Partial Order Verification of Programmable Logic Controllers*

Peter Deussen

Brandenburg University of Technology at Cottbus
Computer Science Institute
Cottbus, Germany
PDeussen@gmx.de,
http://www.Informatik.TU-Cottbus.DE/~wwwdssz/

Abstract. We address the verification of programmable logic controllers (PLC). In our approach, a PLC program is translated into a special type of colored Petri net, a so-called register net (RN). We present analysis methods based on the partial order semantics of RN's, which allow the generation of partial order traces as counter examples in the presence of programming errors. To that purpose, the behavior description 'concurrent automaton', introduced in [3] for safe Petri nets, is upliftet to the dedicated RN's.

1 Introduction

In this paper, we address the verification of industrially applied controllers. We concentrate on software for *programmable logic controllers* (PLC). The international norm IEC 1131-3 [10] defines several languages for PLC programming: *sequential function charts*, *structured text*, *ladder diagrams*, *function block diagrams*, and—most elementary—an assembler-like language called *instruction list* (IL), on which we will focus here.

Let us outline the main ideas:

Organization of the verification process. An IL program is compiled into a dedicated type of colored Petri net, called register net (RN). RN's represent the control flow of an IL (i. e. the order of computation of program parts as determined by jumps and labels) by means of a Petri net (to be precise, an elementary net system (ENS)), and the data flow (memory and hardware addresses and accumulators) by registers containing non-negative integers. Transitions can read and modify data, and their occurrence can depend on data.

The verification process is done as follows: After compiling the IL program into a RN, a model of the environment of the PLC is added to the resulting net, i. e. a model of the controlled facility (also in form of a RN). Such an environment model is necessary because of the following reason: Sensor values tested by the

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

J.-M. Colom and M. Koutny (Eds.): ICATPN 2001, LNCS 2075, pp. 144–163, 2001.
© Springer-Verlag Berlin Heidelberg 2001

PLC change because of actuator actions influenced by the PLC. This behavior has to be captured in some way.

Additionally, we want also to be able to deal with (parts of) plants comprising several facilities and associated PLC's. PLC's of different machines communicate in an implicit manner by the changes of sensor values: If a transport belt is activated by a PLC, a light barrier associated with some other PLC will observe the passing of the transported blank.

The analysis will be done on the composed model consisting of the RN of the environment and the RN's of the several PLC's. In this paper, we focus on three types of system properties:

1. absence/presence of deadlocks (which is merely a side effect of our analysis technique),
2. run-time errors like overflows and division by zero, and
3. simple safety properties. A simple safety property is an assumption on data values describing system states which are not allowed to occur.

Mathematical Considerations. Since RN will be used for analysis purposes, a mathematically sound model of the behavior of a RN is needed. We prefer a partial order semantics instead of the more familiar interleaving semantics for the following reason: Interleaving semantics is based on the notion of sequences of transition occurrences acting on a global state space of the system. In terms of PLC's: A global state is the product of the states of all the PLC's in the plant. Because concurrency and causality are not visible in sequences, such semantics gives the somewhat misleading picture that anything in a plant has to do something with everything else. But in a real plant, many processing steps can be (and are) performed independently of other processing steps.

In using partial order semantics, independence and causality become visible. The idea is to model causal dependence and independence of system actions by the mathematical concept of a partial order. If e_1 and e_2 are system actions (elementary processing steps or the computation of a single command in an IL program), then we use the notation $e_1 < e_2$ to indicate that e_1 has to precede e_2 in time, or that e_1 is a necessary precondition of e_2. On the other hand, if neither $e_1 < e_2$ nor $e_2 < e_1$ holds, then e_1 and e_2 are concurrent or independent of each other. We write $e_1 \text{ co } e_2$.

There are many types of partial order semantics of Petri nets, for instance Mazurkiewicz traces, (prime) event structures, pomsets (partial words), or (branching) processes. We use so-called semi-words [11,15], which are basically partial orders of system actions.

Analysis Technique. For colored Petri nets (and therefore, for RN's), several analysis techniques are available, e. g. place invariant analysis or reachability graph (state graph) generation. Additionally, methods of the classical Petri net theory can be applied to the unfolding of the RN.

We focus on an alternative approach, namely the description of the behavior of a RN by means of a *concurrent automaton* (CA). CA are basically state

graphs, however, with transitions comprising partial order representations of parts of the system behavior.

Our choice is motivated by the following reason: Partial order based techniques have their strength if the system under consideration exhibits a high degree of concurrency, and (more importantly) very few nondeterministic choices. In the given application area nondeterminism can be used to model random events like system faults or human inference. But we expect that the considered systems are 'almost' deterministic. Therefore, it is likely that partial order techniques behave well in the analysis of PLC programs.

Concurrent automata were introduced by Ulrich [12]. Ulrich uses CA for test case generation. A generation algorithm which bases on the input of an *unfolding* of a Petri net [8] is given.

The notion of CA has some similarities to *step covering graphs* [14]. Step covering graphs can be viewed as CA where each transition consists of a semi-order with empty ordering relation, i. e. a step.

Another CA-like approach are *process automata* introduced by Burns and Hulgaard [7]. Process automata comprise global states and transitions labelled by *processes* of safe Petri nets. A *stubborn reduced* reachability graph [13] is used as the input of an generation algorithm.

The paper is organized as follows: In section 2 we examine a very small control problem to explain how IL's are translated into RN's. The example serves only as a motivation for the definition of RN's, not as a running example throughout this paper. To meet the page limit, other examples had to be omitted. Section 3 lists the notions and basic definitions used. In section 4 we introduce RN's formally and define their partial order semantics. Section 5 discusses analysis methods for RN's and introduces CA. In section 6 we present a generation algorithm for CA of a given RN. Simple safety properties and run-time errors are defined in section 7, and a corresponding CA-based analysis algorithm is given. Section 8 summarizes our paper and gives an outlook on further work.

2 Instruction Lists

In this section we discuss a simple controlling problem [9] to motivate our verification method. Figure 1 shows a hydraulic piston. Two valves are used to increase and decrease the pressure of the liquid in the left and right hand part of the piston case. Activation/deactivation of the actors Y_l and Y_r opens/closes the left and the right hand valve, respectively. The sensors X_l and X_r indicate whether the piston has reached its leftmost or rightmost position. Finally, there is a switch to start and stop the piston's movement by a human operator.

The piston is expected to behave as follows: If the operator hits the switch (and keeps pressing it), the piston starts moving until it reaches its leftmost or rightmost position, then it is moving back into the opposite direction. If the operator releases the switch, the piston has to stop.

Figure 2 shows an IL program to solve this simple controlling problem. IL is an assembler-like language. Commands act on variables and on an accumulator.

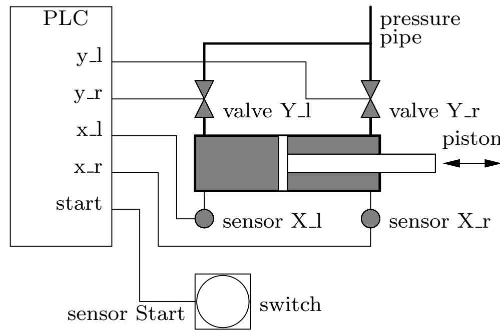

Fig. 1. Hydraulic Piston

For instance, the command LDN z reads: Load the negation of the value of the variable z into the accumulator. S and R are set/reset commands, AND is logical conjunction, and ST means store. Of course, conditional and unconditional jumps are supported by IL, and there are also subroutines in form of functions (procedures without memory) and function blocks (procedures with memory). These 'structured' programming constructs are not allowed to be recursive, i. e. IL's with functions or function blocks can be translated into a formalism with static control structure (e. g. a Petri net). Permitted data types are scalar types like integers, Booleans (the least significant bit (lsb) of a machine word determines its boolean value), or floats.

