Practice of Modeling and Verification of Distributed Controllers using Signal Net Systems

V.Vyatkin, H.-M. Hanisch

Martin Luther University of Halle-Wittenberg
Dept. of Engineering Science
D-06099 Halle, Germany
Valeriy.Vyatkin@iw.uni-halle.de
Hans-Michael.Hanisch@iw.uni-halle.de

1 Introduction

Modeling is a cornerstone of formal analysis and synthesis of industrial control systems. Keeping up with the current practice of discrete control system design requires to make corresponding adjustments of the modeling formalisms. In this paper we discuss basic ideas and a development progress of the formalism of Signal Net Systems motivated by our activities in simulation and verification of certain industrial discrete control systems.

Formal methods for modeling and verification should support the design and implementation process, but dealing with the theory of formal modeling and verification cannot be the central part of daily business of an engineer. Hence, the formalisms of models and methods have to be adapted smoothly to the practical needs, and, more than this, they should be encapsulated and hidden as far as possible. That means that these methods should establish an optional support in improving the quality of a design but not a must in the design which requires detailed background knowledge of formal methods and eventually delays the design process significantly. Otherwise, an engineer would not accept the methods in his daily business.

A model is not the system itself but always an abstraction reflecting some properties of the system which are of interest for a particular question. It must be ensured that the model really copes with the aspects which are of interest. This means that a proper modeling formalism should be chosen to meet the requirements of the application, and that a formal or semi-formal methodology of model building should be provided.

Formal methods give answers to formal questions in terms of the formal model. Questions and answers have to be mapped from/to properties of the real system. According to our experience it is extremely useful that the modeling formalism is as close as possible to the real system for this purpose. Each transformation causes a need for establishing a re-transformation and therefore more formal overhead and possibly loss of relevant information.

Validation of real industrial control systems often requires to deal not only with control logic, but with details of hardware and software architectures. For this reason we pay special attention to modeling of controllers, including relevant features of corresponding operating systems, programming languages, etc. This is partly possible regardless of particular controller details due to existence of widely accepted standards on programming of measurement and control systems, such as IEC-61131 [1], the standard which sets programming languages for programmable logic controllers (PLC).

The latest trends of the control system design development have been reflected in the new standard IEC-61499 [2]. In general these trends combine the distributed architecture of control systems (like those based on the Fieldbus devices), and requirements to outstanding flexibility of production equipment, which ask for easily reconfigurable and reusable software components. The IEC-61499 offers a new concept of function blocks, which is a basically independent software component, connected to other blocks via data and event signals. An application in terms of IEC-61499 is a net of such function blocks, and a system is formed by mapping the (hardware independent) application to a particular (maybe distributed) architecture.

In our modeling we try to extract from the standard those parts which reflect the most essential features of programming for distributed control systems, namely: event connections between components, scheduling of algorithms within a device, communication between devices and resources, and asynchronous execution of algorithms in different devices.

The paper is structured as follows: In Section 2 we describe basics of Signal Net System (abbr.:SNS) formalism and show its application to modular modeling. The basic semantics of SNS is presented in Section 3, and further described in Section 4 with examples of modeling which explain the motivation of the formalism's development. Section 5 introduces our views on the structure of SNS-based verification tools intended to support design of industrial control systems. The paper is concluded by an outline of further research plans.

2 Modular Modeling of Discrete Event Systems

Inspired by the idea of Condition/Event systems as defined by Sreenivas and Krogh in 1991 [7] the prior work of some of the authors dealt with providing a modeling formalism which preserves the graphical notation and the non-interleaving semantics of Petri nets and extends them with a clear and concise notion of signal inputs and outputs.

These models were called Net Condition Event Systems (abbr. NCES), and were further developed to more general Signal Net Systems (SNS). Such models have been applied to a few discrete event system problems, including controller synthesis, verification, and performance analysis and evaluation. A bunch of specific SNS "dialects" have been developed to cover this wide range of problems.

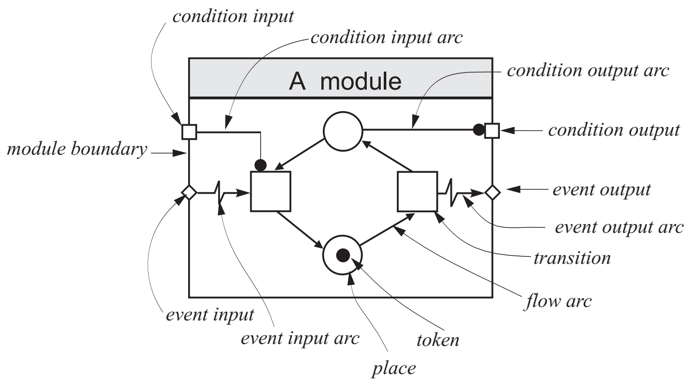


Figure 1: Graphical notation of the module.

From today's point of view, the general idea which is common to each particular "dialect" is quite simple, namely the way of thinking of and modeling a system as a set of modules with a particular dynamic behavior and their interconnection via signals. This way of modeling is a very intuitive one, and the modules can be pre-tailored and used over and over again. Roughly spoken, the behavior of a module can be described by a Petri net in the classical sense or even by a SNS.

Each module is equipped with inputs and outputs which are of two types:

1. Condition inputs/outputs carrying state information, and

2. Event inputs/outputs carrying state transition information.

This way the extension of the system with inputs and outputs clearly reflects the duality of Petri nets, namely the clear distinction between states and states transitions with their own graphical representation, semantics, and formal properties. An illustrative example of the graphical notation of a module is provided in Figure 1.

