# Bringing Automated Model Checking to PLC Program Development – A CERN Case Study –

**Borja Fernández Adiego** * **Dániel Darvas** *
**Jean-Charles Tournier** * **Enrique Blanco Viñuela** *
**Víctor M. González Suárez** **

* *CERN (European Organization for Nuclear Research), Geneva, Switzerland (e-mail: {borja.fernandez.adiego, daniel.darvas, jean-charles.tournier, enrique.blanco}@cern.ch)*
** *University of Oviedo, Gijón, Spain (e-mail: victor@isa.uniovi.es)*

**Abstract:** Verification of critical software is a high priority but a challenging task for industrial control systems. Model checking appears to be an appropriate approach for this purpose. However, this technique is not widely used in industry yet, due to some obstacles. The main obstacles encountered when trying to apply formal verification techniques at industrial installations are the difficulty of creating models out of PLC programs and defining formally the specification requirements. In addition, models produced out of real-life programs have a huge state space, thus preventing the verification due to performance issues. Our work at CERN (European Organization for Nuclear Research) focuses on developing efficient automatic verification methods for industrial critical installations based on PLC (Programmable Logic Controller) control systems.
In this paper, we present a tool generating automatically formal models out of PLC code. The tool implements a general methodology which can support several input languages, like the PLC programming languages defined in the IEC 61131 standard, as well as the model formalisms of different model checker tools. The tool supports the three main stages of model checking: system modelization, requirement formalization and counterexample analysis. In addition, a verification case study of a PLC program, written in Structured Text (ST) language implemented at CERN is described. The paper shows that the verification process is automatized and supported by the proposed tool, thus its difficulty is completely hidden for the control engineer.

*Keywords:* PLC, IEC61131-3, ST, formal verification, model checking

## 1. INTRODUCTION

Control systems based on PLCs (Programmable Logic Controllers) are widely used at CERN (European Organization for Nuclear Research), such as cryogenics, ventilation or vacuum systems. These systems are developed using the UNICOS (Unified Industrial Control System) framework (Viñuela et al. (2011)), which contains a library of objects written in ST (Structured Text) programming language to represent common industrial control instrumentations, such as sensors, actuators, etc. These are the base of all the industrial control systems at CERN.

The control systems are critical to CERN's operation, therefore guaranteeing that their behaviour conform to their requirement specifications is of highest importance. It implies a crucial need to verify the UNICOS objects, the building blocks of all these systems. Formal verification, and especially model checking is a promising technique to ensure that PLC programs meet their specifications. The advantages of this technique is that the verification itself does not need expertise, and the temporal logics, such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) used as specification language provides a rich formalism to describe the real-life requirements. However, these techniques are not wide-spread in this field due to some challenges. First, the system and the requirement to be checked should be both formalized in order to apply model checking. This formalization work is difficult and it requires expertise in the automation and the verification fields. Moreover, its correctness is crucial for the verification process. Therefore the formalization process should be automatized. Second, the formal models of real-life systems are usually complex and it can make model checking impossible due to resource constraints.

In this paper, we present a tool generating automatically formal models out of PLC code and show its applicability to a real-life case study, as well as to demonstrate its benefits. The automatic generation tool implements the methodology presented in Darvas et al. (2013). The tool is based on Xtext (Eysholdt and Behrens (2010)) and Eclipse Modeling Framework (Steinberg et al. (2009)) technologies. It allows to produce formal models for NuSMV (Cimatti et al. (2002)), UPPAAL (Amnell et al. (2001)) and the BIP framework (Basu et al. (2011)). The applicability of the tool is demonstrated through the formal verification of a critical object of the UNICOS library deployed in CERN installations. The case study shows that model checking can be included in the PLC development process without the need of making the automation engineers experts in the field of formal verification.

The rest of this paper is structured as follows. Section 2 overviews the related work. Section 3 briefly introduces our automated formal model generation methodology. Section 4 shows a complete verification case study, from the system and the requirement to be verified to the analysis of the results. Section 5 concludes the paper and shows the future work.

## 2. RELATED WORK

The application of formal methods for PLCs has been studied in previous work for many years (e.g., Frey and Litz (2000); Canet et al. (2000); Bauer et al. (2004); Sülflow and Drechsler (2008); Yoo et al. (2008); Pavlović and Ehrich (2010)), but either they do not address the modelling of real systems, or they do not apply verification, thus they do not have to apply any specific technique to handle the state space explosion problem.