The processing cycle of a PLC is as follows: In a first step the sensor values of the controlled environment are read and mapped to input variables by the operating system of the PLC. (In our example, X_l is mapped to x_l, X_r to x_r, and so on.) The next step is the execution of the user program. Finally, the

```
VAR_INPUT
  x_1: BOOL;
  x_2: BOOL;
  start: BOOL;
END_VAR
VAR_OUTPUT
  y_1: BOOL := FALSE;
  y_r: BOOL := FALSE;
END_VAR
VAR
  z: BOOL := FALSE;
END_VAR

PROGRAM
  LDN z
  AND x_r
  S z
  LD z
  AND x_1
  R z
  LDN start
  R z
  LD z
  AND start
  ST y_l
  LDN z
  AND start
  ST y_r
END_PROGRAM
```

Fig. 2. An IL User Program

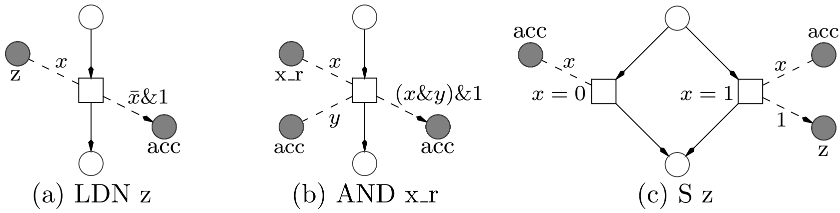

Fig. 3. Register Net Structures for Some IL Commands

values of output variables are mapped to the actuators of the plant. This process is repeated in a fixed time grid, the so-called cycle time.

RN's are Petri nets (ENS) augmented by registers containing non-negative integer values.[1] A transition of a RN is enabled if its pre-places are contained in the current case (marking), its post-places are not (safe firing rule), and if a predicate on register values associated with this transition yields not false (i. e. true or undefined, we will later discuss this point). Additionally, a function on tuples of integers is associated with each transition to determine the effect of the firing of this transition.

Consider fig. 3, 4, or 5 for some examples of RN's. White (unfilled) circles and boxes are places and transitions of the underlying ordinary Petri net, gray shaded circles are registers. A dashed line (without arrow) from a register to a transition means that the value of this register is read by the transition to determine its enabledness and its output values, a dashed arc from a transition to a register identifies this register as an output register. Dashed lines are labeled with variable symbols, dashed arcs are labelled with expressions on these variables—a mechanism adopted from colored Petri nets to define output functions. Predicates are given by Boolean expressions which appear as transition labels. Finally, to avoid edge crossings we fix the convention that the same register may have several graphical appearences.

Figure 3 shows some register net structures associated with several IL commands. Figures 3.(a) and 3.(b) build the RN-semantics of the LDN and AND commands. We use $\bar{x}$ to denote bitwise negation of x, and x & y for bitwise conjunction of x and y. (Recall that the Boolean value of an n-bit integer is determined by its lsb). The set command S (Fig. 3.(c)) and the reset command R are conditional commands: Their arguments get a new value only if the value of acc (the accumulator) is true. We therefore have to use two alternative transitions to describe the semantics for these commands, one which modifies the argument and one which does not: Modification of register clearly defines dependencies between transitions.

Since it is rather obvious, we refrain to give a complete list of RN structures for all IL commands. It should also be easy to imagine how the RN for the user program fig. 2 looks like. Figure 4 shows the RN for the 'operating system' of

[1] Clearly, all the scalar types mentioned above can be traced back to integers.

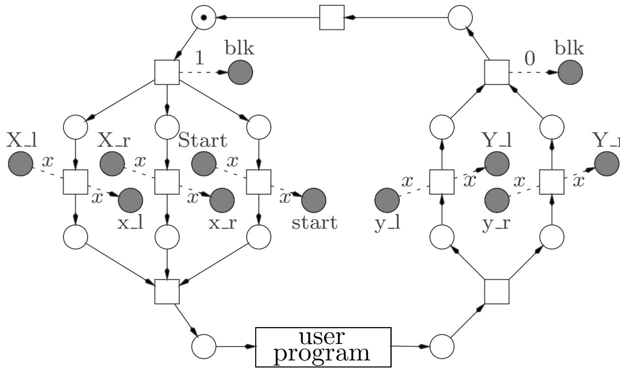

Fig. 4. System Program

a PLC: The system program which maps sensor values to input variables and output variables to actuators. Since we cannot make assumptions on the order in which both mappings occur, we decided to model the mappings as being concurrently executed.

Finally, we have to model the controlled plant (environment model) in order to describe the changes of sensor values in response to the control. In doing this, we are faced with an implicit assumption of any IL program: *The PLC is fast enough to observe any changes of sensor values.* We reflect this assumption by adding a so-called *blocking register* blk to our model: A transition is only allowed to modify a register associated with a sensor if the register blk contains the value 0. This transition modifies the blk register by writing the value 1 into it; therefore, no other transition modifying sensors is allowed to fire until blk is reset to 0. This is done by the RN model of the system program. It also makes sure that the environment model is blocked while the user program is executed.

A very analogous approach to block the environment is described by the authors of [6]: They use safe Place/Transition nets to model both the PLC program and its environment. A blocking place is added to prevent value changing while the PLC program is executed. However, in this approach the environment is never blocked if the PLC program is idle.

Figure 5 shows an environment model for the hydraulic piston. It comprises the basic states r (right hand position), m (middle position), and l (left hand position).

Finally, let us turn to another detail of our modelling approach. We mentioned already that registers contain non-negative integers, which actually means: values of PLC variables. These values cannot be arbitrarily large, but they are restricted to the range $[0..2^m - 1]$, where usually $m = 8$ or $m = 16$ is some constant. What happens if a user program tries to increase such a value beyond $2^m - 1$? Another question is: What happens in the case of dividing by zero? Such ‘undefined behavior’ (even if the programmer knows that $(2^8 - 1) + 1$ equals

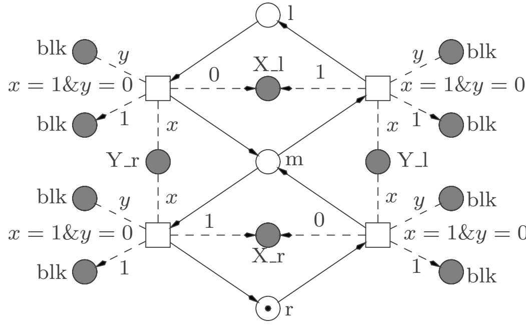

Fig. 5. Piston Environment Model

0 in 8-bit arithmetics) is in almost every case unintended, hence a programming fault.

To capture run-time errors of this type, we allow predicates and functions associated with transitions to return a special value $\bot$ (read 'undefined'), and we assume them to be *strict* (i. e. $f(\bot) = \bot$). $\bot$ models unknown behavior; we cannot define that a RN transition is not enabled if one of it's input registers contains $\bot$—especially, if this transition is located in the environment model, such a definition would model a rather unrealistic behavior. Therefore we decide to make a transition enabled even if the associated predicate yields $\bot$: In case of unknown behavior every behavior is possible.

3 Mathematical Background

This section summarizes the basic notations used throughout this paper.

To avoid tedious notions, we fix the following convention: If a structure $S = \langle A, B, \ldots \rangle$ is introduced, the components of S will always be denoted by $A_S, B_S, \ldots$ $\mathbf{N}$ denotes the set of non-negative integers, $\mathbf{B} = \{\text{false}, \text{true}\}$ is the set of Boolean values.

For some set A, $\mathcal{P}(A)$ is the set of all subsets of A. For $R \subseteq A \times B$ and $a \in A$, we denote the *image* of a under R by $R(a) =_{\text{df}} \{b \in B : a\ R\ b\}$. For $C \subseteq A$ we define $R(C) =_{\text{df}} \bigcup_{a \in C} R(a)$. The *inverse* $R^{-1} \subseteq B \times A$ of R is defined by $b\ R^{-1}\ a \Leftrightarrow_{\text{df}} a\ R\ b$. $R^+ \subseteq A \times A$ denotes the least transitive relation containing $R \subseteq A \times A$. For every set A and $n \geq 0$, A^n is defined by $A^0 =_{\text{df}} \emptyset$ and $A^n =_{\text{df}} A^{n-1} \times A$. For sets $A_1, A_2, \ldots, A_n$ and some $i \leq n$, the *projection* $\text{pr}_i : A_1 \times A_2 \times \cdots \times A_n \to A_i$ is defined to be $\text{pr}_i(\langle a_1, a_2, \ldots, a_n \rangle) =_{\text{df}} a_i$.

