A Petri Net Semantics for the PLC Language Instruction List*)

Monika Heiner, Thomas Menzel

Brandenburg University of Technology at Cottbus,
Department of Computer Science, Postbox 101344, D-03013 Cottbus

{mh, thm}@informatik.tu-cottbus.de
http://www.informtik.tu-cottbus.de

Keywords
Petri net, Programable Logic Controller, Instruction List.

Abstract

In this paper we will describe a Petri net semantics of the PLC language Instruction List (IL) defined in [DIN EN 61131-3] (IEC 1131-3). This is a nessessary prerequisite to be able to analyse functional and especially safety requirements of IL programs. We will define a subset of IL (IL_0) and give formal definitions with reference semantics for this subset. After this the reference semantics is transformed into a Petri net model.

1 Introduction

Programable Logic Controller (PLC) are currently used in many complex industrial areas. It is very important to verify the user program in the PLC in view of given requirements. One possibility to do this is to use Petri nets. There are two approaches:

- Petri nets as modelling tool (synthesis, [Rausch96]) and
- Petri nets as a model to analyse (transformation, [Hanisch97], [Rausch97])

In our department we work on the second approach. The goal is to translate an existing program (here written in IL) into a Petri net. After this translation, we can prove functional and safety requirements by means of the generated Petri net. The requirements can be formulated by formulae of temporal logics, and then we will use model checkers to check the validily of these for formulae in the net. Figure 1 shows the principle cycle of the validation of a PLC user program. This cycle is divided in two main streams:

- generate a model of the IL program and of the environment (plant)
- create formulae of temporal logic for the requirements.

The whole system model is a result of the composition of the control model as a Petri net and the environment model as a Petri net. After the composition it is possible to proof the given set of temporal formulae in the system model.

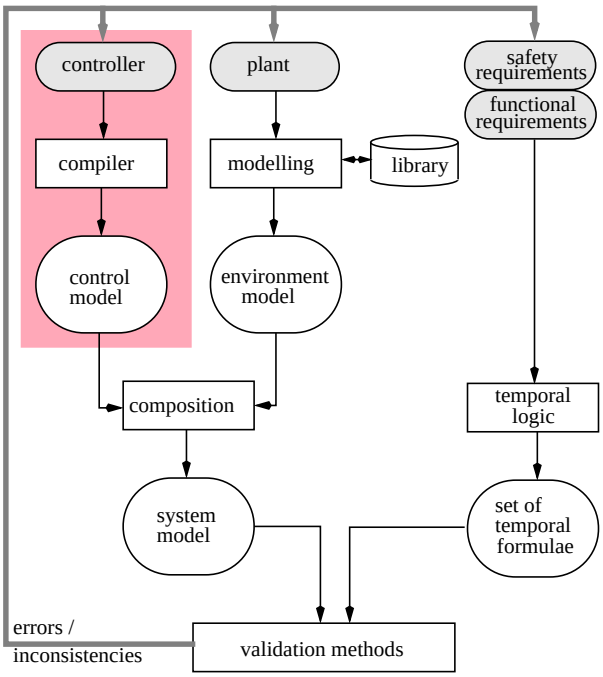


Figure 1 Validation of PLC programs by PetriNets

In this paper we show a possible transformation of an IL program into a Petri net. This transformation process is highlighted in Figure 1. This paper is based upon [Heiner97-1] and [Heiner97-2].

2 IL_0 - a Subset of IL

In this section we describe a subset of the PLC programming language IL, named IL_0. Because of the several libraries with additional commands, the complexity of IL is quite high.
The language IL_0 has some restrictions which make it a subset of IL:

- default commands only
  (no additional commands from any library)
- datastructures to a length of 8-bit only
  (boolean, 8-bit word, unsigned short integer, short integer)
- no commands and data structures for time and date

In the next section we show the way to the formal definition of IL_0. At first we specify an abstract syntax. After that we define a static semantics in the style of a denota-

Proc. of the Fourth Workshop on Discrete Event Systems
*) This work is supported by the German Research Council under grant ME 1557/1-1

tional one. With this semantics we define the allowed semantical context of the language. Finally we give an operational semantics of IL_0 as a reference semantics.
To keep the paper short, we do not show any complete and detailed formal definitions in the sections below. The interested reader is reoffered to [Heiner97-2] for a complete description of IL_0.

