Editing, Visualizing, and Implementing Signal Interpreted Petri Nets

Georg Frey
Lehrstuhl für Automatisierungstechnik
Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern, Germany
frey@eit.uni-kl.de

Mark Minas
Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
minas@informatik.uni-erlangen.de

Abstract

In this paper we present a new tool for editing, visualizing, and implementing Signal Interpreted Petri Nets (SIPNs). SIPNs are used to formally specify control algorithms for Programmable Logic Controllers (PLCs). The presented tool allows to automatically transform a graphically edited SIPN into PLC code using the standardized PLC language Instruction List. The tool was implemented using DiaGen, an innovative workbench for generating graphics editors.

1. Introduction

To model logic controllers we use Signal Interpreted Petri Nets (SIPNs). SIPNs add means for the description of I/O-behavior to the standard PN. In SIPNs the firing of a transition depends on (functions of) input signals from the environment, and the SIPN influences the environment via output signals. With this model logic controllers can be specified, simulated, analyzed [6, 5], and implemented [4]. However, to bring the method into practical applications strong tool support is needed.

For the industrial realization of a controller, standard PLC programming languages according to IEC61131-3 are used. Properties of the SIPN can only be guaranteed for the implemented controller if the generation of PLC code from SIPN preserves the dynamic behavior of the latter. To avoid errors the transformation should be done automatically. This contribution presents a method for automatically compiling an SIPN to Instruction List (IL).

A prototypical tool for editing, visualizing, analyzing, and translating SIPNs (see Fig. 1) has been implemented using DIAGEN (DiaGen Editor Generator), an environment for rapidly developing diagram editors. The tool consists of an graphics editor which allows for easily editing SIPNs. Edited SIPNs are translated by the tool into equivalent IL programs which implement the SIPNs on logic controllers.

Currently, the tool allows to specify the behavior of places and transitions by associating IL code with each place and transition. These code segments are simply transferred into the generated program code without any syntax check. Furthermore, the tool offers an interface to other programs which perform additional tests (e.g., liveness tests) on the edited SIPN.

The rest of this paper is organized as follows: The next section introduces SIPNs and the process of translating them to PLC languages. Section 3 describes DIAGEN, and section 4 briefly introduces into the notion of hypergraphs and grammars which build the foundation of DIAGEN and generated editors. Section 5 then describes the main functionality of the SIPN editor, i.e., how diagrams which have been drawn using the editor are translated into equivalent IL programs. Section 5 concludes the paper.
2. SIPN and PLC languages

SIPN can be seen as a proper subset of Sequential Function Chart (SFC) according to IEC61131-3 standard [7, 9]. The only difference in dynamic behavior is that states in an SIPN can be unstable (transient) whereas in SFC such states are held active for at least one PLC cycle. However, given the short duration of the PLC cycle, such states in SFC can be seen as nearly transient.

Besides the differences in the specific Petri net type considered, the presented work differs from other approaches (see e.g. [3, 13, 14]) in one main aspect. This is the one-to-one correspondence of net elements to code segments that is used. This correspondence allows to easily reinterpret produced code. This reinterpretation is of special importance if a user wants to understand and change the implemented code, which is commonplace in industrial applications.

An SIPN is a condition/event net with the following extensions for the information flow:
• Every transition is associated with a Boolean function of the input signals, the firing condition.
• Every place is associated with an output function, that assigns a subset of output signals while it is marked.

The dynamic behavior of an SIPN is given by the flow of tokens through the net, i.e., the change of its marking. This flow is realized by the firing of transitions. Firing of a transition removes a token from each of its pre-places and puts a token on each of its post-places. For the firing process there are five rules:
• A transition is enabled if all its pre-places are marked and all its post-places are unmarked.
• A transition fires immediately if it is enabled and its firing condition is fulfilled.
• All transitions that can fire and are not in conflict with other transitions fire simultaneously.
• The firing process is iterated until a stable marking is reached (i.e. until no transition can fire anymore). Iterated firing is interpreted as simultaneous. This also means that a change of input signal values can not occur during the firing process.
• After a new stable marking is reached, the output signals are computed by evaluating the output functions of the marked places.

