A Classification of PLC Models and Applications

Angelika Mader[*]
University of Nijmegen
The Netherlands
mader@cs.kun.nl

Abstract

In the past years there is an increasing interest in analysing PLC applications with formal methods. The first step to this end is to get formal models of PLC applications. Meanwhile, various models for PLCs have already been introduced in the literature. In our paper we discuss several classification criteria that characterise different ways of modelling. The criteria include the PLC execution mechanism, the treatment of time and language fragments used. We try to motivate by examples which models are useful for which class of applications. Finally, we briefly reflect on a number of models from the literature according to the criteria discussed.

1 Motivation

Programmable Logic Controllers (PLCs) are applied to a wide range of control tasks, from industrial plants to toys, from milking machines to storm surge barriers, from elevators to household appliances. Many of their applications are safety critical, such as traffic control systems or chemical plants, or in an economical sense critical, where costs quickly get out of hand, e.g. for shutting down a factory, or replication of a bug in many thousands of copies of a car-controller.

In general, our scientific goal is to provide methods and tools that support effective verification of PLC applications. A first step here is finding suitable models for PLC programs and controlled environments that allow for formal reasoning.

Initially, PLCs were characterised by a very simple architecture and program execution mechanism. Nowadays, all kinds of "modern" features like multitasking and interrupts etc., can also be found in PLCs. Such features increase complexity of a formal treatment. Moreover, in most cases programs have to be written in one of the languages of the standard IEC 1131-3 [15]. From a theoretical point of view these languages suffer from ambiguities, ad-hoc features, and a lack of semantics (see also [20], [26], [6]). Under these conditions, when trying to find formal models it is crucial to choose the simplest execution mechanism and language fragment that are sufficient for the application in mind very carefully.

We try to classify PLC models following a few criteria that contrast PLC applications from usual IT applications: the execution mechanism, the use of timers, and programming

[*] Supported by a NWO postdoc grant and in the framework of the EC project VHS

language fragments. The usefulness of the criteria is shown by examples of a number of PLC applications.
Our goal is to obtain a classification of PLC applications. For each class of applications it is desirable to find the best model, meaning the model where most efficient tool support for formal analysis is available.
After an introduction to PLCs we discuss criteria that distinguish PLC models and give examples for classes of applications that can be modelled. Finally, we review some papers with respect to the criteria described.

2 Programmable Logic Controllers

The hardware of a PLC (see figure 1) consists of a CPU that is microprocessor based, a memory, input- and output-points where signals can be received, e.g. from switches and sensors, and sent to actuators, e.g. to motors or valves.

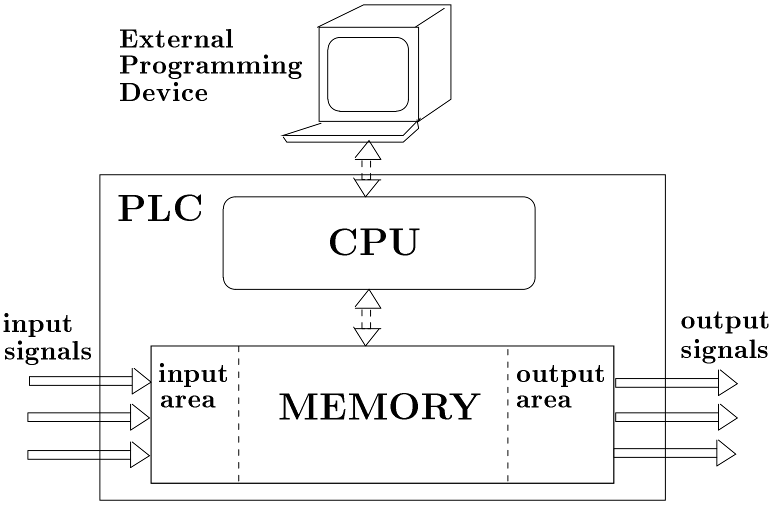

Figure 1: Architecture of Programmable Logic Controller

Usually, a PLC is equipped with an operating system that allows to load and run programs and perform self-checks. Programs for PLCs are developed and compiled on external devices.

