Working draft

Valeriy Vyatkin

Modelling and Verification of Discrete Control Systems

with

Net Condition/Event Systems and

Visual Verification Framework

Version 1/15/2012

Disclaimer

This text is a working draft and may contain some fragments earlier published in papers co-authored with other researchers, in particular with Hans-Michael Hanisch from Martin-Luther University of Halle (Germany), Jose LM Lastra and Andrei Lobov from Tampere University of Technology, (Finland), Gustavo Bouzon from University of Santa – Catharina (Brazil) whose contribution and collaboration is greatly acknowledged and appreciated.

The author is very much grateful to Cheng Pang for valuable contributions to this document, especially for the graphical material in Chapter 11.

The author will be grateful for any comments. Please report to v.vyatkin@auckland.ac.nz

Table of Contents

Structure of the text

First part (Chapters 1-5) introduces the framework and modelling language of Net Condition/Event Systems (NCES) as follows. Chapter 1 introduces the formal verification framework. Chapter 2 starts with providing informal introduction into the formalism of Signal/Event nets. Chapter 3 introduces modular Signal/Event nets called Net Condition/Event Systems. Chapter 4 discusses some challenges to the S/E net semantics brought by the modularity of NCES, and Chapter 5 adds time to the Signal/Event nets.

Second part (Chapters 6-12) presents basics of modelling automation systems and technique of their formal verification in the Visual Verification Framework as follows. Chapter 6 introduces the framework of closed-loop modelling and verification, Chapter 7 presents some basic techniques for modelling objects and physical processes (plant) using NCES, Chapter 8 introduces basic NCES elements to be used in controller models, such as models of variables and operations over them. Chapter 9 discuses some challenges arising from the need to combine purely deterministic and synchronous objects (controller) with asynchronous and non-deterministic processes (plant). Chapter 10 presents an example of a simple automation system modelled and verified in the presented framework. Chapter 11 considers the use of the **Visual Verifier** tool in more detail, and Chapter 12 presents more details on the properties to be verified.

The third part (Chapters 13-15) presents some additional techniques and facts and is structured as follows: Chapter 13 discusses specifics of distributed controller modelling. Modelling of Programmable Logic Controllers is exemplified in Chapter 14. Chapter 15 is devoted to systematic modelling of plants. Chapter 16 introduces the ideas of hierarchical model composition in NCES.

Annex 1 provides examples of XML representation of NCES models.

Annex 2 contains more rigorous definitions of NCES.

1 Introduction: modelling and verification of cyber-physical systems

In computer science *formal verification* is an act of proving the correctness of programs by using mathematical methods and models. It can be used as an automatic alternative to the simulation-based testing and debugging, improving dependability and reliability of automation systems. Unlike testing via simulation, the formal verification can explore the complete set of system's state space and prove mathematically that no undesirable or dangerous behaviour occurs. This can reduce the effort spent on validation the same time increasing its quality which especially important in safety critical applications. Formal verification be also very helpful in proving the compliance with various certification requirements.

Cyber-physical systems is a novel view on embedded systems that takes into account the dynamics and the structure of the environment where the embedded device works. In many control and monitoring applications this view has proven to be beneficial as compared to a more narrow focus only on the computing hardware and software.

In particular, in control systems, usually the control software is the target of verification. This software is further referred to as *controller*, and it is the essential part of the embedded *control device* connected to the *plant under control*. Plant and controller form the interconnected closed-loop control system. For example, in industrial automation, the controller code usually is a variable part of the system, while the hardware remains unchanged. The controller can be programmed in one of general-purpose or specialized programming languages, e.g. following the IEC61131-3 standard [38].

1. The closed-loop system is modeled using an appropriate finite-state or hybrid formalism, e.g. finite state machines, Petri nets, etc. In *closed-loop modeling* the model of the plant needs to be present explicitly. It has to be designed manually by control engineers, while the model of the controller can be built automatically given the code. In *open-loop* modeling only the controller part is verified under some assumptions about its inputs.
2. The desired or forbidden behavior of the plant-controller system needs to be described in form of s*pecifications*, i.e. the properties to hold or to avoid. The specifications have to be formalized using a formal language compatible with the description of the model.

3. Given the model and a number of formal specifications, it can be formally checked whether the specifications hold with respect to the model. This process is called *model-checking*.
4. The results of the model-checking have to be interpreted in terms understandable by the engineers. For this purpose, a bi-directional mapping from the original system to its model and back has to be provided.

This text presents a framework for modelling and verification which is based on the formalism of Net Condition/Event Systems (NCES).

1.1 Supporting Tool Framework

To facilitate the use of NCES by engineers, the formalism is supported by tools and methodologies. The framework is presented in Figure 1.

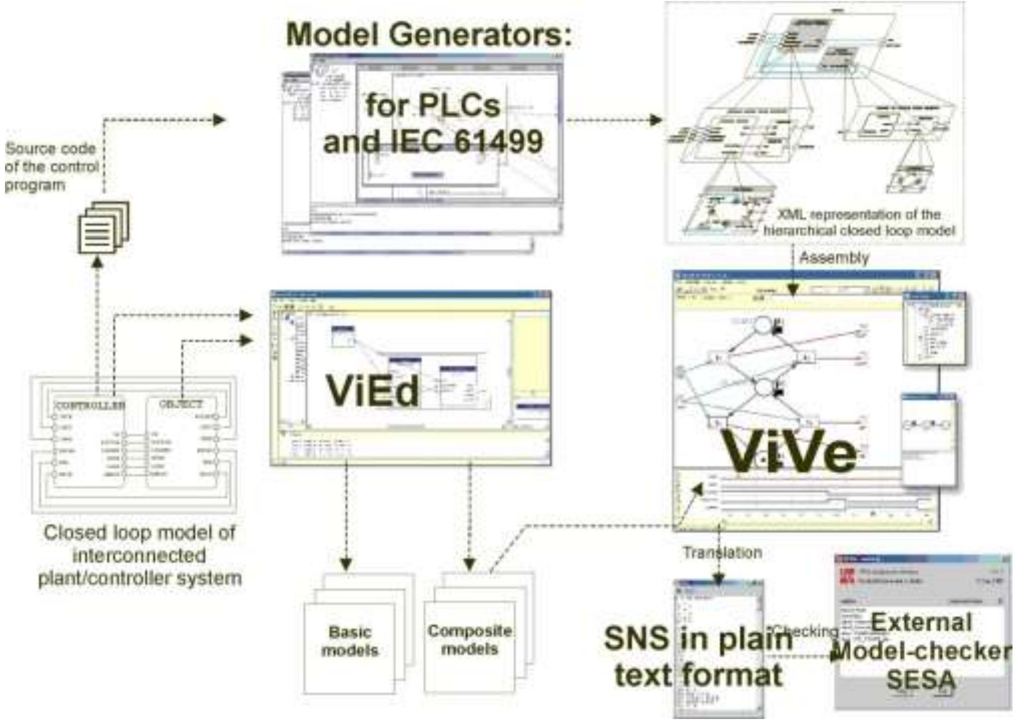


Figure 1. Tool framework for modelling and verification

The functions of the tools are as follows:

- Visual NCES editor (**ViEd**) providing full graphical authoring and editing of the models. Its manual is provided in a separate document;

- Visual Verifier (**ViVe**) – an integrated tool that contains a model builder (assembler), a translator to the flat format for subsequent model-checking, interfaces to several model-checkers, and the means for analysis of scenarios (e.g. their visualization in form of state/time diagrams), or even system simulation along the selected scenarios.
- The model checker **SESA** allows for efficient model-checking of fairly complex systems (millions of discrete states);
- The application methodologies are represented as libraries of standard model elements and by the web-based documentation;

The NCES modelling language is open – an XML based data format allows the development of add-ons to the existing tools, for example model-generators for particular programming languages in which the controllers are programmed.

The graphical editor provides full graphical authoring and editing of the models. The editor uses an open XML-based data format for basic and composite NCES models. The data format of composite model blocks intentionally was made identical with that of IEC61499 function blocks, supported by tool (*FBDK*).

The integrated environment Visual Verifier inputs the model type files given in XML and is capable of:

- Assembling a composite, hierarchically organized model from modules contained in different libraries. The component model types are instantiated into NCES modules.
- Translating the model into a “flat” NCES with the through numbering of places and transitions. The inter-module connections are converted into event and condition arcs between places and transitions. Thus the module boundaries are removed and the model-checking tools can be applied. In particular, the translator generates files in the input format of SESA model checker.

To enjoy the benefits of graphical formalisms the model authoring and maintenance have to be supported in a visual intuitive way. The evolution of graphical tools is described in the following section.

1.2 History of NCES developments

The first version of the tool for editing Timed NCES (TNCES) for was implemented at the University of Halle, (Germany) as a template to **Visio** universal graphic editor. The editor supported only the non-typed approach which did not allow for convenient re-use of previously developed model components. The whole model needed to be developed from scratch and the re-use was possible only by “cut and paste” of some model elements.

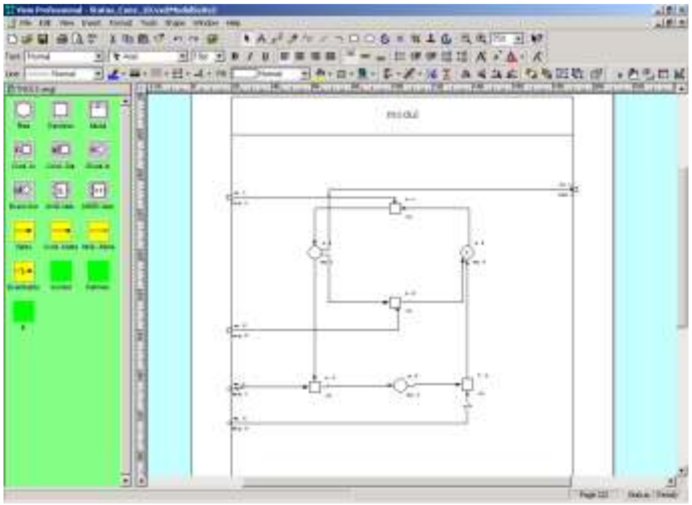
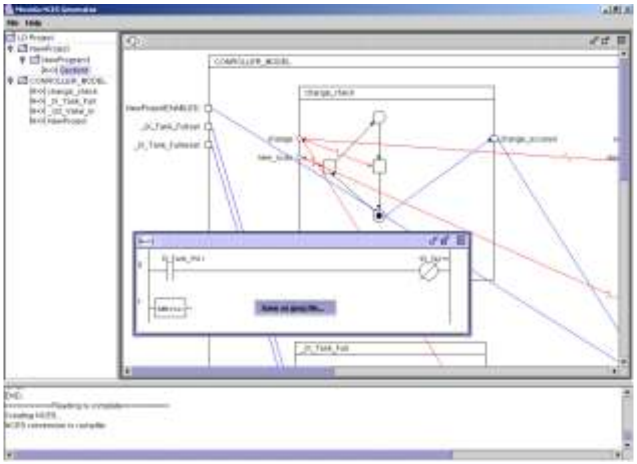


Figure 2. Visio TNCES Template.

The need to re-use models pushed the development of an open XML-based data format for basic and composite NCES models. The data format of composite blocks was intentionally made identical with that of IEC61499 function blocks, supported by tools [17, 18]. Then the export to the XML format was added to the editor in order to create a model type out of a single NCES module. This way the former **Visio**-based editor could be used for populating the library of basic model types, while FBDK could be used for creating complex model types.

However, FBDK is lacking convenience in dealing with module connections. Besides the use of three software tools just for editing models is too complicated. For this reason another editor (ViEd) was conceived that integrates editing of basic and composite models in fully intuitive visual way.

The model of a controller can be generated by the MOVIDA NCES Generator (Fig. 10). The generator takes as an input source code of controllers in several PLC programming languages (for example Omron^{TM}) LD project represented as a textual file), converts it into TNCES, and saves the data in XML based format. The openness and self-explanatory XML representation simplifies the development of the tools that may work with TNCES.



```
END:
===========Reading is complete===========
Creating NCES...
NCES conversion is complete
```

Figure 3. MOVIDA NCES Generator.

2 Signal/Event Nets

2.1 Introduction

In this chapter we give informal definition of Signal/Event nets (S/E nets). A more formal definition is presented in Annex 1 and in the document “Analyzing Signal / Event Nets” [21].

The formalism of Condition/Event systems, suggested by Sreenivasan and Krogh in (1990), provides a convenient framework for modular modelling of discrete-event systems. Internal content of modules can be different: so far finite state and hybrid automata [3, 4], as well as Petri net-like formalisms [7] have been studied in this role.

The Condition/Event model can serve to represent systems’ interface abstractions, internal structure and behaviour of single elements. This model can be easily mapped then onto IEC61499 function blocks [23], thanks to many similarities, namely event and data interfaces and State Chart definition of functionality of single modules.

2.2 Syntax

A Signal/Event net is a place/transition model similar to Petri nets [4-11]. Basic artefacts of the place/transition models are: *places*, which can bear *tokens*; *(net) transitions*, and *arcs* connecting places with transitions and transitions with places, known as *token flow arcs*. S/E nets in addition have two types of arcs: *event arcs* from transitions to transitions (e.g. (t2, t4)), and condition arcs from places to transitions e.g. (p2, t5). The model in Figure 1 is an S/E net.

A state of a place/transition model is determined by marking of its places, i.e. allocation of tokens across the places. Tokens can “flow” from state to state in some discrete moments according to the set of rules, known as “model semantics”. Such a “jump” of tokens leads to a new state of the model, and is called *a state transition*.

It is said that net transitions can *fire* and transfer hereby tokens from a place to place.

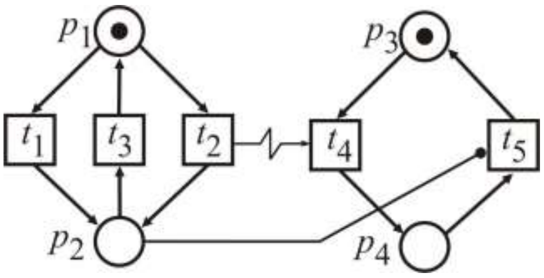


Figure 4. A Signal/Event Net (book/example1).

2.3 Semantics

The semantics of Signal/Event nets is defined by the firing rules of net transitions. There are several conditions to be fulfilled to enable a net transition to fire.

First, as in the 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 as shown in **Figure 5** (or, in case of weighted arcs, with as many tokens as the weight of the corresponding arc from the pre-place to the transition.)

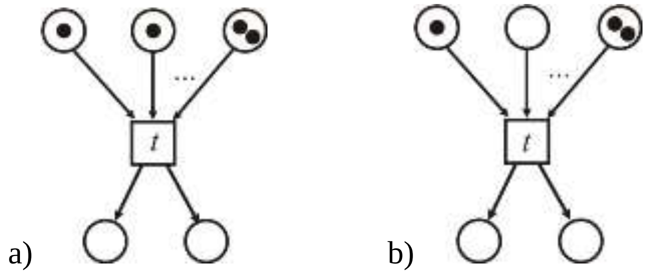


Figure 5. Token concession of transition: a) transition t has token concession; b) there is no token concession.

In addition to the flow arcs from places, a transition in S/E net 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 (more rigorously – as many tokens as the capacity of the flow arc), i.e. if more than one condition arc is connected to a place, the overall influence of the condition arcs is decided by the “AND” of each single arc enableness, as shown in **Figure 5**.

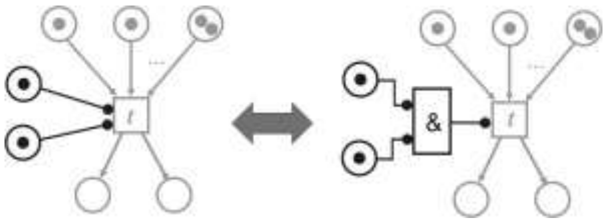


Figure 6. If more than one condition arc is connected to a place, the overall influence of the condition arcs is decided by the “AND” of the each single arc ‘enableness’.

Another type of influence on the firing can be described by event signals which come to the transition from some other transitions in the net. With respect to incoming event arcs a transition can have either OR or AND mode (event signal sensitivity mode). The default event signal sensitivity mode of transition is OR, as shown in **Figure 7**.

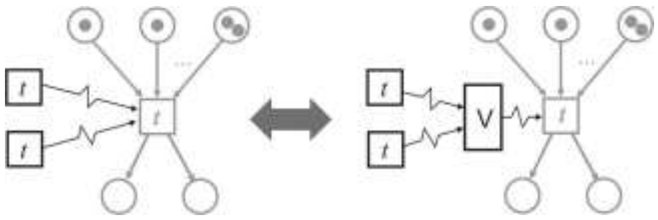


Figure 7. The default event signal sensitivity function of forced transition is OR.

Transitions having no incoming event arcs are called *independent*, otherwise *forced*. A forced transition is enabled if it has token concession and it is enabled by condition and event signals.

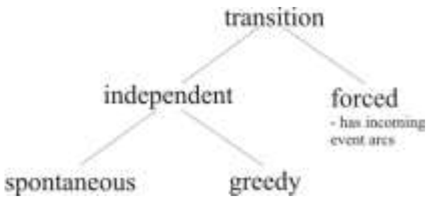


Figure 8. Firing mode of transition.

Several S/E net transitions can fire simultaneously. A set of such simultaneously firing net transitions is called *step*.

A step is formed by first picking up a nonempty subset of enabled spontaneous transitions, and then by adding as many as possible of enabled transitions which are forced to fire by event signals produced by the transitions already included in the step. Such a step is called *maximal* with respect to its forced transitions.

2.4 Conflicts and non-determinism

A *conflict* in classic Petri nets occurs when the number of tokens in some places is “not sufficient” to fire all transitions connected to them by flow arcs. This situation is exemplified in **Figure 9**, a.

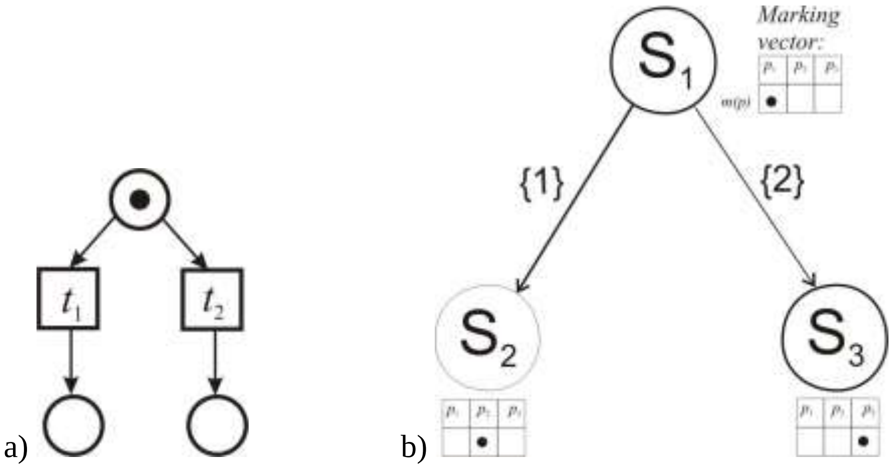


Figure 9. Conflict (a) and reachability graph (b) (book/simple_conflict).

In case of a conflict, not all transitions can fire simultaneously. The reachability graph in **Figure 9**,b shows that there are two steps ‘fireble’ in this state of the model: {t1} and {t2}. Since both these steps can happen, it is said that the choice is non-deterministic. In case if such a model is used for simulation either of this transition steps can happen. In the reachability graph, however, both options are included.

2.5 Condition arcs

Tokens do not flow through condition arcs, so one place with a single token in it can enable many transitions and no conflict will arise, as illustrated in Figure 10 for the place p3.

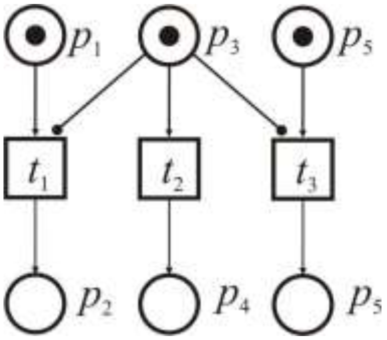


Figure 10. Single token in p3 is sufficient to enable transitions t1,t2 and t3, so no conflict is observed in this situation.

2.6 Arcs with capacities (weights)

The token flow and condition arcs can have capacities determining the number of tokens that will flow through the arc (for token flow arcs), or needed to enable the corresponding destination transition (for condition arcs). If a source place has less tokens than is required then the transition would not get the concession. A net with arc capacities is illustrated in Figure 11.

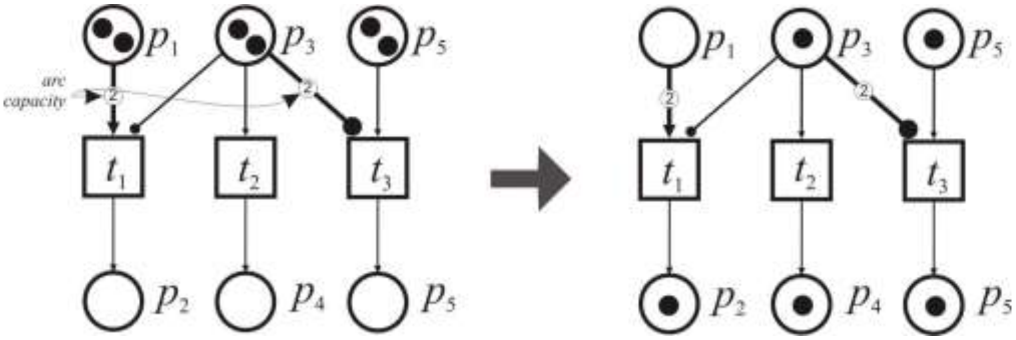


Figure 11. S/E Net with arc capacities.

For example, the flow arc from p1 to t1 has capacity 2, and the condition arc from p3 to t1 has capacity 1. Both places p1 and p3 have 2 tokens, so the transition t1 is enabled. The transition t2 is enabled because it has only one flow arc from p3 and there are enough tokens in p3. The transition t3 is enabled because p5 has as many tokens as required (1) and p3 has as many tokens as required (2). Note, that only one token moved from p1 to p2 and one got lost since the capacity of the arc (t1, p2) is only 1.

