

## PLCTOOLS: Design, Formal Validation, and Code Generation for Programmable Controllers\*

Luciano Baresi<sup>+</sup>, Marco Mauri<sup>\*</sup>, Antonello Monti<sup>\*</sup>, and Mauro Pezzè<sup>+</sup>

(<sup>+</sup>) *Dipartimento di Elettronica e Informazione - Politecnico di Milano*  
Piazza Leonardo da Vinci, 32 - 20133 Milano (Italy)  
+39 02 2399 3400 - baresi|pezze@elet.polimi.it  
(<sup>\*</sup>) *Dipartimento di Elettrotecnica - Politecnico di Milano*  
Piazza Leonardo da Vinci, 32 - 20133 Milano (Italy)  
+39 02 2399 3702 - mauri|monti@etec.polimi.it

### Abstract

*Strong timing requirements and complex interactions with controlled elements complicate the design and validation of software controllers. Different techniques have been proposed to cope with these problems during the different development steps: for example, differential equations for modeling controlled elements, the IEC 1131-3 notations for designing the software controller, and formal models for validating the design, but no definitive solutions have been proposed yet.*

*This paper describes PLCTOOLS, a toolbox that exploits all aforementioned techniques to supply an integrated environment for the design, formal validation, and automatic code generation of software controllers.*

### Keywords

*Programmable Controllers, IEC 1131-3, Petri Nets*

### 1. Introduction

The design of software controllers is often complicated by strong timing requirements and complex interactions between the software controller and controlled elements (hereafter, the plant). The different nature of the components involved in the design can hardly be addressed with a single technique. Most approaches rely on different modeling and development tools for the plant and its software controller. Modeling techniques for the plant have been consolidated through the years and rely on well-known mathematical models. In contrast, the problem of modeling and designing programmable controllers can be approached with different techniques, characterized by complementary advantages and limits: Informal approaches better match domain expertise, but can be analyzed only partially, while

formal methods may be more difficult to adapt to the problem domain, but can be better analyzed. Although the problem of integrating different approaches for modeling, designing, and analyzing the plant and software controller have been widely studied ([9, 4]), no conclusive solutions have been proposed yet.

This paper presents PLCTOOLS<sup>1</sup>, a toolbox developed as part of the Esprit INFORMA Project, that provides control engineers with an integrated environment for designing and analyzing control systems: It integrates differential equations for modeling the plant, IEC 1131-3 FBD (Function Block Diagram, [7]) for designing the software controller, and HLTPN (High Level Timed Petri Nets, [5]) for validating the design and generating the code. The paper illustrates how to address the intrinsic diversity of the different components within a homogeneous toolbox that merges the main advantages of the different approaches overcoming most limitations. The paper presents also the main elements of the underlying methodology, discusses the key structure of the toolbox, and indicates the main experiments conducted so far with the prototype.

### 2. Underlying Approach

PLCTOOLS releases formal validation and automatic code generation through a hidden formal engine. It exploits the formal framework presented in [1, 2] to map FBD models onto functionally equivalent HLTPNs, to represent net executions in terms of visualizations of involved FBD blocks, and to automatically generate code. The formal framework allows for rule-based transformations of operational graphical notations. Control engineers can work with their domain notations (FBD, in this case), but at the same time and transparently they can gain benefits from the formal engine.

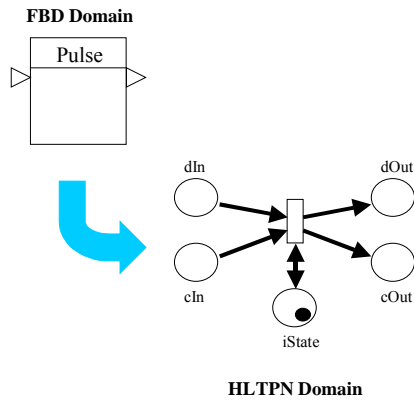
---

\* This work has been partially supported by the ESPRIT INFORMA Project (EP23163.)

---

<sup>1</sup> PLCTOOLS can freely be downloaded from the PLCTOOLS web site: <http://www.etec.polimi.it/PLCTools>

Rules - specified as graph grammar productions ([6]) - define how FBD elements must be rendered with equivalent HLTPNs. For example, Figure 1 presents a HLTPN that formalizes the behavior of *Pulse* blocks. A pulse block has one input and one output port. The output evaluates to *true* when the input switches from *false* to *true*. After the first execution, if the input remains *true*, the output becomes *false*. We need a further input switch from *false* to *true* to make the output be *true* again. The HLTPN of Figure 1 comprises five places and one transition. Places **dIn**, **dOut**, and **iState** contain boolean tokens. Places **cIn** and **cOut** contain control tokens. Control places are auxiliary places that are used to enforce the execution flow among FBD blocks.