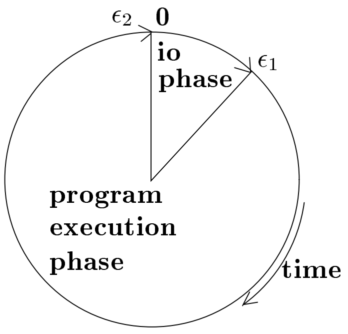

Figure 2: Execution Mode of a Programmable Logic Controller

The main difference to conventional systems is the operation mode illustrated in figure 2: PLC programs are executed in a permanent loop. In each iteration of the loop, commonly called a *scan cycle*, input is read, the program computes a new internal state and output, and the output is updated. There exists an upper time bound for each cycle, which typically is in the order of milli-seconds and depends on the size of the program. It is this cyclic behaviour that makes PLCs suitable for control tasks and interaction with a continuous environment. In this paper, when speaking about *one program execution* we subsume the initial i/o-phase, if not stated otherwise.

There exist different programming languages for PLCs. They have been designed with a main emphasis on applicability, each intended for a specific application domain, and based on the background of the control engineers who use them. One example is Ladder Diagrams, which are still being heavily used, and closely resemble the diagrams of traditional relay logic. From a computer science perspective some of these programming languages lag behind several generations of development in programming methodology.

In order to achieve more conformity of the different PLC programming languages the standard IEC 1131-3 [15] was developed. However, it provides no formal semantics, which makes precise reasoning impossible.

It is to be expected that in several application areas PLCs will be replaced by PCs in the future. However, the main principles such as a cyclic operation mode and the use of timers will be maintained. Therefore, considerations about PLCs in fact apply to a larger class of systems, which we call *polling real-time systems.*

3 Classification of PLC Models

We discuss three, orthogonal criteria:

1. to what extent and how the cyclic operation mode is modelled;
2. the use of timers;
3. programming language fragments.

We characterise the resulting classes and give examples.
Of course, there are more classification criteria for models and applications: e.g. the desired correctness property plays a role; if only safety, but no real-time aspects are relevant then there is no need for a real-time model.

3.1 Modelling the scan cycle

The first criterion is to what extent the scan cycles are modelled. In general, a program on a PLC is executed cyclically as described in section 2. However, when doing only static program analysis the cyclical execution is not of interest: models have to take only a single program execution into account. They are discussed in section 3.1.1. The most "realistic" way of modelling PLCs is to build the cyclical execution into the model equipped with some real-time information about the duration of each cycle. This possibility is discussed in 3.1.2 as *explicit scan cycles.* In some cases, the precise duration of a scan cycle is not of interest, but the cyclic behaviour is. In that case, the real-time information of 3.1.2 is not necessary. This will be discussed in section 3.1.3 as *implicit scan cycles.* Finally, when the environment is much slower than the scan cycle length we can use models where program execution is instantaneously. This is discussed in section 3.1.4 as *abstracting from scan cycles.*
In general, PLCs are embedded systems, and correctness of such systems concerns the interaction between a controller and its environment. Therefore, a composition of two models has to be analysed: one models the PLC with its program and one models the environment. The different ways of modelling scan cycles characterise the abstraction levelsaton which environment and PLC are composed.[1]

3.1.1 Models without scan cycle

Program models that do not consider the scan cycle are mainly intended for static program analysis. They are not composed together with an environment model. Typically, the models do not include real time. The properties that can be investigated based on such models are, e.g., data-independence between parallel components, safe structure of programs, unreachable code, etc. Moreover, it can be checked whether a program conforms to some programming guidelines. It is reasonable to restrict formal analysis only to well-constructed programs, i.e. not every syntacically correct program, but programs using a certain programming discipline.