Also note, that in the next state transition t3 would not be enabled although p3 still has one token. This is due to insufficient number of tokens in p3 to 'activate' the condition arc (p3,t3) which has capacity 2.

2.7 State and reachability

A state of an S/E net is defined by marking of all places. A tuple M=<Z,R,s_0> denotes the reachability structure of a S/E net, where Z is a finite set of reachable states, R is a finite set of *state transitions*[1], and s_0 is an initial state.

A state trajectory is a sequence of states (s_i)= s_0, s_1, …, s_i, … , such that for each pair s_j, $s_{j+1} \in Z$ there is $\tau \in R$ such that s_{j+1} is reachable from s_j by the transition τ (in mathematical terms denoted as s_j [τ> s_{j+1}) . Figure 12 presents the reachability graph for the S/E net from Figure 4.

[1] Note the fundamental difference between *net transition* and *state transition*.

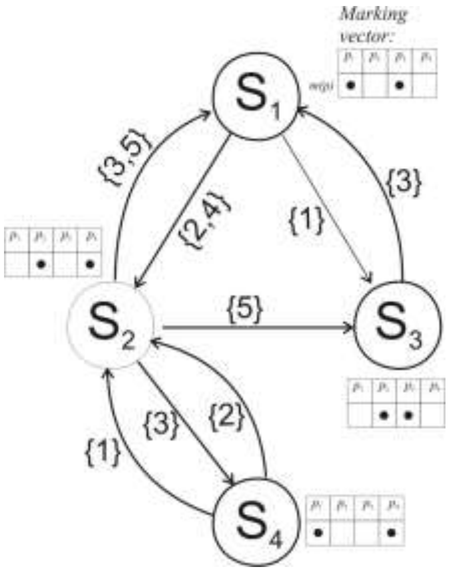


Figure 12. Reachability graph of the model from **Figure 4**.

Nodes of the graph correspond to the states while the arcs correspond to the state transitions. The arcs are marked with their respective steps of net transitions.

2.8 State transition modes

There are three ways to generate the transition step w.r.t. spontaneous transitions:

1. Include all possible combinations of spontaneous transitions (this was illustrated in the previous section in Figure 12);

2. Include only one spontaneous in a step (The corresponding reachability graph is shown in **Figure 13**, a);

3. Include maximum number of spontaneous transitions (the reachability graph is shown in **Figure 13**, b);

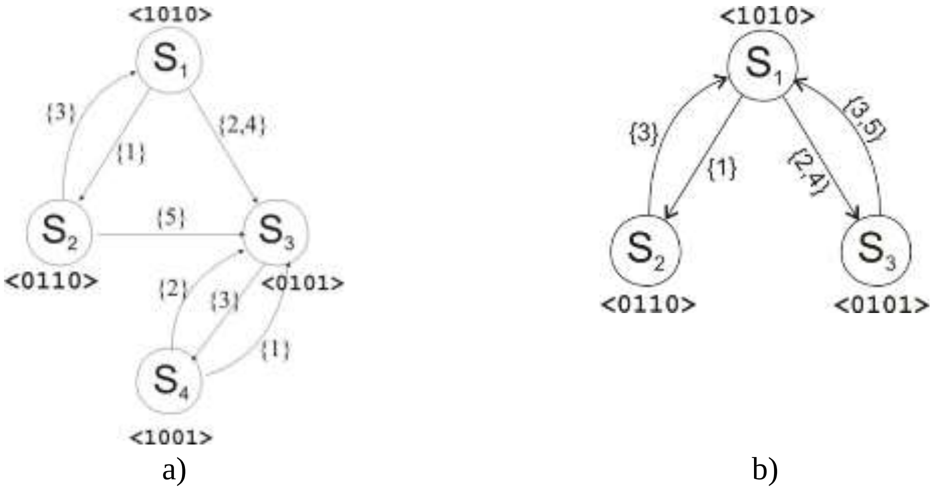


Figure 13. Reachability graphs of the model corresponding to a) single spontaneous transition; b) maximal set of spontaneous transitions.

In all cases forced transitions are included in steps according to the principle of maximal set of forced as discussed in the previous section.

2.9 Synchronous transitions

There are special means provided for description of both asynchronous and synchronous behaviour in the same net, which are especially useful for modelling of interconnected plant/controller systems. This is achieved either by introduction of *synchronous* transitions, firing whenever they are enabled, or by timing.

If a transition is marked with the synchronous (or greedy) attribute, it fires always when enabled. Synchronous transitions should not have incoming event arcs. When a firing step is formed, these are treated as spontaneous, with exception of that all enabled greedy transitions are always included in the step. Let us illustrate the work of greedy transitions on the example in **Figure 14**.

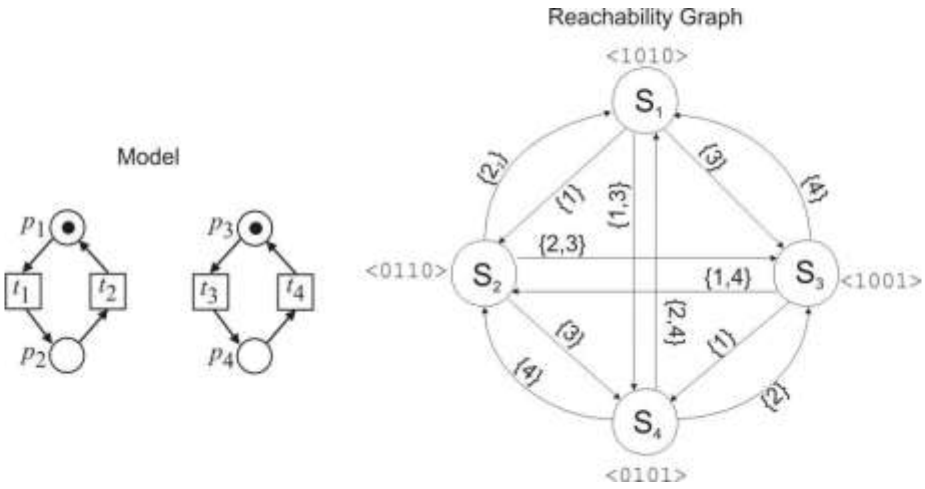
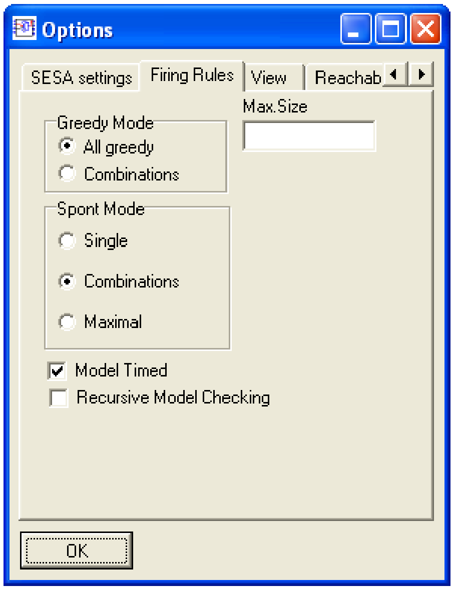


Figure 14. Reachability graph of the model with all spontaneous transitions.

As one sees, the model’s behaviour includes all possible combinations of t1 and t2 with t3 and t4.

This example is provided in the Visual Verifier set of samples as TestSimple2Spont.xml. Check it with the options: Maximal set of greedy transitions and Combinations of spontaneous transitions as illustrated in **Figure 15**. The selected firing mode implies that all greedy transitions will be included in the step and all possible combinations of enabled spontaneous transitions will be added. If the set of Greedy is not empty, then the combination with empty spontaneous set will be also considered.



Greedy Mode
● All greedy
○ Combinations

Spont Mode
○ Single
● Combinations
○ Maximal

Max.Size

☑ Model Timed
☐ Recursive Model Checking

Figure 15. Selection of the firing modes in the Visual Verifier.

In the next example (Figure 15, TestSimple1Spont1Greedy.xml) two transitions are left spontaneous, while two others are made greedy. As a result, some trajectories have disappeared from the reachability graph.

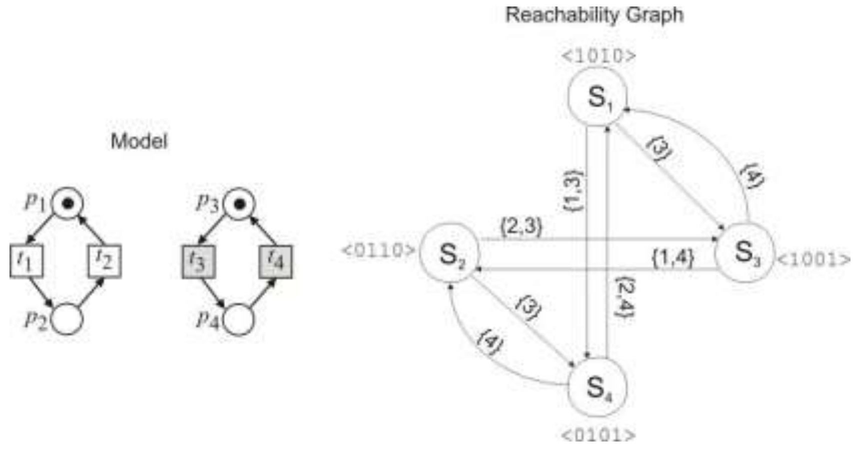


Figure 8. In case if two transitions are spontaneous and two others are greedy, the possible step combinations are limited to those where a greedy transition is always included in the step.

If there is more than one greedy transitions enabled in the moment, they are included into step similarly to spontaneous transitions, i.e. steps are formed from all possible combinations of greedy, as shown in Figure 16, where all four transitions are greedy.

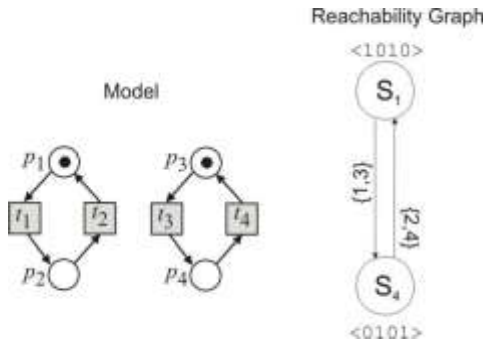


Figure 16. All transitions are greedy.

Note: The “greediness” of transitions can be only used in non-timed models. A similar concept can be achieved in timed models by using synchro sets introduced later in Chapter 14.2.

2.10 Transitions without incoming arcs

A transition without any incoming arcs is always enabled.

2.11 Priorities

In place-transition modelling formalisms a priority is an integer attribute of a transition determining preference of its firing with respect to other enable transitions. Only the transitions with the highest priority (from the set of currently enabled transitions) are included in the executable step. To avoid ambiguities, in S/E Nets priorities can be assigned only to spontaneous transitions.

2.12 Firing rules

Visual Verifier supports several firing rules. The set of firing rules of SESA is a bit different. The reasons for having different firing rules are in the history of these tools. However, having several firing rules available may better fit to particular details of different models.

The firing rules of the Visual Verifier are as follows:

- single spontaneous can fire ;
- all combinations of spontaneous transitions will be considered;
- only the maximal combination of spontaneous can fire.

Certainly for each set of spontaneous transitions as many as possible forced transitions are added to form a step. This is called “maximum step”.

In SESA two firing rules are supported:

- single spontaneous can fire ;
- all combinations of spontaneous transitions will be considered to form the maximum steps on their base;

  In addition, the “greedy” transitions are treated in the VisualVerifier a bit differently from SESA.

  In ViVe two options are provided:
- fire all enabled greedy transitions together or
- consider all combinations of the greedy;

This is applied ‘on top’ of the spontaneous firing rule.

In SESA greedy transitions are treated as normal spontaneous transitions.

3 Modular S/E Nets = Net Condition/Event Systems

The formalism of Net Condition/Event Systems (NCES) was introduced by Rausch and Hanisch in *(Rausch and Hanisch, 1995)* and further developed through the last years, in particular in *(Hanisch and Lüder, 1999)*.

3.1 Encapsulation of models into modules

The general idea of Net Condition/Event systems supports the way of thinking of and modelling a system as a set of modules with a particular dynamic behaviour and their interconnection via signals. An illustrative example of the graphical notation of a module is provided in Figure 17.

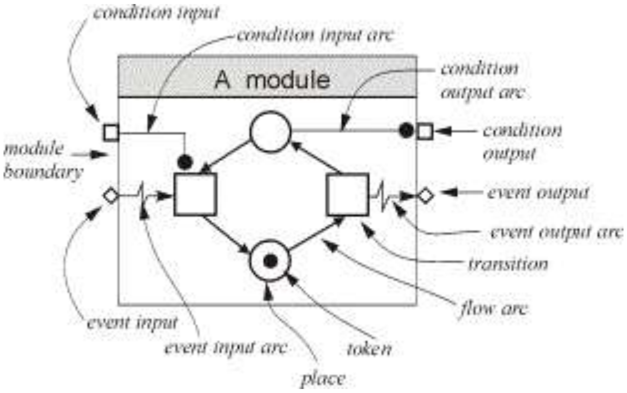


Figure 17. Graphical notation of a module.

Once designed, the modules can be re-used over and over again. Each module has inputs and outputs of two types:

1. Condition inputs/outputs carrying information on marking of places in other modules, and
2. Event inputs/outputs carrying information on firing transitions in other modules.

Condition and event inputs are connected with some transitions inside the module by condition and event arcs. Places of the module can be connected to the condition outputs by condition arcs, and transitions can be connected to the event outputs by event arcs.

This concept provides a basis for a compositional approach to build larger models from smaller components. The "composition" is performed by "gluing" inputs of one module with outputs of another module as shown in Figure 18.

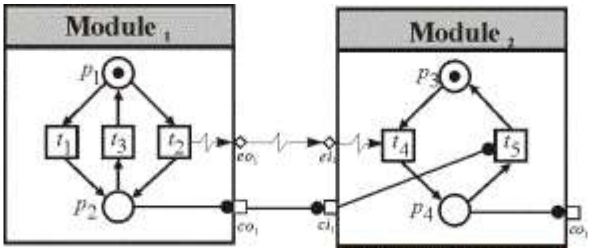


Figure 18. Modular composition.

The result of the composition of two NCES N_1 and N_2 is an NCES N_c obtained as a union of the components and which can be represented as a new module. Inputs and outputs of the "composition" are unions of the components' inputs and outputs, except for those which are interconnected to each other, and hereby "glued", i.e. substituted by the corresponding condition and event arcs, as shown in Figure 19. By the way, the resulting module is equivalent to the S/E from **Figure 4**.

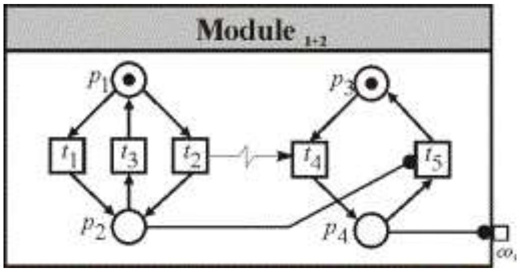


Figure 19. Result of the modular composition.

3.2 Model type definition

In the version of NCES implemented in Visual Verifier a model must be encapsulated in a module. A module is defined by its **interface** and **content**. The interface contains a model name and names of event and condition inputs and outputs. The content can be either a place-transition model, i.e. consist of places, transitions and arcs as described in the previous section (such model types are called **basic**), or be a network of modules interconnected via event and condition arcs (such models are called **complex**).

Once defined and placed in the library, a module defines a **model type**. The module name serves as the type identifier. Type instances can be used over and over again in the complex models (strictly speaking, the modules forming the complex models have to be instances of other modules).

As a consequence of the above definition a model can have a hierarchical structure as the one presented in Figure 20. The hierarchical structure can be transformed into a plain S/E Net by instantiation of a model types.

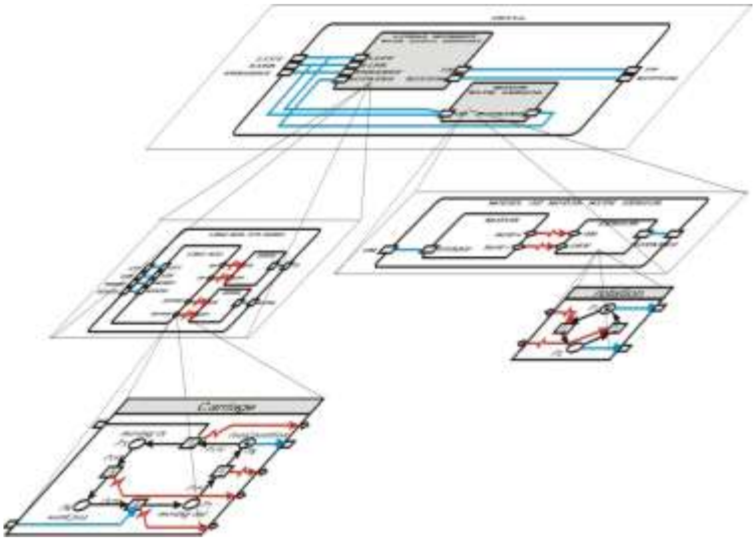


Figure 20. A hierarchical NCES model**.**

Dynamic models of complex objects usually consist of models of their constituent components interconnected by event and condition signals. They may also include an additional model that integrates and coordinates them. Such a "master supervisor model" can also take care about input-output behaviour of the composite model.

3.3 Typed NCES

Further in this text we are using only the typed NCES modelling. This approach is based on the following postulates:

1. All NCES models are encapsulated into modules. A *module* has *interface* that is defined by event and condition inputs and outputs. A modular model, stored in a separate file, defines a *model type* that can be later *instantiated*.

2. NCES models can be *basic* or *composite*.
3. A *basic* NCES model type consists of *places, transitions* and *arcs*. It **cannot** have any nested modules.
4. A *composite* NCES model type consists of *module instances* and *arcs* connecting I/Os of the modules to each other and to the interface elements of the model. The instances are obtained by instantiation of the model types, basic or composite, existing in a storage media (library).

The process of model development can follow both top-down and bottom-up approach. In the former case you may create new module interfaces and as needed specify them and store as model types in the library. After that you can reuse the models over and over again.

In the latter case you start with development of most basic model elements and save them as basic model types in the library. More complex models can be created as composite types using instances of the basic ones. This way you can create hierarchical models of arbitrary complexity always remaining flexible and reusing the repetitive sub-models.

3.4 Capacities of condition arcs

Condition arcs between NCES modules, or between a module and inputs/outputs of another module where its instance is included, can have capacities, that are integer numbers >= 1.

The capacities between modules can differ from the capacities of arcs within modules. When the modules are “glued” into a single S/E net, the capacities of resulting condition arcs are calculated as the minimum capacity of the segments forming them.

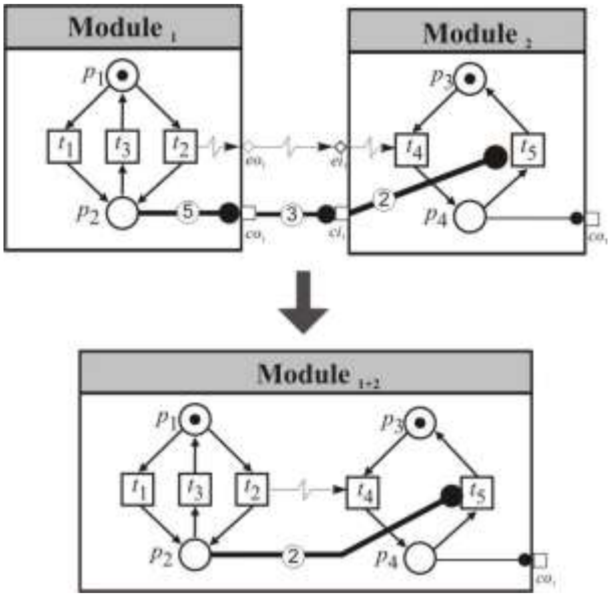


Figure 21. The capacity 2 of the condition arc (p2, t5) is obtained as the minimum of capacities of the arcs forming its segments within modules and between modules.

3.5 Benefits of NCES

There are two main reasons to prefer place-transition formalisms to many others formalisms, e.g. finite automata. The first is their non-interleaving semantics (i.e. possibility of firing several transitions simultaneously), which better fits to modelling of distributed processes and of their interaction.

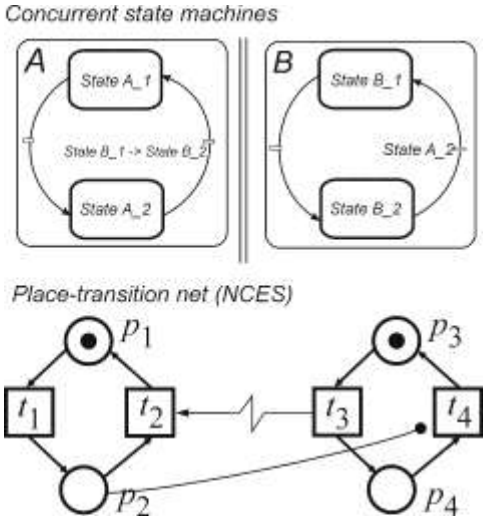


Figure 22. Model of two processes as parallel composition of state machines or NCES.

The second reason is the more compact reachability space, explained as follows.

Modelling of complex distributed systems with automata usually ends up in many concurrent automata models communicating via common variables, as illustrated in Figure 23, left, where two state machines A and B are combined under “asynchronous parallel operator”. Thus, the overall system model is a cross-product of the component automata, and to do model analysis it is necessary to build the cross-product consisting in this case of 9 states, as one sees in the right part of the Figure.

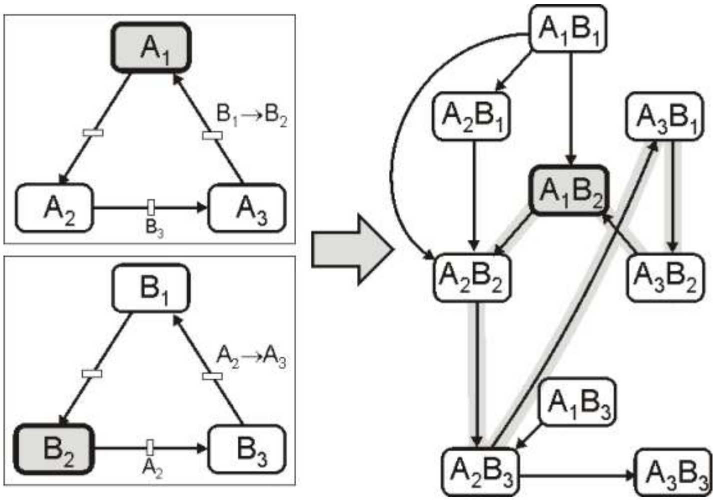


Figure 23. Modelling of two communicating processes by means of concurrent state machines and their cross-product automaton.

