PLC Code Generation Based on a Formal Specification Language

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

***Abstract*—The complexity and quality needs of PLC-based control system software have largely increased. Formal specification methods can help to cope with these needs. Besides formal verification, another benefit of a formal specification language is the possibility to provide automatic generation of the final source code. This paper overviews PLCspecif, our formal specification language for PLC programs and presents a code generation method for the language. The result of the code generator is a Structured Text (ST) code that not only corresponds to the formal semantics of the specification, but is also configurable, readable, understandable, and follows development conventions and standards. The code generation method shows that PLCspecif is applicable and well-adapted to the PLC domain.**

I. Introduction

Programmable Logic Controllers (PLCs) are widely used in various industries for different process control tasks. PLCs are industrial, specialised computers, typically executing a user program in an infinite loop. In each cycle, the physical inputs are read into memory, then the user program is executed using these stable input values, finally the computed outputs are written to the physical outputs. The user program that defines the connections between the inputs and outputs can usually be written in the programming languages defined in the IEC 61131 standard, such as Structured Text (ST) that we use as target language.

PLC programs are often simple and short in many industries, not necessitating advanced specification methods. However, at the European Organization for Nuclear Research (CERN) highly complex and modular PLC programs are in use to control the subsystems of our particle accelerator complex (e.g. for cryogenics, cooling and ventilation, protection systems). The motivation of this work lies here: while the question of specification is not critical for simple PLC programs consisting of dozens of lines of code, a specification method to describe the behaviour of the software is essential in our case, where typical control programs can contain tens of thousands lines of code, reusing many generic components.

The special requirements, the programming languages used in the PLC domain, and the fact that PLC developers often have limited software engineering knowledge reduce the possibility to reuse methods and tools of the "mainstream IT". This is among the main reasons why *model-driven engineering* (MDE) is not widely used in the process control domain yet. Furthermore, as there are no widely-used specification methods, the specification of the control logic is typically informal, making MDE or formal verification difficult.

Previous work [1], [2], [3] proposed PLCspecif, a formal, yet intuitive and easy-to-understand specification language specially targeted to PLC programs. This paper extends this approach with automated code generation. One of the novelties of this work compared to other PLC code generation techniques is the use of a formal specification language tailored to the PLC domain. Being formal implies that the specification can be checked using static analysis and formal verification techniques. PLCspecif aims to be a lightweight formal specification method adapted to the PLC software development domain, in order to make it easy-to-use, even without deep knowledge in mathematics and formal methods. This implies an easier integration to the current development procedures.

The main contribution of this paper is a code generation based on a systematic mapping between the semantics of the specification and the PLC code, preserving the specified and verified behaviours. In addition, our code generation ensures the maintainability by providing configurability for the generation process and readability for the generated code.

The structure of the paper is the following. Section II presents the related work on formal specification and code generation applied to PLC programs. Next, Section III overviews the syntax and semantics of PLCspecif. Section IV is dedicated to the requirements, challenges and solutions of PLC code generation based on PLCspecif. Finally, Section V concludes the paper and describes the future work.

II. Related Work

1) Specification Methods: PLC-based control systems are widely used since the 1980s, but since then no widely applicable formal specification emerged. *Grafcet* [4] is one of the most widespread, standardised description methods. Its usage is restricted (it cannot express all kinds of behaviours), and there are multiple different formalizations for Grafcet. It motivated the Sequential Function Chart (SFC) programming language that is a syntactically similar, but semantically slightly different language. Recent work aimed to provide more domain-specific methods, such as *ProcGraph* by Lukman et al. [5]. While they try to build on the knowledge and habits of developers, the approach is too close to the code,

Authors' manuscript. Published in *14th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 389–396, IEEE, 2016. The final publication is available at IEEE Xplore. DOI: 10.1109/INDIN.2016.7819191.

not improving the understanding. Other works try to approach formal PLC specification by directly reusing general formal specification methods. Dierks proposed PLC-automata in [6], a "language to specify real-time systems", e.g. PLC programs. However, this approach looks too theoretical, making it impossible to be used by PLC program developers without special training. A more detailed analysis on PLC specification methods can be found in [2].

2) PLC Code Generation: Ours is not the first approach aiming to provide code generation for PLC programs. However, former approaches were not formal, did not successfully respect the requirements of the domain, or were based on too complex or not adapted specification methods. *Simulink PLC Coder*[1] seems to be one of the best and most mature available PLC code generator tools. Its input is a Simulink or Stateflow model, which implies its weaknesses: Stateflow is too error-prone for PLC programs because it has a rich semantics and it is not adapted to the PLC domain. Furthermore, PLC Coder cannot handle all features of the Stateflow models, making modelling difficult. Our first experiments showed that even developers experienced in modelling might have problems and doubts while using Stateflow to model PLC programs.

