

PLCTOOLS: Graph Transformation Meets PLC Design

Luciano Baresi¹, Marco Mauri², and Mauro Pezzè³

¹ *Dipartimento di Elettronica e Informazione
Politecnico di Milano – Milano, Italy
bares@elet.polimi.it*

² *Dipartimento di Meccanica
Politecnico di Milano – Milano, Italy
marco.mauri@polimi.it*

³ *Dipartimento di Informatica, Sistemi e Comunicazione
Università degli Studi di Milano – Bicocca – Milano, Italy
pezze@disco.unimib.it*

Abstract

This paper presents PLCTOOLS, a formal environment for designing and simulating programmable controllers. Control models are specified with IEC FBD (Function Block Diagram), and translated into functionally equivalent HLTPNs (High-Level Timed Petri Nets), through *MetaEnv*, for analysis and simulation and obtained results are presented in terms of suitable animations of FBD blocks.

The peculiarity with FBD is that it does not come with a fixed set of syntactic elements; it allows users to add as many new blocks as they want. Consequently, each time users want to add a new FBD block with PLCTOOLS, they must provide the concrete syntax, to add it to the library of available blocks, but also the associated HLTPN, to allow *MetaEnv* to build the formal representation.

1 Introduction

PLCTOOLS is a complete environment for designing and validating software controllers, also known as programmable logic controllers (PLC)¹, based on IEC 1131-3 FBD (Function Block Diagrams) [8]. It combines SIMULINK [7] and *MetaEnv* [3] to supply control engineers with a complete and formal developing environment. It is complete since PLCs can be designed and simulated together with the environment they control. It is formal since modeled

¹ Nowadays, the term PLC is overloaded and it does not mean only programmable hardware components, but also completely software controllers usually deployed on special-purpose CPUs.

controllers are given formal semantics through high-level timed Petri nets (HLTPNs). To this end, SIMULINK supplies the framework to model controlled elements, but also the glue to make the complete simulation (control/controlled elements) happen. *MetaEnv* supplies the infrastructure to automate the transformation between FBD models and HLTPNs and vice versa. Obtained HLTPNs are functionally equivalent to modeled controllers and are used to analyze and simulate designed controllers, displaying the obtained results in terms of suitable animations of FBD blocks.

MetaEnv imposes that the correspondences between FBD blocks and HLTPNs are stated through *transformation rules*. Each rule comprises two graph grammar productions: The *Abstract Syntax Graph Grammar* (ASGG) production, which states how the abstract syntax model² has to be modified, and the *Semantic Graph Grammar* (SGG) production, which describes how the global HLTPN should be modified accordingly.

The peculiarity, in this case, is that FBD does not come with a fixed set of syntactic elements, but users are free to add as many new blocks as they want. The language defines only basic blocks, like AND, OR, and similar simple elements, but users can add their own ones. They can both aggregate existing blocks to obtain composite agglomerates, that is, new blocks that become available for reuse, or design new computational blocks. In this case, the semantics is usually specified using a programming language (in many cases, these definitions are written in C) or a few lines of informal text.

PLCTOOLS lets users extend the set of FBD blocks, but they must specify their behavior in terms of HLTPNs. Once a new block is added, the environment extends also the set of rules that create the HLTPN that corresponds to the designed model. Users do not deal with graph grammars directly, but have to supply only the concrete syntax ascribed to the block and the foreseen HLTPN: In this case Petri nets are not only used as hidden formal engine, but they become also a means to specify the behavior associated with new blocks. It is then the system that builds the rule and adds it to the repository.

In this paper, we do not discuss how PLCTOOLS can be used to better model PLCs, but we briefly present its architecture and explain, through a simple example, how FBD models are transformed into HLTPNs and how new rules are added to the set of available ones. As consequence, the rest of the paper is organized as follows. Section 2 presents the overall architecture of PLCTOOLS and describes also how FBD models are organized. Section 3 explains how PLCTOOLS transforms FBD models into HLTPNs and how users can extend the set of blocks (and automatically transformation rules). Section 4 briefly surveys related proposals and Section 5 concludes the paper.