Some authors addressed the problem of state space explosion by using different abstraction techniques (e.g., Gourcuff et al. (2008); Lange et al. (2013); Biallas et al. (2010)), but they have strong limitations on the system (e.g. only Boolean variables can be used in the PLC program) or limitations on the set of requirements that can be verified (e.g. only reachability properties can be checked).

In addition, only a few works targeting ST code verification can be found on the literature, as it is the most complex PLC programming language (e.g., Gourcuff et al. (2006); Sadolewski (2011b,a)). However, these approaches cannot be applied to our systems, as either they have limitations on the data type declarations or they have strong limitations on the specification language.

These works present interesting results, however either cannot be applied to a real-life PLC programs, as they do not address the state space explosion problem, or they are not addressing ST PLC programs. As it is the main PLC programming language used at CERN, it is unavoidable to be able to formalize them. In the literature, we have not found so far any work addressing the verification of real-world ST programs together with automated model generation. Finally, all the work found in literature target a specific verification tool, not providing a general approach.

## 3. METHODOLOGY

This section introduces the general methodology for verifying PLC programs defined in Darvas et al. (2013). The methodology produces formal models for different verification tools out of PLC programs implemented according to the IEC 61131 standard (IEC 61131). For that purpose, a set of transformation rules are defined to produce input models for e.g. NuSMV (Cimatti et al. (2002)), UPPAAL (Amnell et al. (2001)) and the BIP framework (Basu et al. (2011)). To make the approach general, we defined an intermediate model (IM) based on automata formalism, which facilitates the addition of new PLC programming languages or formal model formalisms by decoupling the inputs and the outputs of the workflow. This IM is also useful to apply various reductions and abstractions in order to make the formal model less complex. More details about the IM and the transformation rules can be found in Darvas et al. (2013).

The methodology is extended with a set of *specification patterns* to help the developer to formalize the requirements to be verified. Usually, the high-level requirements are ambiguous, thus they cannot be verified. LTL provides an unambiguous specification language, but it is too complex to be used by automation engineers. An intermediate solution can be the use of patterns, which provides an easy-to-understand formalism with the possibility of automatic formalization. This approach is similar to the idea proposed by Campos et al. (2008), but more complex patterns have been created supporting state changes, timed behaviour, etc., to cover the real-life needs.

Fig. 1 shows the complete methodology which consists of 6 consecutive steps:

(1) Requirements formalization.
(2) Transformation of the PLC program into the IM.
(3) IM reduction.
(4) Generation of formal models for the model checkers.
(5) Execution of the model checker.
(6) Analysis of the counterexample.

The first step of this methodology is the formalization of the requirements. The following 3 steps are designed to automatically generate the formal models. Once the model and requirements are formalized, model checking can be applied. Finally, if a given requirement does not hold on the model, a counterexample is produced by the model checker, which contains relevant information to identify the source of the problem.

## 4. CASE STUDY

This section describes a verification case study which was performed on a real PLC program with a real requirement by applying the proposed methodology. Each step of our methodology is introduced and measurements are given for the discussed example.

### *4.1 Description of the example case*

As a case study, we have chosen the *OnOff* object of the UNICOS library implemented for S7 Siemens PLCs. This object represents a physical equipment driven by digital signals, e.g. a valve, heater or motor. In the PLC, this object corresponds to a *function block* written in SCL (Structured Control Language), which is the equivalent of ST code in Siemens PLCs. The grammar of the SCL language is close to the standard ST language, with mostly syntactic differences. Some key metrics about this object are shown in Table 1. This object is representative of other UNICOS objects in terms of size and complexity.