Others researchers, like Vogel-Heuser et al. [7], [8] tried to reuse already existing general-purpose modelling languages (such as UML or SysML) and to generate PLC code based on them, however the real applicability of these methods is doubtful: the basis of the code generation is not made public in detail, and the models used in those works seem not to facilitate the specification process enough, as they are relatively far from the target domain.

CIF 3 [9] is an academic tool to model hybrid systems that includes PLC code generation. While the idea and the available tool are promising, the applied textual modelling language looks too complex and mathematical. Its usage would necessitate an extensive training, making its application in our setting difficult.

III. Formal Specification Method for PLC Modules

This section is dedicated to an overview of PLCspecif. Section III-A introduces the main requirements and concepts of the specification method. Section III-B introduces a running example. Next, Section III-C–III-D introduces the syntax, Section III-E introduces the semantics of PLCspecif. The section ends by overviewing the verification possibilities (Section III-F). Detailed description of the syntax and formal semantics of PLCspecif can be found in our report [1].

A. Main Concepts

PLCspecif [3] is a formal specification method that responds to the specification needs of the PLC domain. The main target is the specification of reusable, composable PLC program components, as reusing necessitates deep understanding of the component's behaviour. Reusable blocks are widely used in the PLC domain. This is the case for most PLC-based control software at CERN, which are based on a framework called UNICOS [10]. It consists of a library of base PLC components (representing e.g. an actuator, a sensor or an alarm) and a methodology to instantiate, connect and develop the composition of these base components to build complete applications. Besides this, there is a need to construct new components or change their behaviour according to the new requirements.

Our goal was to design a specification method that is (1) formal, (2) powerful enough and well-adapted to describe PLC components (program units), (3) simple enough to be used in real life, and (4) conveniently implementable.

Requirement (1) is necessary to allow formal argumentation about the correctness of the specification (or the program generated based on the specification). This feature is missing from most of the applied specification methods. PLCspecif is formal, as it has a precise syntax and a semantics defined based on automata theory as described in [1].

To satisfy requirements (2) and (3), PLCspecif is designed based on the needs and experience of PLC developers at CERN. PLCspecif is not a universal formal specification method (such as B Method or Z), instead it specifically targets the PLC domain. In this way it was possible to define simple semantics and convenient representation for the common PLC constructs. We tried to reuse the formalisms known by the PLC developers (e.g. state machines, logic circuits) and to give a simple, PLC-adapted and consistent formal semantics to them. The simple semantics is essential for real usage without excessive training cost.

Many modelling languages suffer from implementability problems because of their expressivity or complex semantics, e.g. considering real concurrency or high-level declarative constructs. However, our method would be useless if it could not be implemented. We kept requirement (4) and the structure and behaviour of the target programs during the design of our method in mind and this paper demonstrates that a valid PLCspecif specification can be implemented. The reader finds a detailed analysis of the requirements for PLCspecif in [2].

B. Example

To illustrate the concepts, we have chosen a simple R-S flip-flop component as an example for this paper. This flip-flop has two reset inputs and a set input. If one of the reset inputs is true, the normal output (`Q`) of the component will become false. If the `Set` input is true, the output becomes true. The outputs keep their state if there is no reset or set input. The reset input has priority over the set input. Besides the normal output `Q`, its negated value should be also produced (`notQ`)[2] The PLCspecif specification of this component is presented in Fig. 2, which will be used as a running example in the rest of the paper.

[1] http://mathworks.com/products/sl-plc-coder/

[2] Defining both `notQ` and `Q` seems to be useless, but it illustrates the fact that PLC programs often define several similar outputs for convenience.