#### Values:

`tmp = FALSE`

#### Transitions:

##### *pulse*

*predicate:* TRUE

*action:* `dOut = dIn && !iState;`  
`iState = dIn;`

*tMin:* `enab + t;`

*tMax:* `enab + t;`

**Figure 1: A HLTPN semantics for *Pulse* blocks**

Transition **pulse** is enabled when there is at least one token in each place of its preset since the associated predicate is *true*. This means that the transition can fire (the FBD block can execute) when there is a datum available (place **dIn**) and the execution flow enables the transition (place **cIn**). The firing makes the action associated with **pulse** execute: It produces a token in **dOut**, whose value depends on **dIn** and on its previous value stored in **iState**, copies the value from the token of place **dIn** in the token in place **iState**, and produces a token in **dOut**. The firing of transition **pulse** makes also the simulated time increase of *t* time

units. This comes from the values of **tMin** and **tMax**: They both add *t* time units to the current value (**enab**) of simulated time. If **tMin** and **tMax** are equal, the FBD block is associated with a fixed execution time; if **tMin** < **tMax**, the execution consumes a random amount of time *t<sub>e</sub>* in the declared interval (**tMin** ≤ *t<sub>e</sub>* ≤ **tMax**.)

Details on how the transformation is carried out are out of the scope of this paper, but interested readers can refer to [1] for an in-depth presentation. The rule-based translation does not impose specific semantics to FBD, but lets control experts formalize the behavior they prefer. Different CPUs, or different operating systems, could require peculiarities that a rule-based approach is able to reflect in the derived HLTPN, but that a fix hard-coded translation process could not accomplish. The same FBD block can be rendered with different HLTPNs. In particular, transitions are associated with both actions and execution delays that can be modified to model different computations and timing behaviors. Rules can produce both generic and special-purpose HLTPNs. In the former case, they would just allow experts to understand how the control behaves. In the latter case, they would be specific to particular CPUs or operating systems and would make the simulated behavior closer to the real one.

The choice among all different alternatives comes from balancing complexity and precision. It goes without saying that the more accurate the HLTPN is, the more complex it is, but the more precise analysis and simulation can be.

The Petri nets generated from FBD models communicate with the plant through special-purpose places: Input values appear in the net as tokens in input places; output values are produced by serializing tokens from output places.

The HLTPN used for simulation is the starting point for generating code. PLCTOOLS generates standard ANSI C code. Code generation exploits both the FBD model and the HLTPN that correspond to a control. The FBD model gives the overall frame of the code and the HLTPN supplies computational aspects. Code generation works on single tasks: It translates hierarchical FBD blocks into C function declarations and uses the subnets associated with leaf blocks to define functions' bodies. Generated code depends on the HLTPN we start from, thus the same FBD control can be transformed in different C code depending on the rules used to transform it. Timing properties are not used during the code generation process; they define requirements on generated code and can be used for an a-posteriori evaluation of derived code.

### 3. PLCTOOLS

PLCTOOLS is a complete environment for designing and validating control software. The main components of PLCTOOLS are presented in Figure 2:

MATLAB/SIMULINK provides suitable means for specifying and simulating the plant.

The *Editor* is developed in MATLAB 5.3 and comprises a *Resource Editor* and an *FBD Editor*. Control software is specified hierarchically: The *Resource Editor* is used to structure the control software in terms of resources (i.e., CPUs) and interconnections among them. The *FBD Editor* is used to specify the “behavior” of each CPU in terms of tasks and FBD blocks.

Every resource hosts a set of tasks and assumes a pre-emptive real-time operating system. PLCTOOLS supports two types of tasks: *periodic tasks*, which are characterized by an offset, an execution period, and a priority level, and *triggered tasks*, which are characterized by a priority level and a triggering event. Moreover, each task has a role.