Condition input signals as well as event input signals are connected with transitions inside the module. Whether a transition of a module fires does not only depend on the current marking (as it is the case in classical Petri nets) but also on the incoming condition and event signals. Incoming condition signals enable/disable a transition by their values in addition to the current marking. Incoming event signals force transitions to fire if they are enabled by marking and by condition signals. Hence, we get a modeling concept that can represent enabling/disabling of transitions by signals as well as enforcing transitions by signals. More than this, the concept provides a basis for a compositional approach to build larger models from smaller components. "Composition" is performed by "glueing" inputs of one module with outputs of another module as depicted in Figure 2.

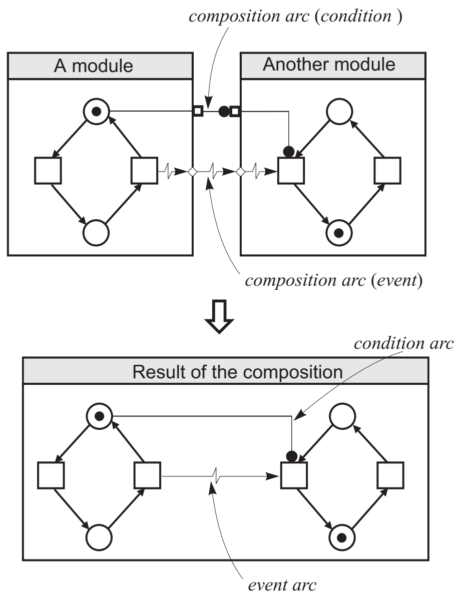


Figure 2: Example of modular composition

This "glueing" is graphically represented by composition arcs. The semantics of these arcs is obvious to any engineer who has ever modeled a system in a block-diagram oriented way. The low part of Figure 2 shows the result of the composition. One sees that we get two kinds of new arcs, namely condition arcs and event arcs.

If such a new module is equipped with inputs and outputs, it can also be interconnected over and over again. If the module is autonomous (it has no inputs), then and it can be analyzed without any additional information about its external environment. Some aspects are worth mentioning before we proceed with the definition of the model for an interconnected, autonomous system. These aspects are:

1. We have an intuitive way to model a system which closely corresponds to the design and engineering practice and trends.
2. The modeling technique supports a bottom-up modeling as well as top-down strategy. One could start with a set of modules and create a larger model by composing them. On the other hand, one could start with a larger system and could decompose it to a set of subsystems (expressed by modules) and a set of interconnections (expressed by composition arcs).
3. Even after composition, the state of the system is distributed, and the original structure of the modules is preserved. Even removing a module and replacing it by another module with the same input/output

interface would be a local operation over the structure of the model. Hence, composition is far less complicated as building the cross product of automata or the interleaving language. This allows us to build models of realistic scale efficiently.

4. On the other hand, local changes in the behavior of a single module may lead to global changes in the behavior of the interconnected system. Hence, one needs a formal model for expressing and analyzing the global behavior of an interconnected system. Such a model is the model of autonomous Signal/Net system as described in the next section.

3 Dynamics of Signal/Net systems

In this section we describe the semantics of (autonomous) Signal/Net Systems rather informally. Mathematically rigorous definitions can be found in [6].

In fact the SNS is a family of formalisms, having some common features on the one hand, and reflecting the wide spectrum of options developed in the Petri net theory, on the other. Among those are concepts of weighted arcs (both flow and condition), colored tokens, timing, etc. which in most cases do not contradict to each other. It would not make sense to describe in this paper all the variety of the options provided by SNS, they can be easily drawn and used if necessary. Instead we present the cornerstone concepts of the SNS first, and then discuss their application and the use of the add-ons as they logically follow from the examples' context. We hope that the reader is familiar with the usual terminology of Petri nets, so we outline here only the features being characteristic to SNS.

3.1 Marking-related firing rules

Semantics of Signal/Event nets is defined by the firing rules of transitions. There are several conditions to be fulfilled to enable a transition to fire. First, as it is in ordinary Petri nets, an enabled transition has to have a token concession. That means that all pre-places have to be marked with at least one token[1] , as it is shown in Figure 3.

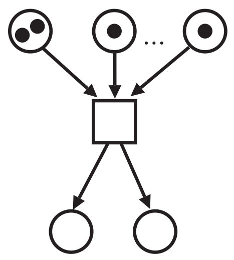


Figure 3: Token concession of transition.

In addition to the flow arcs from places, a transition in SNS may have incoming condition arcs from places and event arcs from other transitions. A transition is enabled by condition signals if all source places of the condition signals are marked by at least one token[2]. Speaking more precisely, the influence of condition signals to firing is defined by the "condition acceptance function", which can be a logical AND or a logical OR. Default value is AND which is interpreted as shown in the Figure 4: value of condition arc is TRUE iff the corresponding source place is marked.

A transition is enabled by conditions iff its condition acceptance function is TRUE at the current marking.

[1]In case of weighted arcs, as many as the weight of the corresponding arc from the pre-place to the transition.

[2]Or as many as the weight of the arc.

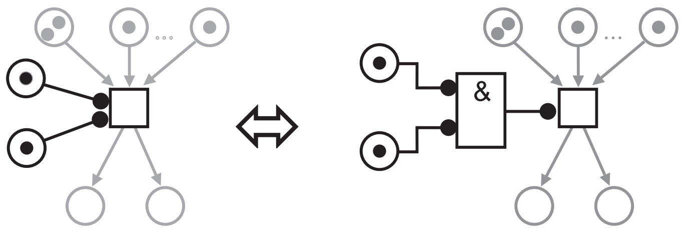


Figure 4: By default a transition has "AND" condition acceptance function.

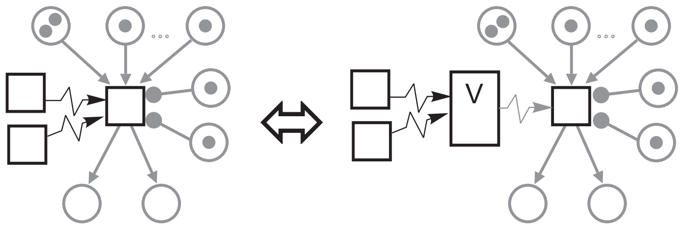


Figure 5: Default event acceptance function of forced transition is "OR".