2.1 Abstract Syntax of IL_0

At first we need an abstract syntax to define the syntactical body of IL_0 formally. An abstract syntax looks like a concrete syntax. The difference is only the degree of detail. In the abstract syntax we end with nonterminals like <PRGNAME> and declare at this point that the nonterminal is unspecified. The details are not interesting in the given context. With the abstract syntax we would like to describe possible derivation trees with nonterminals as root. Our abstract syntax is formulated in Extended Backus Nauer Form, and has altogether 48 rules. To have a concrete example in mind, we take the first of these rules, the so-called starting rule:.

$$\text{<IL}_0\text{>} \quad \rightarrow \quad \text{<PRG> \{<FB> | <FCT>\}}$$

This rule describes the derivation tree of the whole given IL_0 program.

2.2 Static Semantics

In the section above we have introduced a syntactical body of IL_0. Now we define context requirements. We will define a function for each nonterminal in the given syntax. Each nonterminal describes a derivation tree. The function has such a tree as parameter and returns a tuple Env. In this function we check the semantic context and make some changes in the program code. These changes are very important.
The main function is:

$$\mathcal{IL}_0 : \text{IL}_0 \rightarrow Env$$
$$\mathcal{IL}_0(p) = \mathcal{ChangeMark}(\mathcal{Flat}(\mathcal{IL}(p)))$$

where p is a syntactical correct IL_0 program. The result of this function is the tuple
$Env = Name \times Domain \times Env_0 \times Label \times Code$

- $Name$ - name of the program
- $Domain = \{\mathbf{prg}, \mathbf{fb}, \mathbf{fct}\} \times \{\mathbf{n}, \mathbf{p}\}$ where **prg** - program, **fb** - functionblock, **fct** - function, **n** - no parents and **p** - parents
- Env_0 - tupel for variables
- $Label$ - set of used line labels
- $Code = [\langle \varnothing, \perp \rangle, s_1, s_2, \ldots, s_n]$ - the ordered list of code where s_i are program statements

After the evaluation of the function $\mathcal{IL}_0$ for a given IL_0 program p, $Code$ consists of a flat (without procedure- and function calls) IL_0 program without parents and with explicit jump goals (line umbers instead of labels). All procedure calls and function calls are substituted. All labels in jump statements are replaced by the line number intended from the label before. Parent structures are replaced by inserting an new variable, which buffers the accumulator value before the beginning of the parent structures. The resulting strictly sequential IL_0 program is correct in respect to its syntax and the corresponding semantical context.

2.3 Operational Semantics

After scanning and parsing successfully we have a simple list of commands. Now we can define an operational semantics, which describes the transitions between the program states. Such a program state is defined as tupel (i, m, a), where:

- i is the line number (list index from $Code$)
- $m \in Env_0$ a memory state
- $a = (v, t)$ an accumulator where v is the value and t is the type.

As the basic supposition the evaluation from $e = \mathcal{IL}_0(p)$, where p is a IL_0 program, is needed. Now we define for each command a state transition under e. At this point we like to show some examples:

- $(i, m, a) \xrightarrow[e]{\texttt{AND}\ x} (i + 1, m, (m(x) \wedge a[1], a[2]))$
  The result of the logical operation AND between the accumulator value and operand is taken as the new value of the accumulator.

- $(i, m, \langle \mathbf{T}, \mathbf{bool} \rangle) \xrightarrow[e]{\texttt{JMPC}\ x} (x, m, a)$
  $(i, m, \langle \mathbf{F}, \mathbf{bool} \rangle) \xrightarrow[e]{\texttt{JMPC}\ x} (i + 1, m, a)$
  If the accumulator value is TRUE then jump to line number x, otherwise ignore the jump and start with the execution of the line below.