Alternatively, in Signal/Event Nets a state of a model is determined by the marking of model places, so any global state of a distributed system is just one state of the model. This is shown in Figure 24 where the same model is implemented in Signal/Event Nets with places (p_1-p_6) corresponding to states of the automata A or B (in the obvious manner) (Find it in the concurrent.xml file).

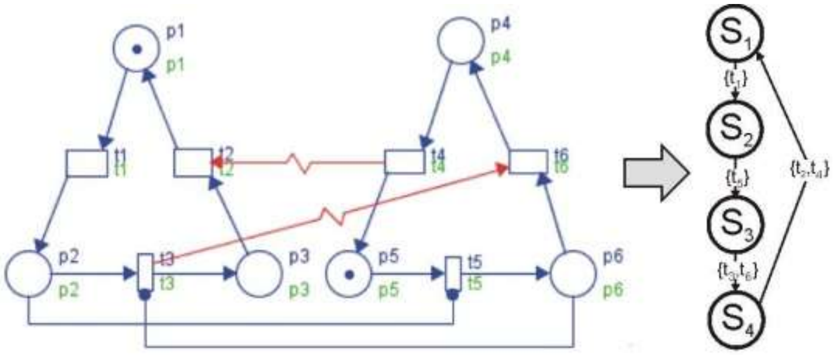


Figure 24. The same model implemented using place–transition nets (S/E Net) and its reachability graph.

In the given initial state the reachability space of the model consists of only 4 states. The same behaviour obviously will be shown by the automata model in Figure 23 (the outlined path A_1B_2→ A_1B_2→ A_2B_2→ A_2B_3→ A_3B_1→ A_3B_2), but to get it the whole cross-product automata needs to be built.

4 Semantics of Modular Models

The modularity of NCES does not bring any extra semantic issues if compared to S/E nets since the module boundaries are removed during the flattening process. However, some “tricky” issues in S/E nets and NCES semantics need to be discussed.

4.1 A condition/event input of a module is not assigned

When an input is not assigned as shown in **Figure 25** there are several possible interpretations.

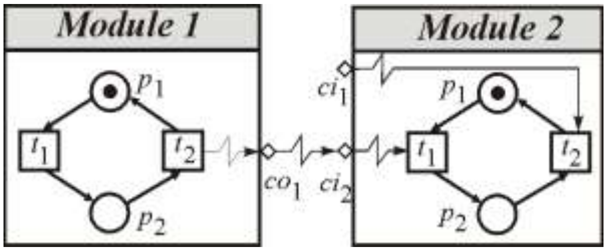


Figure 25. Not assigned input of a module.

The one shown in Figure 26 removes the event arc (ei_1, t_2) making the transition t_2 spontaneous.

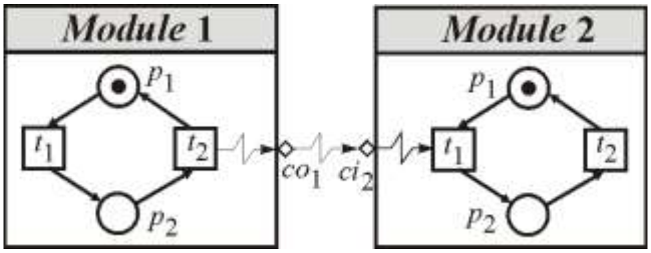


Figure 26. First interpretation: the event arc is removed, transition t2 in the Module 2 becomes spontaneous.

However, this interpretation might not always reflect the intentions of the developer of the module 2, as the presence of the incoming event arc might indicate the forced nature of the transition t_2. Thus, the absence of any input arcs to the input ei_1 may mean that t_2 should not fire at all. This can be implemented as shown in Figure 27 by adding a module (Module 3) with a transition (t_1) that never fires, connected to t_2 of Module 2.

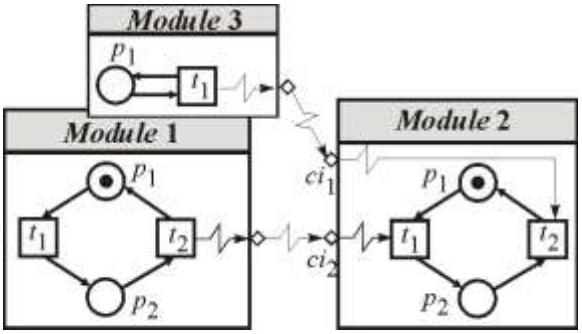


Figure 27. Second interpretation: an always dead transition

4.2 Multiple arc assignments to a module's input/output

Multiple arc connections to inputs and outputs of modules as those shown in **Figure 28** were not allowed in the previous versions of NCES due to ambiguities in interpretation.

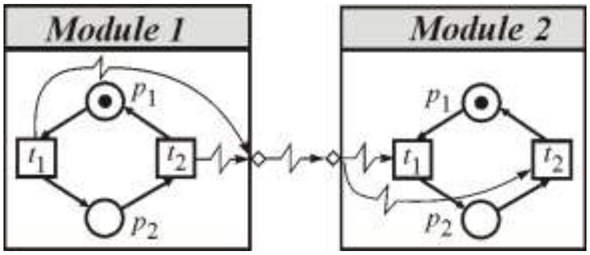


Figure 28. Multiple assignments of arcs to I/Os.

However, since the signal arcs eventually influence the firing of transitions, we can shift the semantic load to the definition of the firing function of transitions, and interpret the concentration of arcs at inputs and outputs by connecting places/transitions in the resulting S/EN with an arc if a connected path exists from the corresponding source place/transition to the target transition in the original NCES. This is illustrated in Figure 29.

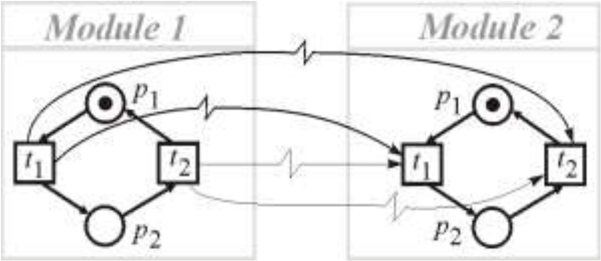
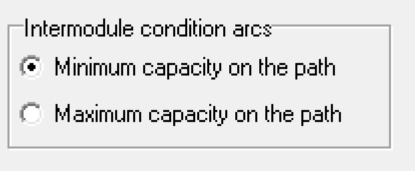


Figure 29. Signal arcs in S/E N as a result of multiple arc resolution in NCES.

4.2.1 Condition arc weights between modules

ViVe provides two options for resolving the weight of the condition arc appeared as a result of structural composition (Options/NCES tab).



Intermodule condition arcs

◉ Minimum capacity on the path

○ Maximum capacity on the path

Thus, if the first option is selected, the arc with the minimum weight determines the weight of the resulting arc after the assembly.

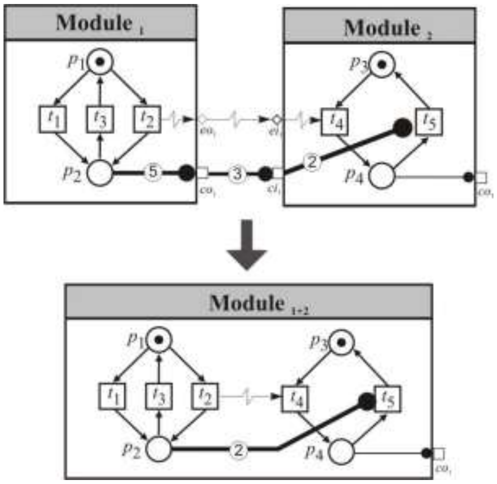


Figure 30. Multiplicity of the resulting condition arc.

4.2.2 Several condition arcs originating in the same place

In process of assembly there could be a situation of several condition arcs ending in the same transition and originating in the same place.

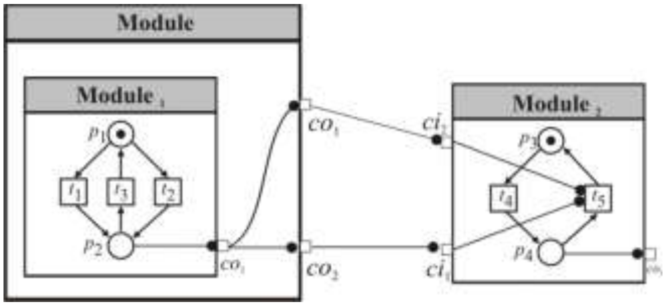


Figure 31. Two condition arcs originating in p2 end in t5.

There are two options to resolve this situation. The first option is to take the maximum capacity across all paths leading from p2 to t5. The result in this case would be as shown in Figure 32.

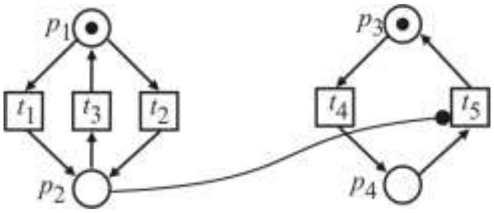


Figure 32. Maximum path capacity is taken (1 in this case).

Another option is to assign the cumulative capacity to the resulting arc.

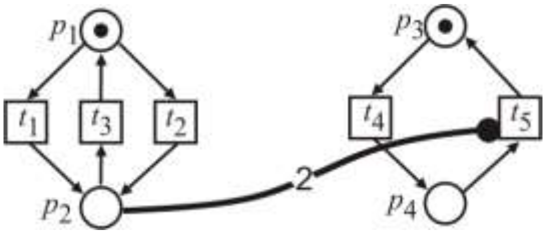


Figure 33. Sum of capacities is taken.

4.2.3 Visual Verifier support of non-assigned module inputs

Ambiguous issue	Support in ViVe	Interpretation
Non assigned event input	Supported	The destination transitions are connected to the “always dead transition”
Non assigned condition input	Supported	The destination transitions are connected to the “always empty place”
Multiple arcs to an event input or output	Supported	The destination transition is connected to the transitions where the arcs are

		originating from.
Multiple arcs to a condition input/ output	Not supported	

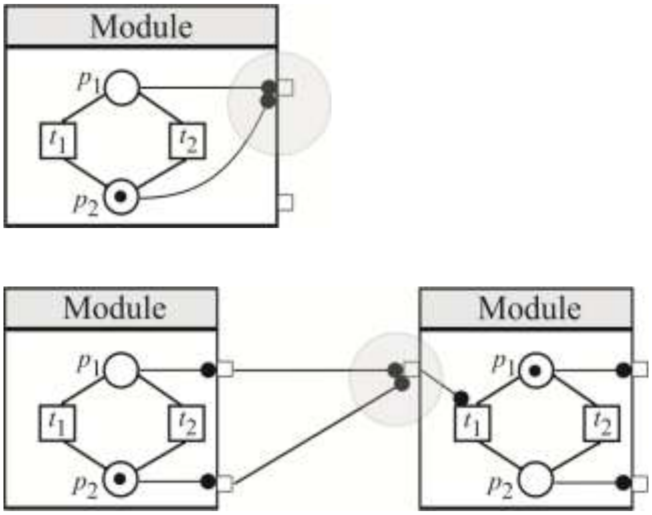


Figure 34. Prohibited condition connections.

5 Timed models

5.1 Discrete timing

The concept of discrete timing is applied to the S/E nets as follows: to every pre-arc [*p*, *t*] of the transition *t* we attach an interval [*l*, *h*] of natural numbers with $0 < l < h < \infty$. 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, ∞]. The interpretation is as follows. Every place *p* bears a clock *u(p)* which is running **iff** (if and only if) the place is marked (*m*(*p*)>0), and is switched off otherwise. All running clocks run at the same speed measuring the time the token status of its place has not been changed. 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 $l(p,t) < u(p) < h(p,t)$.

An example is given in Figure 35.

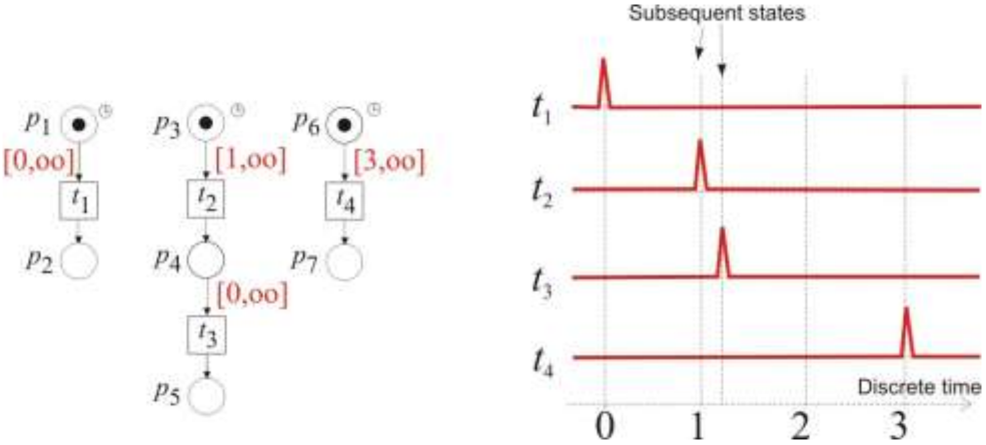


Figure 35. Timed S/E net and firing of its transitions.

Thus, in timed NCES a state is characterized by the marking of places plus the values of local clocks at the places.

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.

There are two slightly different interpretations of time in different NCES implementations. Let us consider illustration in **Figure 36**, a.

In one interpretation (implemented in SESA model checker), time delay is an attribute of the state where the transition originates. If in state S_i there is such a minimum increment Δ that some of the transitions become enabled after elapsing it, then it is said that the state transition τ:S*i*→S*j* has a "delay" Δ. Conversely, it can be interpreted as the state S_i has a "duration" Δ that specifies the time increment of the clocks of this state required to make the transition enabled. So, first the time elapses, and then a state transition occurs. This is illustrated in **Figure 36**,b.

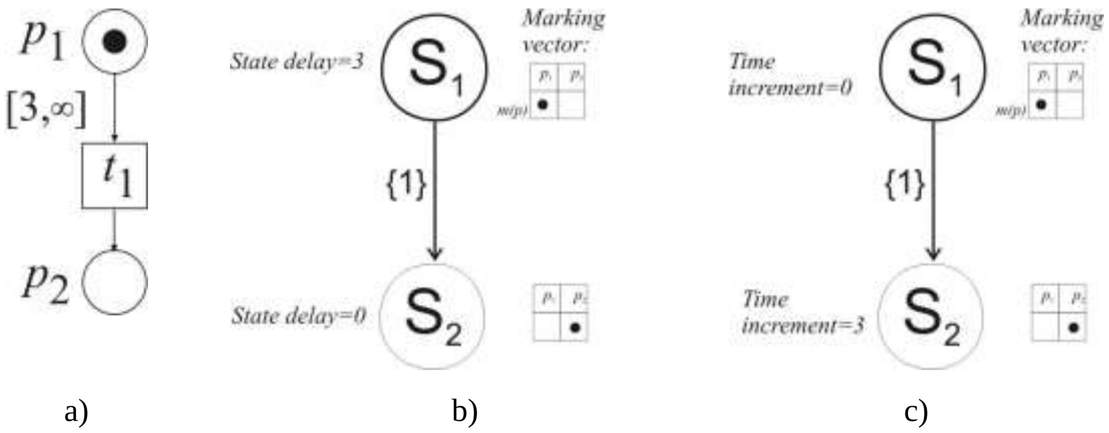


Figure 36 An example of timed S/E net and reachability graphs for two time interpretations.

Another interpretation (implemented in Visual Verifier) uses the concept of time increment. This attribute belongs to the state where the state transition leads to. The reachability graph generated along with this interpretation is in **Figure 36**,c. This interpretation allows interpret the elapsed time as an (implicit) attribute of the state transition.

Although in this example, the number of states in both reachability graphs is the same, in general it can be different.

5.2 Firing rules in TNCES

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 the timing firing rule.

1. **Strong vs. weak firing**: with the *strong* rule all marking enabled (spontaneous) transitions, which have pre-places with clock position equal to either 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). 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 has a pre-place with the clock value equal to the low bound l of the time interval.
3. **Ultimo firing**: is a certain combination of the interval and strong rules: a transition is time-enabled at every discrete time moment within the interval and must fire at the latest at clock position equal to h.

In case if a transition has several incoming arcs with permeability intervals [l_1,h_1],[l_2,h_2],…[l_n,h_n] then, to enable the transition all arcs have to be permeable, which is achieved in the interval [l,h], where l=max(l_i), h=min(h_i).

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, or 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=∞	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 modelling.

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 explodes the reachability space. Perhaps, the variety of choices discussed in this Chapter is a bit confusing, but it extends the modelling horizons and allows more concise description of models. The following example explains the differences between firing options.

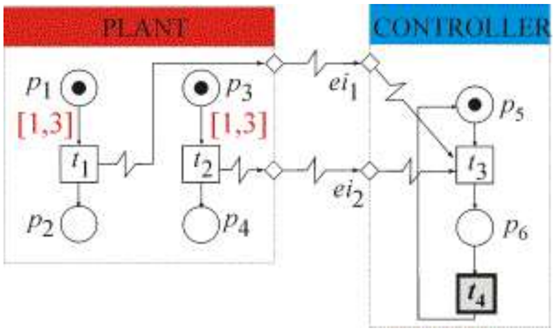
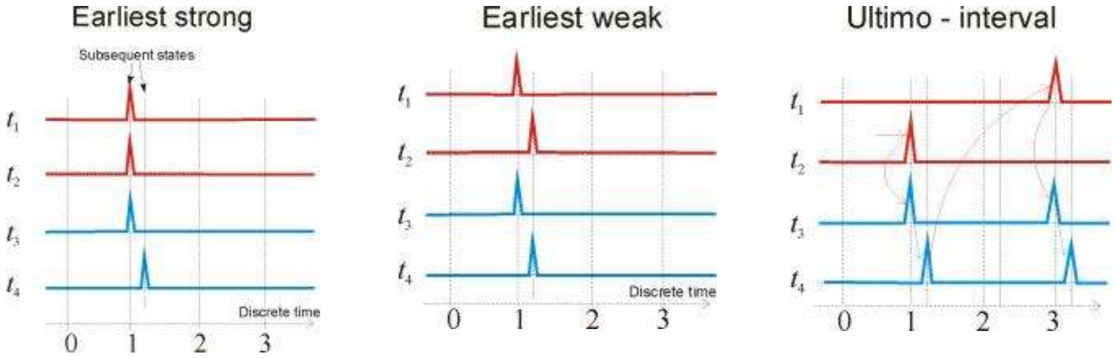


Figure 37. Timed version of a plant-controller interaction model

The following state-time diagrams illustrate different combinations of timed firing options. The earliest strong firing rule forces to fire all transitions when the low time bound is reached by clocks, at the earliest weak rule steps are formed from combinations of time-enabled transitions, at ultimo the firing may occur at every discrete time value within the permeability interval.



a) b) c)

Figure 38. State-time diagrams representing different combinations of timed firing options.

5.3 Implementation

Not all timing modes are currently implemented in the available model-checkers.

5.4 Restrictions

A transition with incoming timed arcs (i.e. [*l*, *h*] where *l*>0) cannot be forced, i.e. cannot have incoming event arcs.

6 Modelling of closed-loop controller-plant systems

An industrial automation system can be considered to be built from two conceptually different parts: controller and plant. The controller is a hardware device driven by software code that performs data processing, communication and decision making, the plant refers to the physical part of the equipment.

Figure 39 shows examples of such control systems. Figure 2,a shows control of the liquid level in a tank The tank has an input valve that controls the liquid supply. Once the tank is filled the valve should be closed. A level sensor (*L*) indicates the level where the filling should terminate.

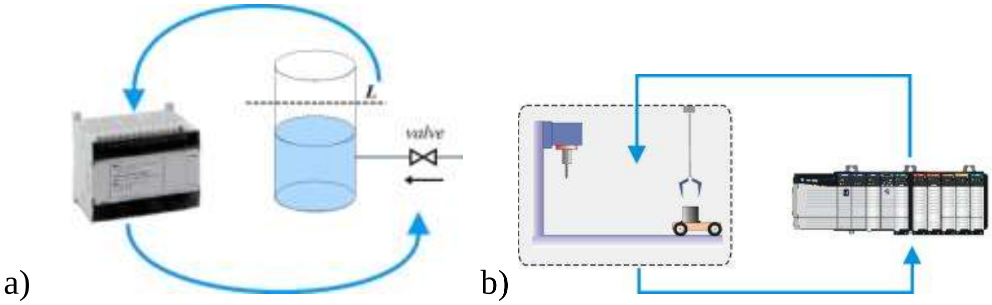


Figure 39. Examples of automated systems: a) control of the liquid level in the tank; b) manufacturing cell - an automated drilling station.

Figure 2,b presents a model of an automated machining process: drilling station. Modelling of automation systems can be done in either open-loop or closed-loop way. The open loop modelling 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.

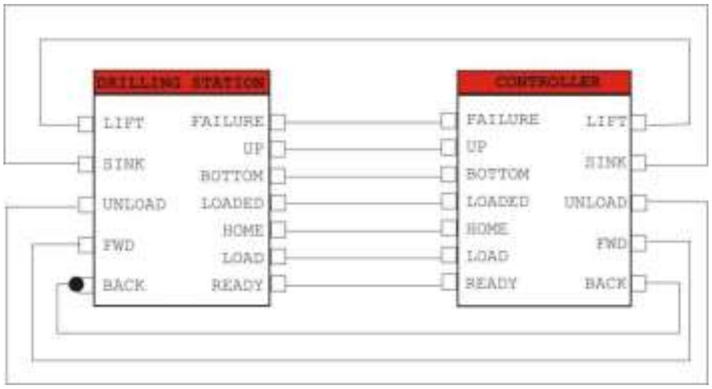


Figure 40 Closed-loop NCES model of the automated drilling station.