Presence of incoming event signals essentially changes the firing rule of a transition. Transitions having no incoming event arcs are called spontaneous, otherwise forced. The default event sensitivity mode of a forced transition is OR: presence of at least one non-zero event signal is required to enable transitions by event signals. A forced transition is enabled if it has token concession, it is enabled by conditions, and its event sensitivity function evaluates to TRUE.

SNS are executed in steps, which are the sets of simultaneously firing transitions. An executable step is formed by first picking up a nonempty subset of enabled spontaneous transitions, and then adding as many as possible of enabled transitions which are forced to fire by signal-events produced by other transitions included in the step. Such step is called maximal with respect to its forced transitions.

3.2 Timing firing rule

A concept of discrete timing is applied to the SNS as follows: to every pre-arc $[p,t]$ of the transition p we attach an interval $[l,h]$ of natural numbers with $0 \leq l \leq h \leq \infty$. The interval is also referred to as *permeability* interval. If a pre-arc has no explicitly designated permeability interval, it is assumed to be $[0,\infty]$. The interpretation is as follows. Every place p bears a clock $u(p)$ which is running iff the place is marked $(m(p) > 0)$ and switched off otherwise. All running clocks run at the same speed measuring the time the token status of its place has not been changed, i.e. the clock on a marked place p shows the age of the youngest token on p. If a firing transition t removes a token from the place p or adds a token to p, the clock of p is turned back to 0. A (marking-enabled) transition t is *time-enabled* only if for any pre-place p of t the clock at place p shows a time $u(p)$ such that $l(p,t) \leq u(p) \leq h(p,t)$.

A state is called *dead* if no transition is time-enabled and no transition would become able to fire after any increments of the clocks. At a given state all (time-)enabled steps have to be computed and placed into the list of enabled steps. Firing of each step brings one more state successor to the current state. Repetitive application of this procedure to every subsequent state forms the reachability space of the model.

Time-enableness is a required but not sufficient condition to include transition to the firing step. The interpretation of the timing intervals is defined by timing firing rule. Several such rules have been studied:

1. **Strong vs. weak firing**: with the *strong* rule all marking enabled (spontaneous) transitions which have pre-places with clock position equal to the low or high time limit are obligatorily inserted into the step

(can be specified to make e.g. either strong earliest firing rule, or strong latest firing rule, etc.). If the *weak* rule is chosen then at least one of the enabled spontaneous transitions has to be included in step.

2. **Earliest vs. interval firing**: In case of the *interval* firing a transition is time-enabled at every clock position within the interval $[l, h]$. In the *earliest* firing rule a transition is time-enabled if it a has a pre-place with clock value equal to the low bound l of the time interval.

3. **Ultimo firing**: is a certain combination of interval and strong rules: a transition is time-enabled every time tick within the interval and must fire at the latest at clock position equal to h.

Processes in discrete modeling are represented as states which can be modeled in SNS as places with safe (0/1) marking. Each stable state can be also explicitly associated with duration. Figure 6,a shows an example of such a model. The time consuming state, represented by p_1, models the action in progress. Transition t_0 has incoming condition arc which starts the process. The process is in progress while the p_1 is marked.

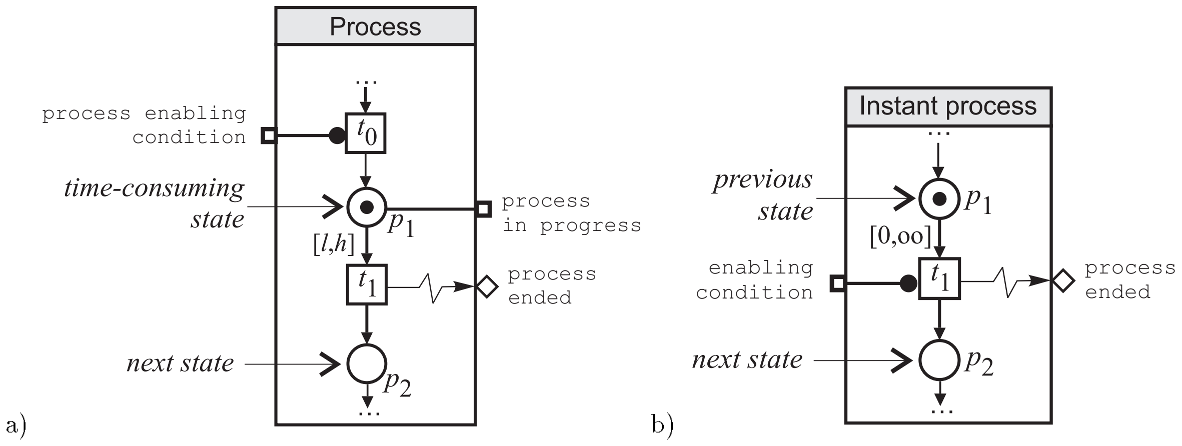


Figure 6: a)Model of plain time consuming process; b)Model of instant process.

Among all possible combinations of time constants and time-firing rules, some were found of interest in some industrial applications. These combinations are presented in Table 1.

	Time constants	Firing rule	Interpretation
1.	$l > 0, h \geq l$	Interval, weak	Event is expected with minimum delay l, maximum delay h, and may not occur at all.
2.	$l > 0, h \geq l$	Ultimo	Process must get terminated within the interval $[l, h]$.
3.	$l > 0, h = \infty$	Earliest, strong	Process has duration l, and all simultaneously started processes with the same duration finish simultaneously.
4.	$l > 0, h = \infty$	Earliest, weak	Process has duration l, but termination of all processes with the same duration may be not synchronized.