For a structure-conserving conversion the token play of the SIPN has to be transferred to the PLC program. Therefore, for each place \( p_i \) of the SIPN a boolean variable \( P_i \) is defined that shows if the corresponding place is marked (\( P_i = \text{true} \)) or unmarked (\( P_i = \text{false} \)). Based on this premise the net elements can be translated to PLC code step by step:

**Transitions**: The compilation of a transition has to test whether the transition is enabled and whether the firing condition is fulfilled. If after the processing of this calculation the accumulator is set to 1, i.e. all conditions are fulfilled, then the transition fires. If not, a conditional jump to the next transition avoids firing. The firing unmarks all pre-places and marks all post-places. To optimize the code, the firing conditions are not evaluated if a transition is not enabled.

**Places**: If a place is marked then the corresponding output function is executed (setting or resetting of variables). If it is not marked a conditional jump to the next place label is performed and the code segment of the output function is not executed.

Note, that the implementation of the output functions results in an output of zero or one (\( \text{true} \) or \( \text{false} \)) for all output variables according to the output of the SIPN. In case of an incorrect output setting in the SIPN the output of the code differs from the SIPN since undefined and contradictory output settings are not possible in the realization. For an undefined output, the PLC code remains at the last defined value for this output. For a contradictory output it depends on the ordering of the involved place code segments if the variable is set to one or to zero. Both cases should be avoided using SIPN analysis prior to code generation [5].

3. DIAGEN

DIAGEN provides an environment for rapidly developing diagram editors. This section first outlines this environment and how it is used for creating a diagram editor that is tailored to a specific diagram language. DIAGEN can be used for creating editors for a wide variety of diagram languages, e.g., finite automata and control flow diagrams, Nassi-Shneiderman diagrams, message sequence charts and visual expression diagrams, sequential function charts, and ladder diagrams [12, 10, 11, 8]. Actually we are not aware of a diagram language that cannot be processed with DIAGEN.
DIAGEN is completely implemented in Java and consists of an editor framework and a program generator. Fig. 2 shows the structure of DIAGEN and the process of using it as a rapid-prototyping tool for developing diagram editors. The framework, as a collection of Java classes, provides the generic editor functionality which is necessary for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer primarily has to supply a specification, which textually describes syntax and semantics of the diagram language. Additional program code which is written “manually” can be supplied, too. Manual programming is necessary for the visual representation of diagram components on the screen and for processing specific data structures of the problem domain, e.g., for semantic processing when using the editor as a component of another software system. The specification is then translated into Java classes by the program generator.

The generated classes, together with the editor framework and the manually written code, implement an editor for the specified diagram language. This editor can run as a stand-alone program. But it can also be used as a software component since the editor framework as well as the generated program code is conformable with the JavaBeans standard, the software component model for Java.

Diagram editors which have been developed using DIAGEN (such editors are called “DIAGEN editors” in the following) provide the following features:

- DIAGEN editors always support free-hand editing, i.e., the editor user can arbitrarily create, delete, and modify diagram components (places, transitions, arcs, and tokens for SIPNs) as with an off-the-shelf drawing tool. The editor uses the syntax specification of the diagram language for analyzing the “drawing” after each editing operation performed by the user. Feedback is provided to the user where the “drawing” is not a correct diagram.
- Correct diagrams are translated into a semantic representation, e.g., an IL program that implements an SIPN on a logic controller. This process is driven by the syntactic analysis and makes use of program code and data structures which are provided as “editor specific program code” in Fig. 2.
- Each DIAGEN editor optionally supports syntax-directed editing, too, i.e., the editor provides a set of editing operations. Each of these operations is geared to modify the meaning of the diagram (e.g. for SIPNs, delete a transition with all of its incoming and outgoing arcs). Syntax-directed editing is only supported if the editor developer has specified syntax-directed editing operations.
- Automatic layout is an optional DIAGEN editor feature, too, but which is obligatory when specifying syntax-directed operations. The automatic layout mechanism adjusts the diagram layout (i.e., the position, size etc. of diagram components) after any modification to the diagram. Automatic layout also assists free-hand editing: After each layout modification by the user, the layout mechanism changes the diagram such that its structure remains unchanged. DIAGEN offers constraints for specifying the layout mechanism in a declarative way [8], or a programming interface for plugging in other layout mechanisms.