To deal with partial functions, we define $A_\bot =_{\text{df}} A \cup \{\bot\}$ for any set A. If $f : A \rightharpoondown B$ is a partial function, then the total function $f^\bot : A_\bot \to B_\bot$ is defined as

$$f^\bot(x) =_{\text{df}} \begin{cases} \bot, & \text{if } x = \bot \text{ or } f(x) \text{ is undefined;} \\ f(x), & \text{otherwise.} \end{cases}$$

This notion also applies to n-ary functions.

A (finite) *labeled partial order (lpo)* $a = \langle E, <, \lambda \rangle$ over some alphabet T consists of a finite set E of *events*, an (irreflexive) partial order $< \subseteq E \times E$, called the *precedence relation* of a, and a *labeling function* $\lambda : E \rightarrow T$. $\epsilon = \langle \emptyset, \emptyset, \emptyset \rangle$ is the *empty lpo*. If $t \in T$ is a symbol, then t is also be used to denote the *letter* $\langle \{0\}, \emptyset, \{\langle 0, t \rangle\} \rangle$.

The relation co_a is defined by $e_1 \;\text{co}_a\; e_2 \Leftrightarrow_{\text{df}} \neg(e_1 \leq_a e_2) \;\&\; \neg(e_2 \leq_a e_1)$. A set $C \subseteq E$ is called a *co-set*, iff we have $e_1 \neq e_2 \Rightarrow e_1 \;\text{co}_a\; e_2$ for all $e_1, e_2 \in C$. A *semi-order* is a lpo a where for all $e_1, e_2 \in E_a$, $e_1 \;\text{co}_a\; e_2 \Rightarrow \lambda_a(e_1) \neq \lambda_a(e_2)$. $\mathbf{SO}(T)$ denotes the class of semi-orders over T.

We now introduce the prefix relation and the notion of sequentialization for lpo's. Since we want to abstract from the specific events of lpo's, both concepts are presented in terms of homomorphism between lpo's.

Let a and b be lpo's. A mapping $h : E_a \rightarrow E_b$ is called a *homomorphism*, iff $e_1 <_a e_2$ implies $h(e_1) <_b h(e_2)$ for all $e_1, e_2 \in E_a$ and furthermore, $\lambda_a = \lambda_b \circ h$. It is called an *embedding*, iff it is an injective homomorphism with the property $h(\leq_a^{-1}(e)) = \leq_b^{-1}(h(e))$. A bijective embedding is called an *isomorphism*. a is called a *prefix* of b, denoted by $a \leq b$, iff there is an embedding $h : E_a \rightarrow E_b$. We write $a \equiv b$, if $a \leq b$ and $b \leq a$ holds. a is called a *sequentialization* of b, denoted by $b \preceq a$, iff there is a bijective homomorphism $h : E_b \rightarrow E_a$.

In [3] we proved that if $a \leq b$ holds for semi-orders a, b over the same alphabet, then the embedding of a into b is unique. We denote it by $H_a^b : E_a \rightarrow E_b$.

Clearly, $\equiv$ is an equivalence relation. A *semi-word* is an equivalence class of semi-orders. We write $[a] = [E_a, <_a, \lambda_a]$ to denote the equivalence class of a lpo a. The same notion applies to semi-words. A *semi-language* is a set of semi-words. $\mathbf{SW}(T)$ denotes the class of semi-words over T.

We fix the following conventions: If a, b, c, ... are semi-orders, then we use boldfaced lowercase letters $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, ..., to denote their equivalence classes $[a], [b], [c], \ldots$ Hence, for instance, E_a will always refer to the event set of a representative of $\boldsymbol{a} = [a]$. Especially, if t is a letter, then $\boldsymbol{t} = [t]$. The equivalence class of ϵ will also be denoted by ϵ.

Now it is easy to prove that both $<$ and $\prec$ are preorders on the class of lpo's. If we put $\boldsymbol{a} < \boldsymbol{b} \Leftrightarrow_{\text{df}} a < b$, and $\boldsymbol{a} \prec \boldsymbol{b} \Leftrightarrow_{\text{df}} a \prec b$ for all $a \in \boldsymbol{a}$, $b \in \boldsymbol{b}$, then $<$ and $\prec$ are partial orders on semi-words.

4 Register Nets and Their Partial Order Semantics

In this section we are going to introduce formally the notion of register nets (RN's). We start with the definition of a net, i. e the graph representation of the control flow part of a RN.

Definition 1 (Net). *A* net $\langle P, T, F \rangle$ *consists of non-empty, finite sets* P *and* T *such that* $P \cap T = \emptyset$*, where the elements of* P *and* T *are called* places *and* transitions, *respectively, and a* flow relation $F \subseteq (P \times T) \cup (T \times P)$. *We assume that* $F(t) \neq \emptyset$ *and* $F^{-1}(t) \neq \emptyset$ *for each* $t \in T$.

In the above definition, places are local system states, and transitions are used to model the changes of local system states according the flow relation.

Definition 2 (Register Net). *A* register net $V = \langle P, T, F, R, \rho, \boldsymbol{G}, \boldsymbol{P}, s\rangle$ *(a RN, for short) consists of the following components:*

1. *P, T, and F are such that $\langle P, T, F\rangle$ is a net.*
2. *R is a finite set of* registers. *We assume $R \cap (P \cup T) = \emptyset$.*
3. *A mapping $\rho : T_V \to R^* \times R^*$; $\rho(t)$ is called* signature *of t. To reduce the notational effort, we use the following shortcuts. Let $\rho(t) = \langle r_0 r_1 \dots r_n, r'_1 r'_2 \dots r'_k\rangle$:*
   a) $\mathrm{in}(t) =_{\mathrm{df}} n$ *(*input arity *of t), and* $\mathrm{out}(t) =_{\mathrm{df}} k$ *(*output arity *of t),*
   b) $\mathrm{rd}(t, i) =_{\mathrm{df}} r_i$ *for $1 \le i \le \mathrm{in}(t)$ (*input register selection *of t), and* $\mathrm{wr}(t, j) =_{\mathrm{df}} r'_j$ *for $1 \le j \le \mathrm{out}(t)$ (*output register selection *of t).*
   c) $\mathrm{mod}(t) =_{\mathrm{df}} \{\mathrm{wr}(t, i) : 1 \le j \le \mathrm{out}(t)\}$ *(the* registers modified *by t).*
   d) $\mathrm{ac}(t) =_{\mathrm{df}} \{\mathrm{rd}(t, i) : 1 \le i \le \mathrm{in}(t)\} \cup \mathrm{mod}(t)$*, (*access registers *of t).*
   We assume $1 \le i \ne j \le \mathrm{in}(t) \Rightarrow \mathrm{rd}(t, i) \ne \mathrm{rd}(t, j)$ and $1 \le i \ne j \le \mathrm{out}(t) \Rightarrow \mathrm{wr}(t, i) \ne \mathrm{wr}(t, j)$
4. *A family $\boldsymbol{P} = \{P_t\}_{t \in T}$ of partial mappings $P_t : \mathbf{N}^{\mathrm{in}(t)} \rightharpoonup \mathbf{B}$ (predicates).*
5. *A family $\boldsymbol{G} = \{G_t\}_{t \in T}$ of partial mappings $G_t : \mathbf{N}^{\mathrm{in}(t)} \rightharpoonup \mathbf{N}^{\mathrm{out}(t)}$.*
6. *$s \in \mathcal{P}(P) \times (R \to \mathbf{N}_\perp)$, an initial state.*

Let us discuss the above definition in more detail: The signature $\rho(t)$ of a transition gives us information of the arguments of the predicate and the function associated with this transition. If $\rho(t) = \langle r_1 r_2 \dots r_n, r'_1 r'_2 \dots r'_k\rangle$, then we will associate a function G_t with t, which obtains the values of the registers $r_1, r_2, \dots, r_n$ and computes a new value for the register $r'_1, r'_2, \dots r'_k$. The enabledness of t depends also on the registers $r_1, r_2, \dots, r_n$: t becomes enabled only in the case where $P_t^\perp(r_1, r_2, \dots, r_n)$ yields not false, i. e. either true or undefined.