The requirement chosen for this case study is connected to the so-called *Mode manager* part of the OnOff object. The OnOff object can operate in one of the following five modes: *Auto*, *Manual*, *Forced*, *Software Local Drive* and *Hardware Local Drive* modes. In Auto mode, the object is driven by the implemented logic. In Manual and Forced

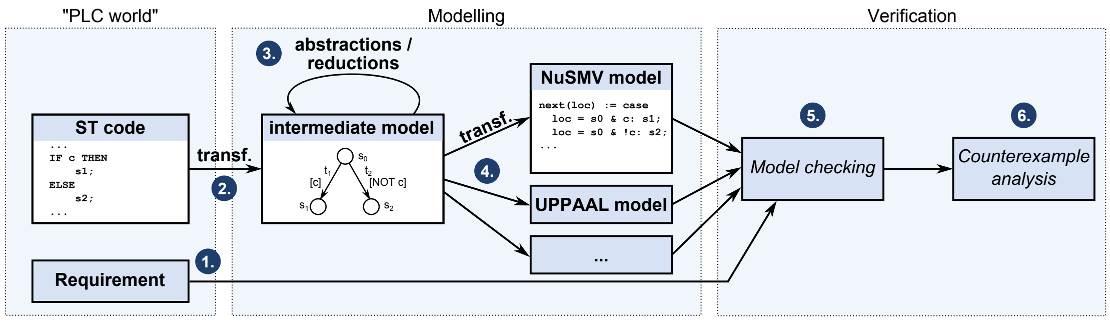

Fig. 1. Overview of the workflow

Table 1. Case Study metrics

| Metric | OnOff PLC code |
|---|---|
| Lines of code | 600 |
| Input variables | 60 |
| Output variables | 62 |
| Data types | Booleans, integers, floats, time, etc. |
| Timers | 3 |

mode, the object is driven by the operator. The other two modes are for special operations. The transitions between the operational modes are performed by "mode requests", coming from the implemented logic, the operator or a local device. To perform an operation mode change, a rising edge is necessary on the corresponding mode request input. One of these inputs is AuAuMoR, which stands for "Auto mode request initiated by the logic". Similarly, there is a MAuMoR input which stands for "Auto mode request initiated by the operator". Fig. 2 shows the state machine that specifies the behaviour of the mode manager for the three main modes. In the figure, "RE($x$)" stands for rising edge on input $x$.

One of the real requirements which can be derived from the specification of the object, shown in Fig. 2, is the following:

> "If there is a rising edge on the AuAuMoR (Auto mode request initiated by the logic) input, the mode will be changed to Auto."

This requirement is ambiguous, thus some refinement is needed in order to be able to be formalized and verified:

> "If the current mode is Auto, Manual or Forced mode and there is a rising edge on the AuAuMoR (Auto mode request initiated by the logic) input, but there are no other mode change requests, the mode will be Auto at the end of the cycle."

This is a real requirement and the corresponding PLC code was developed trying to satisfy this requirement. According to the specification, this requirement should be satisfied by the real implementation.

## 4.2 Formalization of requirements

Verifying UNICOS baseline objects involves complex requirements. For the model checking tools, they should be formalized as temporal logic expressions, such as LTL expressions.

As mentioned in Section 3, the formalization of the requirements is the first step of the methodology (see Fig. 1).

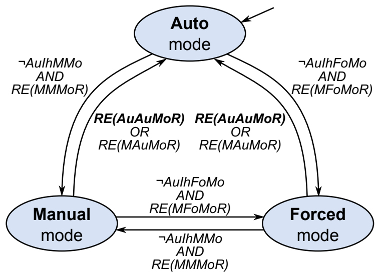

Fig. 2. State machine of the OnOff mode manager

Even if the given example requirement can be considered as unambiguous, it is not straightforward to formalize using temporal logic. The requirement formalization is a difficult task for the control system developers as usually they are not experienced in temporal logic. To help developers formalizing the specification requirements, a set of *patterns* is provided. For example, the pattern corresponding to the current requirement is the following:

> If there is a rising edge on $\mathcal{A}$ and $\mathcal{B}$ is true, then $\mathcal{C}$ is true.

However, this pattern is ambiguous, as there are multiple possible interpretations of "there is a rising edge". Therefore we reformalized it to have an unambiguous meaning:

> If in Cycle $N$: $\mathcal{A}$ is false and $\mathcal{B}$ is true, and
> in Cycle $N+1$: $\mathcal{A}$ is true and $\mathcal{B}$ is true,
> then $\mathcal{C}$ is true (in Cycle $N+1$).

It has to be noticed, that every variable value is checked at the end of PLC cycles, as the output variables are assigned to the real peripheries at this point.