3.1.2 Models with explicit scan cycle

Usually, for each program execution there are a lower and an upper time bound. The lower time bound results from the time needed for update of input and output and the self-checks of the operation system. The upper time bound depends on the longest execution path in the program. Under these assumptions it is important that we do not consider interrupts: they make it difficult to predict the duration of a scan cycle.
Modelling the scan cycle explicitly means that in the model each scan cycle is forced to take time that is between the lower and upper time bound. It is even more precise to assign a

[1] From another point of view, it also describes the underlying data-flow model, which may be discrete or continuous.

duration to each instruction of the program. Short execution paths are then modelled with short execution times and long execution paths with long times. The set of timed executions is restricted in comparison to the lower and upper bound approach. Clearly, all the models used here must have the possibility to express real-time properties.

In general, such models are very close to “reality”. They are useful, when precise timing behaviour is relevant. An example of this is asynchronous communication, where one PLC sends a signal that has to be read by another PLC. The sending PLC must take care that the signal is stable long enough to be read by the receiving PLC. For the sending PLC, the lower time bound is relevant, and for the receiving its upper time bound of the scan cycle length. Of course, two different PLCs have independent scan cycles.

Another possible use of such a model is a precise analysis of delays that are caused by the cyclic operation mode. For example, for a very simple program imitating a kitchen alarm, that ideally is started at some point (input rises) and should ring (output rises) after the specified alarm time. The analysis of the timing behaviour in such a model of a PLC showed that in the worst case there are 5 scan cycles in addition to the alarm time [19].

For some examples (as in the asynchronous communication example above) this may be relevant. For other examples, delays of this magnitude can easily be abstracted away. In any case, when abstracting from such delays, one has to give arguments why this is reasonable. The basis of such an argument must be the order of magnitude of time delays of the PLC compared to that of the timing requirements of the controlled environment.

In this sense a model with explicit scan cycle also can be useful as a reference point for less refined models.

3.1.3 Models with implicit scan cycle

By an implicit scan cycle we mean that the cyclical behaviour is modelled, but the duration of each scan cycle is not considered.

We consider the following examples:

1. For older PLCs it is the case that each program execution takes exactly the same time. There is a program memory for a fixed number of instructions that are all executed in each scan cycle. For shorter programs the remaining space is filled with skip-instructions.

2. In modern PLCs a similar behaviour can be found: an alternative program execution mode to the cyclical one as described above is a periodical execution mode. There, the program execution is started periodically with fixed time intervals (where the interval should be longer than one program execution).

3. For some applications only worst-case behaviour is relevant, in the sense that only maximal delays could do harm. The worst case assumption is then that each scan cycle takes maximal time. The analysis of the worst case then can be done on a model with fixed scan cycle time of maximal length.

4. A typical operation mode of PLCs is that there are several programs running in parallel on one PLC. In this case the execution mechanism of the parallel programs is synchronous, i.e. in each scan cycle each program is executed once. For the analysis

of the behaviour of the synchronous execution the length of each scan cycle is irrelevant. Certainly, synchronous model approaches are well suited to analyse this kind of applications.

For these examples the implicit-scan assumption can be applied only if the program execution is considered alone without combining it with an environment model. If the behaviour of PLC *and* environment has to be analysed, then the environment model has to contain an explicit clock tick at which it synchronises with the program. For examples (1)-(3) above this clock tick is in fixed time intervals, in case (4) the time between clock tick may possibly vary.

3.1.4 Models that abstract from scan cycles

By abstracting from scan cycles we mean that in a model /o-phase and program execution take place in zero time. With this assumption in fact *time-continuous controllers*[2] are modelled, as represented, e.g., by the "old" hard-wired controllers. PLCs are an approximation of time-continuous controllers.

Many applications are "slow" in comparison to the PLC cycle time; the environment cannot distinguish between an ideal, time-continuous controller and a PLC with its scan cycles. In these cases we can model the PLC by the ideal time-continuous controller that it should approximate.