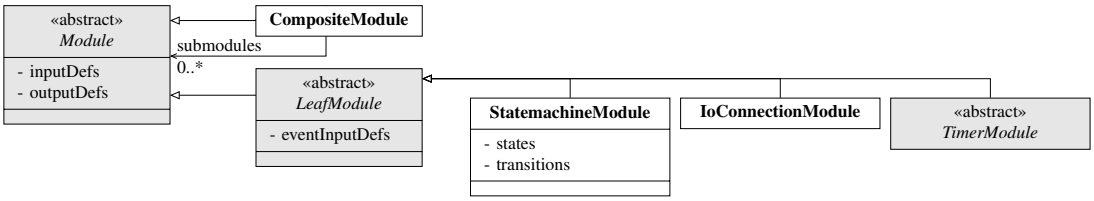

Fig. 1. Abstract syntax (metamodel) of PLCspecif module structure

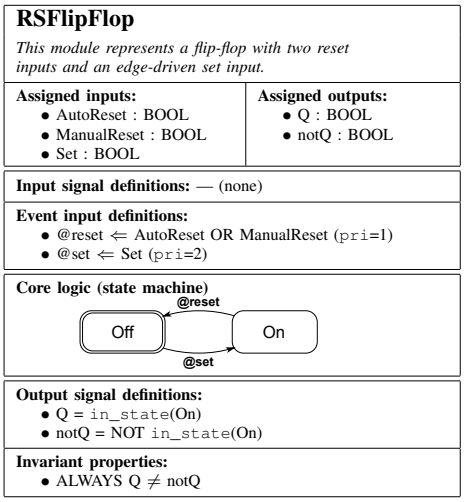

Fig. 2. R-S flip-flop module specification

C. Overview of the PLCspecif Abstract Syntax

The main building concept of PLCspecif is the *module*[3] that is a description of a Mealy machine. A PLCspecif specification is a containment hierarchy of modules: syntactically a module can be either a *composite* module (`CompositeModule`) consisting of an ordered list of other modules, or a *leaf* module (`LeafModule`) containing description of required behaviours. Different kinds of PLC behaviours (state-based, signal-oriented, or time-dependent) necessitate different approaches, therefore a leaf module can be described using three well-defined formalisms. A non-timed module can be described either a) as a state machine (`StatemachineModule`), or b) as an input-output connection description (`IoConnectionModule`, a representation of how output variables/signals obtain their values based on their previous values and the inputs). Time-dependent parts of the program are defined separately, in timer modules (`TimerModule`), semantically matching to the standard PLC timers, i.e. TON, TOFF, TP of IEC 61131. The module structure is summarised in Fig. 1.

Example. *As the example R-S flip-flop is a simple unit, it does not need a module hierarchy to be represented, it can be described by a single state machine-based leaf module.*

Internal Module Structure: The modules are further structured to keep the core logic simple and clean. A statechart could become really complex if it contains not only the core logic, but also the output handling in form of entry and exit actions. In this way it is forbidden to have variable assignments or entry and exit actions in the core logic of a state machine. All output handling is decoupled and defined separately based on the inputs and the state of the state machine, potentially defining intermediate signals for simpler definition. The input handling is similarly structured. This responds to the fact that a significant part of a PLC program is responsible for input/output treatment. Decoupling them results in simpler state machines and a proper separation of concerns. This way each PLCspecif module has at least three parts: 1) input signal definition, 2) core logic definition, and 3) output signal definition.

As mentioned before, there is no single specification language that can conveniently cover all kinds of PLC program units. This implied the need for multiple description methods in PLCspecif: the usage of state machines, input-output connection descriptions and special timer modules. The same principle applies to expressions: while a pure algebraic description (e.g. "$a \vee \neg b$") can be straightforward for simple logic expressions, this representation does not scale up well. Therefore PLCspecif includes the *AND/OR tables* of RSML [11] and another formalism called *switch-case tables* [1] for more complex expressions.

Event Inputs: PLCs typically have "request-like" inputs served by changing the internal state, and "status-like" inputs. Moreover, due to the mainly cyclic PLC behaviour, it is possible to receive multiple requests in one execution. In this case, the request conflicts should be resolved, demanding predefined priorities.

To represent requests, a special kind of input signal called *event input* can be defined in leaf modules. An event input is a special Boolean input that has a priority, unique to the defining module. Therefore at most one event input can be selected as active (i.e. having the highest priority) in each module. This simple solution helps to make the semantics of the state machines simpler (see in Section III-E).

Example. *The general module structure can be observed on the R-S flip-flop example in Fig. 2. After the header, one can see the three mentioned sections: (1) the input signal definitions, (2) the core logic definition, and (3) the output signal definition. Additionally, this module had two event input definitions:* `@reset` *and* `@set`*. The* `@reset` *event input has priority over* `@set`*. Each element of the specification (e.g. an input definition) can have a corresponding textual description to help the understanding of the specification. Here we omitted*

[3]Throughout this paper the term "module" refers to a PLCspecif module.

all these descriptions.

D. Abstract Syntax of the State Machines

As state machine modules are fundamental for the specification of control systems, we describe the abstract syntax of their core logic definition here. For a graphical representation of the metamodel, we refer the reader to [1, Sec. 2.2].

Essentially, a state machine module (`StatemachineModule` in Fig. 1) contains hierarchical states and transitions between states. A state (`State`) can be *basic* (`BasicState`) or *composite* (`CompositeState`). A composite state groups together some (composite or basic) states (which means an *OR* refinement as concurrency is not supported at this level), while a basic state is not further refined. Each transition (`Transition`) goes from a state to a basic state (which implies the activation of its parent states, see later). The transitions can have guards and associated trigger events.