Definition 3 describes system states of a RN V as pairs comprising a marking of V and an assignment of non-negative numbers to registers.

Definition 3 (State). *A* marking *of a RN V is a set $q \subseteq P_V$. A* register assignment *of V is a mapping $\sigma : R_V \to \mathbf{N}_\perp$. A* state *of V is a pair $s = \langle q, \sigma\rangle$, where q is a marking of V and σ is a register assignment of V. By $\Sigma(V) =_{\mathrm{df}} \mathcal{P}(P_V) \times (R_V \to \mathbf{N}_\perp)$ we denote the set of possible states of V. A register assignment σ is K-*restricted *for some constant $K \ge 0$, if $\sigma(r) < K$ for each $r \in R$. $\langle q, \sigma\rangle \in \Sigma(V)$ is called K-*restricted *if σ is K-restricted.*

Data types permitted by the IEC 1131-3 are scalar types like integers, Booleans, floats, etc, but restricted to a representation by m bits. To simplify our definitions, each of these data types is mapped on the set $\mathbf{N}$ of non-negative integers; the specific interpretation of a value (e. g. the interpretation of the number 0 and 1 as the Boolean values false and true) is assumed to be done by the mappings G_t. However, to put the m-bit restriction into our model, we assume that there is an upper bound K of each possible register value.

Definition 4 (K-Restricted RN). *V is restricted to some constant $K \geq 0$ (K-*restricted*, for short), iff for each transition $t \in T_V$ and for all $k_i < K$ we have that $\mathrm{pr}_j(G_t^{\perp}(k_1, k_2, \ldots, k_n)) \neq \perp$ implies $\mathrm{pr}_j(G_t^{\perp}(k_1, k_2, \ldots, k_n)) < K$ ($1 \leq i \leq \mathrm{in}(t)$, $1 \leq j \leq \mathrm{out}(t)$). V is* restricted *if it is K-restricted and s_V is K-restricted.*

We are now going to define the partial order semantics of RN using the notion of semi-orders. The effect of the firing of a semi-order is described by the simultaneous effect of the order-respecting occurrence of each event of the semi-order to both the marking component and the register assignment component of a state.

To simplify the definition of a firing rule of semi-orders in a RN, we need some additional notations:

Definition 5. *Let $a \in \mathbf{SO}(T_V)$ for some RN V. We define $e^- =_{\mathrm{df}} F_V^{-1}(\lambda_a(e))$, and $e^+ =_{\mathrm{df}} F_V(\lambda_a(e))$ for each $e \in E_a$. If $C \subseteq E_a$, we put $C^- =_{\mathrm{df}} \bigcup_{e \in C} e^-$ and $C^+ =_{\mathrm{df}} \bigcup_{e \in C} e^+$. If no confusion is possible, we write $\mathrm{rd}(e, i)$ for $\mathrm{rd}(\lambda_a(e), i)$, $\mathrm{in}(e)$ for $\mathrm{in}(\lambda_a(e))$, G_e for $G_{\lambda_a(e)}$, and so on.*

Definition 6. *Let $a \in \mathbf{SO}(T_V)$ for some RN V and let $C \subseteq E_a$, q be a marking of V and σ be a register assignment of V. We define mappings $\delta_V^0 : \mathcal{P}(P_V) \times E_a \to \mathcal{P}(P_V)$, $\delta_V^1 : (R_V \to \mathbf{N}_\perp) \times E_a \to (R_V \to \mathbf{N}_\perp)$, and $\pi_V : (R_V \to \mathbf{N}_\perp) \times E_a \to \mathbf{B}_\perp$ by*

$$\delta_V^0(q, e) =_{\mathrm{df}} (q - e^-) \cup e^+$$

$$\delta_V^1(\sigma, e)(r) =_{\mathrm{df}} \begin{cases} \mathrm{pr}_j(G_e^{\perp}(\sigma(\mathrm{rd}(e,1)), \ldots, \sigma(\mathrm{rd}(e, \mathrm{in}(e))))), \\ \quad \textit{if } r = \mathrm{wr}(t, j) \textit{ and } 1 \leq j \leq \mathrm{out}(e); \\ \sigma(r), \textit{otherwise} \end{cases}$$

$$\pi_V(\sigma, e) =_{\mathrm{df}} P_e^{\perp}(\sigma(\mathrm{rd}(e,1)), \ldots, \sigma(\mathrm{rd}(e, \mathrm{in}(e)))) \neq \mathrm{false}$$

δ_V^1 and δ_V^0 are inductively lifted to subsets $C \subseteq E_a$ by putting $\Delta_V^0(q, C) \in \mathcal{P}(P_V)$ and $\Delta_V^1(\sigma, C) \in \mathcal{P}(R_V \to \mathbf{N}_\perp)$ to be the smallest sets such that $\delta_V^0(\Delta_V^0(q, C - \{e\}), e) \subseteq \Delta_V^0(q, C)$, and $\delta_V^1(\Delta_V^1(\sigma, C - \{e\}), e) \subseteq \Delta_V^1(\sigma, C)$, where $e \in \max_{\leq_a} C$ and $C \neq \emptyset$; moreover we put $\Delta_V^0(q, \emptyset) = \{q\}$, and $\Delta_V^1(\sigma, \emptyset) = \{\sigma\}$ to terminate this recursive computation rule. Finally, we define $\Delta_V(\langle q, \sigma \rangle, C) =_{\mathrm{df}} \langle \Delta_V^0(q, C), \Delta_V^1(\sigma, C) \rangle$.

The mappings δ_V^0 and δ_V^1 describe the effect of the occurrence of a single event to the marking component and the register assignment component of a state. π_V is used to determine whether a transition t with associated predicate P_t is able to fire or not. Δ_V^0 and Δ_V^1 describe the effect of a set of events C to a state. The idea is that the events of C occur in an order compatible with $<_a$ (i. e, if $e <_a e'$, then e occurs before e', independent events can occur in any order). But for now, we cannot say yet whether two events with $e \ \mathrm{co}_a \ e'$ are independent in V. Therefore, Δ_V^0 and Δ_V^1 cannot be defined as partial mapping $\Delta_V^0 : \mathcal{P}(P_V) \times \mathcal{P}(E_a) \rightharpoondown \mathcal{P}(P_V)$ and $\Delta_V^1 : (R_V \to \mathbf{N}_\perp) \times \mathcal{P}(E_a) \rightharpoondown (R_V \to \mathbf{N}_\perp)$. But as we will see below, if a as

a certain property (V-consistence), both $\Delta^0_V(q, C)$ and $\Delta^1_V(\sigma, C)$ are singletons for each state $\langle q, \sigma\rangle$ and each subset $C \subseteq E_a$ of events, and therefore, they can be considered as partial functions.

Definition 7 (Dependence and Independence Relation). *If V is a RN, then the relation $D_V \subseteq T_V \times T_V$ is defined by*

$$t_1\ D_V\ t_2 \Leftrightarrow_{\mathrm{df}} (\mathrm{mod}(t_1) \cap \mathrm{ac}(t_2) \neq \emptyset \vee \mathrm{mod}(t_2) \cap \mathrm{ac}(t_1) \neq \emptyset) \\ \vee (F_V(t_1) \cup F_V^{-1}(t_1)) \cap (F_V(t_2) \cup F_V^{-1}(t_2)) \neq \emptyset.$$

The complement of D_V, i. e the relation $(T_V \times T_V) - D_V$, is called the independence relation *of V and is denoted by I_V. A co-set C of a semi-order a is called* independent *in V iff for all $e, e' \in C$, $e \neq e'$ implies $\lambda_a(e)\ I_V\ \lambda_a(e')$. a is called V-consistent iff each co-set of a is independent in V. By $\mathbf{SO}_V(T_V)$ ($\mathbf{SW}_V(T_V)$) we denote the class of V-consistent semi-orders (semi-words) over T_V.*

Note: If a is a semi-order over T_V such that $<_a$ is a linear ordering, then a is V-consistent.