The applications we consider are, e.g., in chemical industry. There, delays resulting from the cyclical program execution mode are not relevant: if a solution has reached the correct temperature, it does not make a real difference whether the heater is switched off 15 milliseconds earlier or later; the environment is very slow in comparison to the controlling PLC.

In such cases we can abstract from the scan cycle time, meaning that in a model a PLC scan cycle can be assumed to be executed instantaneously, i.e. with execution time zero.

Without further assumptions zero execution time would allow for Zeno-behaviour, i.e. infinitely many program executions within a fixed time interval. There are several possibilities for solving this problem:

1. Programs are executed in certain time intervals, possibly of variable length. The resulting models are then similar to the ones of sections 3.1.2 and 3.1.3. One difference appears in combination with timers: a timer can give a timeout between the start of a program execution and termination, which could give some behaviour that cannot be modelled with the zero execution time assumption.

2. Another possibility is to execute a program every time when the environment changes the input signals, or more precisely when a change in the input is observable for the PLC. In this case the non-Zeno-ness of the environment guarantees the non-Zeno-ness of program execution. For this approach timer constructions have to be considered carefully: also a (possible) timeout has to enforce a program execution. It is the case that not the full set of PLC programs can be modelled by this approach: programs

[2] We chose the name time-continuous controller to contrast with what control-theorists define as continuous controllers. We rather mean the opposite of what control-theorists understand by discrete controllers.

that can produce different outputs for two subsequent calls with same input cannot be modelled.

3. Examples from chemical plant control provide the following observation: after switching on some actuator (e.g. opening a valve) the program often simply waits until the sensor data tell that the environment reached a new state (e.g. tank is full). During the waiting time the program only checks that the conditions for continuation are not yet satisfied and therefore output remains unchanged. In such a case a program execution is only necessary if the environment changes in a way that the conditions for program continuation are satisfied.

3.2 Use of Timers

There are models that allow to model timers, many of the models do not. Basically, there are two reasons to use timers. They can be characterised as follows.

1. The duration of an *output signal* is controlled, i.e. an output signal has to be stable for a certain time. The easiest example here is a flashing signal indicating a critical process for human users.

2. A timer as a substitute for incomplete knowledge of the environment, such as could be provided by extra sensors. I.e., a timer substitutes an *input signal.* This is certainly the most important use of timers. E.g., in a chemical plant a valve is opened and the process continues after the subsequent tank is full; either a sensor can indicate that the tank is full (closed loop control) or experience tells that after a some time the tank is certainly full and the process just waits for that time before it continues (open loop control). In this sense a filling level sensor can be substituted by a timer. This solution is often chosen, when sensors are too expensive or not available. (Note that this observation is also applicable for other cases: also in protocols time-outs substitute the knowledge that e.g. the communication partner has not received a message.)

For both classes there are applications where the use of timers can be avoided. In the first case it might be sufficient to count scan cycles and multiply them by the lower cycle time bound [3]. In the second case the controlled environment may be fully equipped with sensors and timers are not needed [4].
It is obvious that models not treating timers can be useful for a big class of applications. Furthermore, the intended correctness properties are also relevant: many properties are time-independent.

3.3 Language Fragments

3.3.1 Booleans and Integers vs. Reals

In the PLC languages of the standard types as Booleans, Integers and Reals are available. For many control tasks Reals are not necessary: if some sensor data reach a threshold then

[3] Note that this collides with the approach of section 3.1.4.

[4] However, they can be useful for error detection, when comparing expected processing time and sensor data.

some actuators are set on or off. Moreover, real number computations often cost much time and there is a trend to remove costly real number computations from a PLC to a PC (that communicates with the PLC) in order to keep the scan cycle within predictable bounds. Therefore it is reasonable to investigate the class of programs, where the basic data-types are Booleans (and Integers).