In the closed-loop approach exemplified in **Figure 40**, the model of the automated drilling station system is composed of two independent components: a model of the object (also known as plant) and a model of the controller, connected in a closed-loop by control signals and process data. Both parts are modelled using a common formalism. This approach allows for specification of desired/prohibited behaviour of the automation system in terms of the events/statements related to the object rather than in terms of input/output variables. The closed-loop approach is also beneficial in terms of complexity as a feasible model of plant restricts the controller’s input combinations. The model of plant not only generates the inputs of the controller but also receives the outputs and correspondingly modifies its internal state.

Certainly the latter approach is more complex as the modelling of uncontrolled reactive behaviour of objects is required. However its benefits overweight the extra work needed. Both parts of the system (plant and controller) are modelled by NCES modules with condition signal inputs and outputs. The connection between controller and plant is implemented via logic level signals which are modelled using condition arcs. Event signals are used in both models of plant and controller but not between them. In the model of plant events may be used, for example, to model the causal behaviour 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 IEC61499), as well as a lot of other internal operations: procedure calls, setting/resetting variables, etc.

The closed-loop approach allows for a number of application scenarios that can be derived from the diagram in Figure 41. The scenarios include source-code based modelling of the controller or controller prototyping by a model, as well as formal synthesis of the controller. In all cases, the model of controller is combined with a manually created model of plant.

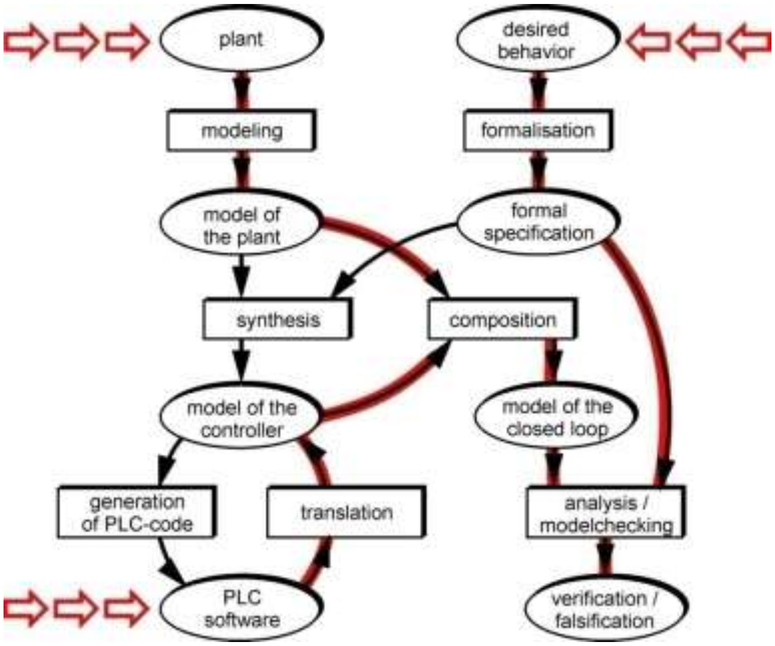


Figure 41. The framework for using formal methods in cyber-physical systems.

The prototyping scenario can be less resource-consuming during the validation if compared to the source code based model generation as the model of controller may cover only essential issues without implementation details.

Depending on the accuracy of modelling, 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 tool, or other material processing and storage units.

Modelling of discrete controllers using NCES simplifies the assembling of the model from the components. Besides, such key features of NCES as event/data connections closely correspond to the latest trends in controller design methodology presented in the new international standard IEC61499.

7 Basics of Plant Process Modelling

7.1 Processes

The behavior of a plant can be seen as a concurrent (usually asynchronous) composition of several processes, each of which has a start event *s* and ending event *e*, and some duration *D*.

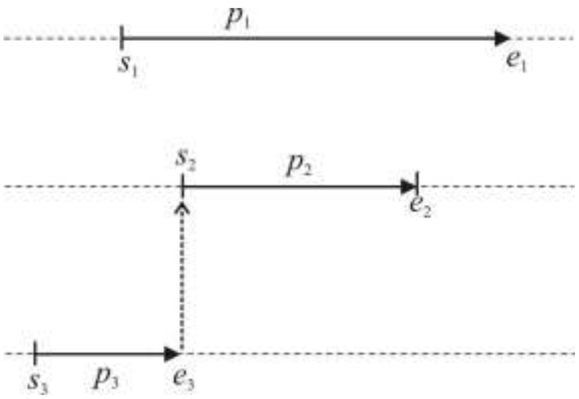


Figure 42. Asynchronous **c**oncurrent processes.
Process 3 initiates process 2 by sending a message.

7.2 Simple process model

Such simple processes can be modeled by the S/E Nets as shown in **Figure 43**.

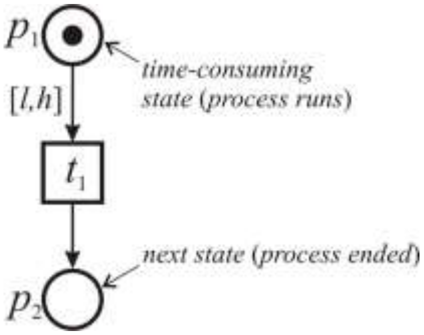

p_1
$[l,h]$
time-consuming state (process runs)
t_1
next state (process ended)
p_2

Figure 43. Model of a simple process.

Usually we model states of the plant components as S/E net places with safe (0/1) marking. In particular, for the processes with known minimal duration D>0 the time limits may look like *l*=D, *h*=∞; In this case the model contains no obligation for the action to occur.

If duration of action is not defined by an exact value, but bounded within the interval [D1, D2] (0<D1<=D2) then *l*=D1 and *h*=D2. In each state with clock value as D1<=clk(p1)<=D2 the action may finish, i.e. t_1 may occur.

Combination [0, D2] is interpreted as "no minimal duration, but maximal time limit exists".

7.3 Process with exception

To model a time consuming action (i.e. with *l*>0) with exception possibility the initial place p_1 is connected with two transitions: one of which stands for the normal operation mode with duration as described in the previous case, while the other models the exception, which interrupts the normal operation and may occur anytime within the normal operation time.

In case if h≠∞, the reached upper time limit forces only one of the transitions: either ending the action normally, or abnormally with an exception thrown.

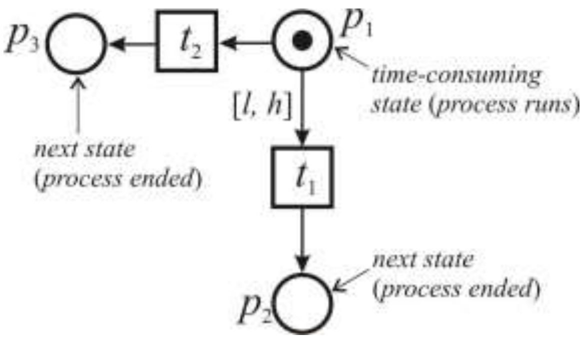

p_3
t_2
p_1
time-consuming state (process runs)
$[l, h]$
next state (process ended)
t_1
next state (process ended)
p_2

Figure 44. Model of a state with an exception.

7.4 Two time scales: ticks in controller and time-elapsing in plant

Timing can be used in both plant and controller models to achieve adequate behavior of the model. In return it allows for quantitative time estimations, or for solving optimization problems like finding control strategy with minimal duration of the technological cycle, etc. Once all the NCES modules have been interconnected into

Signal/Event Net, the resulting net has common time unit. That is why for modeling of objects with different time scale 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 suggest to accept the time scale of the most relevant processes in the plant, and assume that processes in controller (or some 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 S/E N).

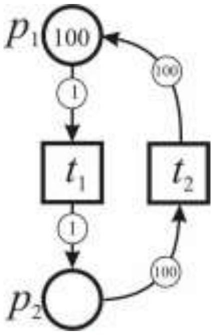


Figure 45. Program delay: model of the controller module which requires 100 commands for execution.

Modeling of a program unit which takes 100 commands for execution can be modeled as shown in Figure: initial place p1 is loaded with 100 tokens, and with every transition t1 one token flows from p1 to p2 through the flow arcs with multiplicity 1 until all the tokens come to p2. Then all the tokens come back to p1 in one transition t2 through the arcs with multiplicity 100.

8 Modelling Control Programs

Special industrial programming languages are applied for implementation of the control algorithms in Programmable Logic Controllers (PLC). The most of the known programming languages in the field were standardized in IEC 61131-3 in 1993 (IEC61131, 1993). The standard includes four programming languages: Instruction List (IL), Function Block Diagrams (FBD), Ladder Diagrams (LD) and Structured Text (ST), and a common element Sequential Function Chart (SFC) that serves for program organization into logical steps and expressing the transitions between the steps.

Despite the successful standardization of PLC programming, there is a number vendor specific programming approaches that have not been included in IEC61131-3, although they are quite popular in certain application areas.

A model of the controller can be built based on the source code of the control program. Relevant properties of system routines also have to be taken into consideration. The source code based validation gives an additional assurance in the correct behaviour of the system after commissioning.

The basics of the modelling of discrete controllers using place-transition formalisms were developed in *(Hanisch and Thieme, 1997)*. In general the modelling of controllers can be split into the following sub problems:

- Modelling of system routines such as scan cycle;
- Modelling of PLC execution is related to the performance of PLC hardware represented by times, instructions execution times, etc;
- Modelling of basic Boolean data and operations;
- Modelling of non-Boolean functions;

The use of NCES simplifies the assembling of the model from the components. Besides, such NCES features as event/condition connections closely correspond to the latest trends in controller design methodology presented in new international standard IEC61499.

The following examples serve to illustrate basic principles of mapping from commands of a programming language to NCES models.

8.1 Data storage and assignment

Boolean variable cell can be modeled by the net having two places p_0 and p_1 and two transitions t_s and t_r as shown in the Figure 1. Setting of the variable is modeled by transition t_s and resetting by t_r.

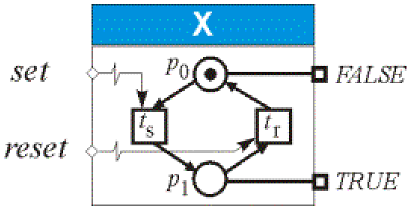


Figure 46. Model of a Boolean variable implementing SET and RESET commands.

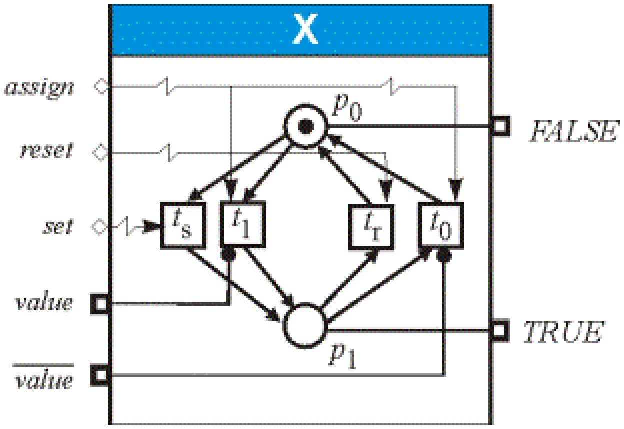


Figure 47 Model of a Boolean variable implementing also ASSIGNMENT of a value.

8.2 Linear sequence of commands

Consider how a linear sequence of two commands (e.g. A; B) can be modeled in NCES. Transitions t_0, t_1 correspond to the commands A and B . Once t_0 fires it forces to start the model of the command A. Upon completion of A, the transition t_1 starts the model of command B.

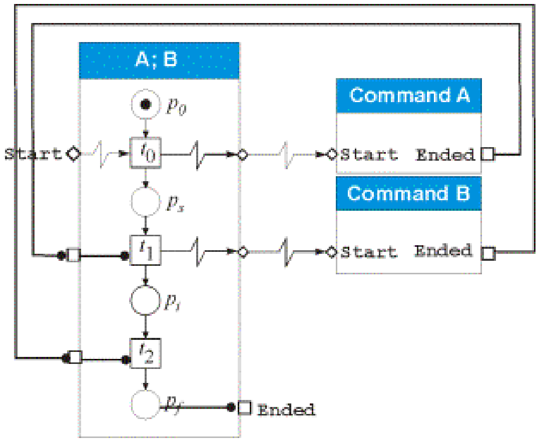


Figure 48 Model of a two commands' sequence.

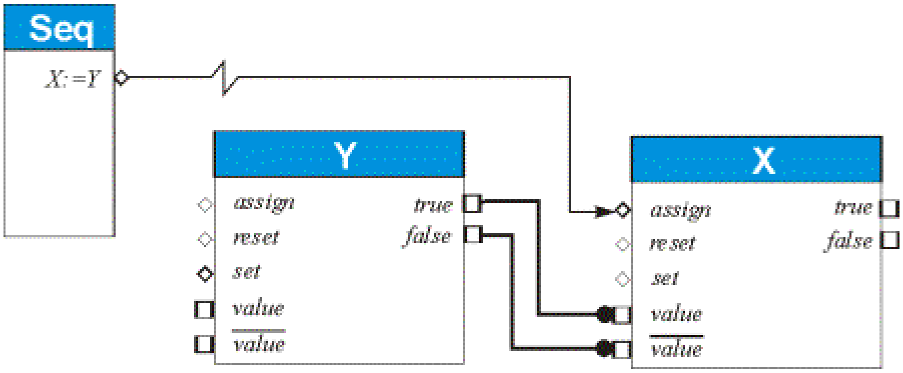


Figure 49. Model of an assignment operator.

8.3 Conditional choice

Conditional choice of type **IF X THEN Sequence A ELSE Sequence B** can be modelled in NCES as shown in Figure 5. Transition t_A has incoming condition arc which relays value of X, and t_B has incoming condition arc marked with not X. Since the conditions are orthogonal, only one of the transitions is able to fire.

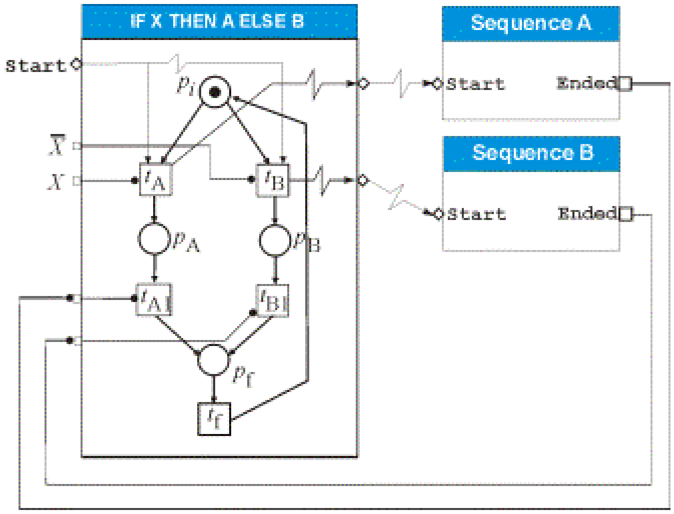


Figure 50. Model of a conditional choice operator.

8.4 Boolean operations

Since every Boolean variable is modelled by two places (as was shown in Figure 46), we do not need a specific model for getting negation of a Boolean variable. As for the AND and OR operations, they can be modelled as shown in Figure 6, a) and b) correspondingly. Both models have two incoming event signals: *compute* and *reset*. Computation of the result takes one state transition.

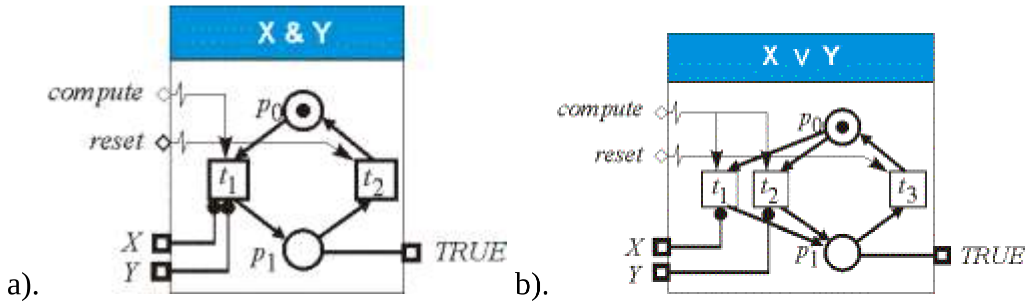


Figure 51 Models of Boolean operations.

8.5 Subroutine call

An example of a flowchart calling a subroutine is presented in Figure 52. The example includes modelling of the data that is passed to subchart, modelling of the Boolean operation, the subchart call and all the flowchart blocks.

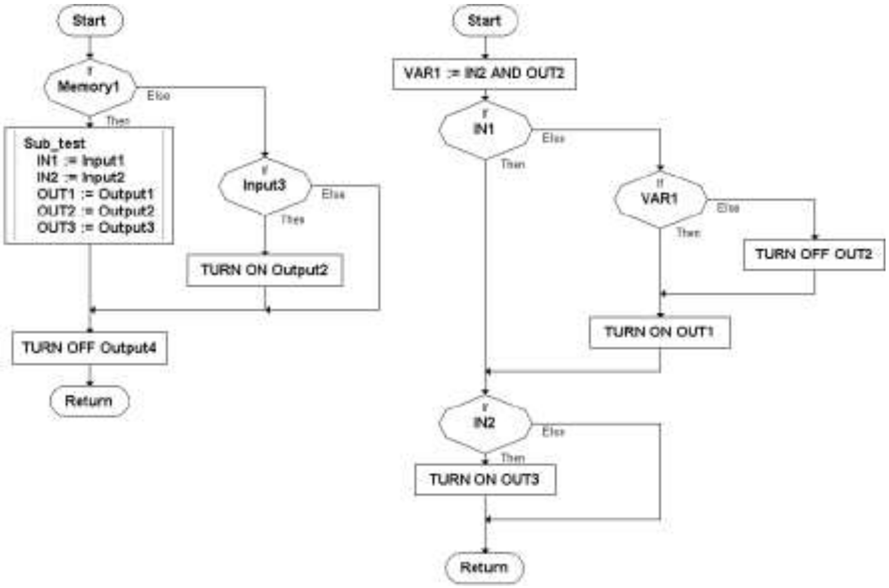


Figure 52. Flowchart with a call to sub-chart

The main flowchart has a sub-chart call block, which invokes the sub-chart. The flowchart passes five variables: IN1, IN2, OUT1, OUT2 and OUT3 to the sub-chart. Four arguments of these variables are passed by reference, one argument passed by value and one is a local variable of the sub-chart. The modelling of the given flowchart and sub-chart starts with the modelling of the local and argument-by-value variables.

The resultant model is shown in Figure 53. For the sake of simplicity it shows only the modules modelling variables VAR1 and IN2, the module implementing the operation AND over two operands; and two modules representing the logic of the main and sub-chart respectively.

The NCES model of local variable VAR1 is modified as compared with the model in **Figure 46** as follows: an event input with an arc to the *t1* transition is added. The input provides variable reset at the moment when sub-chart is called. A NCES module is necessary for the variable IN2 due to its passing by value (although IN2 does not get modified inside the sub-chart in this example). It is modelled as the Boolean input similar to the model in **Figure 58**.

Thus, the local variables and the variables passed by value to the sub-chart are represented as independent NCES modules. Variables passed by reference are treated as global PLC variables and they can be modified directly by the sub-chart.

The NCES module corresponding to the main flowchart has two outputs that are related to the commands - “TURN ON Output2” and “TURN OFF Output4”. All the other provide the representation of sub-chart call mechanism - “SET SUBCH_VARS” and “CALL SUBCHART”, and “YIELD” serves for yielding the control after termination.

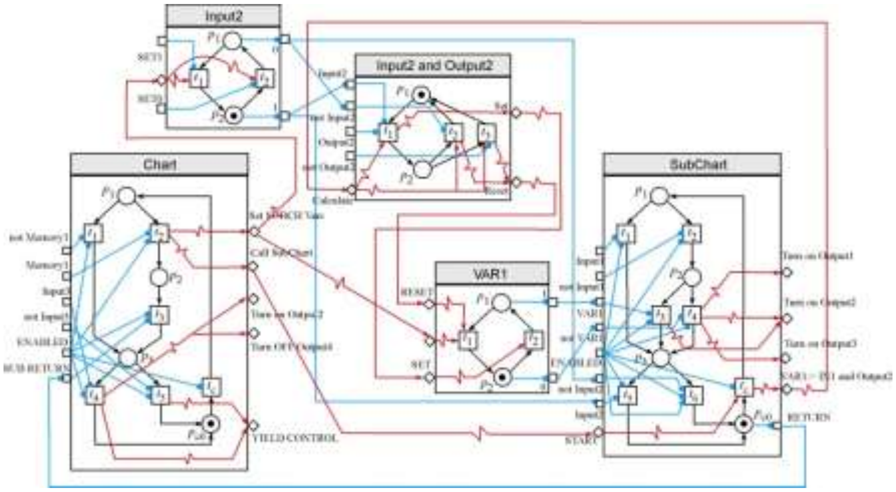


Figure 53. Full model of the sub-chart and main chart.

The sub-chart module has an event input labelled as “START SUBCHART”, which initiates the execution. The input is connected by event arc to the “CALL SUBCHART” event output of the main flow-chart module.

9 Co-existence of synchronous and asynchronous behaviour

Usually transitions in models of plant represent start or finish of some time consuming actions, while transitions in the controller's part of the model 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. Consider a simple example of process/controller communication as shown in Figure 54 (PlantObserverTest.xml).

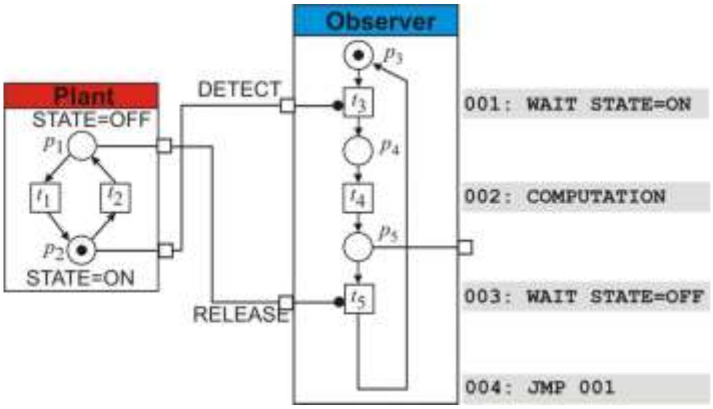


Figure 54 Model of plant and observer.