Lemma 8. *Let V be a RN and let $\langle q, \sigma\rangle \in \Sigma(V)$. For each V-consistent semi-order a and each $C \subseteq E_a$, the sets $\Delta^0_V(q, C)$ and $\Delta^1_V(\sigma, C)$ are singletons.*

Proof. We show the lemma only for Δ^1_V, as the other part follows the same line. If $C = \emptyset$, then we have $\Delta^1_V(\sigma, C) = \{\sigma\}$ by definition, and we are left with the case $C \neq \emptyset$. Let $e, e' \in \max_{\leq_a} C$. We have to prove that $\Delta^1_V(\sigma, C - \{e\}) = \Delta^1_V(\sigma, C - \{e'\})$ for all possible choices of e and e'. For induction let us assume that $\Delta^1_V(\sigma, C - \{e, e'\}) = \{\sigma'\}$ is a singleton. Now it is enough to show that $\delta^1_V(\delta^1_V(\sigma', e), e') = \delta^1_V(\delta^1_V(\sigma', e'), e)$. But this follows immediately from the fact that $\lambda_V(e)\ I_V\ \lambda_V(e')$ holds, i. e. $\lambda_V(e)$ and $\lambda_V(e')$ modify different registers (if any). □

Because of this lemma, we will consider Δ^0_V and Δ^1_V as mappings, as they will be applied only to event sets of V-consistent semi-orders.

Definition 9 (Firing Rule). *Let a be a V-consistent semi-order over T_V for some RN V. Then a is* enabled *at a state $s = \langle q, \sigma\rangle \in \Sigma(V)$ iff for each co-set C of a the following conditions are satisfied:*

$$C^- \subseteq \Delta^0_V(q, <_a^{-1}(C))\ \&\ (C^+ - C^-) \cap \Delta^0_V(q, <_a^{-1}(C)) = \emptyset, \mathit{and} \tag{1}$$

$$\forall e \in C\ \left(\pi_V(\Delta^1_V(\sigma, <_a^{-1}(C)), e)\right). \tag{2}$$

If a is enabled at s, we denote this by $s \stackrel{a}{\Longrightarrow}$. a fires *from s to $s' \in \Sigma(V)$ iff $s' = \Delta_V(s, E_a)$; this is denoted by $s \stackrel{a}{\Longrightarrow} s'$.*

Finally, we call a transition $t \in T_V$ enabled *at some state $s \in \Sigma(V)$ if t considered as a letter is enabled at s. A set $C \subseteq T_V$ of transitions is* enabled *at s if $t_1, t_2 \in C\ \&\ t_1 \neq t_2 \Rightarrow t_1\ I_V\ t_2$ and $s \stackrel{t}{\Longrightarrow}$ for all $t \in C$ holds. If C is enabled at s, the we call C a* step.

Condition (1) is a generalization of the usual enabledness condition for ENS. It reads: C is enabled at q if the marking q' obtained by firing the history of C (the set $<_a^{-1}(C)$) subsumes the pre-conditions of C, and additionally, the places produced by C have not yet contained in q. Condition (2) says that if an event e of C has an associated predicate P_e, then P_e is not false at the register assignment obtained by firing the history of C.

Definition 10 (Reachable States and Semi-Language of a RN). *By* $\mathbf{R}_V(s) =_{\mathrm{df}} \{s' \in \Sigma(V) : \exists a \in \mathbf{SO}(T_V)(s \stackrel{a}{\Longrightarrow} s')\}$ *we denote the set of* states reachable *from some state* s *of* V*, and* $\mathbf{SL}_V(s) =_{\mathrm{df}} \{\boldsymbol{a} \in \mathbf{SW}_V(T_V) : s \stackrel{a}{\Longrightarrow}\}$ *is the* semi-language *of* V *at* s*. Finally, we put* $\mathbf{SL}(V) =_{\mathrm{df}} \mathbf{SL}_V(s_V)$*.*

Lemma 11. *If* s *is a* K*-restricted state of a* K*-restricted RN* V*, then* $\mathbf{R}_V(s)$ *is finite, and moreover, each* $s' \in \mathbf{R}_V(s)$ *is also* K*-restricted.*

Proof. Simple induction on firing sequences. □

5 Analysis of Register Nets

It is not hard to see that RN's are of the computational power of Turing machines. The easiest way to prove this fact is to reduce *counter machines* to RN's. Counter machines are finite state machines equipped with a set of *counters* containing non-negative integer values. Each counter can be decremented or incremented by one. Additionally, a state change of a counter machine can depend on a test whether a counter contains the value zero. It is quite obvious how to translate a counter machine into a RN.

On the other hand, K-restricted RN's can be translated into an ENS and therefore, they are strictly less powerful than Turing machines. The translation is done by adding places for each pair $\langle r, k\rangle$, where r is a register and $0 \leq k < K$ is an integer. The marking of such a place represents the fact that the register r contains the value k. Transitions t which access registers are replaced by transitions which move tokens according to the possible values of the predicates P_t and functions G_t. This transformation is known as *unfolding* in the context of colored Petri nets. Clearly, the unfolding of RN tend to be very large, and therefore, many analysis tools are not applicable for those unfoldings because the size of the input net violates memory limitations.

There are better ways to unfold a RN . One way is to use a binary representation of the register values [5]: For each K-restricted register $N = 2 \times \log_2(K+1)$ places will be added (hence: $K \leq 2^{\frac{N}{2}}$). Let these places be called $r_0, r_1, \ldots, r_{N-1}$ and $\bar{r}_0, \bar{r}_1, \ldots, \bar{r}_{N-1}$, respectively, and let $k_0 k_1 \ldots k_{N-1}$ be the binary representation of a value k. Then the fact that r contains the value k is represented by the marking $\{r_i : k_i = 1\} \cup \{\bar{r}_i : k_i = 0\}$. It is not hard (although quite lengthy) to implement operations like integer addition or division in terms of a binary coding of values by means of ENS. Although binary value coding leads

to smaller nets than unary value coding (ordinary unfolding), to our experience, the resulting nets are in many cases still too large for an analysis.

It should be noted that both transformations preserve the interleaving semantics of a RN (under a suitable notion of language homomorphism), but they *do not preserve* its partial order semantics. The reason is that transitions accessing a common set of registers can occur concurrently (see def. 7), but not those transitions which access a common set of places.

Another way to analyze RN nets is to find dedicated analysis methods for those nets. In this paper, we will describe the *partial order representation* of the behavior of a RN by means of *concurrent automata*. To do that, we need some more math.

Definition 12. *If V is a RN and $\boldsymbol{a} \in \mathbf{SO}_V(T_N)$, then $\langle \boldsymbol{a} \rangle_V$ is defined by $\langle \boldsymbol{a} \rangle_V =_{\text{df}} [E_a, (<_a \cap D)^+, \lambda_a]$, where D is given by $e_1 \; D \; e_2 \Leftrightarrow_{\text{df}} \lambda_a(e_1) \; D_V \; \lambda_a(e_2)$. Moreover, $\odot_V : \mathbf{SW}(T_V) \times \mathbf{SW}(T_V) \to \mathbf{SW}(T_V)$ is the operator $\boldsymbol{a} \odot_V \boldsymbol{b} =_{\text{df}} [E_a \cup E_b, (<_a \cup <_b \cup D)^+, \lambda_a \cup \lambda_b]$, where $D \subseteq E_a \times E_b$ is as defined above (however, with a different domain), and E_a and E_b are assumed to be pairwise disjoint.*

The following lemma states that, if we consider a RN V, for each $\boldsymbol{a}$ member of the semi-language of V there is an uniquely defined least sequential semi-word $\langle \boldsymbol{a} \rangle_V$. Moreover, if least sequential semi-words are concatenated using $\odot_V$, the result is also least sequential. The lemma resembles (the second part of) theorem 2.2.9 in [15].