Example. *The example in Fig. 2 contains two basic states:* `Off` *and* `On`*. The two transitions are both event-triggered.*

E. Semantics of PLCspecif

This part discusses the formal semantics of PLCspecif. Due to space limitation, the focus lies on the main concepts. The formal definitions can be found in [1].

1) Base Formalism: The formal semantics of PLCspecif is defined as an automaton. Automata have well-defined formal semantics, and can be easily extended by variables and variable assignments. The semantics of a PLCspecif specification defined as an automaton extended with variables can be mapped easily to the control flow graph of its implementation, helping to design a code generator that follows the formal semantics.

The rest of this section discusses the semantics of the main features of PLCspecif: the general representation of the composite and leaf modules, then the semantics of the state machines and timers.

2) Composite and General Leaf Module Representation: Each module follows the same structure, therefore their semantics (the corresponding automata) are also structured in the same way. Both the execution of leaf and composite modules start with the evaluation of the input signal definitions and (in case of leaf modules) event input definitions. Then, the module-specific part is executed. For a composite module, this is the sequential execution of each submodule in their pre-defined order. For a leaf module, the core logic can be described as a state machine, an input-output connection diagram or a PLC timer description. The execution of these parts is discussed later. After the module-specific part, the output values are set based on the output signal definitions.

These common parts (i.e. the input and output signal definitions) are unconditional variable assignments, therefore they are represented as one single path in the automaton, where the consecutive variable assignments are attached to consecutive transitions. The core logic between the input and output definitions is defined separately for each module type. Fig. 3 illustrates the structure of the constructed automaton.

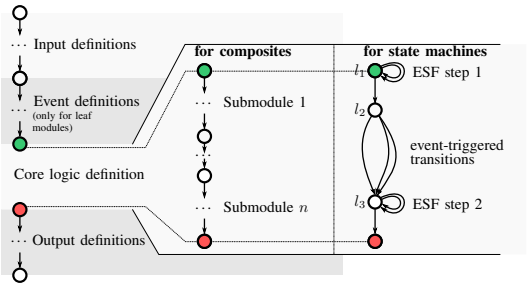

Fig. 3. Illustration for semantics definition based on automata extended with variables

3) State Machine Representation: The syntax of the state machine module is similar to any hierarchical state machine formalism, but the semantics is simplified and adapted to the PLC domain's needs. For instance, parallel regions are not allowed in the state machine, implying that there is always exactly one active basic state in the module. The active composite states are implied by the active basic state (all its parent composite states are active too), therefore the active state configuration can be represented by the active basic state only. This will be represented in the automaton by the variable `activeState` whose type is an enumeration of all *basic states* of the module. The default value of this variable is the initial basic state of the module.

When a transition *fires*, it leaves its `from` state and the new active state will be the transition's `to` state. To fire a transition, three requirements should be satisfied: (1) the active state should be its `from` state or one of its contained states, (2) its guard (`condition`) should be evaluated to true, and (3) the transition's *event input* (`trigger`) should be active (if there is a trigger defined). In addition, state machines can be extended by deep history states, similarly to UML State Machines.

The execution of the state machine (i.e. firing of transitions) has three phases: (1) an "exhaustive stabilisation firing step" (ESF step), (2) the firing of at most one event-triggered transition, and (3) a second ESF step. The goal of the ESF steps is to ensure a stable state (to leave transient states and reach a tangible state configuration in which the state machine waits for the next event input), while the event-triggered transitions are the reactions to the incoming requests. Note that in PLC specifications transient states having non-event-triggered transitions are typically used for error handling purposes.

In Phase 1, the ESF step exhaustively fires all enabled non-event-triggered transitions t_{NT}. A non-event-triggered transition is *enabled* if its source state is active and its guard is currently evaluated to true. Here we assume that the specification respects the well-formedness rules, including that only a finite number of transitions can fire in any ESF step (thus infinite firing sequences are not possible) and in each state at most one non-event-triggered transition can be enabled (nondeterminism is prohibited in the ESF step, but several non-event-triggered transition may fire consecutively).

These well-formedness rules are ensured by the static analysis of the specification. As firing a transition does not have any side-effect (transitions cannot raise events or change variable values), the former well-formedness rule means that there should not be any directed cycle of non-event-triggered transitions in the state machine where the conjunction of the transition guards can be evaluated to true. We check this requirement by translating it into a SAT problem and using a SAT solver (Z3 [12] in our proof-of-concept implementation). The ESF step is represented by a location l_1 in the automaton (see Fig. 3) and an $f = \langle l_1 \rightarrow l_1, g, v \rangle$ loop edge[4] for each non-event-triggered transition t_{NT} of the state machine. The guard of t_{NT} is represented by the automaton guard g of f, while the state change caused by t_{NT} (updating `activeState`) is denoted by the assignment v of edge f.