² To be compliant to the OMG/UML jargon, we should refer to this model as the meta-model.

2 PLCTools

The main components of PLCTOOLS are presented in Figure 1, where boxes represent components and arrows identify data flows between elements.

Fig. 1. The high-level components of PLCTOOLS

MATLAB/SIMULINK™ hosts PLCTOOLS and is employed to specify and simulate the plant. This is a standard approach used by control engineers and is out of the scope of this paper.

The EDITOR, developed with MATLAB 5.3, comprises a RESOURCE EDITOR and an FBD EDITOR. Control software – that is FBD models – 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 that comprises a main task, for the main execution cycle of the control, one or more daemon tasks, which represent critical activities with higher priorities than the main task, and zero or more interrupt and execution tasks, which manage external triggers and exceptions, respectively.

The internals of each task are defined through FBD diagrams. FBD blocks, which belong to the libraries are plugged in a hierarchical way. 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*, where resources, tasks, and hierarchical blocks are presented in a tree-like way.

The ANIMATOR/DEBUGGER, also implemented in MATLAB 5.3, animates the controller during simulation. This component starts by invoking the RULE INTERPRETER, that is *MetaEnv*, to generate the HLTPN by applying the *transformation rules* according to the sequence supplied by the EDITOR.

Once the HLTPN is built, the model can be simulated and debugged together with the environment. The HLTPN ENGINE executes the HLTPN and uses the BRIDGE to exchange information with the SIMULINK model of the plant. The HLTPN ENGINE, written in C++, is a standard HLTPN engine, based on Cabernet: a research prototype developed at Politecnico di Milano. The BRIDGE is a communication channel that lets the HLTPN ENGINE communicate with SIMULINK while executing HLTPNs. Data values and timing information are exchanged in this way. The BRIDGE is implemented using CORBA services.

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 (HLTPN) samples the plant and react to its inputs. The RULE INTERPRETER applies animation rules³ to animate the FBD model when a Petri net transition fires.

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 visualized by the *task scope*, which summarizes the execution of tasks on the different resources, and by standard SIMULINK scopes, which display control values.

During simulation, the DEBUGGER allows users 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.

The LIBRARY MANAGER, 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 behavior be specified. The concrete syntax sets the number, types, shapes, and colors of input and output ports. The behavior is the HLTPN specified through a simple PETRI NET EDITOR. Besides the graphical structure in terms of places and transitions, the specification requires predicates, actions and time intervals for transitions.

The RULE BUILDER, implemented in C++, transforms the definitions provided by the LIBRARY MANAGER into pairs of graph grammar productions represented as C++ code fragments. Once compiled and stored in the repository RULES, transformation rules become available for translating FBD models.

PLCTOOLS distributes a standard library with all blocks defined in the IEC standard. But users are free to both customize this library for specific

³ Animation rules are not discussed in this paper. Interested readers can refer to [2] for a thorough presentation.

purposes and define brand-new blocks which embody new computations.

The `CODE GENERATOR` and `BLOCK GENERATOR` are both implemented in C++. The former automatically derives ANSI C code from FBD diagrams and HLTPNs of each designed task. The generator works on single tasks and merges FBD diagrams and HLTPNs to generate pre-production code. The latter generates the `SIMULINK` block(s) that are necessary to simulate the whole model without the `HLTPN INTERPRETER`. This component exploits the generated code to produce what needed by `SIMULINK`. The simulation becomes faster, but the analysis level is wider because we cannot catch any single event in the running code.

3 Graph Transformation Features

The main characteristics of `PLCTOOLS` as graph transformation environment are: (1) its capability of transforming FBD models into functionally equivalent HLTPNs and (2) the on-the-fly generation of new transformation rules imposed by the extensibility of FBD.

3.1 Transformation Rules

The `RULE INTERPRETER` generates HLTPNs according to the transformation rules stored in `RULES` and the sequence specified by the `EDITOR`. To explain how the translation happens, let us concentrate on the simple *permanent magnet DC motor control* of Figure 2.