Lemma 13. *For each restricted RN V we have*

1. $\langle \boldsymbol{a} \rangle_V \odot_V \langle \boldsymbol{b} \rangle_V = \langle \boldsymbol{a} \odot_V \boldsymbol{b} \rangle_V$
2. $\boldsymbol{a} \in \mathbf{SL}_V(s)$ *implies* $\langle \boldsymbol{a} \rangle_V \in \min_{\preceq}(\mathbf{SL}_V(s))$ *for each state s of V,*
3. $\boldsymbol{a} \in \mathbf{SL}_V(s)$, $\boldsymbol{b} \in \mathbf{SL}_V(s')$, *and* $s \stackrel{\boldsymbol{a}}{\Longrightarrow} s'$ *imply* $\boldsymbol{a} \odot_V \boldsymbol{b} \in \mathbf{SL}_V(s)$

Proof. (1) holds by set theory. (2) Making a semiword less sequential than $\langle \boldsymbol{a} \rangle_V$ would yield a V-inconsistent semi-word. (3) holds by definition. □

Definition 14. *The set $\mathbf{LSL}_V(s) =_{\text{df}} \{\langle \boldsymbol{a} \rangle_V \in \mathbf{SW}_V(T_M) : \boldsymbol{a} \in \mathbf{SL}_V(s)\}$ ist called the* least sequential semi-language *of a RN V at a state $s \in \Sigma(V)$. $\mathbf{LSL}(V) =_{\text{df}} \mathbf{LSL}_V(s_V)$ is the* least sequential semi-language *of V.*

Let us now turn to CA. Essentially, a CA of a net V is a finite automaton. Its state set consists of reachable markings of V. However, the transitions of a concurrent automaton are generally not labelled by single symbols, but by semi-orders.

The following questions arise: What is the language recognized by a CA? Under which circumstances does this language constitutes a complete and correct description of the behavior of a RN?

We choose the following answers to these questions: A CA is complete and correct if it recognizes exactly the least sequential semi-language of the associated RN. Recognition is defined by combining semi-orders obtained by traversing a CA via the $\odot$-operation defined above.

Definition 15 (Concurrent Automaton). *A* concurrent automaton *(CA) over an alphabet T is a structure $A = \langle S, X, \delta, s \rangle$ comprising a finite set S of* states*, a set $X \subseteq \mathbf{SO}(T)$ of semi-orders, partial* transition function $\delta : S \times X \to S$*, and an* initial state $s \in S$*. A* CA *of a RN V is a CA over T_V such that $S \subseteq \mathbf{R}_V(s_V)$ and $s = s_V$ holds.*

Examples for CA can be found in [3].

Definition 16 (Semi-Language of a Concurrent Automaton). *Let A be a CA of a RN V. A* path *through A is a finite sequence of semi-orders $\alpha = a_1 a_2 \dots a_n$ ($a_i \in X_A$ for $1 \leq i \leq n$) such that there are states $s_0, s_1, \dots, s_{n+1}$ with $s_0 = s_A$ and $\delta_A(s_i, a_i) = s_{i+1}$ is defined for $0 \leq i \leq n$. Let $P(A)$ denote the set of paths through A.*

If α is a path through A as given above, then the semi-word $\hat{\alpha}$ is defined by $\hat{\alpha} =_{\mathrm{df}} \boldsymbol{a}_1 \odot_V \boldsymbol{a}_2 \odot_V \cdots \odot_V \boldsymbol{a}_n$. The semi-language *is denoted by $\mathbf{SL}(A) =_{\mathrm{df}} \{\boldsymbol{a} \in \mathbf{SW}(T_V) : \exists \alpha \in P(A) \, (\boldsymbol{a} \leq \hat{\alpha})\}$.*

Definition 17 (Correctness and Completeness). *A CA of a restricted RN V is called* complete*, iff $\mathbf{SL}(A) \supseteq \mathbf{LSL}_V(s_V)$ holds. It is called* correct*, iff we have $\mathbf{SL}(A) \subseteq \mathbf{LSL}_V(s_V)$.*

The following lemma is obvious:

Lemma 18 (Preservation of Dead States). *Let V be a RN and let A be a correct and complete CA of V. $s = \langle q, \sigma \rangle \in \mathbf{R}_V(s_V)$ is dead, iff $s \in S_A$ and $\delta_A(S, a)$ is undefined for all $a \in X_A$.*

6 Algorithm

In [2,3] we discussed an algorithm to generate a concurrent automaton A of a safe Petri net. In this section, we modify this algorithm to work with RN.

Basic Algorithm. The basic algorithm resembles the reachability graph construction algorithm. It works as follows: It starts by introducing the initial state $s_A = s_V$ of A into the set Q, which contains unprocessed states. If a state s is considered, a set of semi-orders enabled at s is generated and appropriate arcs are added to A. If a new state s' is encountered by the firing of a at s, s' is added to S_A and Q. The algorithm terminates if all states in Q have been completely processed.

We have to consider the following problems:

1. If s is a state of A already generated, how do we construct an appropriate set of semi-orders enabled at s, and
2. if a is such a semi-order under construction, do we add another event to a or do we stop extending a?

Let us discuss problem 1. Define the *forward conflict relation* $D_\mathrm{f} \subseteq T_V \times T_V$ by

$$t_1 \; D_\mathrm{f} \; t_2 \Leftrightarrow_\mathrm{df} \Big((\mathrm{mod}(t_1) \cap \mathrm{ac}(t_2) \neq \emptyset \vee \mathrm{mod}(t_2) \cap \mathrm{ac}(t_1) \neq \emptyset) \\ \vee F_V^{-1}(t_1) \cap F_V^{-1}(t_2) \neq \emptyset \Big) \; \& \; t_1 \neq t_2,$$

and the *forward independence relation* by $t_1 \; I \; t_1 \Leftrightarrow_\mathrm{df} \neg(t_1 \; D_\mathrm{f} \; t_2) \; \& \; t_1 \neq t_2$. At a state s under consideration, we generate the set C of all maximal steps in the set T of enabled transitions at s such that

$$\forall t_1, t_2 \in C(t_1 \neq t_2 \Rightarrow t_1 \; I \; t_2 \; \& \; D_\mathrm{f}(t_1) \subseteq T \; \& \; D_\mathrm{f}(t_2) \subseteq T). \tag{3}$$

$t_1, t_2 \in C$ for different t_1, t_2 means that t_1 and t_2 are forward independent and each transition in static forward conflict to t_1 or t_2 is also enabled at s. Hence, for a transition $t \in T$ with $D_\mathrm{f}(t) \not\subseteq T$ a single step $\{t\}$ is generated. Using (3) enables us to deal with confusion situations; see [3] for a detailed discussion. Now each step C is turned into a semi-order a, i. e, if $C = \{t_1, t_2, \ldots, t_n\}$, then $a = \langle \{1, 2, \ldots, n\}, \emptyset, \{\langle i, t_i \rangle : 1 \leq i \leq n\} \rangle$. Events are added to a until some termination criterion holds (see below).

Problem 2 is solved in the following way: We suppose V to be extended by an *initialization part*, i. e. if V is a restricted RN, we construct a RN V^* from V and adding a transition t_I and a place p_I and the arcs $\langle p_I, t_I \rangle$ and $\langle t_I, p \rangle$ for all $p \in q_{s_V}$ to V. t_I has the signature $\rho(t_I) = \langle \epsilon, \epsilon \rangle$, and the predicate true. The initial state of V^* is $s_{V^*} = \langle \{p_I\}, \sigma_{s_V} \rangle$. Obviously, the extension of V to V^* does not change the behavior of the net significantly: We have $\mathbf{SL}_V(s_V) = \mathbf{SL}_{V^*}(s_V)$.