- $(i, m, a) \xrightarrow[e]{\texttt{ADD}\ x} (i + 1, m, (m(x) + a[1], a[2]))$
  The result of the addition between the accumulator value and operand is taken as the new value of the accumulator.

By defining the operational semantics we don't need any type check, because this was done by the execution of the function $\mathcal{IL}_0$.
In the forthcoming section we model these elaborated semantics with Petri nets.

3 Petri Net Semantics

In the sections above we talked about the formal definition of the language IL_0. There we defined an operational semantics as reference semantic, which is now substituted by a Petri net semantic.

3.1 Syntactical Conventions

In the following section we show Petri nets with some extensions in the representation. The Petri nets are mod-

elled hierarchically. But here the hierarchy has only syntactical nature. We don't use any semantic composition. Before analysing these nets, they must be unfolded. A summary of the syntactical extensions is given in Table 1.

By the way, a new kind of edges is added. The readarc has a black circle instead of an arrow as a head (see Figure 2). These kind of edges have always a place as source

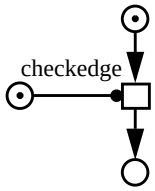


Figure 2 Example of a readarc

and a transition as target. The firing rule is changed in the way, that no tokens are be removed from the source place. The places can interpreted as a side condition to the transition. Under the interleaving semantics for Petri

	*logical place* -- all logical places with the same name inflating to the same place after the unfolding
	*logical transitions* -- like the logical place, but now for the transitions
	*subnet* -- represents a part of the net, which is inserted at this position after the unfolding
	*transition bounded subnet* -- subnet where the bordernodes (nodes with a connection to the upper net) are only transitions
	*place bounded subnet* -- subnet where the bordernodes (nodes with a connection to the upper net) are only places

Table 1 Syntactical Petri net extensions

nets, these arcs can be transformed into two normal arcs (one for each direction between the place and the transition).

The presented Petri nets in the sections below are big and have many repeatitions. Due to this fact we will use the following conventions, to save space. If a subnet would be n times repeated, we write the first and the last subnet only, and mark the repetition with three dots. If the subnets are structurally equal, then we choose intuitive names for places. An example for this compression is shown in Figure 3.

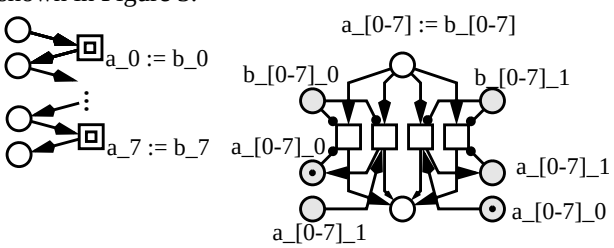


Figure 3 Example of repeatition of a subnet

3.2 Binary representation

We are modelling variables and values in a binary form, where two places are needed to represent a single bit (see Figure 3). Now we can model algorithms for a binary al-

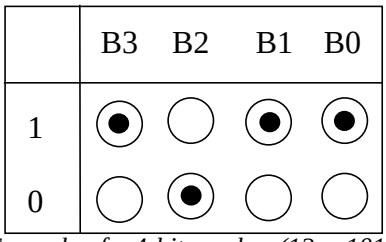


Figure 4 Example of a 4-bit number (12 = 1011)

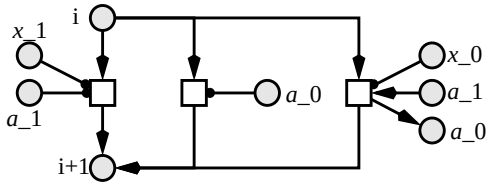
gebra. This we need to formulate the Petri net semantic for IL_0.

