*Parse*(*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**⟩, ⟨(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**⟩, ⟨(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 6: Result of the parsing function.**

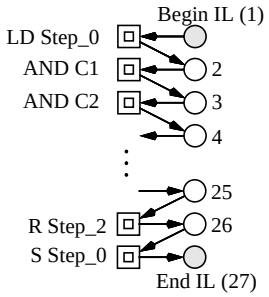



**Figure 7: User program as a Petri net.**

This structure of the net makes it easy to interpret any results of the Petri net analysis in terms of the source code.

The initial marking of the Petri net is also given by the result tuple $e$ from *Parse*. The part $Env_0$ describes the used variables with types (number of places) and initial values (marking of these places).

The Petri net for the user program of the 2-hand switch PLC consists of 45 places and 126 transitions, the total net including the environment model comprises 58 places and 142 transitions. For more details, including the model of the environment and of the system program, see [Heiner 98b]. More challenging case studies (production cells comprising several machines) are under preparation.

## 5. Conclusions and Related Work

Computer-aided verification should rely on verified tools, or at least on tools developed as careful as possible. For that reason, a Petri net semantics has been introduced formally for a subset of the standardized Instruction List language [IEC 1131-3]. The subset's syntax as well as static and operational semantics have been specified strictly. Afterwards, the classical operational reference semantics has been substituted by an equivalent Petri net semantics. Due to this prudent practice, the equivalence proof of the substitution step is quite obvious, and the implementation of the translator has been proven to be straightforward. To stress the analyzability of the generated Petri nets, the net semantics has been described completely in terms of ordinary safe place transition Petri nets. The used graphical enhancements are just syntactical sugar. Due to the structural regularities, many net components might be folded to coloured Petri nets.

Related work may be found in [Hanisch 97], where IL programs are modelled by Timed Net Condition/Event Systems (TNCES). The advantage of TNCES is their dedicated modelling power, which excludes however (at least up to now) most of the sophisticated analysis options. [Rausch 97] discusses generally the transformation between different model forms in discrete event systems.

In the future we intend to weaken the $IL_0$ restrictions step-wise into the direction of the whole language IL. This will include multiprocessor and multitasking systems, where the Petri net based design and analysis methodology promises its best results. Moreover, the incorporation of other standardized PLC languages might be considered (see e. g. [Heiner 97a] for the transformation of Sequential Function Charts into Petri nets).

## References

**[IEC 1131-3]**
IEC Standard 1131-3, Programmable controllers - Part 3: Programming languages; Int. Electrotechnical Commission, 1993.

**[Hanisch 97]**
Hanisch, H.-M. et al.: Modeling of PLC Behavior by Means of Timed Net Condition / Event Systems; Proc. 6th IEEE Int. Symposium on Emerging Technologies and Factory Automation (ETFA '97), Los Angeles, Sept. 1997, pp. 361-396.

**[Heiner 97a]**
Heiner, M.: On Exploiting the Analysis Power of Petri nets for the Validation of Discrete Event Systems; Proc. 2nd IMACS Symposium on Mathematical Modelling (MATHMOD VIENNA '97), Vienna, February 1997, ARGESIM Report No. 11, pp. 171-176.

**[Heiner 97b]**
Heiner, M.; Menzel, T.: Petri Net Semantics for the PLC User programming language Instruction List (in German); Techn. Report BTU Cottbus, I-20/1997, Cottbus December 1997.

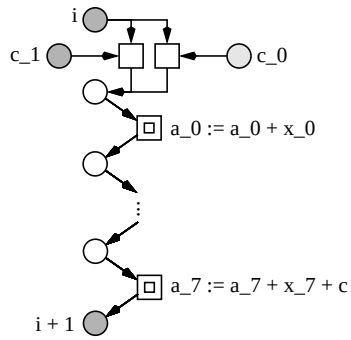
**[Heiner 98a]**
Heiner, M.: Petri Net Based System Analysis without State Explosion; Proc. High Performance Computing '98, Boston, April 1998, SCS Int. San Diego 1998, pp. 394 - 403.