Define for some RN V the *backward conflict relation* $D_\mathrm{b} \subseteq T_V \times T_V$ by $t_1 \; D_\mathrm{b} \; t_2 \Leftrightarrow_\mathrm{df} F_N(t_1) \cap F_N(t_2) \neq \emptyset \; \& \; t_1 \neq t_2$. In [3] the following is shown for safe Place/Transition-nets:

Lemma 19. *Let V be a restricted RN. If there is a infinite sequence of semi-orders $a_0, a_1, a_2 \ldots$ such that $a_0 = \epsilon$ and for all $i \geq 0$, $a_i \in \mathbf{LSL}_{V^*}(s)$ and $a_i < a_{i+1}$, then there is some a_n with the following property: If $e \in E_{a_n}$ is an event of a_n, then there is another event $e' \in E_{a_n}$ with $e \leq_a e'$ such that*

1. $<_{a_k}(H_{a_n}^{a_k}(e')) = \emptyset$ *for all* $k \geq n$, *or*
2. $D_\mathrm{b}(\lambda_{a_n}(e')) \neq \emptyset$.

With other words, if we construct such an infinite sequence of semi-orders by adding successively transitions of V, we will finally end up by adding a transition with non-empty backward conflict relation. The proof of this lemma can be carried out exactly as in [3], because only the ENS part of a RN is used in the definition of D_b.

This solves problem 2. If a is a semiorder under consideration enabled at a state s of the concurrent automaton which we want to construct, a new event e, labelled with some transition t, is only added, if the following conditions hold:

```
procedure extend(a : in out SO_V(T_{V*}); s : in out Σ(V*)) is
    var T : set of T_{V*};
begin
    T ← addable(a, s);
    while T ≠ ∅ do
      select t ∈ T; a ← a ⊙_{V*} t; s ← Δ_{V*}(E_t, s);
      T ← addable(a, s)
    od
end extend;
```

Algorithm 1. Concurrent Automata Generation—Procedure *extend*.

T1. t is enabled after the firing of a at s;
T2. $D_\mathrm{f}(t) = \emptyset$, i. e. events with non-empty forward conflict relation remain minimal in $a \odot_{V^*} t$;
T3. $\forall e \in E_{a \odot_{V^*} t} \left(D_\mathrm{b}(\lambda_{a \odot_{V^*} t}(e)) \neq \emptyset \Rightarrow e \in \max_{<_{a \odot_{V^*} t}}(E_{a \odot_{V^*} t}) \right)$, i. e. if an event for t is added to a, then events of a with a non-empty backward conflict relation remain maximal events of $a \odot_{V^*} t$.

Algorithm 1 shows the heart of the algorithm of [3], the procedure *extend*. It is called if a new state s is encountered. The input of this procedure is a semiorder a associated with a step C of enabled transitions at s, and the state $s' = \Delta_{V^*}(s, E_a)$. It makes use of a function *addable*(a, s), which returns a set T of transitions such that conditions (T1), (T2), and (T3) are satisfied for each $t \in T$.

We improve our basic algorithm in the following way. Let s be a state of V^* encountered by the generation of a concurrend automaton of V^*, and let T be the set of enabled transitions at s. If we have a transition $t \in T$ such that $D_\mathrm{f}(t) \not\subseteq T$, then a single step $C = \{t\}$ is generated at s. Let us further assume that $D_\mathrm{f}(t) \cap T = \emptyset$ and let C' be another step in T generated at s. Let a be the semi-order associated with C'. The procedure *extend* adds only transitions with empty forward conflict relation to a, i. e. if a is extended to a', then t *remains enabled after the firing of* a'. Therefore, it is not necessary to fire t at s; we can postpone the consideration of t until the encountering of some other state where (hopefully) t belongs to a larger than a single step. This rule has two exceptions:

1. Each transition $t' \in T$ has the above property, i. e. $D_\mathrm{f}(t') \not\subseteq T \Rightarrow D_\mathrm{f}(t') \cap T = \emptyset$.
2. The firing of a semi-order a' at s leads to a cycle in the concurrent automaton, i. e the state $s' = \Delta_{V^*}(s, E_{a'})$ is already generated, and moreover, s is reachable from s'. In this case, t would be postponed forever.

This idea leads to algorithm 2. The following data structures are used:

1. Q, a stack of states of V^*, contains unprocessed states;
2. *num* is an array which assigns an unique number to each newly encountered state;

algorithm *generate* **is**
  input V, a RN;
  output A, a CA;
  local variables Q : **stack of** $\Sigma(V^*)$; R : **stack of N**;
    num : **array** $\Sigma(V^*)$ **of N**; i : **N** $\leftarrow 0$; $loop$: **bool**;
    s, s' : $\Sigma(V^*)$; T, C : **set of** T_{V^*}; a : $\mathbf{SO}_{V^*}(T_{V^*})$;
    U, V : **set of set of** T_{V^*};
begin
(1) $s_A \leftarrow s_{V^*}$; $S_A \leftarrow \{s_A\}$; $R_A \leftarrow \emptyset$; $\delta_A \leftarrow \emptyset$; $push(Q, s_A)$; $num(s_A) \leftarrow i$;
(2) **while** $\neg empty(Q)$ **do**
(3) $s \leftarrow top(Q)$; $pop(Q)$; $push(R, num(s))$; $loop \leftarrow$ **false**;
(4) $T \leftarrow enabled(s)$; $V \leftarrow single_steps(T)$; $T \leftarrow T - \bigcup_{C \in V} C$; $U \leftarrow steps(T)$;
(5) **if** $U \neq \emptyset$ **then**
(6) **foreach** $C \in U$ **do**
(7) $a \leftarrow so(C)$; $s' \leftarrow \Delta_{V^*}(s, E_a)$; $extend(a, s')$;
(8) **if** $s' \notin S_A$ **then**
(9) $push(Q, s')$; $S_A \leftarrow S_A \cup \{s'\}$; $i \leftarrow i + 1$; $num(s') \leftarrow i$
(10) **elsif** $member(R, num(s'))$ **then**
(11) **while** $top(R) > num(top(Q))$ **do** $pop(R)$ **od**; $loop \leftarrow$ **true**
(12) **fi**;
(13) $X_A \leftarrow X_A \cup \{a\}$; $\delta_A(s, a) \leftarrow s'$
(14) **od**
(15) **fi**;
(16) **if** $loop \vee U = \emptyset$ **then**
(17) **foreach** $C \in V$ **do**
(18) $a \leftarrow so(C)$; $s' \leftarrow \Delta_{V^*}(s, E_a)$; $extend(a, s')$;
(19) **if** $s' \notin S_A$ **then**
(20) $push(Q, s')$; $S_A \leftarrow S_A \cup \{s'\}$; $i \leftarrow i + 1$; $num(s') \leftarrow i$
(21) **elsif** $member(R, num(s'))$ **then**
(22) **while** $top(R) > num(top(Q))$ **do** $pop(R)$ **od**;
(23) **fi**;
(24) $X_A \leftarrow X_A \cup \{a\}$; $\delta_A(s, a) \leftarrow s'$
(25) **od**
(26) **fi**
(27) **od**
end *generate*;

Algorithm 2. Concurrent automata generation.

3. U and V are sets of transition sets. U contains those steps which can be fired at a state s, V contains single steps which probably can be postponed. The loop (6) – (14) deals with steps in the set U, single steps in V are considered in the loop (17) – (26).
4. R, a stack of state numbers, is used to detect cycles in the constructed concurrent automaton. For each pair of states s, s' considered in the outermost loop of algorithm 2, R contains the sequence of numbers of states from s_A to s which was computed to construct s'. Hence, if s' is already a member of

R, a cycle is detected. In this case, the cycle is removed from R (lines (11) and (22)) and every postponed transition is considered in the loop (17) – (26).

Furthermore, algorithm 2 uses the following subroutines:

1. *enabled*(s) returns the set of enabled transitions at a state s of V^*.
2. *single_steps*(T) returns for a transition set T a set of single steps of transitions which can be probably postponed.
3. *steps*(T) returns the set of all steps in T according to (3).
4. *so*(C) returns a semi-order a with empty ordering for the transition set C.