The introduced requirement can be formalized using this pattern, by using the following values:

> $\mathcal{A}$: AuAuMoR
> $\mathcal{B}$: Mode = Auto or Manual or Forced AND there are no mode change requests other than AuAuMoR
> $\mathcal{C}$: Mode = Auto

This pattern is formal enough to be expressed as an LTL formula. One possible formalization is the following:

$$\mathsf{G}\Big(\big(EoC \wedge \neg\mathcal{A} \wedge \mathcal{B} \wedge \mathsf{X}(\neg EoC \;\mathsf{U}\; (EoC \wedge \mathcal{A} \wedge \mathcal{B}))\big) \\ \rightarrow \mathsf{X}(\neg EoC \;\mathsf{U}\; (EoC \wedge \mathcal{C}))\Big)$$

In the formalization, $EoC$ means "end of PLC cycle" and is true only at the end of PLC cycles. The meaning of

$\mathsf{X}(\neg EoC \; \mathsf{U} \; (EoC \wedge \mathcal{P}))$ is that the end of the PLC cycle is not reached until $\mathcal{P}$ is true. In other words: $\mathcal{P}$ is true at the end of the next cycle.

### *4.3 Model generation*

The application of model checking needs formal models of the system to be verified. In our case, the PLC programs written in ST language have to be transformed into a formal model which is supported by the model checking tools. This step is not the main contribution of this paper, the reader find more details about the model generation in Darvas et al. (2013).

For that purpose, a tool supporting the methodology presented in section 3 have been developed, which drastically reduces the modelization time and the potential human errors. The generation consists of three steps:

(1) Transformation from the PLC program written in ST to the IM (Step 2).
(2) Reduction techniques applied to the IM (Step 3).
(3) Generating inputs for the model checkers (Step 4).

*Transformation from ST code to IM* In the methodology, as it was introduced in the previous section, the PLC programs are first transformed into an intermediate model (IM). For this purpose, the input PLC code should be parsed. In our tool, the Xtext technology (Eysholdt and Behrens (2010)) is applied, which automatically generates an Eclipse-based editor, a parser and the necessary object model (which is essentially an abstract syntax tree) based on the given grammar adjusted to the needs of Xtext.

After parsing the code, i.e. the abstract syntax tree is built, a transformation creates an intermediate model representation, which is basically the control flow graph representation of the PLC program. These transformations are defined as a set of transformation rules. The intermediate model formalism is defined as an EMF (Eclipse Modeling Framework) metamodel (Steinberg et al. (2009)). The EMF technology makes easier to develop and maintain the metamodel, as well as facilitates to create correct instance models which reduces the possibility of faults in the transformation.

*Transformation from IM to specific model checkers* The intermediate model representation of the PLC program is formal and unambiguous, thus it can be used as an input for model checking. However, every model checker tool has a different and custom input formalism to describe the input models. Therefore before using the verification tools, converting the IM to the languages of the desired model checkers is needed (Step 4 on Fig. 1). This transformation is largely syntactical: if the model checker's input formalism is close to a description of an automaton or a state-transition system, the only main task of this transformation is to handle the syntactic differences.

For example, NuSMV uses a finite state machine as input language, which can represent our IM in a simple way. However, small challenges are arisen. For example, the ST language have a strong typing system, but 0 and FALSE are equivalent Boolean literals (i.e. `b:=0` and `b:=FALSE` have the same meaning). However, in NuSMV, the only possible values of the Boolean variables are TRUE and FALSE. Therefore before the semantic transformation is performed, it has to be determined for every element of every expressions what are the corresponding data types, i.e. if a "0" literal represents a Boolean or an integer value.

For this transformation, we used the Xtend technology (Xtend (2014)) which provides good utilities to develop textual model generator.

After the two transformations (Step 2 and 4), the formal model is ready to be checked with the verification tool. However the obtained model is often huge, for example the potential state space of the model of the OnOff object contains approximately $10^{218}$ states, making its verification impossible.

*Model reductions* The performance problem caused by huge models motivated an intermediate reduction step between Step 2 and 4. It is inevitable to apply reductions, simplifications to the model. Therefore a Step 3 was introduced in the methodology and reduction techniques are applied to the IM. Here we present the key ideas of the reduction step and we refer the reader for further details to Darvas et al. (2014).