Roughly, we have a single-resource motor control (Figure 2(a)) with two tasks (*SpeedCtrl* and *CurrentCtrl*, respectively) (Figure 2(b) and (c)). If we concentrate on the *SpeedCtrl* task (Figure 2(d)), it is a simple *proportional speed control* where we subtract the speed reference⁴ (input `in1Spd`) from the actual speed of the motor (input `in2Spd`) and we multiply the result by the proportional gain (input `KpSpd`) to obtain the new reference for current control (output `OutSpd`). The two simple arithmetic operations are done by two blocks of type `Dsub` and `Dmul`, respectively.

The abstract syntax graph underlying the FBD model comprises six nodes and five arcs. Nodes are three of type `Dload`, to read the three input values, two of types `Dsub` and `Dmul`, and one of type `Dstore` to store the output⁵; arcs are untyped.

Given this abstract representation, the `EDITOR` defines the actual sequence of transformation rules that must be applied to build the HLTPN. First, it considers load and store nodes, then all the others. After all rules for creating subnets that correspond to nodes, it adds all rules for connecting the subnets. Besides those connections available from the model (that is, the arcs), there are connections also to define the control flow. FBD imposes that the execution

⁴ All inputs and outputs are treated as if they were real numbers.

⁵ The need for loading and storing values is specific to FBD ([8]).

Fig. 2. A simple *permanent magnet DC motor control*

flow mimics the concrete layout, starting from the top-left corner and moving top-down and left to right. The resulting HLTPN is presented in Figure 3, where dashed lines are used to identify the subnets that correspond to the different blocks, shared places are added by the rules that connect the subnets, they actually merge places, and gray places correspond to control places, that is, those elements that define the execution flow.

Fig. 3. Functionally equivalent HLTPN

If we added a token in place `CtrlIn`, the only feasible execution flow would be: $\langle t_1, t_2, t_3, t_4, t_5, t_6 \rangle$. Tokens in control places are added by the subnets that render resources and tasks, which are not shown in Figure 3.

3.2 Generation of New Rules

Users add a new FBD block to the selected library as shown in Figure 4. They add the block (Figure 4(a)), define its concrete appearance (Figure 4(b)), and specify the corresponding Petri net (Figure 4(c)). The convention used is that all blocks show places as interfaces and the `PETRI NET EDITOR` draws automatically these places once the user has defined the number of input and output ports at the concrete level.

The `LIBRARY MANAGER` adds the new block to the library and produces a simple textual file with all information that must be sent to the `RULE BUILDER`. The file contains the number of input and output ports and the complete HLTPN, that is, places, transitions, arcs, and annotations associated with these elements.

Fig. 4. Some snapshots of the `LIBRARY MANAGER`

The `RULE BUILDER` produces the new transformation rule as exemplified in Figure 5:

- At the abstract syntax level (Figure 5(a)), each FBD block is identified through a B node, to render the block, and two sets of I and O nodes for its input and output ports. The actual block's type and the data type associated with each port is hidden in the textual annotations⁶. The pro-

⁶ If the expression is simply a semicolon (;), this means that the value should be supplied externally, that is, read from the FBD model.

duction adds an `intSum` FBD block to the task. The block has two input ports and one output port. Each input/output port belongs (b edge) to a particular FBD element (B node), while the FBD element belongs to the current task rendered through the T node.

- At the semantic level, the production adds a B marker, which is used to relate all HLTPN elements that correspond to the same FBD block to the T node, that is, the current task. Moreover, it adds all the HLTPN elements that defines the block. In this case, a single transition which has two places in the pre-set and one place in the post-set (i.e., input and output ports). l (link) edges relate places to the transitions of which they define the pre- and post-set. We do not have HLTPN arcs in these productions; they are added when the ports are connected with those of other blocks.

Textual annotations define further properties⁷. Specifically, the `name` of node 2 is the same as the name of node 2 of the ASGG production, the `absNode` of node 2, that is, the abstract element associated with node 2, is node 2 of the ASGG production, and the `action` associated with node 3 computes the sum. The time interval associated with the transition makes it consume t time units when fired, that is, the execution of the FBD block takes t time units.