7 Simple Safety Properties and Run-Time Errors

By a simple safety property we mean a proposition φ on register values of a RN V, which is satisfied at all reachable states of V, i. e. φ characterizes those states of V which are 'good' states. Let $\varphi \equiv \varphi(r_1, r_2, \ldots, r_n)$ be denote a propositional formula containing $r_1, r_2, \ldots, r_n$ as 'parameters' where for $1 \leq i \leq n$, r_i is a register of a RN. Then φ is called a *simple safety property*. A RN satisfies a φ iff for each state $s = \langle q, \sigma \rangle \in \mathbf{R}_V(s_V)$ the formula $\varphi(\sigma(r_0), \sigma(r_1), \ldots, \sigma(r_n))$ is true.

Concerning our example from section 2, a simple safety property is Y_l·Y_r = 0; i. e. at every time point, at least one of the valves is closed.

The validation of simple safety properties in the state graph of a RN is obviously simple. If we want to use concurrent automata for those validations, we have to do some additional work. We use an idea which has deeply buried its origin in the history of Petri net theory: We will add *facts* for each simple safety property to be verified. A fact is a transition which is assumed to be dead at the initial marking of a Petri net, i. e. it is assumed to be never enabled.

A *fact* t_φ of a simple safety property $\varphi(r_1, r_2, \ldots, r_n)$ is a transition with the signature $\rho(t) = \langle r_1 r_2 \ldots r_n, \epsilon \rangle$, the associated predicate $P_t(k_1, k_2, \ldots, k_n) = \neg\varphi(k_1, k_2, \ldots, k_n)$, and the function $G_t = \emptyset$.

Let $\hat{V}$ be the RN obtained by adding a fact t_φ to V. To meet our definition 1, we assume a place p_φ such that $F_{\hat{V}}(t_\varphi) = \{p_\varphi\} = F_{\hat{V}}^{-1}(t_\varphi)$ and $F_{\hat{V}}(p_\varphi) = \{t_\varphi\} = F_{\hat{V}}^{-1}(p_\varphi)$. Clearly, $p_\varphi \in q_{s_{\hat{V}}}$, since otherwise t_φ would be dead regardless whether $\hat{V}$ fulfills φ or not.

Then it is clear that t_φ is not enabled at a reachable state s of V iff this state does not violate φ. Therefore, verification of simple safety properties can be performed on-the-fly while constructing a concurrent automaton of $\hat{V}$. If a semi-order a containing an event e with $\lambda_a(e) = t_\varphi$ is constructed, a violation of φ can be reported; additionally, it is possible to give a counter example for φ:

Lemma 20. *Let $s = \langle q, \sigma \rangle$ be a state of a RN V such that $s \in \mathbf{R}_V(s_0)$, which violates the simple safety property φ. Let a be a minimal semi-order in $\mathbf{LSL}_{\hat{V}}(s_0)$ such that $s_0 \overset{a}{\underset{V}{\Longrightarrow}} s$. Then there is an event $e \in E_a$ such that $\lambda_a(e) = t_\varphi$, and for all $e' \in E_a$ we have $e' \leq_a e$.*

The minimal semi-order a from the lemma above is called a *counter example* to φ.

An algorithm to determine counter examples is immediately at hand: We compute a shortest path $s_A = s_0, s_1, \ldots, s_n$ through a CA A such that $\delta_A(s_n, b)$ is defined and b contains this event e with $\lambda_b(e) = t_\varphi$; the shortest path algorithm by Dijkstra [1] can be used for this purpose. Next we select semi-orders $a_1, a_2, \ldots, a_n$ such that $\delta_A(s_{i-1}, a_i) = s_i$ is defined for $1 \leq i \leq n$, and build the semi-order $c = a_1 \odot_{\hat{V}} a_2 \odot_{\hat{V}} \cdots \odot_{\hat{V}} a_n \odot_{\hat{V}} b$. Now a is obtained by the restriction of c to the event set $\leq_c^{-1}(e)$.

By a run-time error we mean the occurrence of the special value $\perp$ in some of the registers of a RN V. Run-time errors can be formulated as the simple safety property $\chi(r_1, r_2, \ldots, r_N) \equiv \text{true}$, where $R_V = \{r_1, r_2, \ldots, r_N\}$. Note that the associated fact t_χ is enabled only if $\chi(\sigma(r_1), \sigma(r_2), \ldots, \sigma(r_N)) = \perp$ for some reachable state $S = \langle m, \sigma \rangle$ of V^*. Note also: If $\varphi(r_1, r_2, \ldots, r_n)$ is a simple safety property, and $\perp$ is assigned to one of the registers r_i at some reachable state s, then φ is violated at s, because t_φ is enabled at s.

8 Summary and Further Work

Starting with the special requirements to model adequately PLC programs given in IL, we defined (resticted) register nets (RN's), a variation of colored Petri net. RN's are tailored to a concise description of the operational semantics of IL. In order to get both efficient analysis methods and comprehensive behavior descriptions, RN's have been equipped with partial order semantics instead of the usual interleaving semantics. This gave us the chance to define the concurrent automaton (CA) as a semantic model for RN's. CA have been designed to combine the advantages of partial order semantics and state based models.

A generation algorithm for CA has been given. Finally, simple safety properties were defined and an analysis method for those properties based on CA has been given. If a violation of such a property is detected, a counter example is available to give the software developer information on the system behavior in which the error occurs. Due to our partial order semantics, the counter example is given by a concise semi-word instead of one of its arbitrary serializations. We concluded with the observation that run-time errors are expressible as a special type of a simple safety property.

However, the notion of simple safety properties is not powerful enough to capture every relevant analysis question. In industrial applications, it is sometimes important to determine whether a PLC program is able to react 'immediately', i. e. within the next processing cycle. Recalling our case study from section 2, an example of a property of this type is: If the operator releases the switch, does the piston stop moving right after the next processing cycle is completed.

Encouraging results of the available implementation of an algorithm for the construction of CA for safe Petri nets have been published in [2,4]. Our ongoing research focuses on an implementation of the approach presented in this paper

to determine run-times and memory efforts in practice. The analysis of more challenging examples is under preparation.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1987.

[2] P. Deussen. Algorithmic aspects of concurrent automata. In H.-D. Burkhard, L. Czaja, and P. Starke, editors, *Workshop on Concurrency, Specification & Programming '98*, number 110 in Informatik-Berichte, pages 39–50, Berlin, 1998. Humboldt Univ. zu Berlin.

[3] P. Deussen. Concurrent automata. Technical Report 1-05/1998, Brandenburg Tech. Univ. Cottbus, 1998.

[4] P. Deussen. Improvements of concurrent automata generation. Technical Report I-08/1998, Brandenburg Tech. Univ. Cottbus, 1999.

[5] M. Heiner. Petri net based system analysis without state explosion. In *Proc. High Performance Computing '98, SCS Int. San Diego*, pages 394–403, 1998.

[6] M. Heiner and T. Menzel. Time-related modelling of PLC systems with time-less Petri nets. In R. Boel and G. Stremersch, editors, *Discrete Event Systems*, pages 275–282. Kluwer Academic Publishers, 2000.

[7] H. Hulgaard and S. M. Burns. Bounded delay timing analysis of a class of CSP programs. *Formal Methods in System Design*, 11:265–294, 1997.

[8] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. of the 4th Workshop on Computer Aided Verification*, pages 164–174, Montreal, 1992.

[9] T. Mertke. Hydraulic piston example, 2000. private communications.

[10] Programmable logic controllers — programming languages, IEC 1131-3. International Electronical Commission, Technical Commitee No. 65, second edition. Commitee draft, 1998.

[11] P. H. Starke. Processes in Petri nets. *J. Inf. Process. Cybern. EIK*, 17(8/9):389–416, 1981.

[12] A. Ulrich. *Testfallableitung und Testrealisierung in verteilten Systemen.* Shaker Verlaq, Aachen, 1998.

[13] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.

[14] F. Vernadat and F. Michel. Covering step graph preserving failure semantics. In P.Azema and G.Balbo, editors, *18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 253–270. Springer-Verlag, 1997.

[15] W. Vogler. *Modular construction and partial order semantics of Petri nets*, volume 625 of *LNCS*. Springer-Verlag, 1992.