**[Heiner 98b]**
Heiner, M.; Menzel, T.: A Petri Net Semantics for the PLC Language Instruction List; IEE Workshop on Discrete Event Systems (WODES '98), Cagliari/Italy, August 1998.

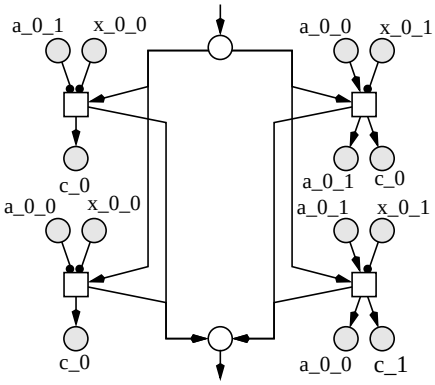
**[Mosses 90]**
Mosses, P. D.: Denotational Semantics; in: Leeuwen, J. v. (ed.): Handbook of Theoretical Computer Science; Elsevier Sc. Publ. 1990, pp. 576-631.

**[Rausch 97]**
Rausch, M.; Krogh, B.: Transformations Between Different Model Forms in Discrete Event Systems; Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics, Orlando, October 1997, pp. 2841-2846.

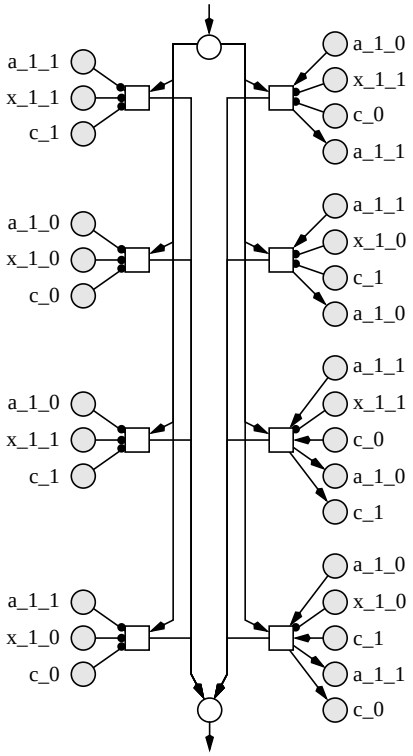
(3) $Compute_e(i, \texttt{ADD}\ x)$=




a_0 := a_0 + x_0




a_[1-7] := a_[1-7] + x_[1-7] + c




## 4. EXAMPLE

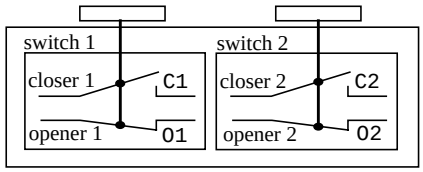



**Figure 5: Structure of a 2-hand switch.**

To illustrate the whole transformation process, let's consider a small, but realistic example. Many safety-oriented equipment like presses use 2-hand switches, which require the operation by both hands. The output is *on* if switch 1 and switch 2 are pressed at the same time (without a significant delay in between). The closers work complementary to the related openers (closer 1 to opener 1 and closer 2 to opener 2). The principle construction is shown in figure 5. A possible coding of the intended PLC behaviour 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        C1, C2, O1, O2:BOOL;
8     END_VAR
9     VAR_OUTPUT
10       Output:BOOL:=FALSE;
11    END_VAR
12
13       LD       Step_0
14       AND      C1
15       AND      C2
16       ANDN     O1
17       ANDN     O2
18       R        Step_0
19       S        Step_1
20       S        Output
21       LD       Step_1
22       AND (    O1
23       OR       O2
24       ORN      C1
25       ORN      C2
26       )
27       R        Step_1
28       S        Step_2
29       R        Output
30       LD       Step_2
31       ANDN     C1
32       ANDN     C2
33       AND      O1
34       AND      O2
35       R        Step_2
36       S        Step_0
37    END_PROGRAM
```

The result of executing $e = Parse(p)$ is given in figure 6. Please note that all bracket structures have been substituted.

Based on the parsing result, we generate now a Petri net (see figure 7). The first place of this net is *Begin IL* and the last one is *End IL*. The other places belong to the subnets of the Petri net semantics for the corresponding commands. These connection places (in the Petri net semantics the places labelled with $i$) are now labelled with the position of the command in $Code$, whereby places with the same name are merged. As a result of this composition we get a sequence of subnets. Each subnet represents a command, and its sequence position corresponds to the position of this command in the transformed source code.



*Proc. 1998 IEEE Int. Conference on Systems, Man, and Cybernetics,*
*Session "Controller Verification and Design", San Diego, October 1998, pp. 716-721.*