Table 1: Combinations of time-firing rule and time intervals commonly used for modeling.

The lower or higher time limits may or may not (depending on the corresponding rule) force transition to fire. The "interval" firing rule accepts presence of empty transition steps, when time elapses even in the absence of any enabled transitions. This option may be useful if aimed at finding of all possible combinations of overlapping processes and, correspondingly, simultaneous events. On the other hand it obviously blows up the complexity of the reachability space. We understand that the variety of choices may be confusing, however, on the other hand, it extends modeling horizons and allows to describe models more concisely.

There are some processes which have no duration, or it can be neglected in comparison to other ones. Such processes are called "instant". An example of a model of such a process is given in Figure 6,b. The process is started by "enabling condition", but no place marking corresponds to the state "process in progress".

More realistic modeling of processes may include exceptional states in which the system comes in case of abnormal process termination. The place p_1 (modeling the time-consuming state as in the previous example) is connected with two transitions: t_1 stands for normal operation mode with duration as described in the previous case, while the t_2 models the exception state, which may occur anytime within the normal operation time. This model is most adequate with the "ultimo" firing rule and $h < \infty$. The reached upper time limit forces only one of the transitions: either ending the action normally, or exceptionally.

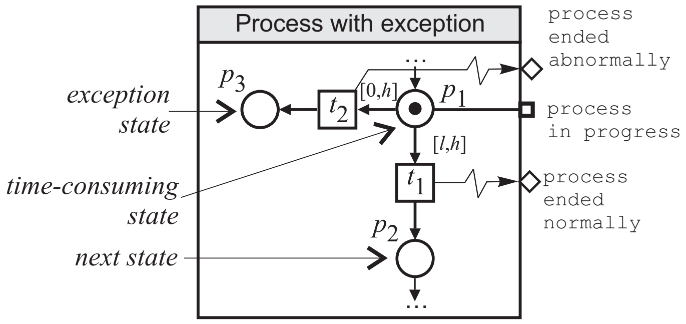


Figure 7: Model of process with possible exception

4 Applications of the SNS modeling

4.1 Models of closed-loop controller-plant system

An industrial automation system can be seen as built from two conceptually different parts: controller and uncontrolled plant. The controller performs data processing, communication and decision making, the plant contains the material-processing part of equipment. Depending on the required accuracy of modeling, the model of plant may include components for each drive, motor, valve, electric relay, sensor, actuator, and other elementary pieces of equipment. These component models may be integrated to composite models of equipment units, such as machine tools, or other material processing and storage units.

While following the general modeling principles layed yet in [4], in this paper we especially emphasize on its application for verification of controllers. To be really useful for verification purposes, the model of the plant must have its own dynamics, i.e. generate events, and respond to interaction from the controller as the real plant would.

The SNS as a direct derivative of Petri nets allows to combine state machine- like modeling of processes (in modules with safe marking) along with efficient concurrency and quantitative modeling of capacities, material flows, etc., in modules with numerical marking.

Spontaneous transitions in the model of plant may represent various non-determined or multi-choice actions, providing the check of controller's behavior on the variety of input combinations which may occur due to dynamic nature of concurrency in the plant, variable duration of operations, and resource conflicts. If several such transitions are enabled it reflects the situation when several simultaneous processes take part. When these are combined with the model of a controller, each of the processes may ask for the corresponding reaction.

Both non-timed and timed models can be used to fulfill different aspects of verification. The non-timed discrete event models usually better cope with complexity constrains, though they are obviously limited in accuracy and provide little information for estimations. However, for the verification purposes these may be quite enough since they generate all feasible trajectories of the plant's behavior. The timed models are expected to be more representative though this inevitably connected with state explosion, since every time tick produces a separate state in the state space of the model. Discrete formalisms with discrete time are limited in precision of the given results (for obvious reasons) if compared with description of the system using hybrid formalisms. The challenge is to squeeze as much as possible from the discrete formalisms, while enjoying their favorable complexity estimations.

Modeling of logic controllers using Petri-net like formalisms also has some background (works [3, 5] can be mentioned as examples). Use of SNS simplifies assembling the model from the components. Besides such key features of SNS as event/data connections closely correspond to the latest trends in controller design methodology presented in the IEC-61499.

Connection between controller and plant in most cases is implemented via logic level signals. Once value of a sensor is set, a certain time passes until the value is polled by the controller. So the value has clearly prolonged nature. The same is true for the output values connected to the actuators - usually they held for certain time.

Such a connection is modeled in SNS as by means of condition arcs. Event signals are used for in both models of plant and controller but not between them. In the model of plant the events may be used for example, to model the causal behavior of sensors influenced from the observed processes. In controllers the event signals model the actions explicitly defined as event-driven (say, event-connected function blocks in IEC-61499), as well as a lot of other operations: procedure calls, setting/resetting variables, etc.

Modeling can be performed in either open-loop or closed loop way. The open-loop modeling usually is a more economical solution which bases on the partial model of controller inputs which help to generate the outputs and then verify their correctness. The closed-loop model requires a comprehensive model of the plant which models its partial or full dynamics. It not only generates the inputs but also receives the outputs and correspondingly modifies its internal state. In both cases inputs and outputs of plant and controller are modeled as condition signal inputs and outputs of the corresponding SNS modules.

4.2 Two time scales: ticks in controller and time-elapsing in the plant

Timing is used in controller and plant to achieve adequate behavior of the model. In return it allows to make quantitative time estimations, or solve optimization problems such as finding a control strategy with minimal duration of the technological cycle, etc.

Once all the SNS modules have been interconnected into Signal/Event Net, the resulting net has a common time unit. That is why for modeling of objects with different time scales the minimum basic time unit over all components has to be selected.