Several reduction rules have been designed that can drastically reduce the size of the model. They can be considered as simple graph transformations. Furthermore, a cone of influence (COI) reduction (Clarke et al. (1999)) algorithm tailored to the PLC domain has been developed. This algorithm eliminates all the variables that are unnecessary for the evaluation of the current requirement. Comparing to the built-in COI included in NuSMV, the obtained performance is much better, because we can exploit the fact that IM is a higher level model and it contains more knowledge about the modelled system.

These reductions do not modify the meaning of the model with regards to the given requirement. In addition, the reductions are not just stand-alone rules, they can "help" each other. Therefore they are applied in an iterative way, until no further reduction can be achieved. By using these reductions, the potential state space of the model can be reduced dramatically, as discussed in the next paragraph.

*Evaluation of model generation* Table 2 shows key metrics on the size of the models and the generation times for the case study. Three different models are compared here: a "non-reduced model" without any model reductions ($M_1$), a "reduced model" which is still general, therefore a large set of different requirements can be verified on them ($M_2$), and a model tailored to the current example requirement which is only suitable for the evaluation of a reduced set of requirements ($M_3$).

As it can be seen in the Table 2, by using the proposed model reductions in the methodology, the size of the potential state space (PSS) of the OnOff object can be reduced from $10^{218}$ to less than $4 \cdot 10^{10}$, without affecting the result of the requirement to be checked.

The reductions are effective in terms of making the models smaller, but naturally they consume time. However, even with the most specific reductions, the run time of the model generation including the ST code parsing, the reductions, and the generation of the NuSMV inputs does not take more than 10–15 seconds, which is orders of magnitude lower than the achieved reduction in the verification time.

Table 2. Metrics of the models

| Metric | Non-reduced model ($M_1$) | Reduced model ($M_2$) | Specific model ($M_3$) |
|---|---|---|---|
| PSS | $1.61 \cdot 10^{218}$ | $4.57 \cdot 10^{36}$ | $3.65 \cdot 10^{10}$ |
| Variables | 255 | 118 | 33 |
| Generation | 0.3 s | 11.3 s | 12.6 s |

### *4.4 Model checking*

The next step is the verification of the formalized requirement on the formal model (Step 5 on Fig. 1). Currently, our tool supports the input formalisms of NuSMV, UPPAAL and the BIP framework. In terms of verification performance and requirements specification, NuSMV appears to be currently the most appropriate tool out of those three for our cases. Therefore we applied NuSMV for the current case study too.

NuSMV is a widely used open source model checker tool which supports a general input language and multiple, generic temporal logics as requirement specification formalism. It also provides various parameters to fine-tune the verification and the run time highly depends on the used parameters. Without any given parameters, NuSMV will explore the full state space before the evaluation of the requirement. However, if this is not necessary for the verification, it can be disabled by the *-df* parameter. The *-dcx* parameter disables the generation of the counterexample which makes the verification time smaller, but we cannot profit from the information given by the counterexample. The *-dynamic* parameter enables the dynamic reordering of the variables, which can drastically reduce the memory consumption of the tool.

Table 3 shows verification time results for the introduced requirement on the OnOff object. As it can be seen, the verification of the non-reduced model ($M_1$) was not successful. The requirement can be verified on the general reduced model, taking 160 s with counterexample generation. However, much smaller run times can be achieved by using the reduced model specific to the given requirement ($M_3$). In this case, the verification and the counterexample generation took less then 1 second together. (The symbol "—" indicates an execution longer than 24 hours.)

For the introduced example requirement, the result of the model checker is false, thus the requirement is not satisfied.