**0-7803-4778-1/98 $ 10.00 © 1998 IEEE**

## Syntactical Conventions

In the following paragraphs, Petri nets with some extensions in the graphical representation are given. E. g. the Petri nets are modelled hierarchically. But here the hierarchy is of syntactical nature only. We don't use any semantic composition. Before analysing these nets, they must be flatten. A summary of the syntactical extensions, used in this paper, is given in table 1.

| | |
|---|---|
|  | *logical place* -- all logical places with the same name collapse to the same place during the flatten process |
|  | *transition bounded subnet* -- subnet where the border nodes (nodes with a connection to the net one level higher) are only transitions |

**Table 1: Syntactical Petri net extensions.**

Additionally, a new kind of edges is introduced. The read arc possesses as a head a black circle instead of an arrow (see figure 2). This kind of edge has always a place as source and a transition as target node. The firing rule is changed in the way that no tokens are moved from/to the source place via a read arc. The place can be interpreted as a side condition of the transition, which do not restrict concurrency. Under the interleaving semantics, these arcs can be transformed into two usual arcs (one for each direction between place and transition).

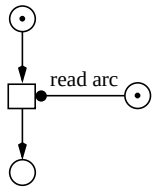



**Figure 2: Read arc example.**

The resulting Petri net components are quite large and exhibit many repetitions. Due to this fact we will use the following conventions in order to save space. If a subnet structure would be $n$ times repeated, we write the first and the last subnet only, and mark the repetition with three dots. We choose intuitive node names for structurally equal subnets. An example for this compression is shown in figure 3.

(Remark: to be conform to all bit-oriented operations, working step-wise from the lowest to the highest bit, the concurrent nature of word instructions, e. g. assignments, is neglected here.)

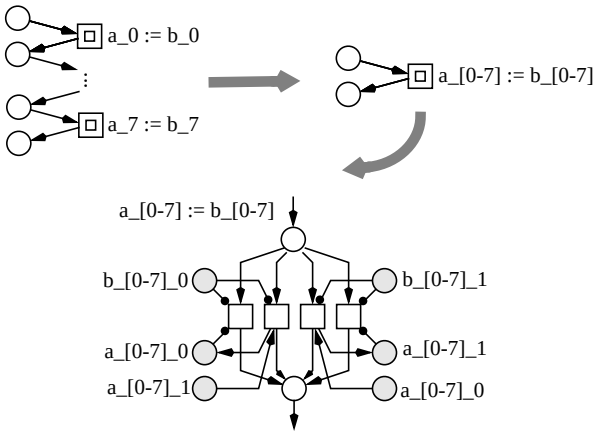



**Figure 3: Example of repetitions of a subnet.**

## Binary Representation

We are modelling variable values in a binary form, where each bit is represented by two places forming a place invariant (see figure 4). Generally, this approach works more efficiently than

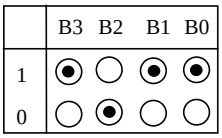



**Figure 4: Example of a 4-bit number ($11_{10} = 1011_2$).**

those way of modelling, where each variable is represented by as many places as there are values which the variable may assume. Now we are ready to model the algorithms of binary algebra, and by this way to formulate the Petri net semantics of $IL_0$ in the next paragraph.

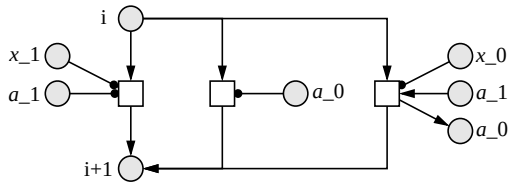
## Petri Net Semantics

In the following we will give a Petri net semantics for the three examples of IL statements from above. Based on these examples, it should become fairly obvious that the definition of the other Petri net components is straightforward.

As precondition we need (like before) a successful execution of the function `Parse` with the result in $e$. Therefore we have - in analogy to the previous operational semantics - a function $Compute_e$, too.

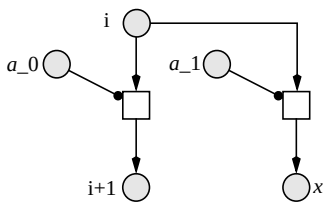