The process is represented by the basic unit of plant which has two flip-flop states (as up--down, left--right, on--off) modelled by places p_1 and p_2. The transitions t1 and t2 are spontaneous. First assume that the model is not timed (by ignoring the time intervals attached to the arcs).

The block “Observer” makes an “instant photo" of the process, i.e. reads the value of state in the loop (input DETECT) and when it is "ON" (i.e. m(p2)=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.

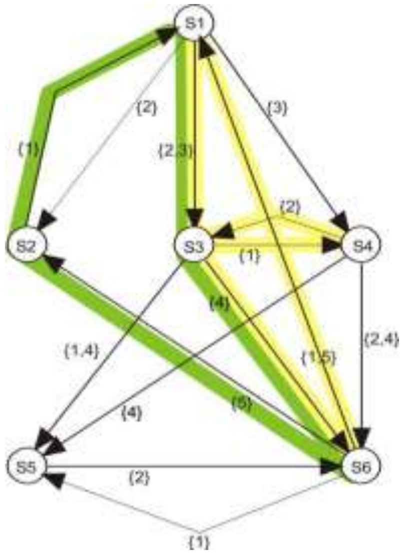


Figure 55 Reachability graph of the model of plant/controller interaction.

If transitions in the observer were also spontaneous then, in the state shown in the Figure 54, there would be the following enabled steps of transitions: s1={t2}, s2={t3}, s3={t2,t3}. However, the first step is not feasible: it reflects the situation when the event occurred (m(p2)=1), but the controller does not start the corresponding action though it was able to do so (was not busy), and the information about the action is lost.

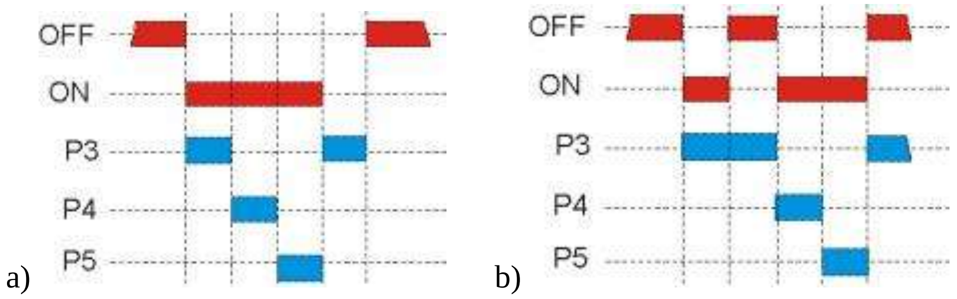
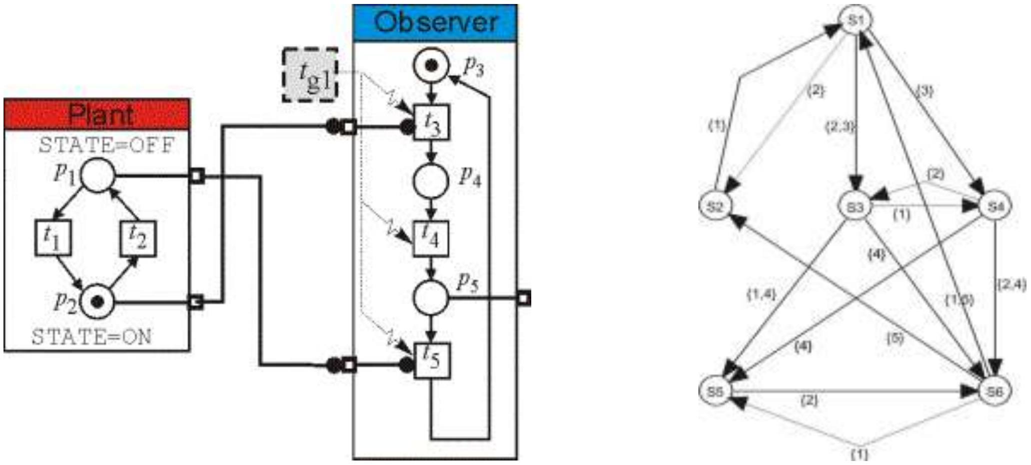


Figure 56 Desirable and incorrect sequence of plant/controller states. The sequence presented in the right Figure can never occur on the reality but is generated by the model.

9.1 Non-timed models

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. This guarantees that all enabled transitions (commands) in controller will be executed until the next action occurs in the plant.



a) b)

Figure 57 Reachability graph of the interconnected system. Trajectories eliminated by the “greedy” tick generator are dotted.

To fix the behaviour of our model we introduce *greedy* transition **tg** connected to t3, t4, t5 via (dotted) event arc. Since **tg** is always enabled, it fires at every state transition sending forcing “ticks” to the commands in the controller no matter what is going on in the plant. All the transitions t3, t4, t5 could be marked as greedy instead and that would yield in this example the equivalent behaviour. But we prefer forcing such transitions from the greedy “tick” generator for the reasons which become clear in the next section.

9.2 Timed models

In case of timed models, the use of greedy transitions in the former example is obsolete - the desired behaviour is obtained automatically since at the clock value u(p2)=0 (i.e. right after place p2 gets marked) the process in the plant is delayed, and the only enabled transition is t3.

The only possible sequence of firing is: t3 -> t4 -> (after 1 time unit) t2 -> t5, if all arcs in the controller model have zero-delay (i.e. time interval [0, ∞]).

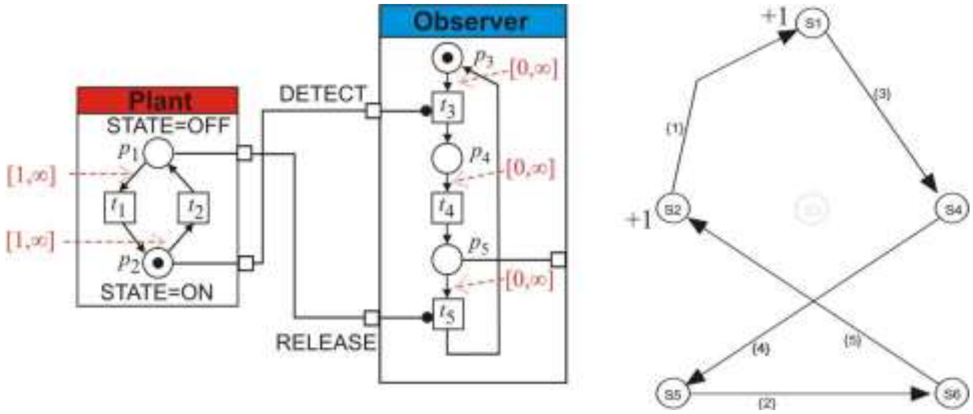


Figure 58 Timed model and its reachability gaph.

9.3 Testing timed NCES modules

For testing of timed models one may develop a structure similar to that in **Figure 58**, but with non-timed left "tester" module emitting random values in a loop. However, this will lead to the situation when the time is not getting incremented in the right, timed part of the model.

A cure can be to make the tester part also timed.

10 Complete Example: Cylinder control

10.1 Object description

As the first example let us consider a very primitive control system of a single linear drive implemented using a pneumatic cylinder.

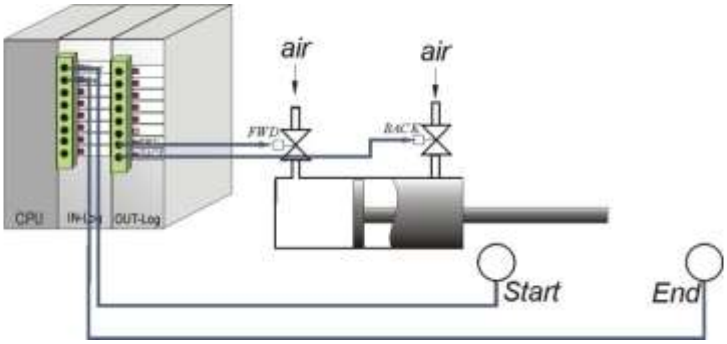


Figure 59. Cylinder with two end-position sensors (Start and End) and two control signals FWD and BACK.

The operation of the object is straightforward. *Suppose we want to retract the cylinder to its leftmost position and from there enter the eternal loop of moving forth and back.* To achieve this suppose we write the following program in Sequential Function Charts (SFC) (as shown in the left part):

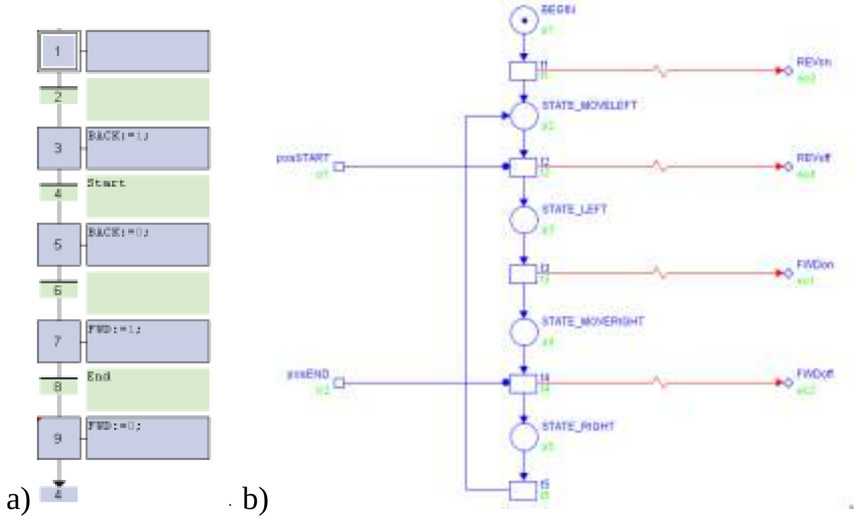


Figure 60. Control program in SFC language (a) and its model in NCES (b).

Now let us try to study this program applying the formal verification technique. For that we create NCES model of the controller, which is quite straightforward, the result is shown in Figure 60, b. Then we connect the model of controller with the model of plant as shown in Figure 61.

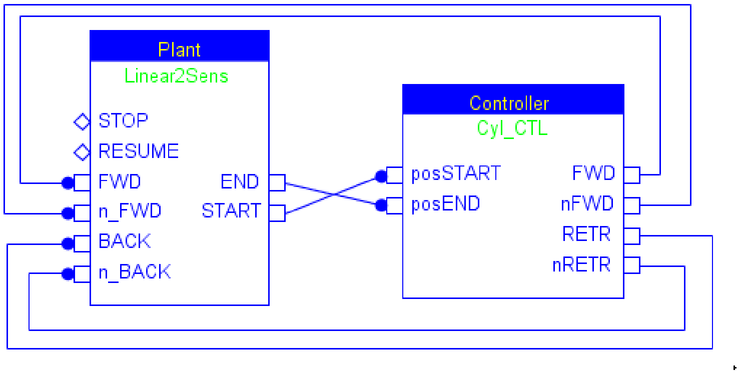


Figure 61. Block diagram of the Plant-Controller NCES model.

This model is provided with ViVe tool set and is called **Cylinder**.

10.2 Modelling the plant: Linear drive

The most typical motion process in automated machines can be modelled and encapsulated for further re-use in the model of linear drive shown in Figure 62.

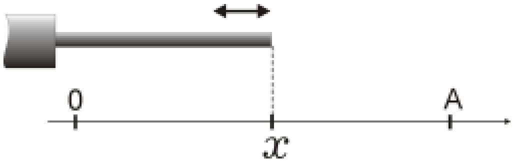


Figure 62. Linear drive.

The model which quite precisely represents uncontrolled behaviour of the drive is shown in Figure 63. Main modelled parameter is linear coordinate of the drive *x*.

There are 6 states distinguished in the behaviour of drive, two of which are *dynamic* that means the speed is more than 0. The dynamic states are shown as circles, while the other *static* states are represented by rectangles with rounded corners. Model's inputs include two control signals FWD and RETR for forward and backward motion respectively. Input event STOP serves to relay all sorts of possible failure situations during the motion.

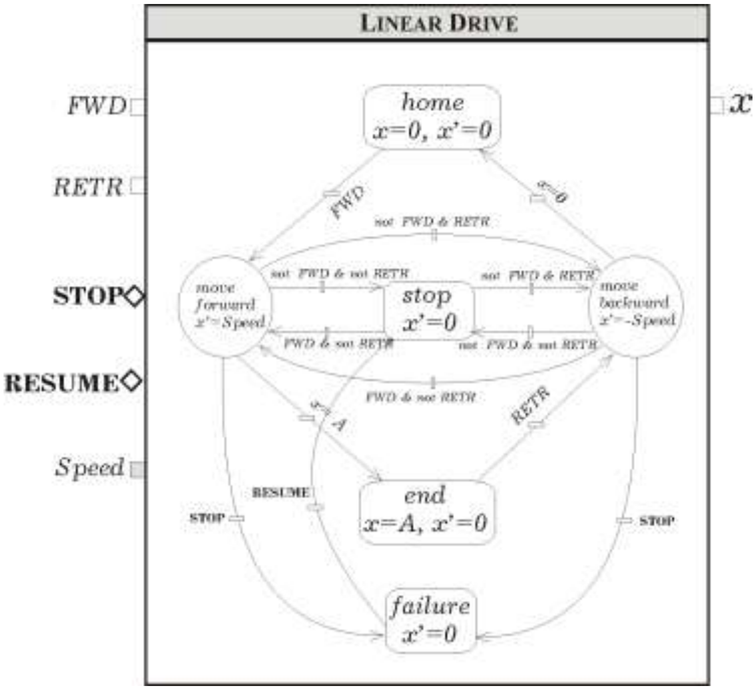


Figure 63 Continuous/discrete (hybrid) state-chart model of the linear drive.

The source of the failure is external to the model of drive, but consequence of the failure has to be taken into account within the model. According to the model, input event STOP leads to the static state "failure", and input event RESUME enables transition from that state to state "stop". The model above provides the numeric value of x as a function of time conditioned by the values of inputs.

In order to represent similar model by means of a discrete state formalism we decompose functionality of the drive to most basic characteristics, for example as follows:

- motion status (stands ready to move, moves forward, moves back, stands in failure);
- motion position (depending on the used formalism can be either represented as a numeric coordinate value, or as a discrete position).

This template is illustrated in Figure 64.

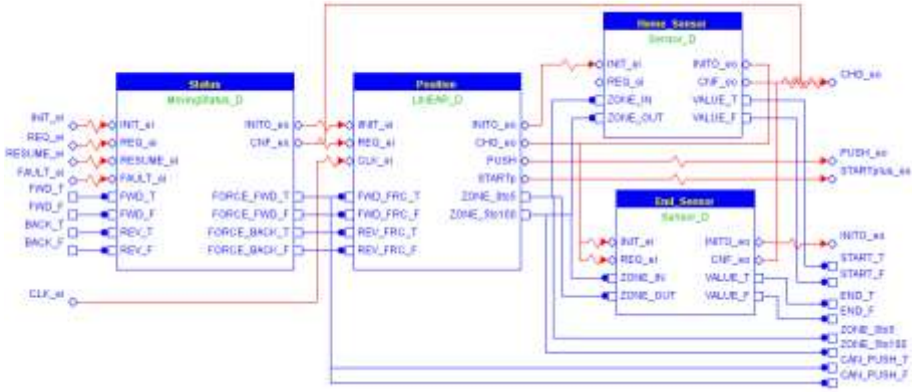


Figure 64. Status-Position-Sensors template illustrated on the cylinder's example.

Discrete model of the motion status is very similar in structure to the state-chart model. In contrast to that, it is purely discrete and does not include time.

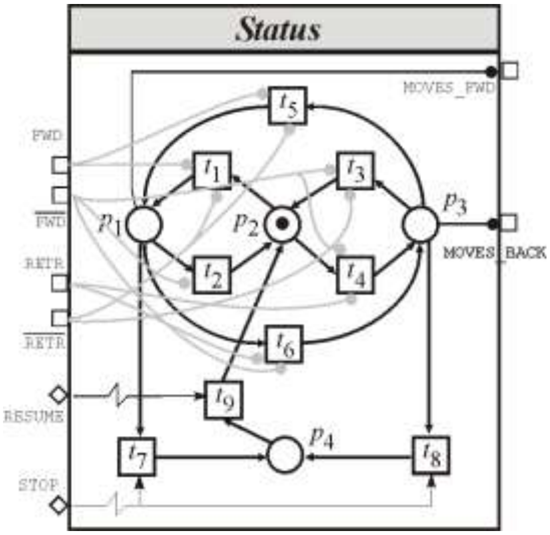


Figure 65. Motion status of the linear drive.

The motion status model is illustrated in Figure 65. The model converts the values of Boolean control signals to the status of motion – thus when all control signals go off, the status must change from movement (either place 1 or 3) to stop (place 2). Besides, external event FAILURE causes transition to the state "Emergency stop". Places in the model represent the following discrete states:

P_1	Motion forward
P_2	Stands still
P_3	Motion back
P_4	Emergency stop

The input signals of the model have the following meaning:

FWD	Control signal “Move forward”
RETR	Control signal “Retract”
STOP	Event causing failure
RESUME	Event that recovers failure

A specific feature of NCES modelling is that control signals need to be represented by two condition inputs for positive and negative value of the signal. It explains by the fact that sources of condition inputs are places, so to obtain the negation of a certain place marking we would need to connect to a place complement to the source.

Transitions between states are driven by the values of inputs relayed to NCES transitions by means of event and condition arcs. Thus, transition from state “Stands still” (p2) to “Moves forward” (p1) is conditioned by the input combination “FWD and not RETR”. The corresponding NCES transition t_1 has two incoming condition arcs from inputs “FWD” and “not RETR”. Abnormal halt of the drive is conditioned by an external event delivered via event input “STOP”. This event brings model from whichever motion state to the state “Stands in failure”, which is recoverable by event signal “RESUME”.

The model delivers information about its motion status by means of two condition outputs “Moves forward” and “Moves back”. Since we cannot get the continuous value of x out of a discrete state model, there are several possible options exist how to approximate it.

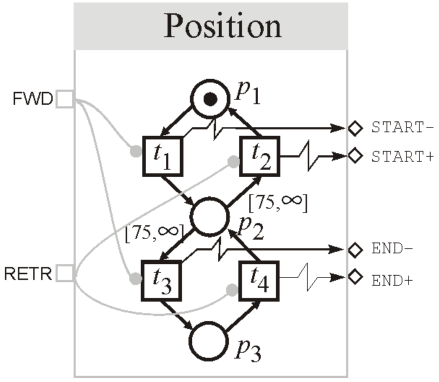


Figure 66. Simple timed model of the position change.

The most primitive way to identify position is to distinguish three positions: start, end and in the moving in between the two. The model in Figure 66 gives correct results assuming that the status of the movement never changes once it has started. Place p2 stands for the state "in between" and it takes 75 ms to get in either state "start position" (p1) or "end position" (p2). Should both condition inputs go off while the token is in p_2 there would be no way to figure out the coordinate of the drive.

A more precise way to represent the coordinate with a discrete formalism could be to divide the interval on segments and represent each of them in a way similar to that in Figure 67.

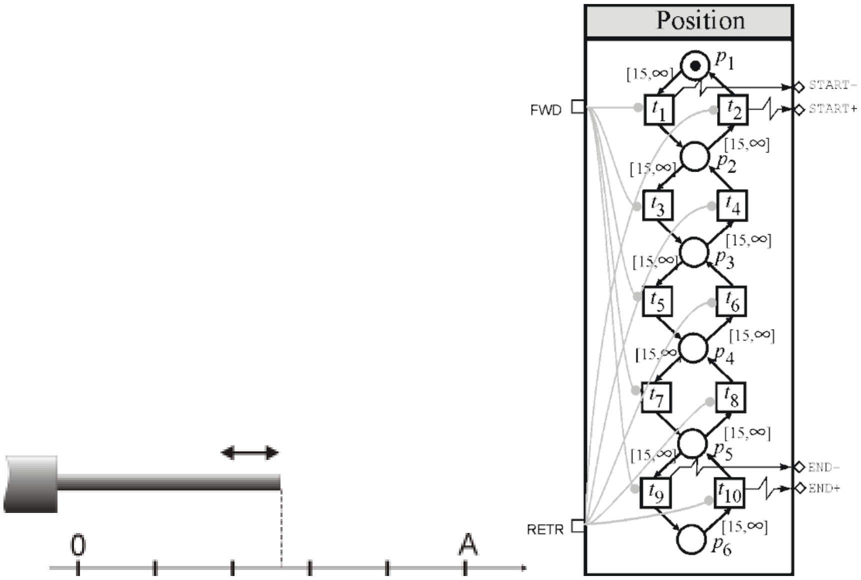


Figure 67. Model of position in the interval divided on 5 segments.

The resulting model of the cylinder can look like the one in Figure 68. Status and Position modules here are grouped under composite model "Cylinder", although this is just a matter of convenience.

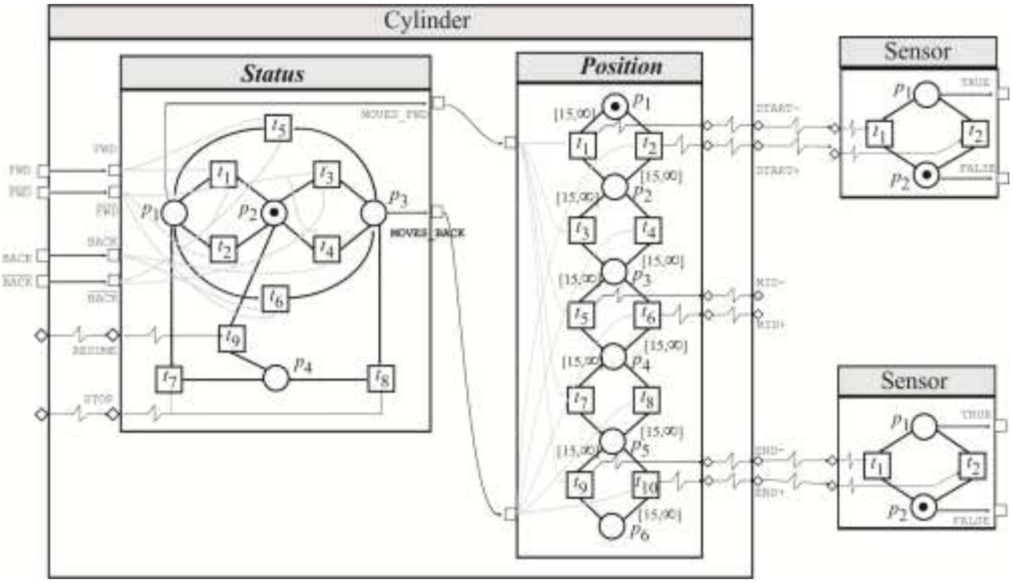


Figure 68. Complete NCES model of the cylinder.