3.3.2 Languages of the Standard

Here, we briefly comment on the languages of the standard [15]. For critical discussions we also refer to [20], [26], and [6].
PLCs are built by many different manufacturers. Many of them used to define their own programming languages. The intention of the standard was to bring conformity in the confusing variety of languages. The standard contains five different languages:

- Instruction List (IL), an assembly-like language,
- Structured Text (ST), which is similar to Pascal,
- Ladder Diagrams (LD), a graphical language for Relay Ladder Logic, that, in the first place, is a notation for Boolean expressions,
- Function Block Diagrams (FBD), a graphical language, representing the data-flow described,
- Sequential Function Charts (SFC), which is a meta-language that allows to put a Petri-Net related structure on top of one of the other languages. It is inspired by the French formalism Grafcet.

Beside providing conformity, a standard should also be accepted by a large number of manufacturers. This happens only, if the step between their traditional programming environments and the ones provided by the standard is not too big. Therefore, the standard has also a collecting function. A sensible restriction of language constructs and a sharp definition of semantics of the languages would simply throw several brands out of the game. As a consequence, these topics are not addressed by the standard on purpose. However, this creates bad conditions for a formal treatment of the languages. From the theoreticians' point of view the languages (to different extent) are overloaded with constructions that are not at all necessary for expressiveness. Instead, some of them introduce non-trivial semantic ambiguities. On the other hand, there is a precise, but un-implementable semantics. An example are the timers: there is a precise definition based on a continuous data-flow model. It *cannot* and *does not* translate to any implementation as specified, because every PLC only approximates a continuous control.
In this situation our conclusions are:

- It does not make sense to give semantics to a full language.
- When trying to formally analyse PLC programs then it is essential to carefully select the fragments of languages that are treated.

- It is useful to develop programming guidelines and a programming discipline, that restrict the class of programs that should be verified. Even in a defined language fragment it is not desirable to be able to verify every thinkable (and possibly unstructured) program.
- Programming methodologies are useful for program development that supports verification.

However, when considering only language fragments, this should be in the first place horizontal restriction. What is certainly necessary, is to integrate different languages into one formal approach: many PLC programs are (and have to be) written in more than one language. In a typical case, a program structure is given in SFC, the transition conditions are Boolean expressions written in ST, and the actions attached to each step may be in IL.

3.3.3 Feature Fragments

Initially, PLCs were characterised by a very easy and also stable operation mode, as described in section 2. Nowadays, all sorts of modern features also can be found in PLCs: multitasking, interrupts and event-handling, watchdogs, various execution modes and more. From the theoretical point of view, these features increase complexity in a way that makes formal analysis difficult. The natural consequence is to restrict the applications to those not making use of "complicated" features. E.g., a non-restricted number of interrupts can increase the scan-cycle length in a way that the cycle length gets unpredictable. In extreme cases they can lead to an alarm of a watchdog leading to other interrupts, which altogether makes the correct behaviour of the controller difficult to guarantee.

On the other hand, there may be cases,where interrupts are necessary: e.g. the unfolding of the air-bag in a car has to happen as soon as possible, without delay of even one scan cycle. In such a case interrupts are certainly sensible, and can be treated formally, if we allow only for a restricted number of interrupts within one scan cycle.

Therefore, in order to contribute to real applications also more complex features have to be investigated. Here, the same holds as for the programming languages: the features used should be well-chosen and restricted in their application. Furthermore, it would be useful to characterise the kind of restriction that still allows for formal treatment.

4 Existing Models

The intention of this section is to look closer at some PLC models published, comparing them with the criteria above, and see what tools are available to support verification. This survey does not in any sense pretend to be complete, but is hopefully useful as a guide to present approaches. The order of the list is insignificant. A different survey can be found in [23].

- In [19] a timed automaton semantics is given to Instruction List. The language fragment does not contain function and function block calls. Timers, i.e. of type TON, can be modelled. Variables are restricted to Booleans. The scan cycle is modelled explicitly, with lower and upper time bound.

Based on this work a tool was developed which is described in [29]. It translates IL programs to timed automata. The IL programs considered there also allow integers as variable types. The resulting timed automata are represented in the format as used by the model checker UPPAAL [4]. The environment also has to be modelled as timed automaton synchronising with the automaton modelling the PLC on input and output variables. For model checking UPPAAL is used.