If there are no more enabled non-event-triggered transitions, Phase 1 is over. This is represented by an edge $l_1 \rightarrow l_2$ in the automaton. Then, in Phase 2, at most one event-triggered transition fires: the enabled transition that is triggered by the triggering event input. According to the well-formedness rules of PLCspecif, for each state of the state machine module, any event input e cannot have multiple outgoing transitions triggered to e where the guards can be evaluated to true at the same time. This is also checked using a SAT solver. For each event-triggered transition t_{ET} we have an edge $l_2 \rightarrow l_3$ with guard containing that (1) the source state of t_{ET} is active, (2) the guard of t_{ET} is true and (3) its triggering event input is active. According to the assumption above, this automaton representation does not have any nondeterminism. If none of the event-triggered transitions can fire, we proceed to the next phase without any state modification.

Finally, in Phase 3, a second ESF step finishes the execution of the state machine in l_3. This step can be skipped if there was no event-triggered transition firing. The principles discussed up to this point are illustrated in Fig. 3.

Example. *Both transitions in Fig. 2 are event-triggered, therefore in one execution at most one of them can fire. Although there is a cycle in the state machine, an infinite fire sequence is not possible. In this example, there is no ESF step. The automaton-based semantics of this example can be seen in Fig. 4. As there is no ESF step in this example, the self-loops like $l_1 \rightarrow l_1$ in Fig. 3 are not present. The labels and colours correspond to Fig. 3.*

4) PLC Timers: We recall that PLCs can have timed behaviours that are isolated in timer modules in PLCspecif (`TimerModule` in Fig. 1). As these modules follow the standard timer semantics defined in IEC 61131, we do not need an automata description for the semantics definition or for the implementation. However, the formal verification requires a consistent model describing both the non-timed and timed parts. The semantics of the PLCspecif timer modules are defined based on timed automata (TA) [13] in [1]. Accordingly, the semantics of state machine modules can also be represented by TA, but without any clock variables. We have defined these TA in a straightforward way that matches to the previously used automaton semantics. This way we can have a consistent semantics for verification, but we do not have to face additional complexity in the implementation or if no timers are used.

[4]To avoid the confusion, we will consistently use *transition* for transitions that are defined in PLCspecif state machines and *edge* for the state transitions in the constructed automaton.

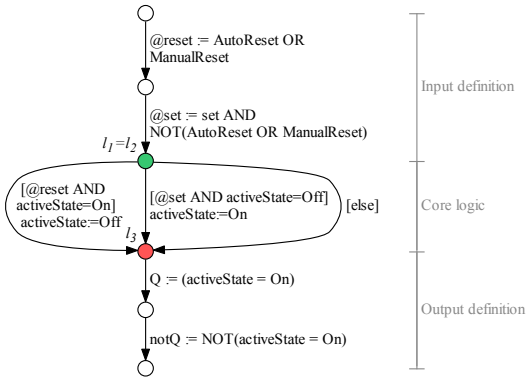

Fig. 4. Automaton (extended with variables) describing the semantics of module RSFlipFlop

F. Verification Possibilities

One of the main reasons for using a formal specification method is to allow to formally argue about the properties of the program. *General features* (that are independent of the application program) can be checked on the specification, such as deadlock and livelock freedom, absence of nondeterminism or absence of dead (unreachable) parts. *Program-dependent invariant properties* can also be defined in addition to the behaviour described by the specification, such as safety properties for outputs or other external requirements.

All these requirements can be (explicitly or implicitly) included in and checked on the specification. The well-formedness of the models can be checked using static analysis methods. If the requirements can be formalised by temporal logic, model checkers can analyse their validity on the automata. On the basis of the definition of the formal semantics, (timed) automata can be explicitly generated to represent the specified behaviour. Using a correct code generation method, the properties proven to be satisfied by the specification will be satisfied by the generated code as well. Other verification methods can be used as well, such as equivalence or conformance checking between the specification and a (model extracted from the) legacy implementation in a re-engineering use case. The discussion of these methods is out of scope, these were successfully applied at our department [14].