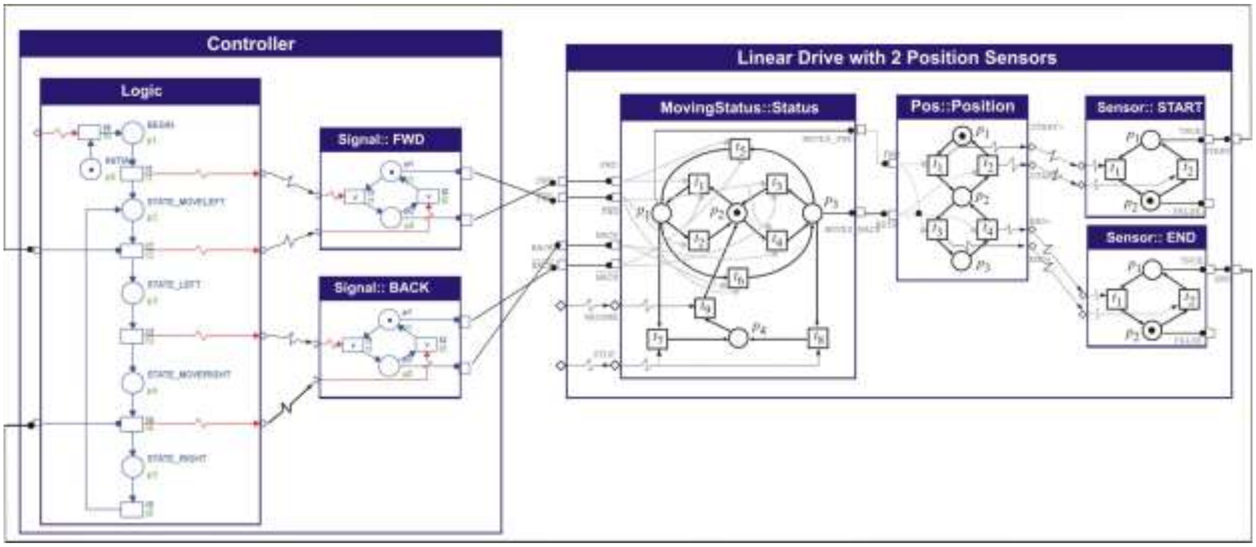


Figure 69. Complete closed-loop plant-controller model.

10.3 Deadlocks

The first step in analysis of this model (**Cylinder**) using the Visual Verifier (ViVe) is generation of its reachability graph. In this case it consists of just 3 states. While doing this ViVe reports that the reachability graph contains deadlocks. **Thus, without spending any time for step-by-step testing of the program on the real object (or its model) we were able to tell that the program drives the object to a deadlock state!** This already is quite significant result provided that for a more complex object the reachability space may include a lot more states and manual finding of a deadlock can take very long time.

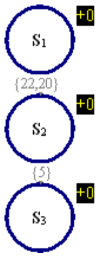


Figure 70 Reachability graph of the plant-controller model.

The reachability graph for this simple control system consists of 3 states as shown in Figure 70. Arcs of the graph are marked with the numbers of transitions of the NCES model. The graph contains 1 deadlock state. ViVe can help with understanding of why the model gets into the deadlock.

Thus, as Figure 71 shows, in the deadlock state 3 the controller cannot leave the state MOVELEFT since the sensor START never becomes TRUE. The reason of that can be found in the model of the plant. Models of both sensors initially are in state ZERO and nothing makes them transiting to the state ONE. As a result, the controller stuck, waiting for sensor START to become TRUE

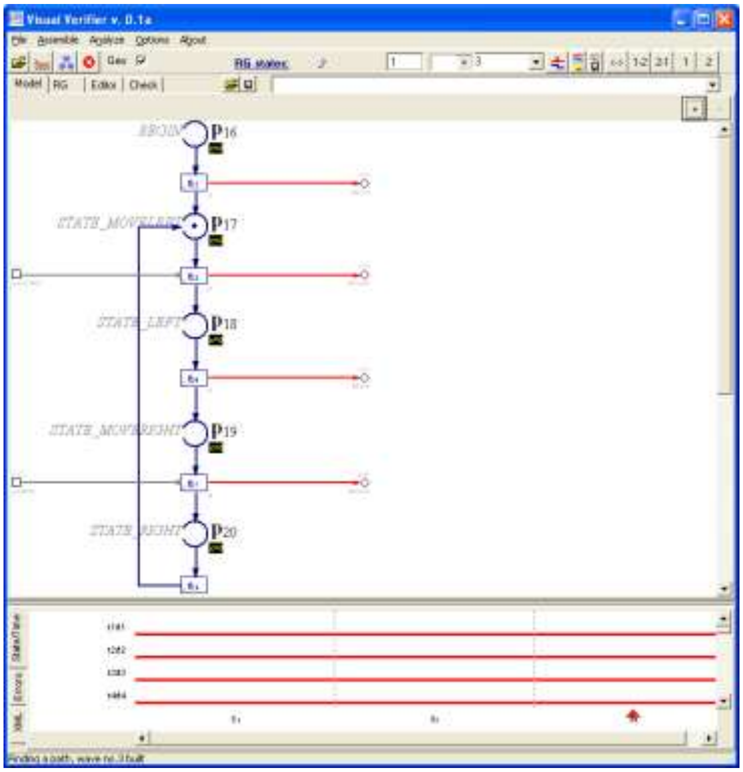


Figure 71. ViVe shows the controller status in state 3.

The remedy for this would be initialization preceding model's operation. The initialization needs to set parts of the model to the appropriate initial states, e.g. model of sensor START to the state 1. Given this hint we can modify the model of the plant so that the overall behavior gets closer to the desired.

First, after introducing the INIT input signal as shown in Figure 73 (**Cylinder_INIT** model) the plant adjusts values of sensors before the controller starts its operation.

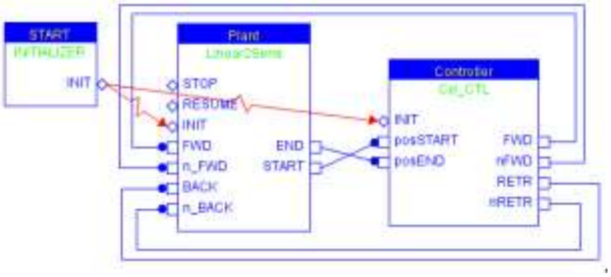


Figure 73. Closed-loop model with initialisation.

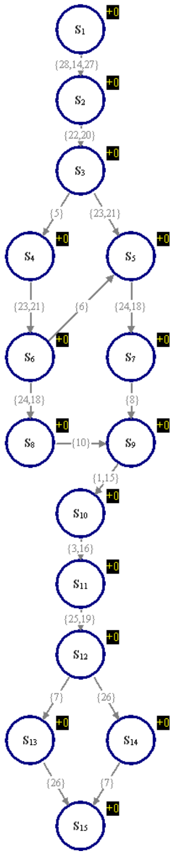


Figure 72. Reachability graph of the model with initialization.

As a result, the overall behavior changes to have 15 reachable states as shown in Figure 72. This model however also has a deadlock state. The reason for the deadlock becomes clear after looking into models behavior in the last 3 states before the deadlock. While the cylinder is in its extended state (END), the controller jumps to the ‘STATE_MOVINGLEFT’ and waits for the sensor START to become true. However, the controller did not issue the control signal BACK !

This problem can be easily fixed by modifying step 9 of the controller by adding the ‘BACK:=1’ command. The NCES model also needs to be modified accordingly. As a result the behavior becomes deadlock – free as expected in the system that operates in eternal loop (Figure 74). The corresponding model is provided in **Cylinder_CTLcorrected**.

10.4 Branching

The reachability graph, however, contains a few states with several outgoing branches and one may wonder what this model’s behavior corresponds to in the real object’s operation.

Thus, state 3 has two alternatives: either t_5 fires or t_{23},t_{21} fire together. The former corresponds to the model’s of plant evolution from the state stSTOPPED to stM_BACK in the module MovingStatus.

The latter is controller’s transition from the state STATE_MOVELEFT to the STATE_LEFT with setting to zero the output signal BACK.

Since plant and controller operate concurrently either of these actions can occur first in the real settings. However, if the latter occurs first it will make the former obsolete, as there is no more signal BACK present at the input of the plant and therefore there is no need to transition from the status stSTOPPED to the status stM_BACK. One can see that in the branch S3→S5→S7→… transition t_5 never occurs, while in the branch S3→S4→S6→… the transition step t_{23},t_{21} occurs right after t_5.

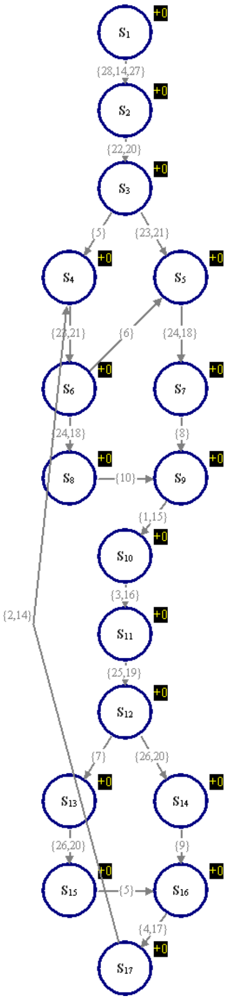


Figure 74. Deadlock-free reachability graph.

Technically speaking, the reason for branching is *the interleaving semantics* of spontaneous transitions used for generation of theses reachability graphs – only one spontaneous transition can fire at time (ViVe supports also other options: all combinations of enabled spontaneous and maximum set of enabled spontaneous transitions to fire simultaneously).

Other branching cases in the states S6 and S12 are of the similar nature.

10.5 Deeper analysis

Although ViVe allows following various traces of a model's behavior in the visual way, for more complex models more analytic methods of analysis are required, namely using specifications formulated in CTL. Usually the following classes of properties are of interest:

a. Liveness – i.e. deadlock-free behaviour
b. Checkpoints of the process or properties of the product – assurance that the product always satisfies specifications;
c. Safety – not entering certain 'prohibited behaviour' scenarios.

10.6 Exercises

1. Develop an example plant – controller system where the controller gets into a deadlock while the whole model is alive.
2. Develop a model illustrating the idea of a 'dynamic trap' where e.g. controller enters eternal loop but the model of plant stuck.

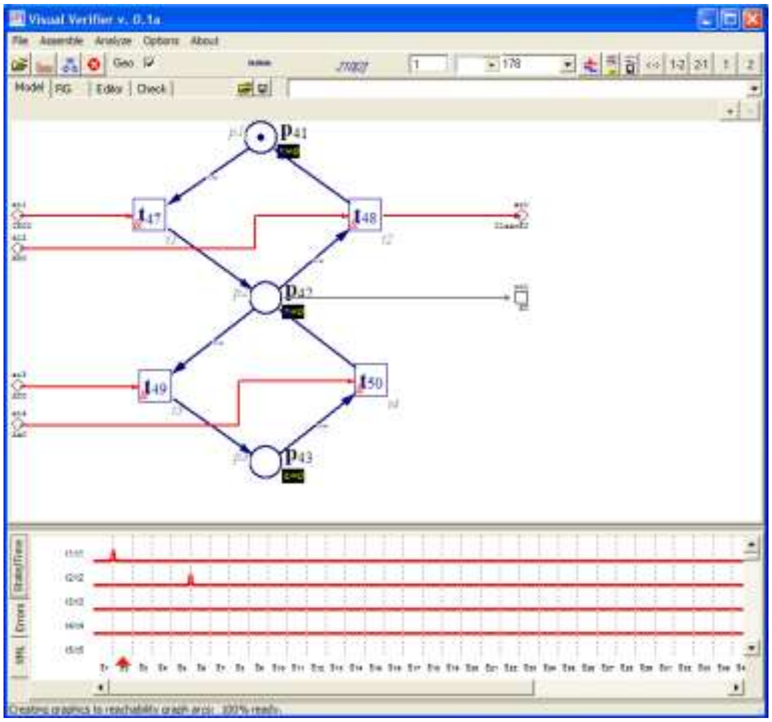
10.7 Review questions

1. Why branching is observed in the behavior of the automated cylinder?

11 Model Verification with Visual Verifier

11.1 Visual Verifier functions

The integrated environment for Model Assembly, Translation, and Checking (Visual Verifier) inputs the model type files (in the XML-based format) and is capable of assembling, translating and checking the models.



Creating graphics to reachability graph arcs: 100% ready.

Figure 75. Visual Verifier screenshot.

Assembling means creation of a flat model from a composite, hierarchically organized modular model using the modules from different libraries of model types. The component model types are instantiated into NCES modules.

Translating the model into a “flat” NCES with the through numbering of places and transitions (**Figure 76**). The inter-module connections are converted into event and condition arcs between places and transitions. Thus the module boundaries are removed and the model-checking tools can be applied. In particular, the translator generates files in the input format of SESA model checker.

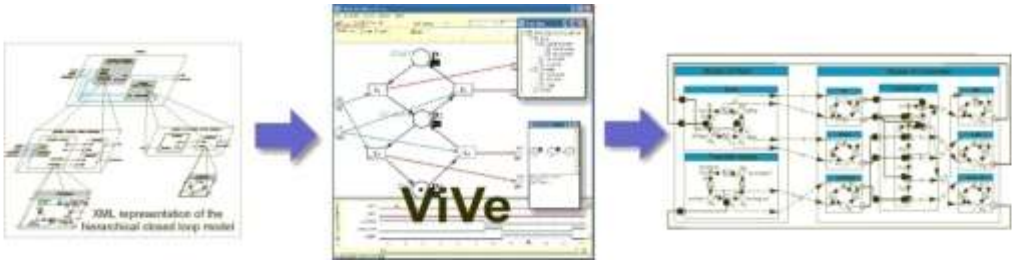
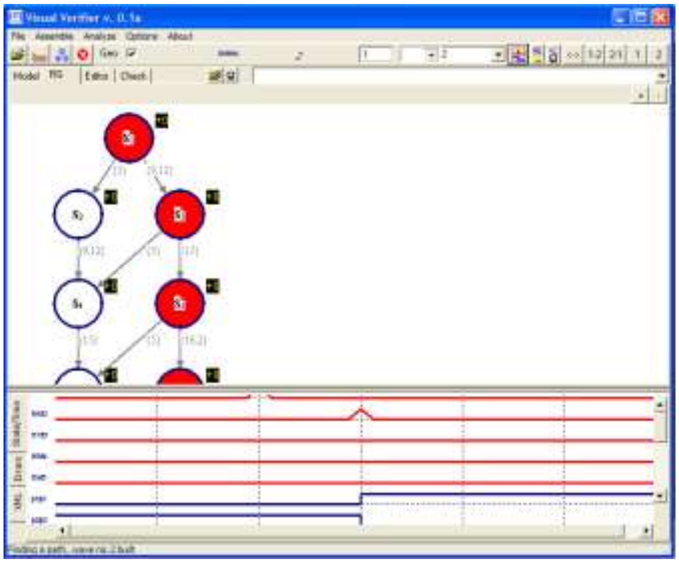


Figure 76. Visual Verifier creates a flat model from a hierarchical model.

Visual Verifier can **prove** specifications in the form of the first order predicates or can pass the temporal logic formulae to the SESA model checker. The internal model checker of Visual Verifier generates the reachability graph for the model, either completely or dynamically while it checks the formula. It can also import a reachability graph generated by SESA and visualize it.

Once a state with particular properties is found in the reachability space, Visual Verifier can **visualize** a path from initial (or any other state) to the found one. The visualisation is done in a form of state-time diagram for a selected set of system variables (both from plant and controller). A user can select between different views and see the model in each state. The visualisation options proved to be very useful in practical verification.

For documentation it is possible to **export** the picture of model or the reachability graph in the BMP format.



Finding a path... wave no.2 built

Figure 77. Visualisation of a reachability graph

11.2 Data formats

The data format of Condition/Event Nets is based on XML. The Document Type Definition (DTD) of the S/EN model types is given in Annex 1.

The installation package of Visual Verifier contains a number of examples in that format that will be commented here.

Model-checker SESA has its specific data formats which are explained in detail in its documentation. It is important to remember that input for SESA does not include any module information and has a through enumeration of network places and transitions each starting from 1. For this reason SESA input format will be referred to as “flat” model format.

FBT format is used to provide compatibility with IEC61499 and its supporting tools. FBT is XML-based, but contains only a description of *model interface*. The XML format for S/E Nets also contains interface part but their syntax is different.

The FBT files can be generated automatically for each S/EN module along with XML file containing full model description. Then they can be used for creating *composite models* as networks of interfaces interconnected via event and data connections in Function Block Editor.

File extension:	**Basic S/EN module**	**Composite module**
XML – full S/E model	Created by S/EN editor	Created by S/EN editor
FBT – interface of the model	Created by S/EN editor	Created by FBEditor

Visual Verifier currently supports several input formats on NCES models:

For basic modules:

S/EN editors:

1. ViEd format: the editor of typed NCES developed at the University of Auckland, New Zealand. The centre of axes is in the middle of the picture. The coordinates are in pixels.

2. TNCES Editor

The editor was developed in Martin Luther University of Halle-Wittenberg, Germany. The coordinates of elements are given in pixels, centre of coordinates is in the top left corner.

Older editing tools and generators:

3. VISIO format

Coordinates are given in millimetres, the vertical coordinate axis is directed upwards; Element's coordinate indicates the centre of the element.

4. Generated by MOVIDA generator

In addition to position coordinates some elements may have explicitly assigned size, either via Width, Height or via Diameter parameter.

Visual Verifier has default sizes for graphic elements. If the size is assigned once to any element in the model, Visual Verifier scales correspondingly sizes. If the size is not present, then Visual Verifier takes this default scaled size.

For composite modules:

1. FBDK format: coordinates are measured in basic units that are equal to 1/10th of interval between parameters of a module.

2. ViEd and TNCES Editor format.

3. MOVIDA Generator format.

11.3 Limitations

There are many input syntax limitations of ViVe which are not always properly detected by the input parser.

1. Symbolic names of places, transitions, I/Os cannot contain spaces. Example:

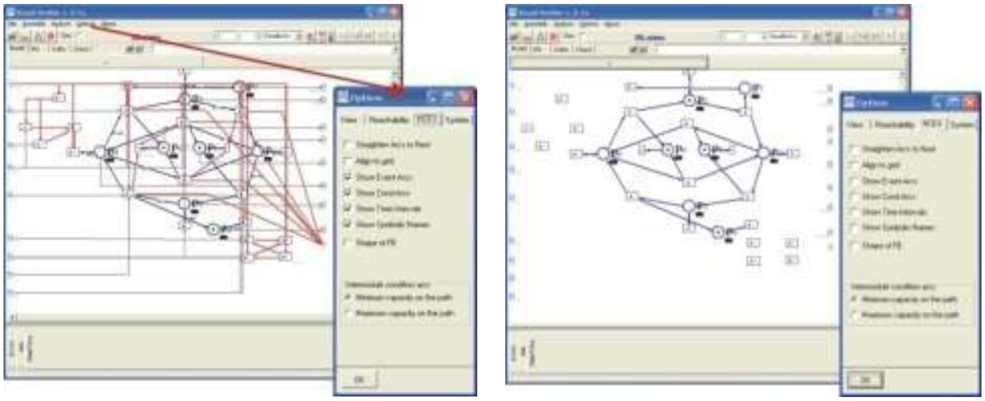
   Correct: TrueToFalse , True_To_False

   Incorrect: True To False

2. Objects within a module cannot have same symbolic names, even in different case. For example, event input “END” and place name “end” will be treated as the same.

11.4 A hint for clearer models

Sometimes NCES models can be overloaded with arcs which reduces their clarity. To cope with this problem ViVe suggests 2 solutions. First you can select which graphical elements to show, and which not, on the pane NCES of Options, as illustrated in Figure 78.

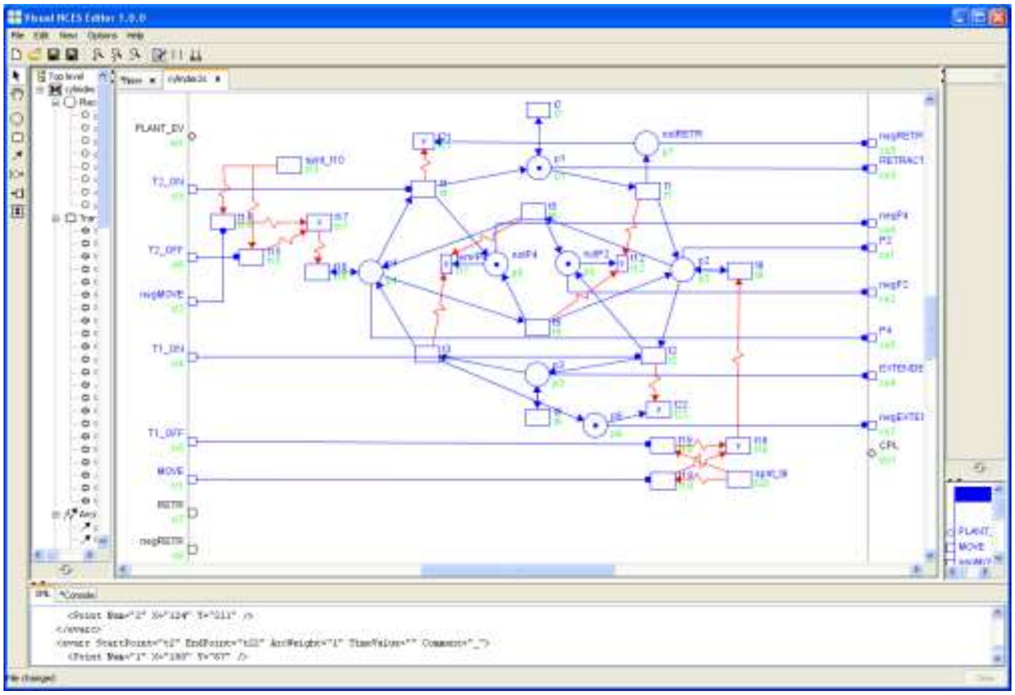


a) All graphical elements are shown

b) Only token flow arcs and numeric identifiers of elements (e.g. p_1) are displayed)

Figure 78. Hiding event and condition arcs and timing intervals.