Every time increment brings the model to a new state, which obviously means state explosion if we try to decrease the basic time unit. This is especially sensitive in the closed-loop models. The common sense suggests to accept the time scale of the most relevant processes in the plant, and assume that processes in controller (or some 'fast' (electric) units of the plant, such as sensors, or relais) as having zero duration. However, some estimations in controller still can be done by measuring the number of executed commands (or number of transitions in SNS).

Modeling of a program unit which takes 100 commands for execution can be modeled as shown in Figure 8: initial place p_1 is loaded with 100 tokens, and with every transition $t1$ one token flows from p_1 to p_2 through the flow arcs with multiplicity 1 until all the tokens come to p_2. Then all the tokens come back to p_1 in one

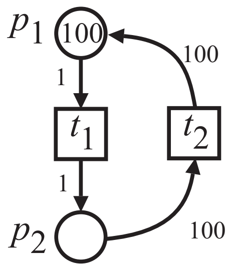


Figure 8: Program delay - model of the controller module which requires 100 commands for execution.

transition t_2 through the arcs with multiplicity 100.

4.3 Interaction between process and sensor

Event arcs are able to express the variety of instantaneous actions, one of which is operation of a sensor which detects the changes of the plant's state (see Figure 9). Once transition t_1 in the model of the process fires, it also switches the sensor ON by means of the event arc. Model of sensor comes first to the transitional state (marking in p_2) and after the delay D - to the state with $p_3 = 1$. Reading of the sensor usually comes to controller as a logic value modeled in our formalism as a condition signal.

Note that in the Figure 9 we model the "malfunction free" sensor which always produces the required value upon elapsing the specified time D. Model of sensor can be either simpler (just a bi-stable) or much more complicated, depending on the required accuracy of modeling.

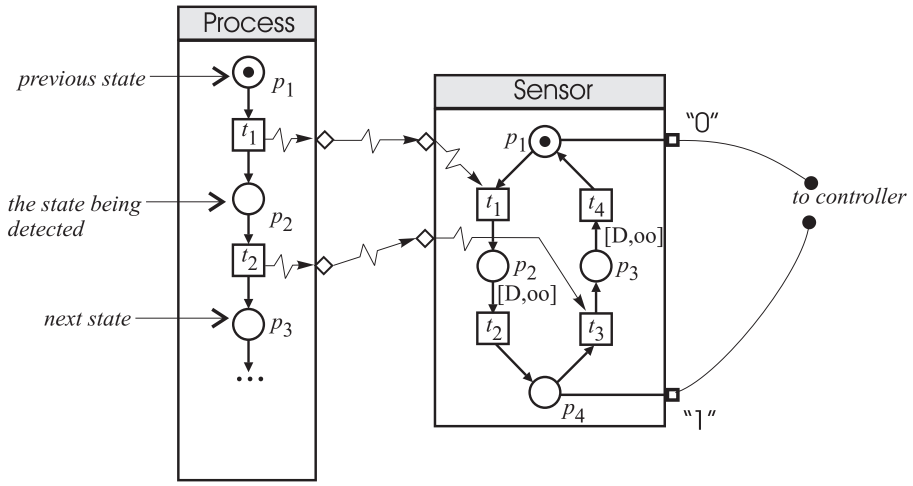


Figure 9: Model of sensing: sensor detects when the process comes to the observed state and with delay produces the required logic value (places p1,p4 of the module "Sensor").

4.4 Modeling of controller and its interaction with plant

Controller's data are modeled by marking of places, each data element is represented by a module. For logic data the marking is safe. The executed control algorithms (both system and user-defined) are modeled by nets in which places represent the current point of execution and transitions model actions such as basic or macro commands. Such "execution logic" modules interact with the data modules using event arcs to change the data values and using the condition arcs to read them.

Plants and controllers cannot be modeled uniformly. Transitions in the plant represent usually start or finish of some time consuming actions, while transitions in the controller part of the module represent almost instantaneously executed commands. Hence, when two transitions are enabled, one in the plant, and the other in the controller, first the latter has to be executed. On the other hand we try to make the component model as much independent from each other as it is possible to provide better reusability of both. To avoid further complications in the description of the required behavior two mechanisms were introduced. All these modifications apply only to spontaneous transitions. More precisely, they reduce the number of executable steps as compared to the previously explained firing rule eliminating some unfeasible ones from the total list of steps.

The first, "greedy" transition approach is aimed at non-timed models. All spontaneous transitions in the controller (like those in the "execution logic" part) are marked as greedy. According to the greedy firing rule, if a greedy transition is enabled, then each executable step must include at least one greedy transition. This guarantees that all enabled transitions (commands) in the controller model will be executed until the next action occurs in the model of plant.

The other option, intended for timed models, is to define *synchronization sets* of non-empty, pairwise disjoint sets of spontaneous transitions, such that different sets have no pre-places in common. All enabled transitions in the same synchronization set should fire simultaneously (except for those in conflict to each other): if at least one member of the synchronization set is included into a step, that means that all other enabled transitions from the same synchronization set have to be also included in the step (as far as they do not make conflicts to each other). In a particular case when all the spontaneous transitions form one synchronization set, we get the "strong" firing rule. Use of the synchronization sets will be illustrated in the next section.

Consider a simple example of process/controller communication as shown in Figure 10.