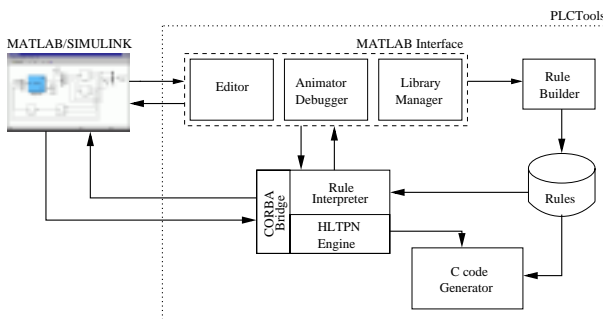


Figure 2: High-level architecture of PLCTOOLS

Each resource must have:

- A *main task*, which embodies the main execution cycle of the control.
- One or more *daemon tasks*, which are periodic tasks that typically represent critical activities with higher priorities than the main task. For example, multi-rating applications require as many daemon tasks as the possible sampling times.
- Zero or more *interrupt tasks*, which are triggered tasks whose execution is triggered by an external event associated with a digital input of the resource. A priority level is always associated with these tasks to manage their executions.
- Zero or more *exception tasks*, which are triggered tasks that depend on the hardware. Some microprocessor provides an interrupt resource related to a particular hardware or software exception that users can associate with these tasks.

The behavior of each task is defined in terms of FBD diagrams. Control engineers can plug FBD blocks, which belong to the libraries<sup>2</sup> (i.e., sets of FBD blocks) associated with the resource the task belongs to, in a hierarchical way.

<sup>2</sup> For example, a 16 bit fixed-point processor needs different libraries than a 32 bit floating-point processor.

Leaf diagrams contain only flat FBD blocks, which correspond to actual computations, while intermediate diagrams contain also hierarchical blocks, which are simply containers for lower-level diagrams. The *FBD Editor* associates a window with each FBD diagram. The overall control organization is summarized in a special-purpose window called *session navigator*. Notice that the graphical organization of an FBD diagram reflects how PLCTOOLS executes it: Blocks are enabled when all their input data are available and are executed top-down and left to right.

The *Library Manager* is developed in MATLAB 5.3, organizes FBD blocks in libraries, and associates them with HLTPNs. The definition of a new block requires that both its concrete syntax and dynamic semantics be specified. The concrete syntax sets the number, type, shape, and color of input and output ports. The dynamic semantics uses a *Petri Net Editor* to specify the HLTPN. Besides the graphical structure in terms of places and transitions, the specification requires predicates, actions and time intervals for transitions. Time intervals can be both fixed and parametric with respect to a user-defined time granularity  $t$ .

PLCTOOLS distributes a standard library with all blocks defined in the IEC standard, which are associated with general-purpose HLTPNs. Experts are free to both customize this library for specific purposes and define brand-new blocks which embody new computations. As already pointed out while drafting the approach, a specific set of libraries should be associated with each processor to allow both correct simulation of timing aspects and acceptable code generation. Thus, if we started the definition of a completely new application or we used a new processor for the first time, we should define a new library.

The *Rule Builder* is implemented in C++ and transforms the definitions provided by the *Library Manager* into pairs of graph grammar productions represented as C++ code fragments. Once compiled and stored (repository *Rules*), rules become available for transforming FBD models.

The *Animator/Debugger* is implemented in MATLAB 5.3 and animates the plant/controller simulation. This component starts by invoking the *Rule Interpreter*, which reads the FBD model and generates the HLTPN according to currently set rules. Once the HLTPN has been built, control experts can start simulating and debugging the control, together with the plant. The *HLTPN Engine* executes the HLTPN and uses the *CORBA Bridge* to exchange information with the SIMULINK model of the plant. SIMULINK “executes” the plant and the engine executes the HLTPN. The time-discrete FBD (HLTPN) model is executed together with the time-continuous model of the plant by setting a time frame at which the controller samples the plant and react to its inputs.

Simulation produces both on- and off-line results. On-line results are animation events that highlight blocks showing

the execution flow, that is, what blocks execute when. Off-line results are the *PLCTools Task Scope*, which visualizes tasks executions on control's resources, and usual SIMULINK scopes, which visualize control values.

Animation - block highlighting when executing - can be switched on and off according to user needs. Interactive animation allows users to understand what is executing when, but at the same time slows down the simulation. Typically animation is used more intensively during early design phases to analyze system behavior, and less often during late design phases while checking performances.