After the execution of the function `Parse` we have just to parse sequentially the ordered list of $Code$, and for each statement in $Code$ we insert an appropriate Petri net component. The label $i$ is substituted by the line number, and $x$ is substituted by the name of the operand. The composition of the net components (each command has its own) results into the model of the whole $IL_0$ program. This composition works simply sequentially and is realized via the logical place mechanism.

(1) $Compute_e(i,$ `AND` $x)=$




Please note: the middle transition summarizes two of the four value combinations of the two Boolean operands.

(2) $Compute_e(i,$ `JMPC` $x)=$



semantics definition, we just quote - without any comments - the functions of the remaining syntax rules given above.

$$Prg: \text{PRG} \to (FbNames \to Env)$$
$$Prg[\![\texttt{PROGRAM PRGNAME PRGVARDECL CODE END\_PROGRAM}]\!](n) = \langle [\![\text{PRGNAME}]\!], \langle \mathbf{prg}, \mathbf{n} \rangle \rangle$$
$$PrgVarDecl[\![\text{PRGVARDECL}]\!](n), CodeLabel[\![\text{CODE}]\!], Code[\![\text{CODE}]\!][2]\rangle$$

$$PrgVarDecl: \text{PRGVARDECL} \to (FbNames \to Env_0)$$
$$PrgVarDecl[\![v_1, \ldots, v]\!](n) = \bigsqcup_{i=1}^{k} PrgDeclCon[\![v_i]\!](n)$$

$$Code: \text{CODE} \to (Env \pounds Domain \to Bool \pounds Env)$$
$$Code[\![s_1 \hookleftarrow s_2 \hookleftarrow \ldots s_k]\!](e, d) =$$
$$\langle \wedge_{i=1}^{k} Stmt[\![s_i]\!](e, d)[1], \bigsqcup_{i=1}^{k} Stmt[\![s_i]\!](e, d)[2] \rangle$$

$$Stmt: \text{STMT} \to (Env \pounds Domain \to Bool \pounds Env)$$
$$Stmt[\![\text{BOOLOP}]\!](e, d) = BoolOp[\![\text{BOOLOP}]\!](e, d)$$
$$Stmt[\![\text{ANYOP}]\!](e, d) = AnyOp[\![\text{ANYOP}]\!](e, d)$$
$$Stmt[\![\text{BROP}]\!](e, d) = BrOp[\![\text{BROP}]\!](e, d)$$
$$Stmt[\![\text{JMPOP}]\!](e, d) = JmpOp[\![\text{JMPOP}]\!](e, d)$$
$$Stmt[\![\text{CALLOP}]\!](e, d) = CallOp[\![\text{CALLOP}]\!](e, d)$$
$$Stmt[\![\text{RETOP}]\!](e, d) = RetOp[\![\text{RETOP}]\!](e, d)$$

After the evaluation of the function Parse for a given $IL_0$

$$BoolOp: \text{BOOLOP} \to (Env \pounds Domain \to Bool \pounds Env)$$
$$BoolOp[\![\text{BOOLCMD OPERAND}]\!](e, d) =$$

$$\begin{cases} \langle \mathbf{T}, \langle Name(e), d, Env_0(e), Label(e), \\ \quad \langle Code(e) \cdot \langle (\text{BOOLCMD OPERAND}), \{\mathbf{bool}\} \rangle \rangle \rangle \\ \quad \textit{if } \mathbf{bool} \in DType[\![\text{OPERAND}]\!](Env_0(e)) \wedge \\ \quad last(Code(e))[2] \wedge \{\mathbf{bool}\} \wedge \\ \quad Access[\![\text{OPERAND}]\!](Env_0(e)) \in \{\mathbf{r}, \mathbf{rw}\} \wedge \\ \quad [\![\text{BOOLCMD}]\!] \in \{\texttt{AND}, \texttt{ANDN}, \texttt{OR}, \texttt{ORN}, \texttt{XOR}, \texttt{XORN}\} \\ \langle \mathbf{T}, \langle Name(e), d, Env_0(e), Label(e), \\ \quad Code(e) \cdot (\text{BOOLCMD OPERAND}), \{\mathbf{bool}\} \rangle \rangle \\ \quad \textit{if } \mathbf{bool} \in DType[\![\text{OPERAND}]\!](Env_0(e)) \wedge \\ \quad Access[\![\text{OPERAND}]\!](Env_0(e)) \in \{\mathbf{r}, \mathbf{rw}\} \wedge \\ \quad [\![\text{BOOLCMD}]\!] \in \{\texttt{LDN}\} \\ \langle \mathbf{T}, \langle Name(e), d, Env_0(e), Label(e), \\ \quad \langle Code(e) \cdot \langle (\text{BOOLCMD