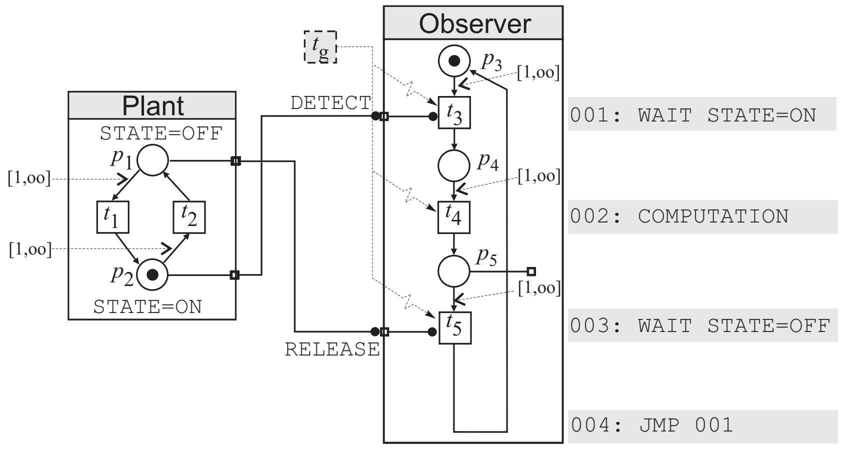


Figure 10: Simple example of plant/controller interaction

The process is represented by the basic unit of plant which has two flip-flop states (as up–down, left–right, on–off) modeled by places p_1 and p_2. Transitions between the states are spontaneous. First assume that the model is not timed (omit the time intervals attached to the arcs).

The block "Observer" makes a "moment photo" of the process, i.e. reads the value of state in the loop (input DETECT), and when it is "ON" (i.e. $p_2 = 1$), performs some computations and stores the value in the memory. When the state becomes "OFF" (input "RELEASE"), the observer clears the memory and returns to the initial state.

If the case of timed model at the clock value $u(p_2) = 0$ (i.e. right after the place p_2 gets marked) the process in the plant is delayed, and only transition t_3, corresponding to the information reading command, is enabled. The only possible sequence of firing is: $t_3 \rightarrow t_4 \rightarrow$ (after 1 time unit) $t_2 \rightarrow t_5$.

If the non-timed model is used, and transitions in the observer are also spontaneous, then in the state, shown in Figure 10, there would be the following enabled steps of transitions: $s_1 = \{t_2\}$,$s_2 = \{t_3\}$,$s_3 = \{t_2, t_3\}$. However, the first step is not feasible: it reflects the situation when the state is changed (marking in the p_2), but controller does not start the corresponding action, although it was able to do so (was not busy), and information about the action is lost. Figure 11,a shows the state diagram reflecting the correct modeling of the interconnected system, while Figure 11,b presents the diagram of the modeling scenario which can never occur in the reality. Due to the simultaneous enableness of the transitions t_2, t_3 this scenario reflects the situation when state of the plant changes as fast as the first command in the controller gets executed (initiated by the previous change of the plant's state).

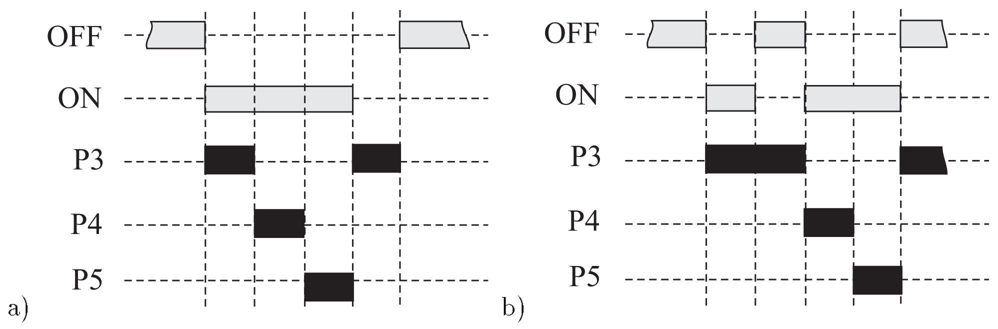


Figure 11: Desirable and incorrect sequence of plant/controller states. The sequence presented in the part b) can never occur in the reality but is generated by the model.

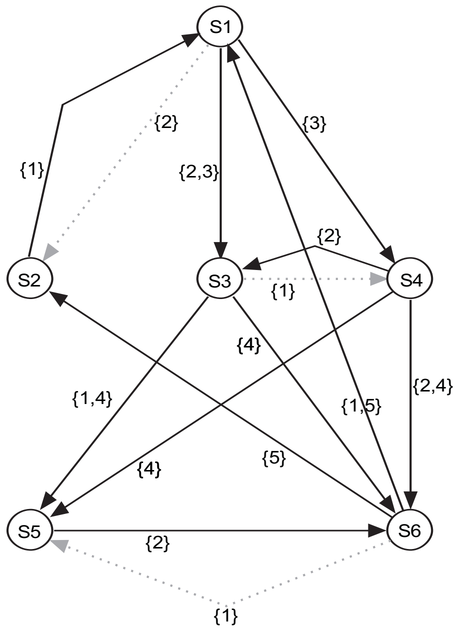


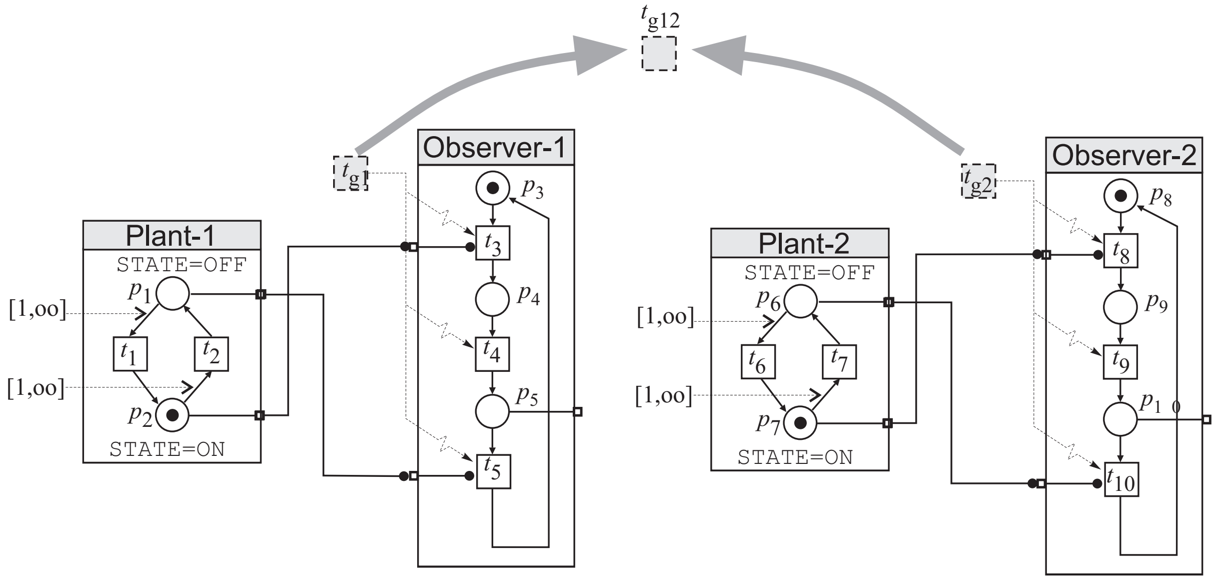
Figure 12: Reachability graph of the interconnected system. Trajectories eliminated by the "greedy" tick generator are dotted.