The tool allows also for minimisation of the timed automata. The minimisation is done with help of the tool CAESAR/ALDEBARAN [1] on a LOTOS representation.

There are two execution mechanisms treated: cyclical execution with time intervals of variable length (as in 3.1.2) and instantaneous execution when input signals (or timer output) change (as in 3.1.4.2).

- In [7] the class of models used is condition/event-systems (including real-time). The programs that can be modelled are in a fragment of Instruction List without jumps. Data-types are restricted to Booleans. Timers of all three types can be modelled, under the assumption that timers are started only at the beginning of a program, resulting in a model similar to those of section 3.1.4.1. The scan cycle is modelled explicitly, but with constant cycle length. The environment is also modelled as a condition/event-system. The verification of the whole system can be done with help of the tool VERDICT [16]. VERDICT provides an interface to already available verification tools, such as KRONOS [22] and HyTech [14] and SMV [3].

  In [8] this work is extended by including also jumps and a representation of scan cycles that allow for variable cycle length. There is a duration assigned to each instruction and the duration of a scan results from the durations of all the instructions executed in this particular program execution.

- In [13] programs of a fragment of Instruction List are modelled as Petri Nets. The fragment includes the standard set of instructions without commands from libraries. Possible data structures are what is expressible within a 8-bit word. The cyclic execution mode of the program is modelled in the way in which the Petri nets representing the environment and the program are composed into one net. Real-time, however, is not represented in the model. The properties that can be verified are those expressible in CTL. Verification tools used are PEP [2] and SMV [3].

- Also [12] deals with Instruction List programs. The models are Net Condition/Event systems extended by time. There is an explicit scan cycle considered with lower and upper time bounds. The data structure available seems to be Booleans. Instruction execution is modeled by a sequence of transitions, which suggests that branching is not considered (which should not at all be a problem in this way of modelling). Furthermore, the only instructions mentioned in the paper are LD(N), AND, OR and ST. Timers, as e.g. TON, can be modelled. However, it is not completely clear how the timer model is related to a timer call in an IL program. The connection with an environment model should not be difficult, but is not discussed in the paper. There is a tool automatically translating IL programs to timed Net Condition/Event systems.

- In [18] Sequential Function Chart programs are modelled. The models are described as timed automata. They abstract from scan cycles, meaning that a change in the environment causes an instantaneous reaction of the program. The use of timers is restricted to a special case: with activation of each step an associated timer is started and transitions between steps may depend on these timers. The data types are restricted to Booleans. It is not discussed how integration with other PLC languages (as required with SCFs) should be performed. There exists a compiler from SFC programs to timed automata and the verification is done with KRONOS [22].

- The goal of [10] is the derivation of correct PLC programs. Specifications are written in a subset of Duration Calculus. They can be transformed to a graphical representation, that is called "PLC-automata" [9]. From PLC-automata programs in the language Structured Text can be automatically derived. Moreover, PLC-automata have an intuitive operational meaning and can be used for specification in place of Duration Calculus.

  Basically, PLC-automata are based on a Duration Calculus semantics. In [11] also a timed automaton semantics is given. Consequently, PLC-automata can be transformed to timed automata and be verified with e.g. KRONOS [22]. There is a tool available, Moby/PLC [24].

  With respect to our classification criteria the semantics of PLC-automata explicitly models scan cycles. The use of timers is restricted to the special case that certain input may be ignored for a specified time. Of course, real-time behaviour can be analysed. The variable types are Booleans.

- In [25] fragments of FB and SFC are reconstructed and based on an operational semantics. Time is included in the model. The semantics is general enough to deal with variable scan cycle length, but in this thesis it is mostly is assumed that scan cycles are of fixed time. The definition of the semantics provides also proof rules as worked out in [6]. Composition with an environment is not discussed.