IV. Code Generation

In this section we overview the code generation method developed for PLCspecif. Obviously, the *primary requirement* towards a code generation method is to provide *correct code*, more precisely an implementation whose semantics corresponds to the semantics of the specification. In the process control domain, a secondary, but still important requirement

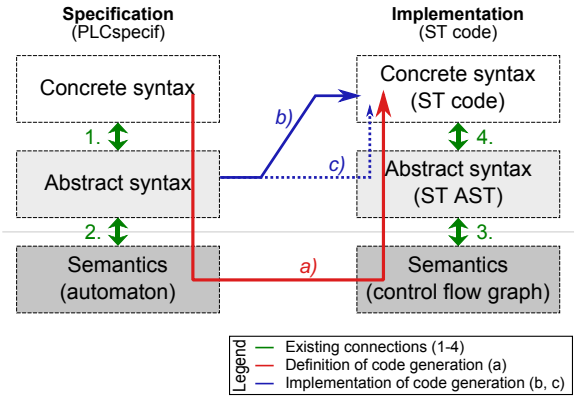

Fig. 5. Overview of the definition and implementation of code generation

is to generate an understandable code that can eventually be modified manually. In cases when the PLC program should be modified without stopping the PLC (e.g. urgent intervention), for technical reasons the whole application cannot be redeployed, therefore the expert developers should be able to read and manually modify the code.

Overview of the Solution: Both the source and target languages (in our case PLCspecif and ST code, respectively) have a concrete syntax, an abstract syntax and a semantics behind. The abstract syntax plays a central role, as typically the concrete syntax and the semantics of the language are both defined based on that. The *code generation problem* is to provide an implementation (i.e. source code) in concrete syntax for a given specification in concrete syntax having corresponding semantics. The link to be established is illustrated by the arrow *a)* in Fig. 5. However, the implementation of the code generation is typically a model-to-text (M2T) transformation from the abstract syntax of the specification or model to the concrete syntax of the implementation (arrow *b)* in Fig. 5). To ensure that the semantics of the generated code matches to the semantics of the specification, the transformation should be constructed based on the semantics definitions.

To provide the link *a)* of Fig. 5, the following steps are needed.

1) The concrete syntax of the specification has to be represented in its abstract syntax. This was briefly presented in Sections III-C and III-D. As it is not specific to the code generation, we omit its details in this paper.
2) The abstract syntax of the specification has to have well-defined semantics. In our case, the semantics of the specification is based on its translation into an automaton (extended with variables). Section III-E already discussed this step.
3) The semantics of the implementation should correspond to the semantics of the specification. The automaton-based semantics of the specification is systematically mapped to the control flow graph of the implementation, thus this is the basis of establishing the link. This step is described in Section IV-A.
4) The concrete and abstract syntax of the implementation should correspond to the control flow graph semantics. This can be achieved as presented in Section IV-B.

A. Semantics Based on Control Flow Graphs

1) Control Flow Graph: We consider a control flow graph (CFG) as a generic low-level representation of program code, providing semantics for the code. A CFG [15] is a directed graph, where each node represents a basic block (i.e. a linear sequence of program instructions having one entry point and one exit point); and each edge represents control flow paths with the corresponding conditions between blocks. One of the blocks represents the entry point of the program. An alternative view of CFG is the control flow automaton (CFA), where the instructions (practically: variable assignments) are assigned to the edges instead of nodes representing the blocks. We should simply move the instructions assigned to each node n to all of the incoming edges of n. The conditions are evaluated first, then the instructions are performed. The result of this transformation is a graph where nodes only serve as "junction points". This simple transformation emphasizes the similarities between automata and CFGs or CFAs.

2) Automaton to CFG/AST: An automaton where the transitions are labelled with guards and variable assignments is similar to a control flow graph as defined above. The typical constructs used in the automaton can be mapped to CFG structures. The generated code is based on the code representation of these CFG constructs.

- The linear paths of the automaton (without any junction, e.g. the input and output definitions) are basic blocks, they can be represented as sequential variable assignments.
- The loops used for the ESF steps of the state machines can be realised using loops in the CFG (corresponding to `while` loops in the abstract syntax tree (AST), where the exit condition of the `while` loop is the guard of the one single transition that leaves the location with the loop). As the guards of the different loop edges in the automata should be mutually exclusive, their order (the order of the corresponding state changes) can be arbitrary.
- The parallel transitions used to represent event-triggered transitions of the state machine module can be represented using one or more conditional branches. The conditions of the automata transitions represent checking the actual state of the state machine, the active event input and the guard of the corresponding event-triggered transitions. The variable assignment connected to an automaton transition modifies the active state of the state machine. As the guards of these transitions are mutually exclusive, it does not matter in which order they are checked in the implementation.
- Timed transitions and clocks are only permitted in the timer modules. As they are defined to match the semantics of standard PLC timers, they can be represented directly with their standard implementation, their TA representation is only used for verification purposes.

B. Generating the Concrete Implementation

In the previous sections we described the main concepts of the transformation to the implementation. Section IV-A presented which constructs of the concrete syntax could provide the required CFGs. However, some questions were left open and also some modifications are practical to be made.