Greedy transitions help to cope with this problem. We add (an always enabled) greedy transition t_g, connected to t_3, t_4, t_5 via (dotted) event arcs. It fires always (i.e. it is included to every executable step) sending forcing "ticks" to the transitions corresponding to the commands in the controller no matter what is going on in the plant. As a result some trajectories are eliminated from the reachability graph of the model, as it is illustrated in Figure 12.

Remark: we could mark transitions t_3, t_4, t_5 as greedy, and that would bring us in this example the equivalent behavior. However the structure with the "tick" generator is more flexible: we can easier synchronize (or model as asynchronous) distributed components of controllers, as this will be illustrated in the next section.

4.5 Distributed controllers

More realistic architectures of distributed control systems may include several autonomous controllers, some of which work asynchronously with data exchange via a (distributed) network, while the others co-exist within one device, thus being synchronized. The architecture may also influence the occurrence of new trajectories in the system's state space which are not feasible in local centralized architectures. Consider the system built from two concurrently working earlier considered subsystems.



t_{g12}
t_{g1}
t_{g2}
Observer-1
Observer-2
Plant-1
Plant-2
STATE=OFF
STATE=ON
[1,oo]
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10}
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}

Figure 13: Two concurrent processes in the plant being observed by two independent controllers.

In the non-timed model the greedy transitions ensure that the model generates all possible sequencing of commands. In the state presented in the Figure 13, there are 12 enabled steps: $\{\{t_{g1}, t_3\}, \{t_{g2}, t8\}, \{t_{g1}, t_3, t_{g2}, t_8\}\} \times \{\{\}, \{t_2\}, \{t_7\}, \{t_2, t_7\}\}$. For example: $\{t_{g1}, t_3, t_2\}, \{t_{g2}, t_8, t_2, t_7\}, \{t_{g1}, t_3, t_{g2}, t_8\}$, etc. For instance, the step $\{t_{g1}, t_3\}$ models the situation when observer 1 has started processing, while the other observer still has not.

The proposed solution allows easy change of the architectures. Thus, if the two observers would be placed within the same controller device (becoming thus syncronized), it is enough to substitute the two tick generating transitions t_{g1} and t_{g2} by t_{g12} as shown in the Figure 13. Then, only 4 steps would be possible in the given state: $\{t_{g12}, t_3, t_8\} \times \{\{t_2\}, \{t_4\}, \{t_2, t_4\}, \{\}\}$.

Behavior similar to the "greedy" transitions is modeled in timed models by means of synchronization sets. All transitions in the controller part of the model, which are driven by the "greedy" transitions in the non-timed model, are marked as members of a particular synchronization set, associated with the controller. Membership in a particular synchronization set has the only consequence on the firing of transitions: all enabled transitions belonging to the same set are included in the firing step only all together. It has no consequences in the simplest case when a module has the safe marking and one firing spontaneous transition at once. But in case of several simultaneous actions taking place in controller, there is a need to separate them out from other groups of actions taking place in other controllers. In the example in Figure 13 we define two synchronization sets : $S_1 = \{t_3, t_4, t_5\}$ and $S_2 = \{t_8, t_9, t_{10}\}$ to model the allocation of the observers to separate devices. The reachability graph for this model is presented in Figure 14 with initial state S_1 equivalent to that shown in Figure13 and with clock values equal to 0 in all places.

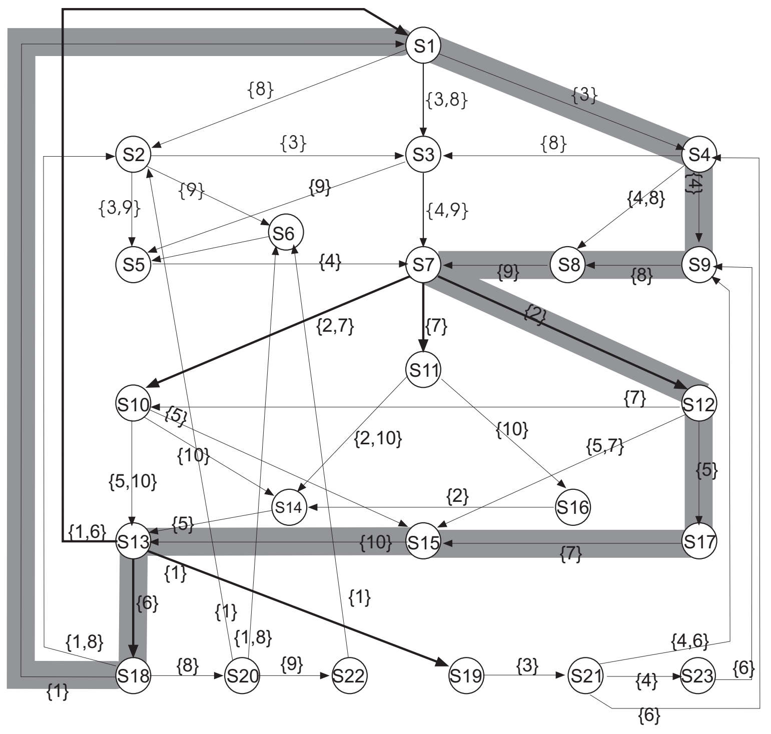


Figure 14: Reachability graph for the timed model with two synchro-sets (The "observers" reside in independent devices). The bold arcs have duration 1.