3.3 Petri Net semantics

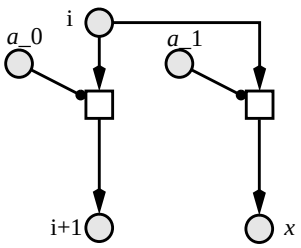
In this section we will show 3 examples for the Petri net semantics. As supposition we need (like in the operational semantic) an execution of the function $\mathcal{IL}_e$ with the result in e. In analogy to the operational semantic we have a function $Compute_e$ too.

After the execution from the function $\mathcal{IL}_0$ we have to parse only the ordered list of $Code$, and for each statement in $Code$ we insert a Petri net. The label i is substituted by the line number, and x is substituted by the name of the operand. The composition of the subnets (each command has its own) results in the whole IL-program. This composition is only sequential.

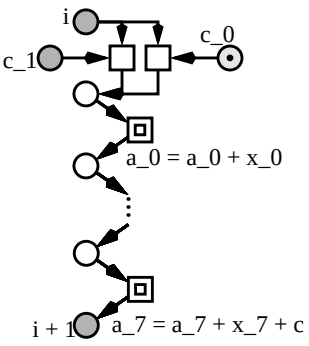
$Compute_e(i, \texttt{AND}\ x)=$

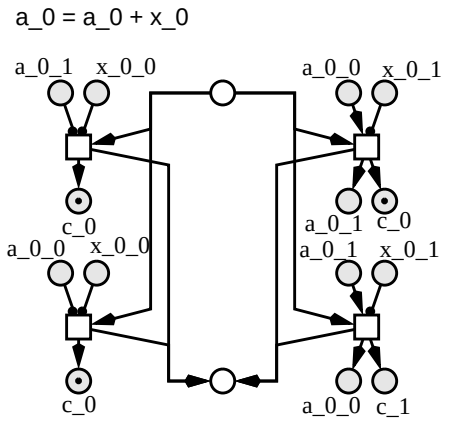


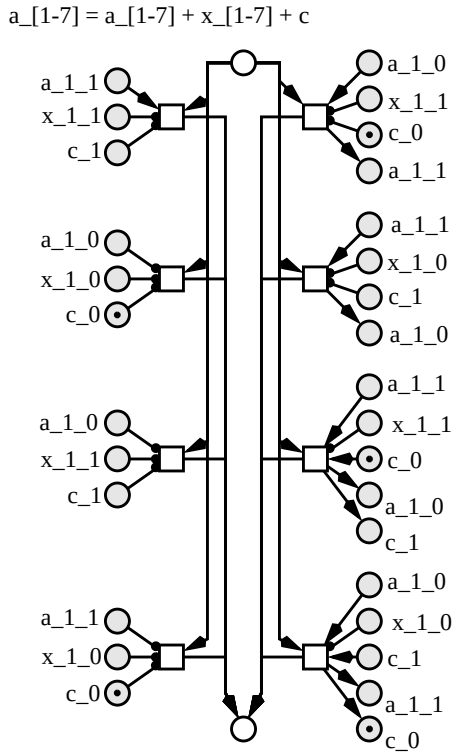
$Compute_e(i, \texttt{JMPC}\ x)=$



$Compute_e(i, \texttt{ADD}\ x)=$







4 Example: 2-hand Switch

Many production cells focused on safety use such a 2-hand-switch, which conditioned the use with 2-hands ([DIN EN 574]). The output is *on* if switch 1 and switch 2 are pressed at the same time (a little delay is possible). The closers work complimentary to the related openers (closer 1 to opener 1 and closer 2 to opener 2). The principle construct is shown in Figure 5.