OPERAND}), \{\mathbf{bool}\} \rangle \\ \quad \textit{if } \mathbf{bool} \in DType[\![\text{OPERAND}]\!](Env_0(e)) \wedge \\ \quad last(Code(e))[2] \quad \{\mathbf{bool}\} \\ \quad Access[\![\text{OPERAND}]\!](Env_0(e)) \in \{\mathbf{w}, \mathbf{rw}\} \wedge \\ \quad [\![\text{BOOLCMD}]\!] \in \{\texttt{STN}, \texttt{S}, \texttt{R}\} \\ \langle \mathbf{F}, \langle Name(e), d, Env_0(e), Label(e), Code(e), \langle ?, ? \rangle \\ \quad \langle ?, ? \rangle \rangle \\ \quad otherwise \end{cases}$$

program $p$, $Code$ consists of a flat (without procedure and function calls) $IL_0$ program without any brackets and with explicit jump goals (line numbers instead of labels). The resulting strictly sequential $IL_0$ program is correct with respect to its syntax and static semantics.

### Operational Semantics

After parsing and context checking successfully, we get a simple list of commands. Now we are able to define conveniently an operational semantics describing the transitions between program states. Such a program state is defined as tuple $(i, m, a)$, 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 current value and $t$ its type (in the following $a.v$ stands shortly for the value, and $a.v$ for the type).

Let $p$ be an $IL_0$ program, and $e$ the environment obtained by the successful evaluation of $e = Parse(p)$. Now we define for each command a state transition under $e$. Let's give three examples:

(1) $(i, m, a) \overset{\texttt{AND}\ x}{\underset{e}{\longrightarrow}} (i + 1, m, (m(x) \wedge a.v, a.t))$

The result of the logical operation *and*, applied to the accumulator value and the value of the operand $x$, yields the new value of the accumulator.

(2) $(i, m, \langle \mathbf{T}, \mathbf{bool} \rangle) \overset{\texttt{JMPC}\ x}{\underset{e}{\longrightarrow}} (x, m, a)$

$(i, m, \langle \mathbf{F}, \mathbf{bool} \rangle) \overset{\texttt{JMPC}\ x}{\underset{e}{\longrightarrow}} (i + 1, m, a)$

If the accumulator value is TRUE, then jump to the line number $x$, otherwise ignore the jump and start with the execution of the next code line. There are no changes in the accumulator.

(3) $(i, m, a) \overset{\texttt{ADD}\ x}{\underset{e}{\longrightarrow}} (i + 1, m, (m(x) + a.v, a.t))$

The result of the integer operation *addition*, applied to the accumulator value and the value of the operand $x$, yields the new value of the accumulator.

While defining the operational semantics we don't need any type check, because this has been done during the execution of the function Parse. In the forthcoming section we describe this polished operational semantics by Petri nets.

## 3. PETRI NET SEMANTICS

In the section above, a formal definition of the language $IL_0$ has been introduced. Now we define the operational semantics as Petri net substituting the classical reference semantics.



*Proc. 1998 IEEE Int. Conference on Systems, Man, and Cybernetics,*
*Session "Controller Verification and Design", San Diego, October 1998, pp. 716-721.*

program into a Petri net, which is highlighted by a grey background in figure 1. To keep the paper within the given limits, we do not claim to give here complete and detailed formal definitions. The interested reader is referred to [Heiner 97b] for a self-contained description of $IL_0$ and its Petri net semantics.