- In [17] the authors model specifications and implementations in higher order logic (HOL). Function blocks are modelled as relations on streams. According to the framework, there are no restrictions on data types. Time is treated implicitly; in an example only the upper time bound for a scan cycle was considered. In the logical framework controller and environment specifications are simply composed by conjunction. Proofs are done with help of a theorem prover, the Isabelle/HOL system.

- In [5] programs of the language Ladder Diagrams are investigated. Variables may only be Booleans. The programs are modelled as set constraints. Only single program executions (one scan) are considered. The main correctness property that is checked is absence of races. A race is a kind of unwanted history: as an assumption here, the output of the program should only depend on input and status of timers and counters. Different outputs for same input at same status of timers and counters is the defined incorrect behaviour. As a consequence, the model does not include real-time. There is no model of the environment. Instead, input signals are chosen randomly for a finite number of times, which does not model the complete behaviour. The program analysis is done by a general constraint resolution engine.

- Also in [21] PLC programs in relay ladder logic (another name for Ladder Diagrams) are investigated. The model is a finite transition system. The Ladder Diagrams are straightforwardly translated to Boolean assignments. The assignments define the transitions between states that represent the values of the Boolean variables. Time is only treated implicitly. The environment is not taken into account. Model checking is done with help of the tool SMV [3].
- In [28] two case studies are presented for chemical processes. The PLC is programmed with Ladder Diagrams. The programs are modelled as in [21]. Additionally, also the environment is modelled as finite state machine. Time is not treated (i.e. no real-time). The composition of environment model and controller model is in an alternating way: when the controller has reached a stable state (i.e. the control variables do not change any more), the environment may take a step, which may have consequences on the control variables etc. Model checking is done with the tool SMV [3].

5 Conclusion

Our interest is analysis of PLC applications with formal methods. Doing this requires formal models for PLC applications. There are already several ways of modelling introduced in the literature. Trying to understand the differences between these models, and doing case studies in verification of PLC applications lead to a number of classification criteria for models.

One goal of the classification is to find the most suitable model for a given application. To be more general, this can also mean different models for different levels of abstraction of *one* application. An example here is some relatively “slow” plant that does not require precise timing analysis in the range of milliseconds for the task it has to do. However, in the plant there may be a sensor that gives signals in intervals of a few milliseconds, as e.g. a rotation sensor that gives 16 different signals for one full rotation and that has to measure about 5 rotations per second. In this case there are already 80 signals per second which results in approximately 12ms per signal. The analysis of whether the sensor data are received and processed by the program in the intended, correct way, requires precise timing analysis. On a more abstract level the sensors then can be assumed to be interpreted by the program in the correct way.

In general, when trying to verify a PLC application it may turn out that different models have to be considered, possibly depending on the nature of the application, the properties to show, or on the size of the state space of the model. Here, we tried to provide criteria that give arguments for the choice of the suitable models.

Acknowledgement: for critical questions and discussions I want to thank Hanno Wupper, Ed Brinksma, Eric Rutten, the colleagues of the VHS project, and Heinz Treseler for his notes [27]. Judy Romijn and Hans Meijer gave useful comments that helped to improve the paper.

References

[1] *CADP home page.* see: http://www.inrialpes.fr/vasy/cadp.html.

[2] *PEP Homepage.* URL: http://theoretica.Informatik.Uni-Oldenburg.DE/ pep/.

[3] *The SMV System.* URL: http://almond.srv.cs.cmu.edu/project/modck/pub/www/smv.html.

[4] *UPPAAL home page.* see: http://www.docs.uu.se/docs/rtmv/uppaal/.

[5] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in relay ladder programs. In *Proceddings of TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 1998.

[6] S. Anderson and K. Tourlas. Design for proof: An approach to the design of domain-specific languages. volume 10, pages 452–468, 1998.

[7] N. Bauer. Übersetzung von Steuerungsprogrammen in formale Modelle. Master's thesis, University of Dortmund, 1998.