DIAGEN has been used for generating an SIPN editor with the following features:

- Free-hand editing is supported as with regular drawing tools: places, transitions, arcs, and tokens can be created and deleted with the mouse; when selected, diagram components are visualized with handles which can be used to modify position or size of the component. Syntax-directed editing is not supported since there is no need for complex operations for this simple class of diagrams.
- The editor uses a simple layout mechanism which ensures that places and transitions “snap” to a grid. The mechanism also takes care of adjusting arcs when moving places and transitions such that connections are not lost. But the layout mechanism does not change the position of places and transitions nor does it auto-route arcs. We believe that this is a editor user’s task.
- The diagram visualizes the graphical structure only, i.e., the places and transitions together with their names, the arcs, and tokens. For each place and transition, additional IL code can be added in a property editor when the corresponding diagram component has been selected. The IL code of a place has to be executed if the place is active, whereas the IL code of a transition specifies a firing condition.
- After each modification to the diagram, i.e., SIPN, the (correct) diagram is translated into an IL program as a semantic representation. This code implements the behavior of the SIPN on a logic controller. IL code which has been added to places and transitions by the user is properly included in the resulting program.

Fig. 1 shows a snapshot of the generated editor in action. Some lines of the semantically equivalent IL code which has been created by the editor are depicted in Fig. 6.

The following sections briefly survey the main concepts of DIAGEN and the SIPN editor which has been generated with DIAGEN.

4. Hypergraphs and Grammars

DIAGEN editors are based on hypergraphs as internal diagram models and hypergraph grammars as a means for syntax specification. This section briefly surveys these concepts.
Each graph consists of a set of labeled nodes and a set of labeled edges. Each edge visits two nodes which need not be different. Hypergraphs are generalizations of directed graphs: they have a set of labeled hyperedges instead of edges. Each hyperedge has a fixed number of labeled tentacles which is determined by the hyperedge’s label. Tentacles connect the hyperedge with nodes visited by the hyperedge. A regular directed graph is a hypergraph where each hyperedge has two tentacles with labels source and target. Nodes will be represented by black dots, directed edges by arrows, and hyperedges by boxes containing the hyperedge label. Thin lines are used to represent tentacles connecting the hyperedge with visited nodes. Tentacle labels are omitted where possible.

Hypergraph grammars are similar to string grammars. Each hypergraph grammar consists of two sets of terminal and nonterminal hyperedge labels and a starting hypergraph which contains nonterminally labeled hyperedges only. Syntax is described by a set of productions which consist of the same LHS and RHS, but with an additional (“embedded”) hyperedge on the RHS, i.e., this hyperedge is embedded into the context provided by the LHS when applying such a production (the last two productions of Fig. 5). Parsing algorithms and a more detailed description of both grammar types can be found in [10, 2].

DIAGEN uses hypergraphs as diagram representations and hypergraph grammars for specifying syntactically correct diagrams. The following section describes how these concepts are used by the SIPN editor which has been generated with DIAGEN.

5. Translating SIPNs

The main task of an editor is translating a “drawing” which is supposed to be a correct diagram (i.e., an SIPN here) into a semantic representation (i.e., equivalent IL program here). During this translation process, the editor has to check the drawing for correctness and has to provide visual feedback to the editor user if the drawing contains errors. The editor performs this task in a sequence of four steps after each editing operation: the scanning, the reduction, the parsing, and the attribute evaluation step.

Scanning step: Diagram components (e.g., places, transitions, arcs, and tokens in SIPNs) have attachment areas, i.e., the parts of the components that are allowed to connect to other components (e.g., start and end of an arc). The most general and yet simple formal description of such a component is a hyperedge which connects to the nodes which represent the attachment areas of the diagram components. These nodes and hyperedges first make up an unconnected hypergraph. The scanner connects nodes by additional edges if the corresponding attachment areas are related in a specified way, which is described in the specification. The result of this scanning step is the spatial relationship hypergraph (SRHG) of the diagram. Fig. 3 shows the SRHG of the SIPN shown in Fig. 1. Nodes are represented by black dots, hyperedges either by gray arrows (relationships between attachment areas) or by rectangles (diagram components) that are connected to their nodes (“attachment areas”) by thin lines.

Reduction step: SRHGs tend to be quite large even for small diagrams (see Fig. 3). In order to allow for efficient parsing, a reduced hypergraph model (HGM) is created from the SRHG first. The reducer is specified by some transformations that identify those sub-hypergraphs of the SRHG which carry the information of the diagram and build the HGM accordingly. This step is similar to the lexical analysis step of traditional compilers. Fig. 4 shows the HGM for the SIPN of Fig. 1. Please note the similarity to the original diagram.