The two files (the two productions), which are fragment of C++ code, are automatically included in the main file of the library and then the whole library – with the new transformation rule – is recompiled. The result, that is, the set of transformation rules for all blocks in the library, are then available for transforming models that use the new block.

4 Related Work

The problem of interpreting informal models through formal notations by means of proper mappings has been widely studied [4,9]. Initially the different proposals concentrated mainly on Structured Analysis, while nowadays almost all approaches have shifted towards UML and similar notations. In this paper we do not have enough room to discuss all interesting approaches, but we want to mention at least the work by France [6] since he is the first who classifies the different ways for integrating formal and informal notations. According to his simple taxonomy, the approach proposed in the paper is both *interpretative* and *suppletive*. It is interpretative since HLTPNs are used to interpret (ascribe a formal behavior to) FBD models; it is suppletive since HLTPNs are also used as modeling notations by those users who want to extend the set of available building blocks.

As far as graph grammars as transformation means are concerned, our

⁷ The convention is that if an expression is between two `@`, the value should be read from the ASGG production. For example `1.name = @3.name@` means that the name of node 1 of the SGG production is the same as the name of node 3 of the ASGG production.

<pre> 2.name = ; 2.type = "intSum"; ... </pre>	<pre> 2.name = @2.name@; 2.type = "intSum"; 2.absNode = @2.name@; ... 3.predicate = "TRUE"; 3.action = "6.value = 4.value + 5.value"; 3.tMin = "enab+t"; 3.tMax = "enab+t"; ... </pre>
--	---

(a) ASGG production

(b) SGG production

Fig. 5. A simple transformation rule for `intSum` FBD blocks

work is clearly inspired by *pair grammars* (Pratt [10]) and *triple graph grammars* (Schürr [11]). In both cases, our approach borrows the main concepts, but simplifies the way they are used: Our main goal with *MetEnv*, and thus PLCTOOLS, was a simple and usable rule-based transformation engine, instead of a sound and complete theoretical framework.

5 Conclusions

The paper presents PLCTOOLS and concentrates on its global architecture, informal-to-formal transformation, and on-the-fly generation of transformation rules. The aim is to explain how the environment exploits graph grammars to improve the current practice in modeling software programmable controllers. Space constraints preclude us to show realistic examples of what can be done with PLCTOOLS: Interested readers can refer to [1] for a more control-oriented presentation of the environment and to [5] for a complete example on how PLCTOOLS was used to model an industrial-like control.

References

- [1] L. Baresi, M. Mauri, A. Monti, and M. Pezzè. PLCTools: Design, Formal Validation, and Code Generation for Programmable Controllers. In *Proceedings of the IEEE Conference on System, Man, and Cybernetics*, October 2000.
- [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] L. Baresi and M. Pezzè. A Toolbox for Automating Visual Software Engineering. In *Proceedings of the 5th ETAPS FASE Conference*, pages 189–202, April 2002.
- [4] M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors. *Workshop on Precise Semantics for Software Modeling Techniques*, 1998. In conjunction with ICSE 1998.
- [5] S. Carmeli, E. Cosatto, and C. Penno. Ansaldo Demonstration: Design of Application Components. Technical Report INFORMA-AI-17, Ansaldo Sistemi Industriali, January 2000.
- [6] R. B. France and M. M. Larrondo-Petrie. From Structured Analysis to Formal Specifications: State of the Theory. In *Proceedings of Computer Science Conference*, pages 249–256. ACM Press, April 1994.
- [7] The MathWorks Inc. MATLAB 5.3. *www.mathworks.com*, 2000.
- [8] R.W. Lewis. *Prgramming Industrial Control Systems Using IEC 1131-3*. IEE Publishing, 1998.
- [9] R.F. Paige. A meta-method for formal method integration. *Lecture Notes in Computer Science*, 1313:473–485, 1997.
- [10] T.W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
- [11] A. Schürr. Specification of Graph Transformations with Triple Graph Grammars. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer-Verlag, June 1994.