A path in the reachability graph corresponds to a particular scenario of the system's operation. Consider, for example, the path outlined in Figure 14. The state/timing diagram of the model propagating along this path is shown in Figure 15.

The first observer starts computation in response to the occurrence of the marking in p_2. It makes two computation steps before the second observer starts its execution in response to another event (marking in p_7).

Note that results of the computation may be used by other controllers (not present in this example), so that their sequence is important.

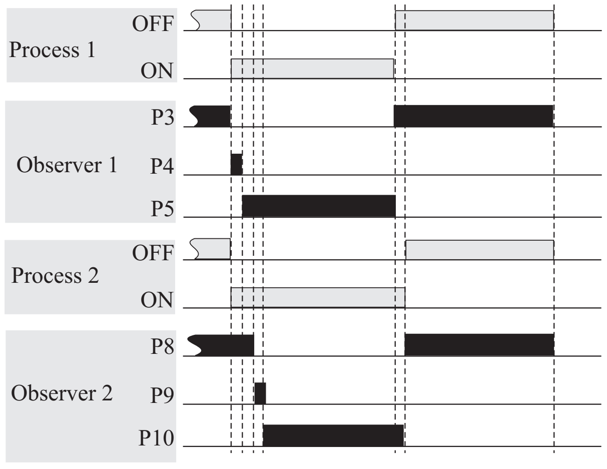


Figure 15: State/timing diagram of the outlined path in the reachability graph. Short intervals represent states with zero-duration, larger intervals represent states with duration 1.

5 Conclusion

We presented in the paper some ideas for discrete-event modeling of closed-loop control systems which may serve as a basis for developing of a practically applicable tool for testing and verification of discrete control systems.

Such a tool, called VEDA - Verification Environment for Distributed Applications, - is being developed by the authors in cooperation with the group of Prof. P.Starke at Humboldt University, Berlin. The tool has the following characteristic features:

- It deals with source code of controllers following IEC-61131 and IEC-61499, and it automatically generates the formal model of the controller given the controller source code.
- It provides facilities to develop SNS models of plants: a graphic editor for Petri net-like models plus a means to specify visualization of the model.
- It allows simulation and model-checking of the controller/plant closed-loop system. It therefore offers a number of visual tools to analyze the behavior of the models (tracing trajectories with timing diagrams of all the parameters);
- It facilitates formulation of specifications using a visual editor of specifications presented in a sort of state/timing diagram;

For more details on VEDA see [8]. We applied the presented methods of modeling to automatic model generation of IEC-61499 function blocks, including such their features, as event-driven data and execution control, distribution of blocks over several devices, scheduling and execution of algorithms, and data communication between devices. The modeling also helped to reveal some incorrectness in the descriptions of function block internal behavior as they were presented in the draft of the standard. Verification of distributed controllers using SNS modeling is discussed in more details in [10, 9].

Plans for future work in this direction include further development of VEDA, along with its testing on real-scale examples of control applications. This work includes also continuation of the theoretical studies, in particular those related to formulation of specifications, their consistency check and transformation to equivalent SNS models, as well as development of more adequate specification language, and model-checking methods, allowing to deal with both condition and event formula.

References

[1] *International Standard IEC 1131-3, Programmable Controllers - Part 3.* International Electrotechnical Commission, Geneva, Suisse, 1993.

[2] *Function Blocks for Industrial Process Measurement and Control Systems.* International Electrotechnical Commission, Tech.Comm. 65, Working group 6, Geneva, 1998.

[3] P. J.Zaytoon De Loor and G.Villerman-Lecolier. Abstraction and heuristics for the validation Grafcet controlled systems. *European Journal of Automation*, 31:561–580, 1997.

[4] H.-M.Hanisch. *Petri Netze in der Verfahrenstechnik.* Oldenburg, München, 1992.

[5] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behavior by means of timed net condition/event systems. In *International conference on Emerging Technologies and Factory Automation (ETFA'97)*, Los Angeles, USA, September 1997.

[6] P.Starke, S.Roch, K.Schmidt, H.-M. Hanisch, and A.Lüder. *Analysing signal-event systems.* Humboldt Universität zu Berlin, Institut für Informatik, July 1999. Internal report.

[7] R.S.Sreenivas and B.H.Krogh. On condition/event systems with discrete state realizations. *Discrete Event Dynamic Systems: Theory and Applications*, 2(1):209–236, 1991.

[8] Vyatkin V. and Hanisch H.-M. VEDA - a prototype tool for deep debugging of distributed applications. Information leaflet, June 2000. http://www.et.uni-magdeburg.de/∼vyatkin/refs/veda.pdf.

[9] V.Vyatkin, H.-M. Hanisch, P. Starke, and S. Roch. Modelling and verification of execution control of the function blocks following the standard IEC 1499 by means of net condition/event systems (NCES). Technical report, Uni-Magdeburg, IFAT, Magdeburg, March 2000. http://www.et.uni-magdeburg.de/∼vyatkin/refs/report.pdf.

[10] V. Vyatkin and H.-M. Hanisch. Modeling of IEC 61499 function blocks as a clue to their verification. In *XI Workshop on Supervising and Diagnostics of Machining Systems*, Karpacz, Poland, March 2000.