The task scope lets experts evaluate the activation sequence among tasks and their duration. At the beginning, for example, if all tasks were ready, we could get at a glance how long the higher daemon task delays the others and check the model against time constraints (e.g., over-flown tasks). SIMULINK scopes help control engineers get the system dynamics and evaluate control performance.

During simulation, the *Debugger* allows experts to set breakpoints on selected FBD blocks (i.e., HLTPN transitions) and to start step-by-step execution, where a single step corresponds to executing a single FBD block.

Finally, the *Code Generator* automatically derives ANSI C code from the FBD diagrams and HLTPN of each designed task. The generator works on single tasks and merges FBD diagrams and HLTPNs to generate pre-production code. Task-by-task generation precludes the whole-code view, but guaranties enough flexibility and maximizes modularity and reuse. All generated code can be linked to and used through a pre-emptive real-time operating system.

#### 4. An Example

In this section we show how PLCTOOLS can be used to design, analyze, and code generate a simple drill controller taken from the FMS (Flexible Manufacturing Systems) domain. The system, shown in Figure 3, comprises two drills and a cart: Two pistons move the cart with a to-be-worked piece back and forth to position it under drills. The working sequence is organized as follows: (1) Piston 1 moves the cart under drill 1. (2) Drill 1 operates on the piece for  $t_a$  time units. (3) Piston 2 moves the cart under drill 2. (4) Drill 2 works on the piece for  $t_a$  time units. In the end, the two pistons move back and the cart returns to the initial position, ready for new pieces.

The design and validation process can be organized in a sequence of steps:

**Plant-Control Interface Design** In Figure 3, a presence sensor indicates a new piece on the cart and two position sensors identify the positions of the two pistons. Both pistons and drills are activated by binary control signals. Thus, the control interface comprises three input variables, which store the information from the three sensors, and four

output variables, which correspond to the four signals that control pistons and drills. Pistons move back and forth when their control signals are equal to 0 and 1, respectively; drills are on and off when the control signals are equal to 1 and 0.

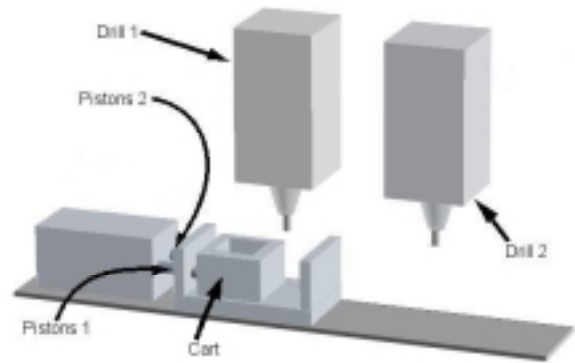


Figure 3: An example drill system

**Plant Specification** The SIMULINK model of the plant is presented in Figure 4:

- Blocks **piston1** and **piston2** model the pneumatic pistons. The input to these blocks is the activation signal (*in1*), while the three outputs correspond to the piston's position (*pos*) and to the state of the piston's position sensors: *Fc1* and *Fc2* correspond to pistons completely opened and completely closed, respectively.

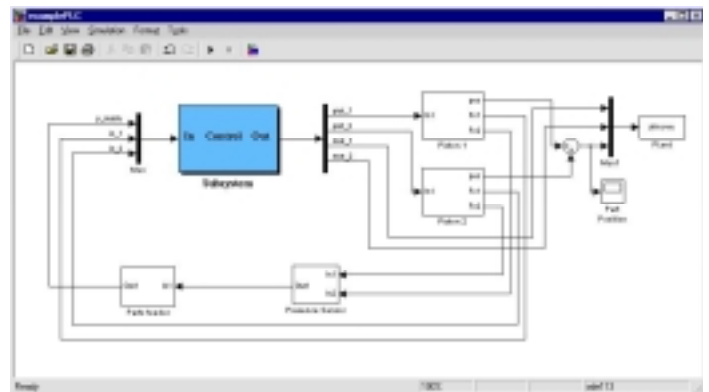


Figure 4: SIMULINK specification of the plant for the drill system

- Block **presence sensor** identifies whether there are new pieces on the cart.
- Block **parts feeder** is used to always feed the system (cart) with new pieces.
- Block **control** is the actual control, which will be specified in the next paragraph.

- The auxiliary block **plant** visualizes the whole system within SIMULINK by masking the MATLAB **sfmove** function.