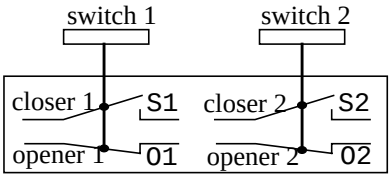


Figure 5 structur of a 2-hand-switch

A possible coding in an IL_0 program is given below:

```
1    PROGRAM switch
2    VAR
3       Step_0:BOOL:=TRUE;
4       Step_1, Step_2:BOOL:=FALSE;
5    END_VAR
6    VAR_INPUT
7       S1, S2, O1, O2:BOOL;
8    END_VAR
9    VAR_OUTPUT
10      Output:BOOL:=FALSE;
11   END_VAR
12
13      LD      Step_0
14      AND     S1
15      AND     S2
16      ANDN    O1
17      ANDN    O2
18      R       Step_0
19      S       Step_1
20      LD      Step_1
21      S       Output
22      LD      Step_1
23      AND (   O1
24      OR      O2
25      ORN     S1
26      ORN     S2
27      )
28      R       Step_1
29      S       Step_2
30      LD      Step_2
31      R       Output
32      LD      Step_2
33      ANDN    S1
34      ANDN    S2
35      AND     O1
36      AND     O2
37      R       Step_2
38      S       Step_0
39   END_PROGRAM
```

4.1 Enviroment Model

First we model the plant. In this example the plant are the two switches. The model is shown in Figure 6.

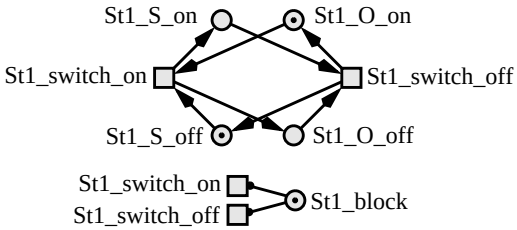


Figure 6 A Petri net model for one switch with blocking place

In this figure we see a place St1_block. This place is a blocking place, which prevents the model of the plant to perform any action until this place is unmarked. This we need to solve the general requirement on a PLC. During the runtime of a PLC user program, no changes in the plant are possible.

4.2 System Program

Each PLC program is included in a PLC system. Such a system has a system program which transforms the physical date from the plant into data which can be used by the program. The system program maps external values in internal values of variables. The system program is shown in Figure 7, and the mapping in Figure 8.

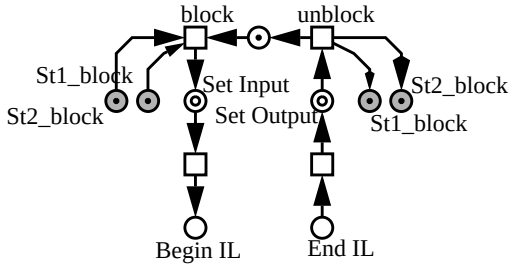


Figure 7 System program with plantblocking and valuemapping

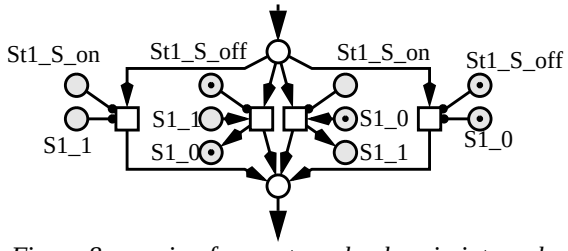


Figure 8 mapping from external values in internal

In the section below we talk about a blocking place. In Figure 7 you can see a transition named block. This transition removes the token from the blocking place. In fact of this, no transition in the enviroment model has concession. After the execution of the user program, the transition named unblock puts tokens back to the blocking places. Now the enviroment is able to change its state. No change in the enviroment model is possible, during the execution of the user program.