## 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 several libraries with additional commands, the complexity of IL is quite excessive. Opposed to that, the language $IL_0$ exploits some restrictions, the essential ones are:

- default commands only
  (i. e. the complete statement set of IL - altogether 47 commands, but no additional commands from any library),
- data types to a length of 8 bit only
  (Boolean, 8-bit word, unsigned short integer, short integer),
- no commands and data types for time and date,
- single processor / single task only.

The basic intention of the chosen language restrictions is to start with simpler problems before going ahead to more complicated ones. $IL_0$ is already powerful enough for many realistic PLC programs. Nevertheless, a step-wise weakening - as far as possible - of the imposed simplification is under consideration.

In the next paragraphs we follow the path of a step-wise formal definition of $IL_0$. At first we specify an abstract syntax. After that, we define a static semantics in the style of a denotational one. By this way, the allowed semantic context of all language constructs is specified strictly. Finally, we give an operational semantics of $IL_0$ in a classical style as reference semantics which will be substituted later by a Petri net based one.

### Abstract Syntax of $IL_0$

At first we need an abstract syntax to define strictly the syntactical frame of $IL_0$. An abstract syntax looks like a concrete one, with the exception that they differ in the degree of details. The abstract syntax ends with nonterminals like <PRGNAME>, leaving them unspecified, if the details are irrelevant for the intended purpose of the syntax. The abstract syntax of $IL_0$ is formulated in Extended Backus Nauer Form, consisting altogether of 48 rules. To have concrete examples in mind, we quote six of these rules.

<IL$_0$> → <PRG> {<FB> | <FCT>}

<PRG> → 'PROGRAM' <PRGNAME>
<PRGVARDECL>
<CODE>
'END_PROGRAM'

<PRGVARDECL> → <FBINSTDECL> | <INVARDECL> |
<OUTVARDECL> | <GLOBALVARDECL>

<CODE> → {<STATEMENT>}

<STMT> → <BOOLOP> | <ANYOP> | <BROP> |
<JMPOP> |<CALLOP> | <RETOP>

<BOOLOP> → <BOOLCMD> <OPERAND>

Any syntax describes possible derivation trees with nonterminals as root. The so-called starting rule (the first one given above) describes the derivation tree of the whole $IL_0$ program. According to this rule, an $IL_0$ program consists at least of a program part, possibly followed by an arbitrary set of function blocks and functions in any order.

### Static Semantics

In the paragraph above, a syntactical frame of $IL_0$ has been introduced. Now we are ready to define context conditions in a denotational style [Mosses 90]. For that purpose, we are going to specify a function for each nonterminal of the given syntax to check its semantic context. Additionally, these functions perform some semantics-preserving program code transformations lightening the following operational semantics definition.