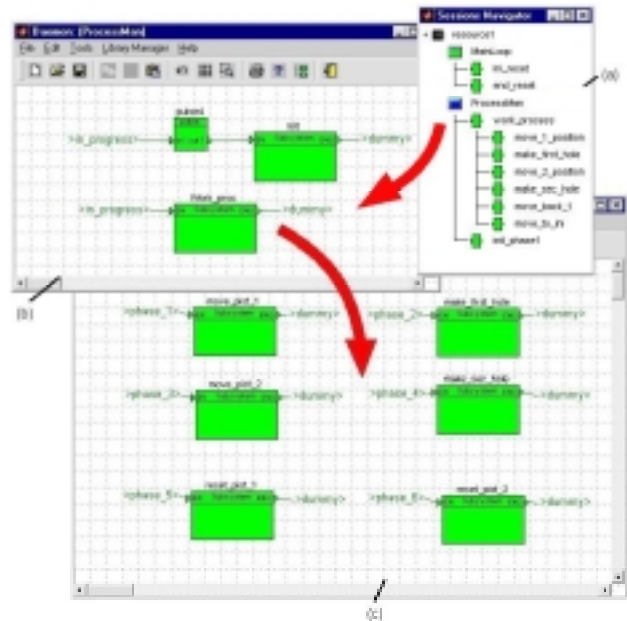
**Control Design** According to the IEC standard ([8]), we can split control design in three sequential activities: *configuration design*, to identify resources and their relations, *task design*, to identify tasks and their relations, and *task specification*, to define tasks' behaviors. In this example, we assume a single resource that runs two tasks:

- Task **MainLoop** is a low priority task that starts new working sequences by polling the control input *p\_ready* (from block **presence sensor**) to discover the presence of a new piece.
- Task **ProcessMan** is a high priority daemon task that manages the evolutions of working sequences.

Moving to task specification, each task is organized in a hierarchy of FBD blocks as shown in the *PLCTOOLS Session Manager* of Figure 5(a). The daemon task, for example, performs two separate activities that correspond to the two main hierarchical blocks of Figure 5(b). Block **init** executes once at the start of the working sequence and resets all internal variables; block **work\_proc** controls the whole working cycle.

The **work\_proc** phase, Figure 5(c), is a sequence of activities represented with different hierarchical blocks. Recalling the working sequence, the working process (1) moves the first piston (block **mov\_pist\_1**), (2) makes the first hole (block **make\_first\_hole**), (3) moves the second piston (block **mov\_pist\_2**), (4) makes the second hole (block **make\_sec\_hole**), (5) resets the first piston (block **reset\_pist\_1**), and then the second piston (block **reset\_pist\_2**).

**Simulation** After completing the specification of both the plant and control, domain experts can simulate the whole system. The HLTPN that corresponds to the control is not presented here due to lack of space; here we concentrate on experts' viewpoint. Figure 6(a) freezes the execution of task **MainLoop** and more precisely, it shows the activation of block **ini\_reset**. Figure 6(b) presents an excerpt of the task scope of the example, where gray (light gray in the figure) means that the task is executing and blue (dark gray in the figure) means that the task is delayed. Figure 6(c) shows the scope that represents the position of the cart (piece). Figure 6(d) offers a 3D model of the plant to visually touch how the system evolves. Thus, in an integrated view, PLCTOOLS offers both low-level information (through SIMULINK scopes and the task scope), and high-level global animations, through special-purpose MATLAB models.



**Figure 5: The Session Manager and some excerpts of the FBD model of the daemon task of the drill system**

**Code Generation** After validating the model through simulation, we can generate the ANSI C code that corresponds to the two tasks.

## 5. Conclusions and Future Work

The paper presents PLCTOOLS, a toolbox for formal design, validation, and code generation of software controllers. So far, PLCTOOLS has been used to design and validate the control of a DC motor, the control of the Ansaldo<sup>3</sup> Cycloconverter ([3]) and the control of a robot arm (jointly with the University of Parma). These case studies demonstrated the soundness of the technology and the suitability of PLCTOOLS as supporting environment. Both PLCTOOLS and its supporting technology are still evolving and our future plans include:

- The use of PLCTOOLS to model further control problems, both to assess its soundness and receive feedback;
- The definition of special-purpose static analyses on the HLTPN and their meaningful mappings in terms of visualizations and animations of FBD blocks;
- The extension of PLCTOOLS to support identified static analysis strategies;
- The improvement of code generation to cope also with those details that are not currently addressed

<sup>3</sup> Ansaldo is the major Italian electro-mechanical company.