[8] N. Bauer, S. Kowalewski, and H. Treseler. Model checking of control software under consideration of the plc behaviour. Submitted to the 5th Workshop on Discrete Event Systems.

[9] H. Dierks. PLC-automata: A new class of implementable real-time automata. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development (ARTS'97)*, volume 1231 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997.

[10] H. Dierks. Synthesising controllers from real-time specifications. In *Proceedings of Tenth International Symposium on System Synthesis*, pages 126–133. IEEE CS Press, 1997.

[11] H. Dierks, A. Fehnker, A. Mader, and F. Vaandrager. Operational and logical semantics for polling real-time systems. Technical report, University of Nijmegen, 1998.

[12] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behaviour by means of timed net condition/event systems. In *Proc. of IEEE Int. Symposium on Emerging Technologies and Factory Automation (EFTA '97)*, pages 361–369, 1997.

[13] M. Heiner and T. Menzel. A Petri net semantics for the PLC language Instruction List. In *Proceedings of the International Workshop on Discrete Event Systems (WoDES)*, pages 161–166. IEE Control, 1998.

[14] T. A. Henzinger, P. S. Ho, and H. Wong-Toi. *A User Guide to HyTech.* UC Berkley, USA, 1996. available at: http://www-cad.EECS.Berkley.EDU/ tah/HyTech.

[15] International Electrotechnical Commission. *IEC International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages*, 1993.

[16] S. Kowalewski, N. Bauer, J. Preußig, O. Stursberg, and H. Treseler. An environment for model-checking of logic control systems with hybrid dynamics. In *Proc. IEEE Int. Symp. On Computer Aided Control System Design*, 1999.

[17] B. J. Krämer and N. Völker. A highly dependable computer architecture for safety-critical control applications. *Real-Time Systems Journal*, 13(3):237–251, 1997.

[18] D. L'Her, P. Le Parc, and L. Marcé. Proving sequential function chart programs using automata. In *Proceedings of 2nd AMAST workshop on Real-Time Systems*, 1995.

[19] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proceedings of the 11th Euromicro Conference on Real Time Systems*, pages 114–122. IEEE Computer Society, 1999.

[20] O. Maler. On the programming of industrial computers. VHS deliverable in Workpackage IP.1, http://www-verimag.imag.fr/VHS/IP/iec1131.ps, May 1999.

[21] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[22] A. Olivero and S. Yovine. *KRONOS: A tool for Verifying Real-Time Systems. Userguide.* VERIMAG, Grenoble, France, 1993. available at: http://www.imag.fr/VERIMAG/TEMPORISE/kronos/.

[23] O. ROSSI, J. J. LESAGE, and J. M. ROUSSEL. Formal validation of plc programs: A survey. In *Proceedings of European Control Conference 1999 (ECC'99)*, 1999.

[24] J. Tapken. Moby/PLC - A Design Tool for Hierarchical Real-Time Automata. In *Proceedings of FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 326–329. Springer Verlag, 1998. available at: http://theoretica.Informatik.Uni-Oldenburg.DE/ moby/.

[25] K. Tourlas. Semantic analysis and design of languages for programmable logic controllers. Master's thesis, Department of Computer Science, The University of Edinburgh, 1996.

[26] K. Tourlas. An assessement of the IEC 1131-3 standard on languages for programmable controllers. In Peter Daniel, editor, *SafeComp'97*, pages 210–219, 1997.

[27] H. Treseler. Ein überblick über alternative Ansätze zur formalen Modellierung der Steuerungssoftware. 1999.

[28] A. L. Turk, S. T. Probst, and G. J. Powers. Verification of real time chemical processing systems. In O. Maler, editor, *Proc. International Workshop on Hybrid and Real-Time Systems (HAR'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 1997.

[29] H.X. Willems. Compact timed automata for PLC programs. Technical Report CSI-R9925:, University of Nijmegen, Computing Science Institute, 1999. http://www.cs.kun.nl/csi/reports/info/CSI-R9925.html.