4
(4)	Restricting non-local references	9
(5)	Expression element restrictions	13
(6)	Complex structural checks	18
(7)	Type constraints	10
(8)	Complex semantic checks	3
	Total	72

However, neither code generation, nor conformance checking can guarantee a correct implementation if the specification is incorrect, contradictory or malformed. Therefore we developed additional methods to detect malformed or unintentional behaviour descriptions and therefore to increase the confidence in the correctness of the developed formal specifications. In the following sections two such methods are introduced: well-formedness checking and invariant checking.

IV. Well-Formedness Checking

The syntax and semantics of the PLCspecif language is defined in our report [11]. It provides a metamodel (abstract syntax) and a concrete syntax for the language, furthermore informal and formal semantics. However as usual, the metamodel could not express all constraints that a specification (instance model of the metamodel) has to respect in order to be considered correct and meaningful. Therefore further restrictions are formulated as well-formedness rules.

We defined 72 well-formedness rules for PLCspecif in [11], which are additional restrictions to the abstract syntax besides the metamodel of the language, ensuring that the specification is well-formed, meaningful and deterministic. A possible categorisation of the rules is shown in Table I. For example, rules in group *(3)* define additional restrictions for the references of the objects. Rules in group *(4)* restrict the reference to non-local elements (e.g. referring to a state of a state machine from a different module). The rules in group *(5)* restrict the set of elements of an expression (e.g. an expression which is not in a state machine module should not refer to a state). Group *(7)* contains rules constraining the expression types (e.g. a transition guard should have a Boolean type).

Most of these rules (mainly in groups *(1)–(5)*) are simple, restricting the use of certain incorrect or undefined constructs that are not forbidden by the metamodel; restricting the types of references; or prescribing name uniqueness.

Obviously, the definition of these rules is not enough, it has to be checked automatically whether they are respected or not. This checking is done after reading and parsing the specification, on its abstract syntax graph (instance model of the defined metamodel). Most of the simple well-formedness

rules (in groups *(1)–(5)*) are implemented in Java. Many of these rules can be efficiently implemented in a dozen lines of code, not necessitating any more advanced methods. Others, typically rules in groups *(6)–(7)*, need more implementation effort, e.g. to check whether the guard of a transition has a Boolean type or not, the type inference has to be implemented.

The implementation of the complex semantics checks (rules in group *(8)*) is more difficult, for example to analyse that the outgoing transitions of a certain state are mutually exclusive (no conflict is possible); or that it is not possible to have an infinite transition firing run in the state machines. In these cases the manual implementation of checking these rules would require significant effort. However, these rules can be transformed into a SAT (Boolean satisfiability) problem. For example, to check whether a state does not have any conflicting outgoing transitions, the guards and priorities of these transitions have to be collected, then it has to be checked whether there exists a pair of transitions on the same priority level with guards that can be satisfied at the same time.

We have used the Microsoft Z3 SAT solver [14] for this purpose, as it provides state-of-the-art algorithms, good performance and Java integration. The usage of Z3 provides an efficient and automated way (thus hidden from the user) to check these more complex rules. Besides checking the satisfaction, the witness ("model" in SAT terminology) generated by Z3 can help the user by pointing out the source of the violation of the well-formedness rules.

PLCspecif was used for the specification of two real examples since its creation: for a reusable PLC module library of the UNICOS framework[3], and the logic of a safety-critical controller used at CERN [15]. In both cases the well-formedness checking was able to identify mistakes made during specification, for example conflicting definitions, unused variables, ambiguous priority settings and conflicting guard expressions.

V. Invariant Checking

The static analysis of well-formedness rules helps to ensure that the specification follows the rules of PLCspecif which are not enforced by the syntax itself. This is required for any specification.

However, in certain cases additional requirements have to be checked too. The formalisms provided for the core logic definition in PLCspecif (state machines, input-output connection diagrams, PLC timers) focus mainly on the *functionality* of the defined module. The specification of the behaviour may hide safety or invariant properties that are required to be satisfied by the specified module. For example, it might not be obvious to see the satisfaction of requirements such as "Outputs `a` and `b` shall not be *true* at the same time." or "Output `c` shall always be within the range `d..e`." based on the state machine-based core logic definitions and the output definitions.