The second trick applies to the ViEd NCES editor. To make the picture in the editor clearer a special textual notation can be used in comments of the corresponding arc destination elements. For example, in Figure 79 the same cylinder2s.xml model is opened in ViEd. As one can see, not all inputs have graphical links. For example, the condition input RETR in ViEd has no outgoing graphical link connections.



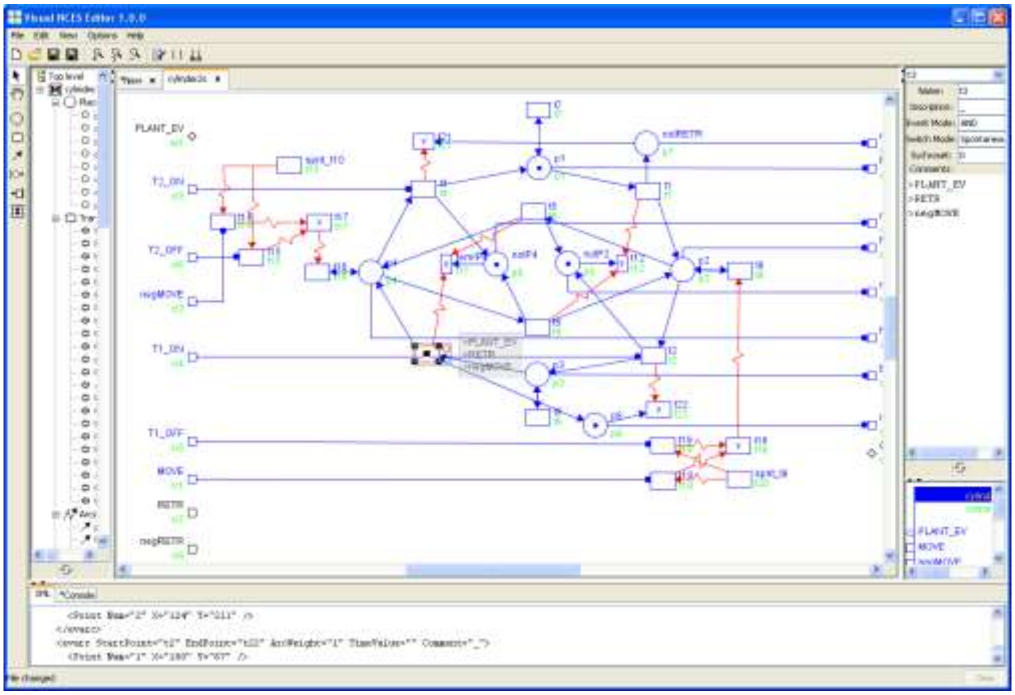
```
  <Point Num="2" X="124" Y="211" />
</evarc>
<evarc StartPoint="t2" EndPoint="t12" ArcWeight="1" TimeValue="" Comment="_">
  <Point Num="1" X="180" Y="67" />
```

Figure 79. Same model opened in ViEd.

One of these links goes to the net transition t_3. If this transition is selected, as illustrated in **Figure 81** one would see its parameters in the right pane, including the comment:

```
>PLANT_EV
>RETR
>negMOVE
```

This comment defines arcs having their destination in t_3 by enlisting their sources. Actual type of the arc (condition or event) is not important as far as it is not ambiguous. When the model will be opened in ViVe, the arcs will be created automatically.



t3
Name: t3
Description: _
Event Mode: AND
Switch Mode: Spontaneous
SubFacet#: 0
Comments:
>PLANT_EV
>RETR
>negMOVE

```
  <Point Num="2" X="124" Y="211" />
</evarc>
<evarc StartPoint="t2" EndPoint="t22" ArcWeight="1" TimeValue="" Comment="_">
  <Point Num="1" X="180" Y="67" />
```

Figure 80. Transition t_3 is selected.

Such “symbolic” arcs can be established between:

Event input –> transition

Condition input -> transition

Place -> transition (condition arcs, not flow arcs)

Transition -> transition

Place -> condition output

Transition -> event output

In all cases, the link is defined at the destination elements using the syntax described below:

```
> source [,source][,source]
```

12 User Interface of Visual Verifier

A Visual Verifier screenshot with annotated screen areas is presented in Figure 81.

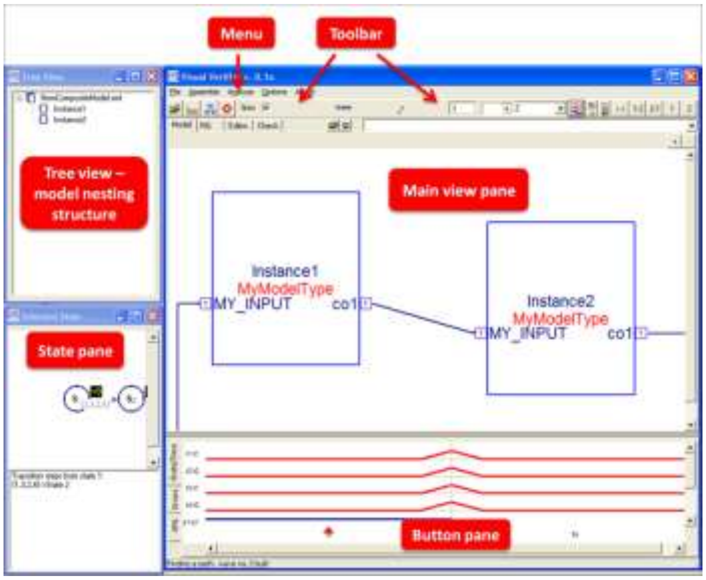


Figure 81. Visual Verifier screen.

12.1 Tabs

Model tab – Viewer of NCES model currently opened (if Basic) or currently selected in the model tree view (after assembly).

RG – Reachability Graph – shows the generated reachability graph if

1) The graph has been generated (for that the model needs to be built);

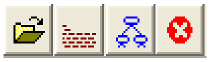
2) The **Geo** checkbox is checked;

Editor – Edit the flat S/E Net.

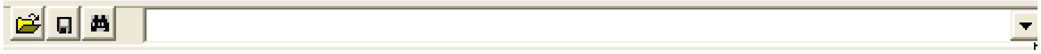
Check – provides the tools for specifying CTL formulae and verifying the current NCES model.

12.1.1 Functional toolbars

Files toolbar

	
1	Open model file (*.xml)
2	Build flat model
3	Create reachability graph in the memory area 1
4	Interrupt the model checking/ reachability graph generation

Specification toolbar:

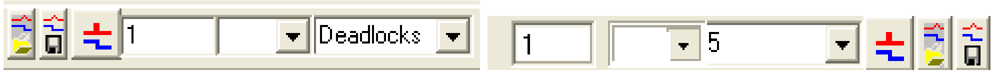


1	Open specification file;
2	Save specification
3	Look for a state complying with spec in the reachability space. If the space is not generated yet, it will be generated on the fly until the desired state is available

Reachability graph toolbar

1	Geo check box – creates geometry of the reachability graph. Needed to visualise the reachability graph (if unchecked the reachability graph still can be used for model checking but cannot be visualised)

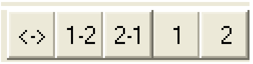
Trace toolbar

1	First state number of the trace
2	List of intermediate state numbers. To add a number to the list: type the number in the field, press “Enter”. To delete number from the list: select the number in the pull-down menu and press “Space”. The path will be created trough all the selected states (if such path exists).
3	Last (target) state number of the trace Generate Trace
4	Generate Trace
5	Load a trace from file
6	Save trace to file

Figure 82. To delete element from the states list: select it in the pull-down menu, and press “Space”

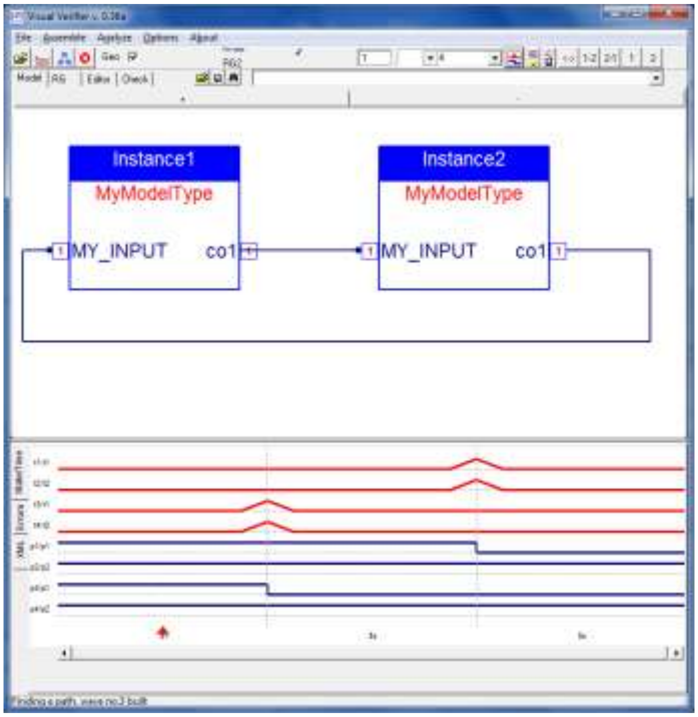
Multiple reachability graphs toolbar

	
1	Compare states of RG1 and RG2 (until first discrepancy is found)
2	Compare topology of RG1 with RG2 (until first discrepancy is found)
3	Compare topology of RG2 with RG1 (until first discrepancy is found)
4	Navigation in reachability graph: Find the target state in RG1 and show it on screen
5	Find the target state in RG2 and show it on screen

12.2 Typical sequence of steps using Visual Verifier

Step 1: Open the header file of the model

The header file is the module of highest hierarchy. If the header is a composite model, then it refers to other model types whose instances make a network of modules. Open, for example, src/book/NewCompositeModel.xml and you will see the following:



Step 2: Build a flat Condition-Event net model

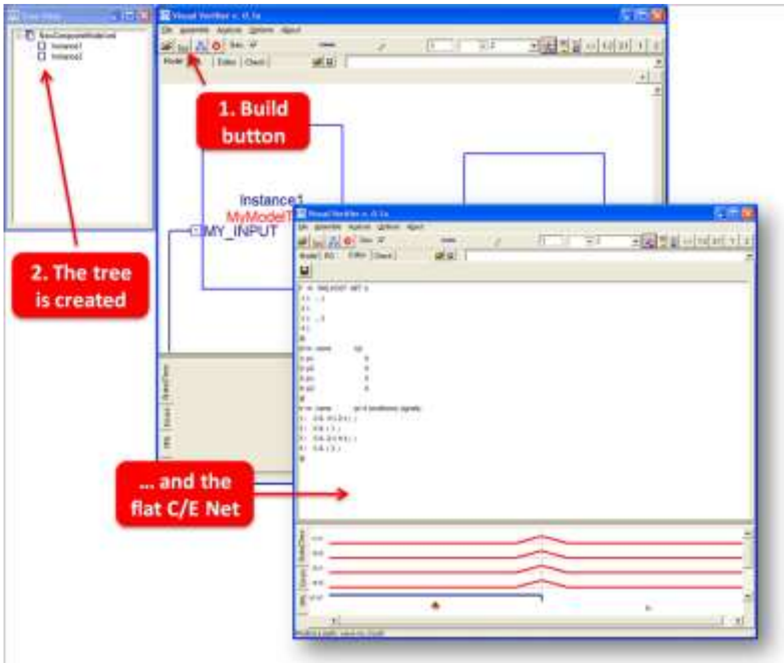
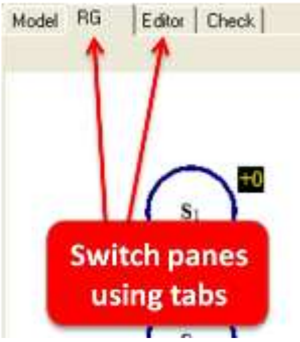
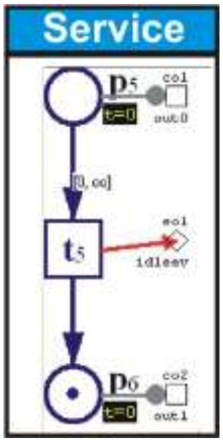


Figure 83. Generation of the flat S/E net from hierarchically dependent NCES modules.

After the button "Build" is pressed, the model is assembled from model types. Then the nested structure of the model appears in the Tree View window, and the flat model in the Edit pane window.



You can switch back to the Model view and you will see that after the "Build" operation one module has added to the network of modules. This is the module **dummy** of type "Service". This module is needed to set values of those inputs of other modules which are not assigned.



The dummy module has two places – one always empty and the other always full. It also has one transition which is never enabled.

This transition is connected during the build to all unassigned event inputs of all modules in ALL LEVELS of hierarchy!

The always empty place is connected to all condition inputs of all modules via condition arcs.

Step3: Generate reachability space of the flat S/E model

Then, if you check the "Geo" radio button the corresponding reachability graph will be created. It can be viewed under the RG tab in the main pane.

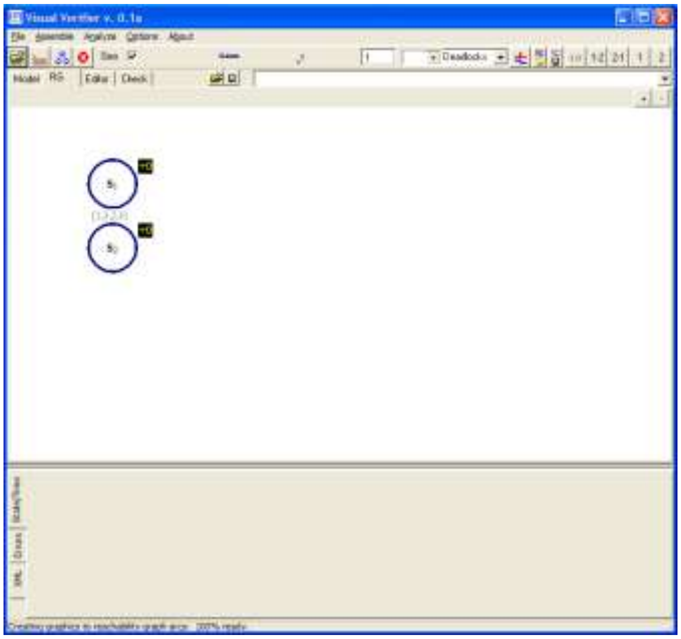


Figure 84. Reachability graph visualized in ViVe.

Step 4. Check specifications using internal model checker

The internal model checker allows check specifications given in predicate logic over the state (marking) variables.

If the reachability graph has been already generated then the specification will be checked on that graph. Alternatively, the graph will be generated “on the fly” until a counterexample is found.

12.3 Model-checkers

ViVe has two built-in model-checkers and, additionally, can call external model-checker SESA. The first built-in model-checker checks only specifications in form of first order predicates, the other understands temporal logic formulae in CTL. As illustrated in Figure 85, access to both built in model-checkers is provided from the pane “Check”. The CTL checker (also referred to as STARk) is based on SESA and supports the same syntax of the specifications as SESA.

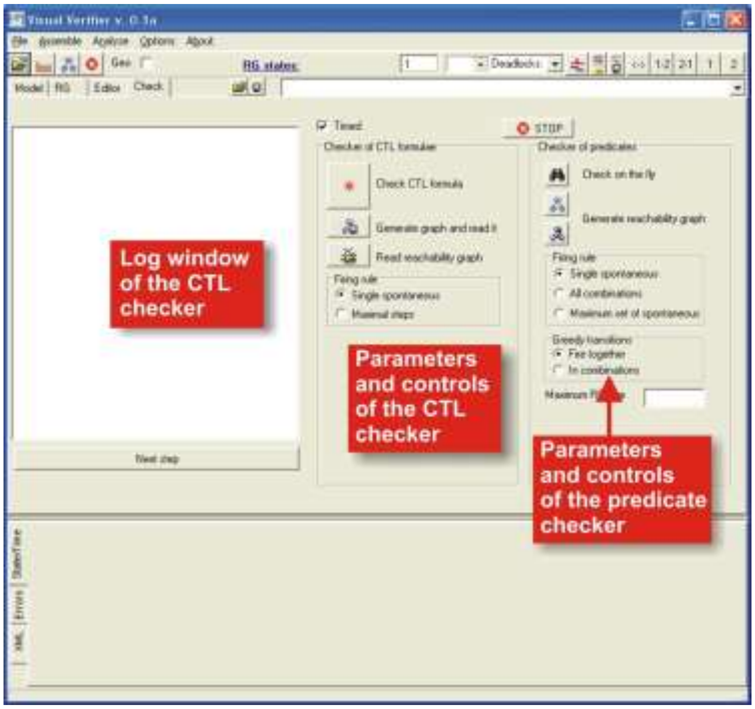
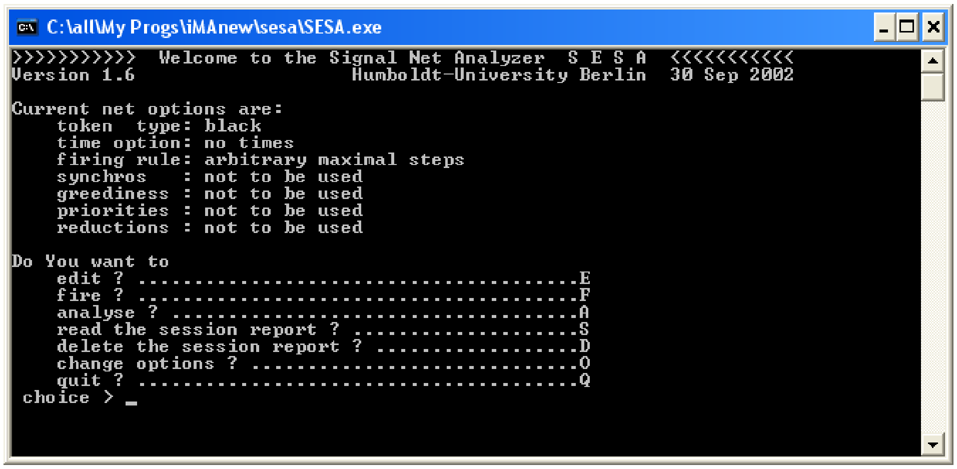


Figure 85. The Check pane contains controls of both built in model-checkers.

12.4 Command line SESA

The command line version software tool SESA for the analysis of Signal-Net Systems. SESA has been developed in 1999-2002 at Humboldt University of Berlin (Germany) by Professor Peter Starke and his group.



```
>>>>>>>>>>>  Welcome to the Signal Net Analyzer  S E S A  <<<<<<<<<<<
Version 1.6                      Humboldt-University Berlin  30 Sep 2002

Current net options are:
    token  type: black
    time option: no times
    firing rule: arbitrary maximal steps
    synchros    : not to be used
    greediness  : not to be used
    priorities  : not to be used
    reductions  : not to be used

Do You want to
    edit ? .......................................E
    fire ? .......................................F
    analyse ? ....................................A
    read the session report ? ....................S
    delete the session report ? ..................D
    change options ? .............................O
    quit ? .......................................Q
 choice > _
```

Figure 86. Screenshot of SESA.

SESA was developed in collaborative R&D project "Function Blocks" conducted with the group of Professor Hans-Michael Hanisch at Martin Luther University of Halle (Germany). The project was funded by DFG - German Research Foundation. After the retirement of Professor Starke SESA support by Humboldt University has been discontinued.

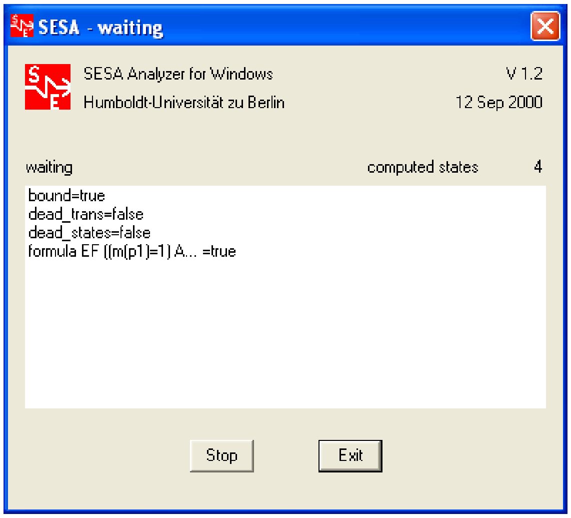
In 2008, the group at The University of Auckland's has created a new build of SESA (SESAcmd) which is 64-bit capable. This means it can use RAM beyond the 2GB limit of the 32-bit SESA.

The description of the command line version is in (SESA Manual, 2005).

With permission of Prof. Starke the code of SESA has been adapted and integrated into ViVe as the second model-checker STARK (also referred here as CTL-checker).

12.5 SESA from ViVe

Another version of SESA can be called from within Visual Verifier (Menu 'Analyze/Call SESA'). The built-in version of SESA allows checking specifications in both Computational Tree Logic (CTL) language and in Predicate Logic.