1) Representing Enumerated Types: Previously we allowed enumerated types for variables in the automaton, e.g. `activeState` was defined with an enumerated type on all the basic states of the module. Some PLC manufacturers do not support enumerated types, in these cases these variables can be implemented by an integer, or by n integers, with an 1-out-of-n encoding.

2) Loop Unfolding: The ESF step is essentially a loop until no more non-event-triggered transition can be fired. While it is straightforward to implement this as a `while` loop, using loops is regarded as bad practice in PLC programs since it is difficult to predict at compilation time the execution time of a loop. The ESF steps are expected to be short, as they typically represent error-handling related behaviour (e.g. maintaining a "healthy state"). Therefore we unfold these ESF steps, explicitly representing each possible firing sequence. This also allows to demonstrate the absence of infinite loops.

3) Event-Triggered Transitions: To obtain a CFG that exactly matches the automaton semantics, the implementation of firing the (at most one) event-triggered transition between the two ESF steps would consist of one single `if-then-elsif-else` block. A big monolithic block like this can be difficult to understand. Instead, we split it into several conditional blocks based on the triggering events. As at most one event can be active in an execution, this does not modify the semantics of the implementation.

The code generated for the R-S flip-flop example (Fig. 4) can be found in Listing 1. The variables starting with `s_` represent the currently active state, the variables starting with `_E_` represent the currently active (at most one) event input.

```
FUNCTION_BLOCK RS_FF
   VAR_INPUT
      AutoReset : BOOL; // Reset request from logic
      ManualReset : BOOL; // Reset request from operator
      Set : BOOL; // Set request
   END_VAR
   VAR_OUTPUT
      Q : BOOL; // Normal (positive) output
      notQ : BOOL; // Negated output
   END_VAR
 // Events
   VAR
      _E_reset : BOOL; // Event Resets the RS flip-flop.
      _E_set : BOOL; // Event Sets the RS flip-flop.
   END_VAR
 // State variables
   VAR
      s_Off : BOOL := TRUE;
      s_On : BOOL;
   END_VAR

 // RS_FF (RSFlipFlop)
 // ==========================================
 // This module represents a flip-flop with two reset inputs and an edge-
      driven set input.
 // ------------------------------------
 // Events
   _E_reset := (AutoReset OR ManualReset); // Event reset (pri=1)
   _E_set := Set AND NOT _E_reset; // Event set (pri=2)

   IF _E_reset AND s_On THEN // Transition tReset
      s_On := FALSE; s_Off := TRUE;
   END_IF;
   IF _E_set AND s_Off THEN // Transition tSet
      s_Off := FALSE; s_On := TRUE;
   END_IF;

 // Outputs
   Q := s_On;
   notQ := NOT (s_On);
 // End of RS_FF
END_FUNCTION_BLOCK
```

Listing 1. Code generated from the R-S flip-flop example specification

C. Providing Readable Code

The code generation based on the principles discussed before can satisfy the primary (correctness) requirement. However, it did not take into account the requirement of providing readable code. To tackle this problem, this secondary requirement should be further refined. The code generated based on PLCspecif provides the following features to fulfil the secondary requirement.

- *Readability.* The code is correctly indented and uses the whitespaces consistently.
- *Understandability.* The code is easily understandable. It follows a general structure similar to the specifications' structure, it is formatted to support the meaning (e.g. expressions are well-formatted, an expression described by an AND/OR table in the specification follows a similar structure in the implementation too).
- *Configurability.* The developer is able to configure the way of implementation, where the used solution can be chosen from a set of possible ones (e.g. representing an enumeration using an `INT` or several `BOOL` variables).
- *Simplicity.* The code should be regular, but as simple as reasonably possible. For example, redundant variables should be avoided. Also, unnecessary constructs, such as always true expressions (tautologies) and unnecessary `if` statements should be avoided, trivial nested conditional statements should be flattened, etc. This can be achieved by different configuration options and by using simplification algorithms on the implementation (cf. Section IV-D).

  Example. *In case of Listing 1, it is unnecessary to have two separated variables for `s_On` and `Q`. In some cases, merging these variables could help the user to easier understand the code, and it also reduces the memory consumption. In other cases, this could have the opposite effect, causing difficulty in understanding. Therefore the user is able to configure these decisions too.*

D. Generation Process

Most of these requirements can be satisfied by having a M2T (model-to-text) transformation. The input of the transformation is the specification in abstract syntax, and the output is implementation in concrete syntax, in our case the ST code representation (represented by arrow *b)* in Fig. 5). This is the workflow we have chosen for our current, proof-of-concept code generator.