Previous work on the model checking of PLC programs [16] demonstrated that checking various, typically invariant or safety properties on PLC programs using model checkers is feasible and may uncover well-hidden faults in the implementations. One of the difficulties of this approach is the formalisation of models and the properties to be checked for the model checkers. To reduce this obstacle, we have developed *PLCverif* [17], a tool that automatically generates artefacts for model checkers using inputs that are known to the PLC developers. The *models* are generated from the source code of the PLC programs, via an intermediate model (IM). The IM formalism allows the model checker-independent reduction of the formal models and facilitates to use multiple widely-known model checkers (e.g. nuXmv, UPPAAL, ITS). The *property* for model checking is defined using requirement patterns which are easy-to-fill fixed sentences in English with given placeholders, e.g. "If α is true (at the end of the PLC cycle), then β is true (at the end of the same cycle)", where α and β are placeholders of Boolean expressions, composed of input and output signals, constants, comparison and logic operators. The result of the verification in PLCverif is a human-readable verification report. It demonstrates the violation of properties (if any) by diagnostic traces from the model.

The defined PLC modules should often satisfy certain safety and invariant properties. Explicitly declaring and verifying these properties may greatly improve the quality of the PLC software modules. Therefore we have included specific support in PLCspecif to capture these invariant properties. In each module the specifier may define properties which have to be always satisfied by the given (sub)module, more precisely after the execution of the (sub)module the defined invariant properties have to be satisfied. As discussed above, the satisfaction of these requirements may be checked using model checkers, similarly to the approach followed in PLCverif. This is discussed in the rest of the section.

a) Representing the Requirement for Model Checking: For model checking, the only need is to be able to represent the invariant properties as temporal logic formulae. The usage of requirement patterns to hide the complexity of describing properties with temporal logic seems to be a convenient way, but in the future additional property specification methods can also be incorporated, provided that they can be represented as computation tree logic (CTL) or linear temporal logic (LTL) formulae. For example, the above-presented requirement pattern can be represented in CTL as "$\mathsf{AG}((EoC \wedge \alpha) \rightarrow \beta)$", where EoC is true at the end of the PLC cycle only.

b) Representing the Specification for Model Checking: We recall that the formal semantics of PLCspecif is given as a construction of an equivalent timed automaton. The choice of this semantics description was partially to facilitate the formal verification of the PLCspecif specifications. The intermediate model used in PLCverif is also an automata-based formalism with matching semantics. The timed automata elements (e.g. edges) can be systematically translated to the corresponding IM elements (e.g. transitions). The only exception is the clock of the timed automata, which do not have corresponding element in IM. The representation of time-related behaviour in the IM is further discussed in [18].

The semantics of the corresponding elements are defined

[3] http://cern.ch/unicos/

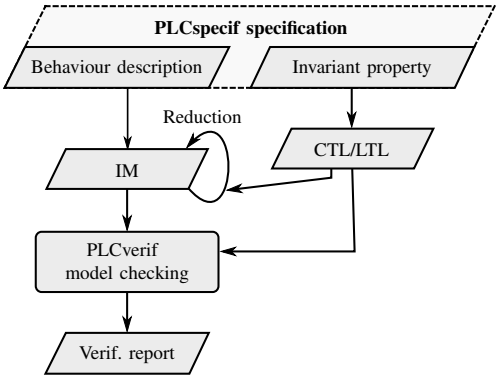

Fig. 1. Workflow of invariant checking

identically. Therefore by using the semantics description of PLCspecif it is feasible to develop a systematic, element-by-element mapping from a PLCspecif specification to a PLCverif IM. Then the IM can be transformed by PLCverif into the format required by the applied model checker tool. This method also benefits from the model reduction methods that are already included in PLCverif.

The complete invariant checking workflow based on PLCverif is depicted in Figure 1. The behaviour description of the specification is transformed into the IM formalism, then automatically reduced. The invariant properties are represented in CTL or LTL. The CTL/LTL formulae are not only used to check the satisfaction of the invariant property, but they also influence the reductions. PLCverif executes the selected external model checker tool, then the result is presented to the user.