Parsing step: The syntax of the hypergraph models of the diagram language—and thus the syntax of the language—is defined by a hypergraph grammar. Fig. 5 shows a context-free hypergraph grammar with embeddings. The attribute evaluation rules are used in the following step. The starting hyperedge of the grammar consists of a Net hyperedge which does not visit any node. Similar to compilers for (textual) programming languages, a hypergraph parser which is built-in into each DIAGEN editor is used for creating the syntactic structure of the HGM of the diagram, i.e., for finding a derivation sequence from the starting hypergraph to

2Please note that the second and third production allows an arbitrary number of Place resp. Trans hyperedges. This more readable notation is used here as a shorthand for recursive productions.
the HGM. The parser is capable of identifying syntax errors which are then visualized to the editor user.3

**Attribute evaluation step:** The final step of the translation process creates the semantic representation of the diagram by some kind of syntax-directed translation based on an attribute grammar as it is also used in compilers for (textual) programming languages [1]: terminal and nonterminal hyperedges are augmented by attributes, and hypergraph grammar productions by evaluation rules. Some attribute values are already defined by the diagram (e.g., name and code for each place and transition edge which contain the name and the IL code which are associated with the corresponding place resp. transition); the values of the other attributes are determined by the evaluation rules which are used whenever the corresponding grammar productions are used in the derivation sequence of the HGM. The evaluation rules which are shown in Fig. 5 create a Java object structure which directly represents the SIPN. The implementation of the invoked functions is straightforward. This object structure is then easily translated into an equivalent IL program (see Fig. 6 for the code generated for the SIPN of Fig. 1).

6. Conclusions

We have described a tool for editing, visualizing, and translating Signal Interpreted Petri Nets (SIPNs) which are used to model causality as well as the concurrency of control algorithms. The current prototype, which has been generated with a rapid prototyping tool for diagram editors, mainly provides a tailored graphics editor for SIPNs which translates such SIPNs to equivalent IL programs implementing the SIPN behavior on a logic controller. The tool moreover provides interfaces for further analyzing and accessing SIPNs, e.g. for importing or exporting SIPNs in an XML format. Further work will also integrate the SIPN tool with existing development tools for logic controller software in order to advertise SIPNs as a high level language for logic controllers.
\[ n.net := \text{createNet}(p.p\text{laces}, t.t\text{ransitions}) \]

\[ p.p\text{laces} := \text{createList}(p_1.p\text{lace}, p_2.p\text{lace}, p_3.p\text{lace}, \ldots) \]

\[ t.t\text{ransitions} := \text{createList}(t_1.t\text{rans}, t_2.t\text{rans}, t_3.t\text{rans}, \ldots) \]

\[ p.p\text{lace} := \text{createPlace}(p.p\text{name}, p.t\text{code}, \text{true}) \mid p.p\text{lace} := \text{createPlace}(p.p\text{name}, p.t\text{code}, \text{false}) \]

\[ t.t\text{rans} := \text{createTrans}(t.t\text{name}, t.t\text{code}) \]

\[ t.t\text{rans}.a\text{ddPre}(p.p\text{lace}) \mid t.t\text{rans}.a\text{ddPost}(p.p\text{lace}) \]

**Figure 5.** Hypergraph grammar for SIPNs resp. their hypergraph models. Terminal hyperedges are represented by rectangular boxes whereas nonterminal ones are shown as ovals.

**References**


**Declarations:**

VAR

P1: B0SL := TRUE; (* Stand by *)

P2: B0SL := FALSE; (* Filling *)

P3: B0SL := FALSE; (* Heating *)

P4: B0SL := FALSE; (* Emptying *)

P5: B0SL := FALSE; (* Stirring *)

END_VAR

**Statements:**

(* Transition Start button pressed *****)

ANDN P5 (* post place Stirring *)

ANDN P2 (* post place Filling *)

JMPCN 11 (* transition specific code *)

JMPCN 11

R P1 (* pre place Stand by *)

S P5 (* post place Stirring *)

S P2 (* post place Filling *)

11: (* Transition Filled & temp low *****)

... 15:

(* Place Stand by *****)

LD P1

JMPCN 15 (* place specific code *)

16: (* Place Filling *****)

LD P2

JMPCN 17 (* place specific code *)

... 19:

(* Place Heating *****)

LD P3

JMPCN 19 (* place specific code *)

... 20:

(* Place Emptying *****)

LD P4

JMPCN 20 (* place specific code *)

... 21:

(* Place Stirring *****)

LD P5

JMPCN 21 (* place specific code *)

... 22:

(* Special Transition, e.g. short cut for Stand by and Filling)"

**Figure 6.** Generated IL code for the SIPN of Fig. 1