Table 3. Verification run time with NuSMV

| Metric | Non-reduced model ($M_1$) | Reduced model ($M_2$) | Specific model ($M_3$) |
|---|---|---|---|
| no parameters | — | — | 8.398 s |
| -dynamic | — | $\approx$ 7 h | 1.334 s |
| -dynamic -df | — | 160.8 s | 0.547 s |
| -dynamic -df -dcx | — | 3.787 s | 0.141 s |

### *4.5 Counterexample analysis*

One of the strengths of the model checking method is that if a requirement is not satisfied, a counterexample can be generated by the model checker. This counterexample can be analysed to find the source of the problem (Step 6 on Fig. 1).

```
-> State: 1.17 <-
  inst.HLD = FALSE
  inst.MANREG01[8] = TRUE
  inst.MANREG01[9] = FALSE
  inst.MANREG01[10] = FALSE
  inst.MANREG01[11] = FALSE
  inst.AUAUMOR = FALSE
  inst.AUIHMMO = FALSE
  inst.AUIHFOMO = FALSE
...
```

(a) Extract from the generated counterexample

```
onoff1.HLD := false;
onoff1.ManReg01 :=
      2#0000000000000001;
onoff1.AuAuMoR := false;
onoff1.AuIhMMo := false;
onoff1.AuIhFoMo := false;
...
CPC_FB_ONOFF.onoff1();
// Check:
b := onoff1.AuMoSt;
```

(b) Extract from the PLC demonstrator

Fig. 3. Examples for the counterexample and the PLC demonstrator

With appropriate settings, NuSMV generates detailed counterexamples, containing the value of each variable after each transition in the model (corresponding to the execution of an instruction in the PLC code). However, these counterexamples are usually too long. For the current requirement the detailed counterexample contains about 5000 lines for the specific model ($M_3$) and 37 500 lines for the generic model ($M_2$). These are too detailed for human analysis. However, they can be reduced, as it is enough to know the value of the input and output variables at the end of each PLC cycle, thus the variable valuations of the intermediate states can be removed. By removing the unnecessary states and the temporary variables, the counterexample of the current requirement can be reduced to approximately 100 lines.

The counterexample can also be used to demonstrate the problem in a real environment. Using the NuSMV's counterexample, a demonstrator PLC code can be generated easily by adding a module simulating the variables from the counterexample and checking the expected result. In this way, it can be proved that the counterexample is not caused by a mistake during the model generation or due to bad reductions. In this case study, the counterexample given by NuSMV was confirmed in a real PLC, showing that the model and the real system are equivalent for this specific requirement. Fig. 3 shows an extract from the generated counterexample and the corresponding part of the PLC demonstrator source code.

Furthermore, the PLC demonstrator can help us to reduce the counterexample. Usually, most of the input variables do not have any effect on the evaluated requirement. Those variables can be fixed to constant values which can help to focus on the input value changes that causes the violation of the requirement.

In the current example, by using the counterexample and the reduced PLC demonstrator we were able to identify the source of the problem. It turned out, that from Forced mode the logic cannot switch to Auto mode, it can only be done by the operator. The corresponding part of the implementation can be seen in Fig. 4[1]. The highlighted part shows the code implementing the transition from

[1] In the source code, the variables starting with "E_" indicate the rising edges of the corresponding inputs. AuMoSt_aux is true if

Forced mode to Auto mode, which does not take into account the AuAuMoR input. This behaviour is correct and intentional, in this case the source of the requirement, the specification contained a mistake.

```
IF (MMoSt_aux AND (E_MAuMoR OR E_AuAuMoR)) OR
   (FoMoSt_aux AND E_MAuMoR) OR
   (SoftLDSt_aux AND E_MAuMoR) OR
   (MMoSt_aux AND AuIhMMo) OR
   (FoMoSt_aux AND AuIhFoMo) OR
   NOT (AuMoSt_aux OR MMoSt_aux OR
       FoMoSt_aux OR SoftLDSt_aux) THEN
       (* Setting mode to Auto *)
       AuMoSt_aux := true;
       MMoSt_aux := false;
       FoMoSt_aux := false;
       SoftLDSt_aux := false;
END_IF;
```

Fig. 4. Extract of the source code causing the violation of the requirement

Many safety and liveness requirements have been verified using this methodology and the number of discrepancies between the specification and PLC program were significant even if testing was applied previously. In all these cases, 3 possible situations are identified as the source of the problem: a bug in the implementation, a mistake in the specification or an incomplete specification.

## 5. CONCLUSION AND FUTURE WORK

The paper presents a real-life case study to validate a general approach meant to apply formal verification to PLC programs. The paper discusses each step of the methodology and illustrates them with measurements.

We have shown that the methodology hides the difficulty of building the formal model and of formalizing the requirements from the control engineer, as it does not require any expertise in the field of formal methods from the engineer. This methodology is being applied extensively to all the 20 objects of the UNICOS library, each containing hundreds of specification requirements.