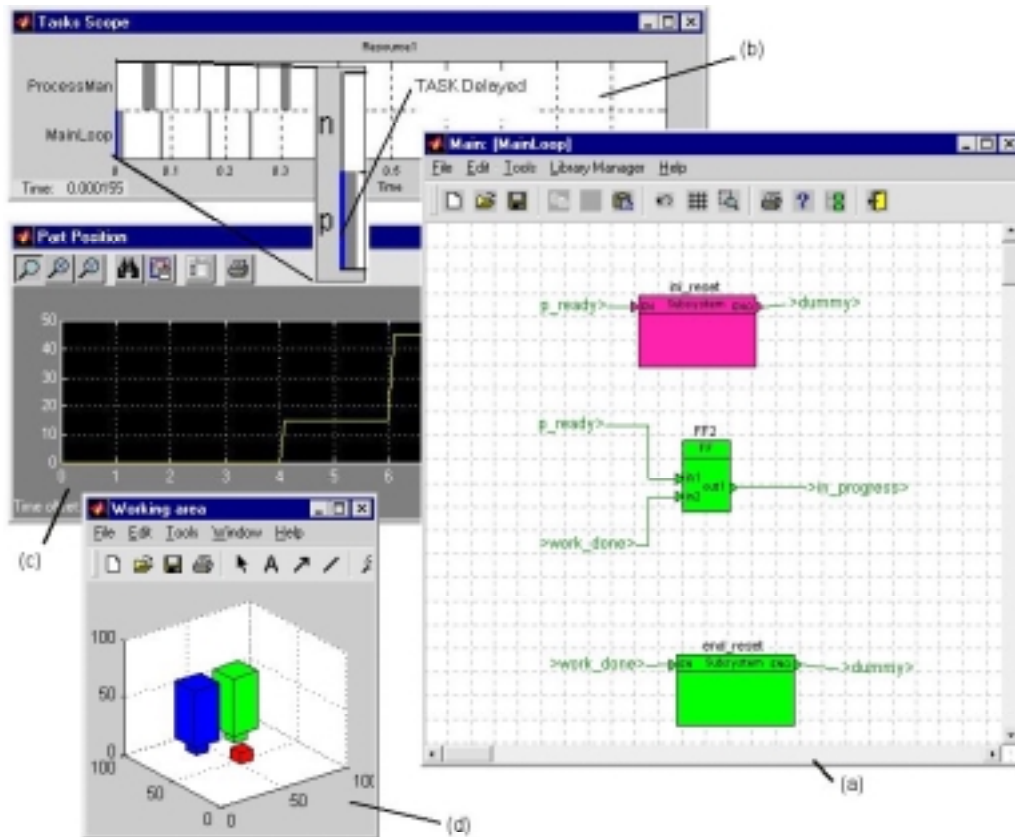


Figure 6: Integrated simulation of the drill system

## References

- [1] L. Baresi. Formal Customization of Graphical Notations. PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. In Italian.
- [2] L. Baresi, A. Orso, and M. Pezzè. Introducing Formal Methods in Industrial Practice. In *Proceedings of the 19th International Conference on Software Engineering*, pages 56-66. ACM Press, 1997.
- [3] S. Carmeli, E. Cosatto, and C. Penno. Ansaldo Demonstration: Design of Application Components. Technical Report INFORMA-AI-17, Ansaldo Sistemi Industriali, Jan. 2000.
- [4] G. Frey. Simulation of Hybrid Systems Based on Interpreted Petri Nets. In *Proceedings of the IEE International Conference on Simulation - Innovation Through Simulation*, pages 168-175. IEE, Oct. 1998.
- [5] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A Unified High-Level Petri Net Model For Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160-172, Feb. 1991.
- [6] H.Göttler. Attribute Graph Grammars for Graphics. In *Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 130-142. Springer-Verlag, 1983.
- [7] IEC. Part 3: Programming Languages, IEC 1131-3. Technical report, International Electrotechnical Commission - Geneva, 1993.
- [8] R. Lewis. Programming Industrial Control Systems Using IEC 1131-3. IEE Publishing, 1998.
- [9] S. Pamperiere-Couffin, O. Rossi, J. Lesage, and J. Roussel. Formal Validation of PLC Programs: A Survey. In *Proceedings of European Control Conference 1999 (ECC'99)*, Sept. 1999.