However, some advanced features might necessitate a more

complex workflow. For example, merging the variables with the same meaning or simplification of conditional statements can make the simple M2T code generator really complex, thus error-prone. Decoupling the simplifications from the core code generation features can solve this issue, but then a new, three-step workflow is necessary. First, the AST of the implementation is generated (model-to-model transformation). Then the simplifications can be made on this AST, and finally the AST is translated to the real implementation using a M2T transformation. This is represented by arrow *c)* in Fig. 5.

Decoupling the two transformations can also help to solve the portability problem of PLCs. Even if the IEC 61131-3 standard defines the syntax of the PLC programming languages, each manufacturer have slightly different variants, making it difficult to change the hardware supplier. By having a manufacturer-independent abstract syntax and a manufacturer-dependent M2T transformation to generate the program code, the problem of vendor-incompatibility could be reduced.

V. Summary and Future Work

1) Validation on a Real Example: The presented example is obviously too small to properly motivate a formal specification and a code generation method. We have conducted an experiment with a much bigger, real PLC component from the UNICOS framework. The example taken is a generic object, representing a valve, heater, or similar actuator providing control for two distinct positions. Currently, we are using an implementation developed around 10 years ago, and extended with new features during the last years. This implementation is significantly more complex than the R-S flip-flop example: the current implementation has about 120 individual input/output variables and 600 lines of code.

With the aim of creating a formal specification for this component, we were able to capture the key parts in PLCspecif. This leaded to an increased understanding, as for the specification its behaviour had to be understood in detail. On the basis of the work with and the review of the specification, we have found several problems in our specification. By studying them, we were able to create 10 new validation rules that can help future specifiers by warning them of potential problems. Also, we have found some bugs in the current implementation just by creating a specification based on the legacy code, its documentation and intended behaviour. The specification of this PLC component is much more complex than the example presented here, it contains 24 modules (leaf and composite), and 32 internal input/output definitions in addition to the module's input/output variables. 9 of 24 modules are state machine modules, containing in total 31 states and 41 transitions. The specification of this PLC component uses all features of the PLCspecif state machine module.

Finally, we generated an implementation for the specified parts of the module based on the specification and we tried it on a real hardware. The generated code contains 1100 lines of code and comments, and has about 120 internal variables, thus it is a significantly large component. The current tool is not optimised for the code size, the size of the generated code could be reduced using simplifications on the ST abstract syntax tree as described in Section IV-D.

2) Conclusion: This paper presented PLCspecif, a formal specification method for PLC programs, and specifically a PLC code generation method for PLCspecif. This code generation demonstrates that PLCspecif is adapted to the PLC domain and that the specifications made using PLCspecif are implementable. Also, the method takes the readability of the generated into consideration, that is required for real-life usability.

3) Future Work: A prototype tool was made for the evaluation, making it more robust and providing more convenient interfaces is a future goal. Also, expecting manual modifications of the generated implementation as a valid use case, later followed by the corresponding manual modifications on the specification necessitates a solution to check the conformance between the code and the specification. This may also help to put existing UNICOS components on formal basis by preparing their specifications and checking their conformance with the original implementation.

References

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

[2] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Requirements towards a formal specification language for PLCs," in *Proc. of the 22th PhD Mini-Symposium*. Budapest University of Technology and Economics, DMIS, 2015, pp. 18–21, doi: 10.5281/zenodo.14907.

[3] D. Darvas, E. Blanco Viñuela, and I. Majzik, "A formal specification method for PLC-based applications," in *Proc. of the 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems*. JACoW, 2015, pp. 907–910.

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

[5] T. Lukman *et al.*, "Model-driven engineering of process control software – beyond device-centric abstractions," *Control Engineering Practice*, vol. 21, no. 8, pp. 1078–1096, 2013.

[6] H. Dierks, "PLC-automata: a new class of implementable real-time automata," *Theoretical Computer Science*, vol. 253, no. 1, pp. 61–93, 2001.

[7] B. Vogel-Heuser, D. Witsch, and U. Katzke, "Automatic code generation from a UML model to IEC 61131-3 and system configuration tools," in *Proc. of the Int. Conf. on Control and Automation*. IEEE, 2005, pp. 1034–1039.

[8] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of manufacturing automation software projects – A SysML-based approach," *Mechatronics*, vol. 24, no. 7, pp. 883–897, 2014.

[9] D. van Beek *et al.*, "CIF 3: Model-based engineering of supervisory controllers," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2014, vol. 8413, pp. 575–580.

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

[11] 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 Sys. Conf.*, vol. 1, 1998, pp. C43/1–C43/8.

[12] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2008, vol. 4963, pp. 337–340.

[13] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, ser. LNCS. Springer, 2004, vol. 3098, pp. 87–124.

[14] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Formal verification of safety PLC based control software," in *Integrated Formal Methods*, ser. LNCS. Springer, 2016, vol. 9681.

[15] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.