By applying this automated methodology to the UNICOS framework, many discrepancies have been detected between the specification and the real PLC code, even if the UNICOS library was tested extensively.

This case study has emphasized the importance of having a correct, unambiguous and complete specification. Future plans for this project are to invest a big effort on producing unambiguous requirement specification which will improve the quality of the PLC code.

## REFERENCES

Amnell, T. et al. (2001). UPPAAL – now, next, and future. In *Modeling and Verification of Parallel Processes.* Springer.

Basu, A. et al. (2011). Rigorous component-based system design using the BIP framework. *IEEE Sw.*, 28, 41–48.

Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., and Stursberg, O. (2004). Verification of PLC programs given as sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*, 517–540. Springer.

Biallas, S., Brauer, J., and Kowalewski, S. (2010). Counterexample-guided abstraction refinement for PLCs. In *Proc. of SSV'10.* USENIX A., Berkeley, USA.

Campos, J.C. et al. (2008). Property patterns for the formal verification of automated production systems. In *Proc. of 17th IFAC World Congress*, 5107–5112. IFAC.

Canet, G. et al. (2000). Towards the automatic verification of PLC programs written in Instruction List. In *Proc. of SMC 2000*, 2449–2454. Argos Press.

Cimatti, A. et al. (2002). NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *LNCS*, 359–364. Springer.

Clarke, E.M., Grumberg, O., and Peled, D.A. (1999). *Model Checking.* The MIT Press.

Darvas, D., Adiego, B.F., and Viñuela, E.B. (2013). Transforming PLC programs into formal models for verification purposes. Internal note, CERN. CERN-ACC-NOTE-2013-0040.

Darvas, D., Fernández Adiego, B., Vörös, A., Bartha, T., Blanco Viñuela, E., and González Suárez, V.M. (2014). Formal verification of complex properties on PLC programs. In *Proc. of FORTE 2014.* To appear.

Eysholdt, M. and Behrens, H. (2010). Xtext: Implement your language faster than the quick and dirty way. In *SPLASH/OOPSLA Companion*, 307–309. ACM.

Frey, G. and Litz, L. (2000). Formal methods in PLC programming. In *IEEE Int. Conf. on Systems, Man, and Cybernetics*, volume 4, 2431–2436. IEEE.

Gourcuff, V., de Smet, O., and Faure, J.M. (2006). Efficient representation for formal verification of PLC programs. In *Proc. of 8th WODES*, 182–187.

Gourcuff, V., de Smet, O., and Faure, J.M. (2008). Improving large-sized PLC programs verification using abstractions. In *Proc. of 17th IFAC World Congress.*

IEC 61131 (2013). IEC 61131: Programming languages for programmable logic controllers.

Lange, T., Neuhäußer, M., and Noll, T. (2013). Speeding up the safety verification of programmable logic controller code. In *Hardware and Software: Verification and Testing*, volume 8244 of *LNCS*, 44–60. Springer.

Pavlović, O. and Ehrich, H.D. (2010). Model checking PLC software written in function block diagram. In *International Conference on Software Testing*, 439–448.

Sadolewski, J. (2011a). Automated conversion of ST control programs to Why for verification purposes. In *Federated Conference on CSIS 2011*, 849–854.

Sadolewski, J. (2011b). Conversion of ST control programs to ANSI C for verification purposes. *e-Informatica*, 5(1), 65–76.

Steinberg, D., Budinsky, F., et al. (2009). *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional.

Sülflow, A. and Drechsler, R. (2008). Verification of PLC programs using formal proof techniques. In *FORMS/FORMAT 2008*, 43–50.

Viñuela, E.B. et al. (2011). UNICOS evolution: CPC version 6. In *Proc. of 12th ICALEPCS.*

Xtend (2014). `http://www.eclipse.org/xtend/`.

Yoo, J., Cha, S., and Jee, E. (2008). A verification framework for FBD based software in nuclear power plants. In *Proc. of APSEC'08*, 385–392. IEEE.

the current mode is Auto. The meaning of variables MMoSt_aux, FoMoSt_aux, and SoftLDSt_aux are similar, with the other modes.