Each nonterminal describes a derivation tree. Therefore, each function to be defined has (at least) a tree as parameter and returns (at least) a tuple $Env$, describing the total environment of the program (all variables and labels) as well as the program code itself (but without any variable declarations).

The main function is:

$$\mathtt{Parse}: IL_0 \to Env$$
$$\mathtt{Parse}(p) = \mathtt{ChangeMark}(\mathtt{Flat}(\mathtt{IL}_0(p)))$$

where $p$ is a syntactical correct $IL_0$ program. The result of the function `Parse` is the tuple

$Env = Name \times Domain \times Env_0 \times Label \times Code$, with

- $Name$ - name of the program
- $Domain = \{\mathbf{prg}, \mathbf{fb}, \mathbf{fct}\} \times \{\mathbf{n}, \mathbf{b}\}$, where
  **prg** - program,
  **fb** - function block,
  **fct** - function,
  **n** - no brackets, and
  **b** - brackets
- $Env_0$ - tuple for variables
- $Label$ - set of used line labels
- $Code = [s_0, s_1, s_2, \ldots, s_n]$ - the ordered list of code, where $s_i$ are program statements

All procedure and function calls are substituted by the procedure `Flat`All labels in jump statements are replaced by the corresponding line numbers by the procedure `ChangeMark`

The function $\mathtt{IL}_0$ corresponds to the starting rule of the abstract syntax above.

$$\mathtt{IL}_0: IL_0 \to Env$$
$$\mathtt{IL}_0[\![prg\,fb_1 \ldots fb_k\, fct_1 \ldots fct_n]\!] =$$
$$\mathtt{Prg}[\![prg]\!](FbNames([\![prg\,fb_1 \ldots fb_k\, fct_1 \ldots fct_n]\!])) \left[ \left( \bigsqcup_{i=1}^{k} \mathtt{Fb}[\![fb_i]\!] \right) \left[ \left( \bigsqcup_{i=1}^{n} \mathtt{Fct}[\![fct_i]\!] \right) \right. \right.$$

During the evaluation of $\mathtt{IL}_0$ any bracket structure (`AND(`, `ANDN(`, `GE(`, `...`) is resolved by inserting a new variable, which buffers the accumulator value at the beginning of that bracket structure, and which is stored back at the end of the bracket structure.

To give at least a flavour of the denotational style of static

# Instruction List Verification Using a Petri Net Semantics *)

**Monika Heiner, Thomas Menzel**

Brandenburg University of Technology at Cottbus
Department of Computer Science
Postbox 10 13 44, D - 03013 Cottbus, Germany

mh@informatik.tu-cottbus.de, thm@informatik.tu-cottbus.de
http://www.informatik.tu-cottbus.de
Phone: (+ 49 - 355) 69 - 3885, Fax: (+ 49 - 355) 69 - 3830

## ABSTRACT

In order to adapt a Petri net based verification framework to programmable logic controllers, a Petri net semantics is introduced formally for a subset of the standardized Instruction List language [IEC 1131-3]. For that purpose, the subset's syntax as well as static and operational semantics are specified strictly. Having that, the operational reference semantics is substituted by an equivalent Petri net semantics. Due to this prudent practice, the equivalence proof of the substitution step is obvious.

## 1. MOTIVATION

The development of provably error-free discrete event controllers is still a challenge of practical system engineering - in spite of encouraging results of world-wide promoted academic research on formal methods. This paper aims at decreasing the still existing gap between current practice and promising theory.