4.3 User Program

In section Section 3.3 we said, that we need an execution of the function $\mathscr{IL}_{\mathscr{O}}$ with a specific IL_0 program p. In the case of our example this program is our IL-program. The result of executing $e = \mathscr{IL}_{\mathscr{O}}(p)$ is given in Figure 9. Here you see, that all parent structures are substituted. Now we translate the result in a Petri net (Figure 10). The first place of this net is `Begin IL` an the last is `End IL`. The other places are the subnets from the Petri net semantic for the related command. The connection places (in the Petri net semantic the places labelled with i) are now labelled with the position of the command in $Code$. Places, except the logical, with the same names, are merged. As a result of this composition we have a sequence of subnets. Each subnet represents a command, and the position in the sequence represents the position of this command in the source code. This structure of the net makes it easy to interpret results from the analysis of the Petri net in terms of the source code.
The initial marking of the Petri net is also given by the result tupel e from $\mathscr{IL}_{\mathscr{O}}$. The part Env_0 describes the used variables with types (number of places) and initial values (marking of this places).

$\mathscr{IL}_{\mathscr{O}}(switch)$={⟨switch, ⟨**prg**, **n**⟩,
{⟨`Step_0`, **rw**, ⟨**bool**, **T**⟩⟩,
⟨`Step_1`, **rw**, ⟨**bool**, **F**⟩⟩,
⟨`Step_1`, **rw**, ⟨**bool**, **F**⟩⟩,
⟨`S1`, **r**, ⟨**bool**, **F**⟩⟩,
⟨`S2`, **r**, ⟨**bool**, **F**⟩⟩,
⟨`O1`, **r**, ⟨**bool**, **F**⟩⟩,
⟨`O2`, **r**, ⟨**bool**, **F**⟩⟩,
⟨`Ausgang`, **rw**, ⟨**bool**, **F**⟩⟩,
⟨`A`, **rw**, ⟨**bool**, **F**⟩⟩,
⟨`B`, **rw**, ⟨**bool**, **F**⟩⟩ }, ∅,
[⟨(`LD Step_0`), **bool**⟩, ⟨(`AND S1`), **bool**⟩,
⟨(`AND S2`), **bool**⟩, ⟨(`ANDN O1`), **bool**⟩,
⟨(`ANDN O2`), **bool**⟩, ⟨(`R Step_0`), **bool**⟩,
⟨(`S Step_1`), **bool**⟩, ⟨(`LD Step_1`), **bool**⟩,
⟨(`S Output`), **bool**⟩, ⟨(`LD Step_1`), **bool**⟩,
⟨(`ST A`), **bool**⟩, ⟨(`LD O1`), **bool**⟩,
⟨(`OR O1`), **bool**⟩, ⟨(`ORN S1`), **bool**⟩,
⟨(`ORN S2`), **bool**⟩, ⟨(`ST B`), **bool**⟩,
⟨(`LD A`), **bool**⟩, ⟨(`AND B`), **bool**⟩,
⟨(`R Step_1`), **bool**⟩, ⟨(`S Step_2`), **bool**⟩,
⟨(`LD Step_2`), **bool**⟩, ⟨(`R Output`), **bool**⟩,
⟨(`LD Step_2`), **bool**⟩, ⟨(`ANDN S1`), **bool**⟩,
⟨(`ANDN S2`), **bool**⟩, ⟨(`AND O1`), **bool**⟩,
⟨(`AND O2`), **bool**⟩, ⟨(`R Step_2`), **bool**⟩,
⟨(`S Step_0`), **bool**⟩] }

Figure 9 Result of the parsing function

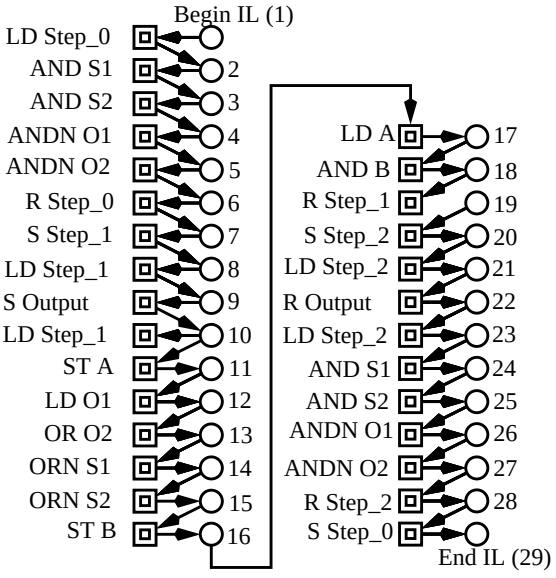


Figure 10 User program as a Petri net

4.4 Composition to the System Model

At this point we have a system program as as Petri net and the user program as a Petri net. Now we have to compose these two parts, because we need a Petri net for the whole controller to the analysis. This composition goes over the places `Begin IL` and `End IL`. The result is shown in Figure 11. The subnets represent the parts of the PLC-