The invariant checking was used in the re-specification of the previously mentioned reusable PLC module library of UNICOS. It was possible to define and verify invariant properties such as "If the *manual* mode is inhibited, the module should not switch to *manual* mode", directly on the intermediate model generated from the specification, before generating the corresponding source code.

VI. Summary

This paper discussed the static well-formedness analysis and invariant property checking features that are incorporated in the PLCspecif specification approach. The well-formedness checking methods help to ensure that the formal specification is consistent and well-formed; that it respects properties that are required for any formal specification. By using invariant checking it can also be checked whether the designed specification matches the intentions of the specifier. If the user declares safety or invariant properties explicitly in a PLCspecif specification, their satisfaction can be checked by reusing the model checking approach included in the PLCverif tool. This may reveal problems which would be hidden despite the use of formal specification. By checking these general and specific properties of the specification, the quality of the specification can be improved, which has a positive effect on the correctness of the implementation through the code generation and conformance checking methods.

References

[1] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, "Guide to industrial control systems (ICS) security," National Institute of Standards and Technology, Special Publication 800-82 rev. 2, 2015.

[2] *IEC 61131-3:2003 Programmable controllers – Part 3: Programming languages*, IEC Std., 2003.

[3] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Requirements towards a formal specification language for PLCs," in *Proc. of the 22nd PhD Mini-Symposium*. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2015, pp. 18–21.

[4] D. Darvas, E. Blanco Viñuela, and I. Majzik, "A formal specification method for PLC-based applications," in *Proc. of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, L. Corvetti *et al.*, Eds. JACoW, 2015, pp. 907–910.

[5] S. Stattelmann, S. Biallas, B. Schlich, and S. Kowalewski, "Applying static code analysis on industrial controller code," in *19th IEEE International Conference on Emerging Technology and Factory Automation (ETFA)*. IEEE, 2014.

[6] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger, "Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application," *IEEE Transactions on Industrial Informatics*, 2016, advance online publication, http://doi.org/10.1109/TII.2016.2604760.

[7] O. Ljungkrantz, K. Åkesson, M. Fabian, and C. Yuan, "A formal specification language for PLC-based control logic," in *8th IEEE International Conference on Industrial Informatics (INDIN)*, 2010, pp. 1067–1072.

[8] 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.

[9] G. Godena, T. Lukman, M. Heričko, and S. Strmčnik, "The experience of implementing model-driven engineering tools in the process control domain," *Information Technology and Control*, vol. 44, no. 2, 2015.

[10] S. R. Koo, P. H. Seong, J. Yoo, S. D. Cha, C. Youn, and H. Han, "NuSEE: An integrated environment of software specification and V&V for PLC based safety-critical systems," *Nuclear Engineering and Technology*, vol. 38, no. 3, pp. 259–276, 2006.

[11] D. Darvas, E. Blanco Viñuela, and I. Majzik, "Syntax and semantics of PLCspecif," CERN, Report EDMS 1523877, 2015. [Online]. Available: https://edms.cern.ch/document/1523877

[12] ——, "PLC code generation based on a formal specification language," in *14th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2016, pp. 389–396.

[13] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Conformance checking for programmable logic controller programs and specifications," in *11th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2016, pp. 29–36.

[14] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, vol. 4963, pp. 337–340.

[15] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Formal verification of safety PLC based control software," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, E. Ábrahám and M. Huisman, Eds. Springer, 2016, vol. 9681, pp. 508–522.

[16] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[17] D. Darvas, B. Fernández Adiego, and E. Blanco Viñuela, "PLCverif: A tool to verify PLC programs based on model checking techniques," in *Proc. of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, L. Corvetti *et al.*, Eds. JACoW, 2015, pp. 911–914.

[18] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, V. M. González Suárez, and J. O. Blech, "Modelling and formal verification of timing aspects in large PLC programs," in *Proc. of the 19th IFAC World Congress*, ser. IFAC Proceedings Volumes, E. Boje and X. Xia, Eds., vol. 47 (3). IFAC, 2014, pp. 3333–3339.