In current practice, controllers are often implemented by Programmable Logic Controllers (PLC). Up to now, the Instruction List (IL) is one of the favourite programming languages of PLC design. The instruction list language is part of the international standard IEC 1131-3, moreover several in-house variations exist in outstanding companies.

On the other side, Petri net theory offers in the meantime a rich amount of promising sophisticated analysis techniques alleviating the state explosion problem. Related tools have reached, at least partly, an acceptable standard. For an overview and experience report, demonstrating the Petri net's analysis power by case studies, see e. g. [Heiner 98a]. To get the richest amount of analysis facilities, we restrict ourselves in the following to ordinary safe (1-bounded) place transition Petri nets.

In order to be able to apply a Petri net based verification framework on IL-written PLC, we introduce formally a Petri net semantics for the international standard's IL, which may be adopted very easily to any of its variations. For that purpose, we define an IL subset (the $IL_0$), comprising only default commands and data structures up to a length of 8 bits. Based on an abstract syntax of $IL_0$, we give its static semantics in the style of a denotational one to specify context conditions, and its operational semantics in a classical style playing the role of a reference semantics. Having that, the classical operational reference semantics is substituted by an equivalent Petri net semantics. Due to the efforts done up to here, the equivalence proof of the substitution step is obvious. Moreover, based on this formal semantics definition, the realization of an automatic IL translation into Petri nets has been proven to be straightforward.

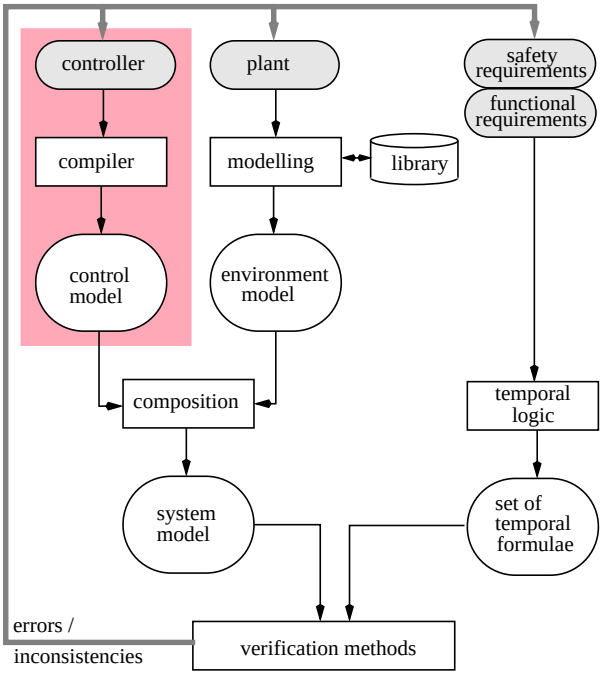



**Figure 1: Model-based validation of PLC software (e. g. by Petri nets).**

To get a self-contained verifiable system model, two additional aspects have to be taken into account: at first the embedding of the generated Petri net model of the IL user program into a PLC's system program model, and the interconnections between the models of the controller program and of the uncontrolled plant. For a discussion of both aspects in the context of a case study see [Heiner 98b].

After this translation, we are able to prove functional and safety requirements, the PLC is expected to fulfil, by means of the generated Petri net. For that, the requirements have to be specified by formulae of temporal logics. Afterwards, model checkers may be used for verification/falsification of these formulae in the net. Figure 1 shows the principle cycle of the model-based verification of PLC user programs.

In this paper, we restrict ourselves to the transformation of an IL

*) This work is supported by the German Research Council under grant ME 1557/1-1.



*Proc. 1998 IEEE Int. Conference on Systems, Man, and Cybernetics,*
*Session "Controller Verification and Design", San Diego, October 1998, pp. 716-721.*

**0-7803-4778-1/98 $ 10.00 © 1998 IEEE**