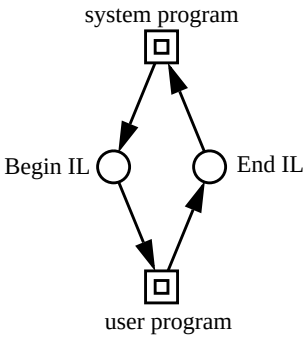


Figure 11 connection between system program and user program

controller.
In the figure we dont have represent the enviroment model, because it is implicit given in the Petri net of the system program. There would be realized the mapping from the external values to internal values of variables.
The numbers of places and transitions of the whole net is shown in Table 2.

Part	Places	Transitions
user program	45	126
system program	5	12
enviroment model	8	4
	58	142

Table 2 number of places and transitions

The Petri net is safe, and live, when dead transition under the initial marking are removed. Its reachability graph has 551 states. This net can be used to prove temporal logic formualae with a modelchecker.

5 Summary and Further work

In this paper we described a method how a PLC controller, whose userprogram is written in IL_0, can be translated into a Petri net. The Petri net has no special elements with a new semantic. The used elements have only syntactical nature. At this point the Petri net can be analysed with most of the analysis methods, because it is a safe place/transition net and the most analysing methods work fine with this class of Petri nets. The Petri net can be used as a simulation of a execution of a PLC, too.
In further works we like to implement this transformation. Also the language IL_0 will be extended to the whole language IL. In the area of analysing, we investigate the possibilities to formulate the requirements from the standardisation papers in formulas of temporal logic. Finally we like to extend this method to multiprocessor and multitasking systems, where the Petri net based design and analysis methodology gives its best results.

Bibliography

[DIN EN 61131-3] DIN EN 61131-3 (IEC 1131-3) **Speicherprogrammierbare Steuerungen, Teil 3: Programmiersprachen** 1993

[DIN EN 574] DIN EN 574 (EN 574) **Sicherheit von Maschinen, Zweihandschaltung** 1992

[Hanisch97] H.-M. Hanisch, J. Thieme, A. Lüder, O. Wienhold **Modeling of PLC Behavior by Means of Timed Net Condition / Event Systems** Otto-von-Guericke University of Magdeburg, Magdeburg, 1997

[Heiner97-1] M. Heiner, H. Meier, T. Menzel, T. Mertke **Petri-Netz-basierte Methoden zur sicherheitstechnischen Zertifizierung von SPS-Anwenderprogrammen** BTU-Cottbus Reihe Informatik I-19/ 1997, Cottbus December 1997

[Heiner97-2] M. Heiner, T. Menzel **Petri-Netz-Semantik für die SPS-Anwenderprogrammiersprache Anweisungsliste.** BTU-Cottbus Reihe Informatik I-20/ 1997, Cottbus December 1997

[John95] Karl-Heinz John, Michael Tiegelkamp **SPS-Programmierung mit IEC 1131-3** Springer-Verlag Heidelberg, 1995

[Rausch96] Mathias P. Rausch **Modulare Modellbildung, Synthese und Codegenerierung ereignisdiskreter Steuerungssysteme** VDI Verlag Düsseldorf, 1997

[Rausch97] M. Rausch, B. Krogh **Transformations Between Different Model Forms in Discrete Event Systems** IEEE Int. Conf. on Systems, Man, and Cybernetics, Orlando, Florida, USA 12.-15. 10. 1997