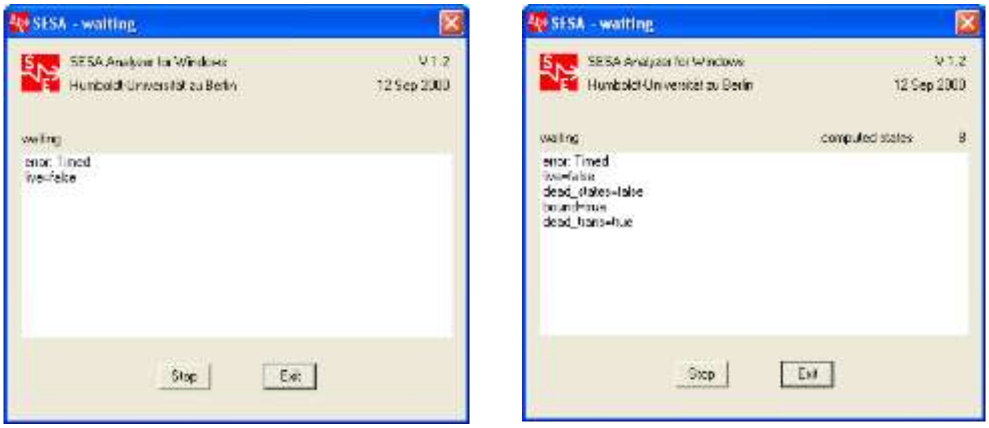


```
bound=true
dead_trans=false
dead_states=false
formula EF ((m(p1)=1) A... =true
```

Figure 87. Calling SESA from Visual Verifier produces such a window

SESA goes through a particular verification scenario and stops after net pre-check is completed. The button "STOP" is a bit confusing: in reality it means "CONTINUE"! So, while SESA is busy pre-checking the "STOP" button is blocked, but when it stops, the button becomes available. Pressing the button will conclude the reachability graph generations and will start the formulae verification. After formula has been proved, press EXIT.

The current version of SESA does not support timed NCES checking. So, if the checkbox “Timed” in ViVe is ON SESA would stop reporting an error. In this case press the “STOP” button to continue model-checking as shown in Figure 88.


SESA - waiting
SESA Analyzer for Windows
V 1.2
Humboldt-Universität zu Berlin
12 Sep 2000
waiting
Stop
Exit
SESA - waiting
SESA Analyzer for Windows
V 1.2
Humboldt-Universitat zu Berlin
12 Sep 2000
waiting
computed states
8
Stop
Exit

```
error: Timed
live=false
```

```
error: Timed
live=false
dead_states=false
bound=true
dead_trans=true
```

a)

b)

Figure 88 a) SESA stops after encountering the “Timed” option. Just press “STOP”; b) The model-checking continues.

The following table summarizes the dialects of NCES supported by these model checkers.

Features	Editor	SESA	Visual Verifier
Timed firing rules			
Interval ultimo	-	X	X
Earliest weak	-	X	X
Spontaneous transition firing		*-maximal steps* *- single*	*- all spontaneous,* *- all combinations,* *- single*
Greedy transitions	+	X	X
Synchro sets	-	X	-
Specifications	-		
Predicate logic	-	X	X
CTL	-	X	
Analysis			
Static analysis		X	-
Reachability graph		X	X
NCES			
Colours	-	X	-
Priorities	-	X	Yes: - for spontaneous

transitions

12.6 Hints for analysing complex models

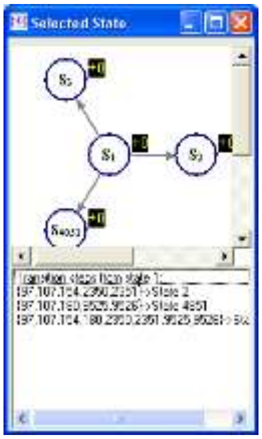
The three available model-checkers have different performance. For that reason the following sequence of steps can be recommended.

1. Assemble the model.
2. Call external SESA (without entering any specification) in order to estimate the number of states in the reachability graph. You can also enter some CTL specifications, but SESA will be able to give only 'YES' or 'NO' answer without providing a counterexample.
3. The built-in CTL checker is currently about 10 times slower than SESA as such, but it can export reachability graphs and provide counterexamples for CTL properties (which takes extra time).
4. The predicate checker (which is 3 times slower than the CTL checker, but does not take extra time for loading the reachability graph) can be used for initial check of model's feasibility. With it you can quickly create a smaller part of the reachability space and check if your model behaves reasonably. An indication of non-reasonable behaviour can be too large reachability space (generated by SESA). The predicate checker has two options:
   a. Breadth – first search (default)
   b. Depth – first search (selected by the "Recursive model-checking" check box);
   c. The "Check-on the fly" option allows checking a predicate without prior creation of reachability graph. Graph generation stops when a state satisfying the predicate is found. The created graph can be re-used for checking other predicates and it can be incrementally extended if necessary if the "->" button is pressed.
5. The CTL checker provides a choice of two firing rules: 'single spontaneous' and 'maximal steps'. The first rule leads to interleaved firing of spontaneous transitions and can eliminate some effects of a modelled concurrency. The second rule handles concurrency better.
6. The predicate checker offers an additional rule: "all combinations of spontaneous" which can be useful for modelling asynchronous concurrent processes.
7. ViVe can store two reachability graphs, generated by different model-checkers and/or with different firing rules. The switch of the "Current RG" implies that a relevant operation (such as generation of a trace to a state) will be applied to the currently selected reachability graph.
8. After reachability graph created by the CTL checker has been loaded, it can be also used for checking predicates with the "Predicate checker" using the "Search in the created graph" button.

12.7 Exploring reachability space

ViVe provides an option of looking into the reachability space by visualising the reachability graph. After the reachability space is generated in either of the two internal model-checkers, it can be visualised by checking the “Geo” box, which will assign geometrical layout to the generated states. Then the reachability graph will appear in the “RG” pane. This option, however, is beneficial only until the space becomes large. Here the available navigation options are:

- Zoom/ Unzoom the graph;
- Select a particular state by clicking on it; The selected state becomes current, so marking of any model part will be shown in this particular state if selected in the navigation tree.



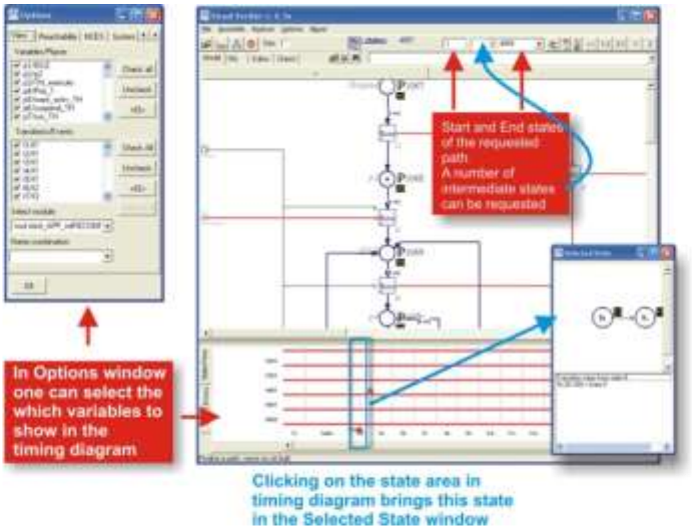
Transition steps from state 1:
197.107.154.2350.2351->State 2
197.107.160.9525.9526->State 4851
197.107.154.180.2350.2351.9525.9526->St

Figure 89. Selected State window.

Clicking on a state opens the new window providing a detail look on the state as shown in **Figure 89**. The state and its immediate successors are shown in the upper part. All states here are “clickable”, i.e. one can “travel” through the graph without having visualised the whole graph.

The lower part of the window shows the transition steps from the selected state.

Another navigation option is via the “Timing diagram view” of a particular path in the reachability space as shown in **Figure 90**. The path can be specified by its start state, end state and a number of intermediate states.



Start and End states of the requested path
A number of intermediate states can be requested

In Options window one can select the which variables to show in the timing diagram

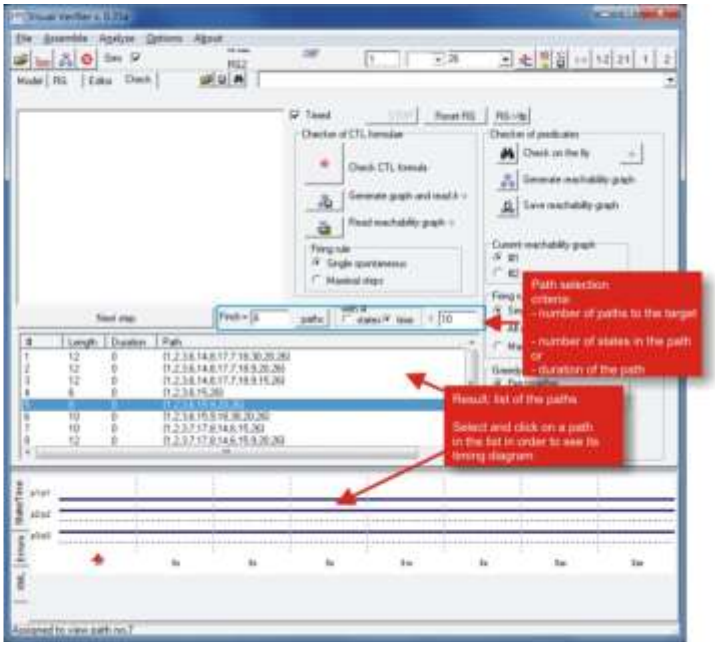
Clicking on the state area in timing diagram brings this state in the Selected State window

Figure 90. Navigating with timing diagram view of path.

12.8 Finding paths satisfying certain criteria

ViVe can find a group of paths from the initial state to the target state. This facility is located in the “Check” tab. It works for a generated reachability graph.

Currently, it can find either certain number of paths satisfying one of the criteria: maximum number of states or time duration not exceeding a limit. Clicking on a path in the list will display it in the timing diagram window.



#	Length	Duration	Path
1	12	0	[1,2,3,6,14,6,17,7,18,30,20,26]
2	12	0	[1,2,3,6,14,6,17,7,18,9,20,26]
3	12	0	[1,2,3,6,14,6,17,7,18,9,15,26]
4	6	0	[1,2,3,6,15,26]
5	8	0	[illegible]
6	10	0	[1,2,3,6,15,9,18,30,20,26]
7	10	0	[1,2,3,7,17,6,14,6,15,26]
8	12	0	[1,2,3,7,17,6,14,6,15,9,20,26]

Figure 91. The paths search.

12.9 Metrics

The largest SNS model built with ViVe so far was built of 744 modules and contained more than 6700 places and 10000 transitions. Its reachability space, however, was quite small – less than 5000 states.

SESA reportedly can deal with reachability spaces of millions of states. More hints for handling large model spaces in the CTL model-checker

If the model-checker stops without notice, most likely it is due to the “Out of memory” problem. In this case, however, the reachability graph is saved to disc (file with the name of your model and extension *.arc). You can explore this reachability space without visualising the graph as follows:

1) Quit and Restart ViVe;

2) Open and build the model again;

3) To minimize memory requirements in Options/View unselect all variables and select just those needed;

4) Read the reachability graph (but don't try to create its geometry!);

5) Generate timing diagram to the last state;

6) Go along the timing diagram and by clicking on states see in more details how many successors the state has. This may give you some idea of the RG structure.

In case if the model-checker crashes without a message (which is a symptom of the “Out of memory” situation), you can do the model-checking in 2 steps.

First, build the model, then exit ViVe, then open model but not build it and directly start generation of RG. The CTL checker takes the flat model saved during the previous run and starts RG generation. Then follow the steps described above. The model-building is taking extra memory which can be saved in this case.

13 Verification of Properties

13.1 Overview

The validation of automation systems modelled by NCES can be performed by simulation and formal verification via model checking. The simulation usually can follow a limited number of scenarios in the system's behaviour while the potential flaws can be in those paths left out unvisited. The multiple scenarios may result from the influence of some unpredictable factors, such as variable durations of some operations, communication delays, malfunctions, etc. In contrast, the model-checking explores all the existing scenarios.

The verification consists in proving specifications with respect to the dynamic behaviour of the model. The specifications can be given either in form of second order predicates, or in form of temporal logic expressions, for example in Computational Tree Logic (CTL). The basic terms of these expressions in most cases are the "values" of inputs and outputs (either of plant or controller) or, literally, the marking of the corresponding NCES modules modelling the data variables. As the hierarchical NCES model is converted into a flat S/E Net model this provides the through place/transition numbering, and these numbers are used as references to the values.

In case of the lifter the following groups of specifications were of the primary interest:

- Avoidance of potentially dangerous situations that may lead to a breakdown of the lifter or to damage of the product being transferred by the lifter. Example: when used in manufacturing of precise electronic components, such as hard drives, the lifter introduced and described in Section 15.12 must never allow the situations when the pallet leans or jumps. Such problem can be caused by inexact synchronization of conveyors' levels, which, in turn, may be a result of wrong synchronization of control programs;
- Robustness of the system in case of malfunctions of some sensors;
- Control programs can have branching logic of execution. Formal verification helps to prove that the response time is never exceeded in any feasible I/O combination in any branch;
- Avoidance of deadlocks or "dynamic traps" that may result from wrong synchronisation of operations;

- Presence of certain "checkpoints" in any possible scenario of behaviour that guarantees all necessary operations have been applied to the product in any circumstances;

The overall model of the automated Lifter had 3 hierarchy levels and after assembly from modules encountered 571 places and 828 transitions. However, the model-checking of the normal behaviour (without modelling malfunctions in sensors) resulted in a reachability space not exceeding 60000 states which was generated on a usual laptop less than in a minute. This result reflects the efficiency of distributed state modelling with NCES.

Besides the possibility to verify or falsify certain properties of the system, another important advantage is that the method may be applied in absence of physical controller and physical plant. Consider the following scenario: the manufacturing line where conveyor modules, lifters, workstations, robotic cells, etc., is being installed. Mechanical and electrical engineers do installations and tests of the equipments. The time for project runs out, the deadline is approaching, but the control engineer had no chance to test the line, since the physical equipment is not ready yet. In this situation the application of this method (model-based validation with the model of controller derived from the source code) may provide an environment for independent development of control, while the physical plant is being set up.

13.2 Syntax of specifications

The combination of Visual Verifier and SESA allows verify specifications given as first order predicates, and temporal logic formulas given in Computation Tree Logic language (CTL). Note that at the moment syntax of specifications allowed by Visual Verifier and SESA is a bit different.

A first order predicate is a Boolean expressions that uses marking of places or firing of transitions as variables along with usual Boolean operations such as & - and, ! – or, ~ - not, and brackets. In Visual Verifier specification can use only Boolean expressions over marking (i.e. place is marked or not marked).

Example:

Visual Verifier	p1 & ~p4 p1 and not p4	Search for states where marking of place p1>0 and marking of place p4 is 0.
SESA	m(p1)=5	Search for states where marking of place p1 is 5
SESA	(m(p1)=5)AND (m(p2)=2) For Boolean marking: AG(p1ANDp2)	No spaces are allowed between terms in SESA

NOTE: In Visual Verifier operands and operations MUST be separated by spaces. No spaces are allowed between terms in SESA.

13.3 How to check specifications

1. Enter a specification, e.g. a CTL formula in the specification area. For example: EG ((m(p1)=1) AND (m(p4)=1))

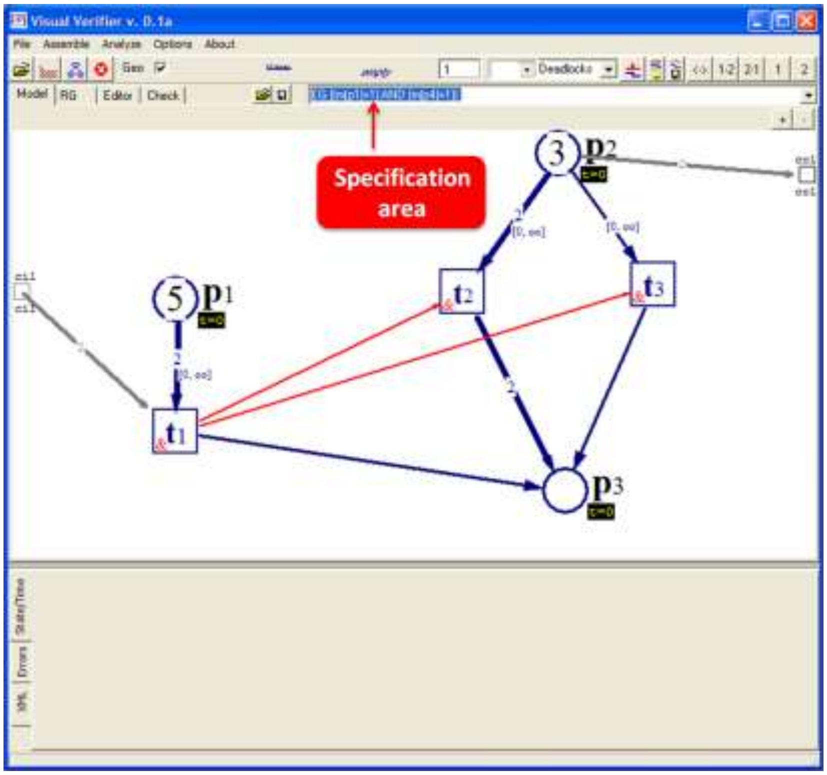
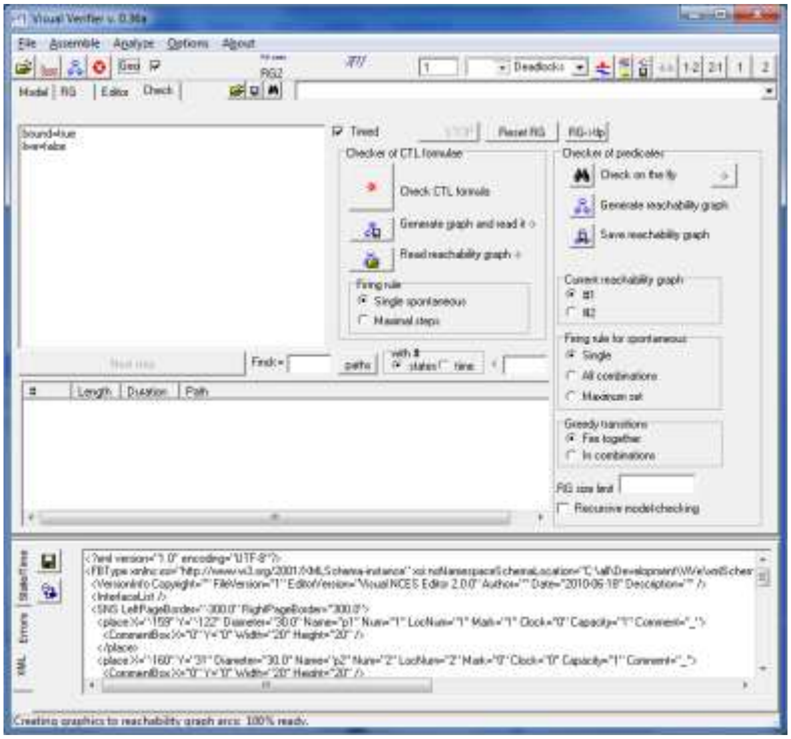


Figure 92. Field for entering **s**pecifications in Visual Verifier.

2. Go to the “Check” tab.



```
bound=true
live=false
```

```
<?xml version="1.0" encoding="UTF-8"?>
<FBType xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="C:\all\Development\VVe\xmlSchem
 <VersionInfo Copyright="" FileVersion="1" EditorVersion="Visual NCES Editor 2.0.0" Author="" Date="2010-06-18" Description="" />
 <InterfaceList />
 <SNS LeftPageBorder="-300.0" RightPageBorder="300.0">
  <place X="-159" Y="-122" Diameter="30.0" Name="p1" Num="1" LocNum="1" Mark="1" Clock="0" Capacity="1" Comment="_">
   <CommentBox X="0" Y="0" Width="20" Height="20" />
  </place>
  <place X="-160" Y="31" Diameter="30.0" Name="p2" Num="2" LocNum="2" Mark="0" Clock="0" Capacity="1" Comment="_">
   <CommentBox X="0" Y="0" Width="20" Height="20" />
```

Creating graphics to reachability graph arcs: 100% ready.

3. Press the “Check CTL formula” button.

Examples of specifications are presented in 15.12 for the Lifter introduced in Section 15.11.

14 Distributed controllers

14.1 Discrete-state model

Distributed control systems may include several autonomous controllers either working asynchronously (with data exchange via network), or co-existing within one device thus being synchronized. The distributed architecture may create new trajectories in the system's state space which are not the case in local centralized architectures. Consider the case of two previously presented subsystems working concurrently.

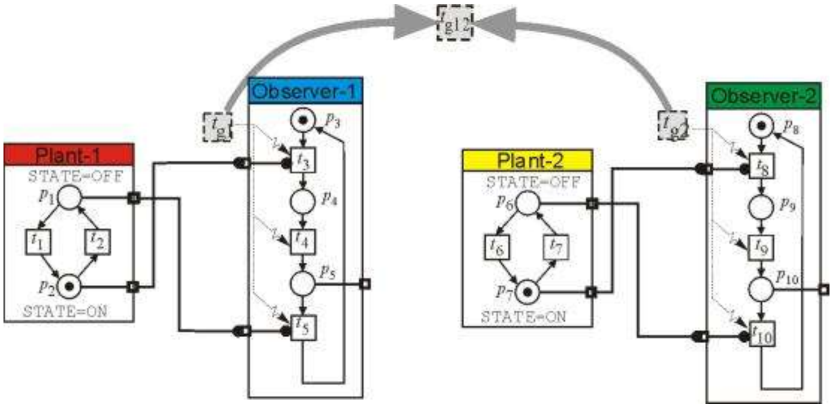


Figure 93 Two concurrent processes in the plant being observed by two independent controllers.

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 there 12 enabled steps: {{tg1,t3},{tg2,t8},{tg1,t3,tg2,t8}} X {{},{t2},{t7},{t2,t7}}.

For example, { tg1,t3,t2},{tg2,t8,t2,t7}, {tg1,t3,tg2,t8}, etc. Thus, the step {tg1, t3} models the situation when the Observer1 has started processing, while the other observer still has not.

The proposed solution allows easy change from distributed to centralized architecture. To model placing both observers in the same controller device (synchronisation), it is enough to substitute the two tick generating transitions tg1 and tg2 by tg12 as shown in the Figure 1. Then only 4 steps would be possible in the given state: {tg12, t3, t8} X {{t2},{t4},{}}.

14.2 Timed model

In timed models behaviour similar to the "greedy" transitions can be modelled by means of synchro-sets. **The synchro set model is implemented only in SESA model checker (and in STARK).** All transitions in the controller part of the model, which are enforced by the arcs from "greedy" transitions in the non-timed model, are marked as members of a particular synchro-set, associated with the controller. Membership in a synchro-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 impact 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 need to separate them out from other groups of actions taking place in other controllers.

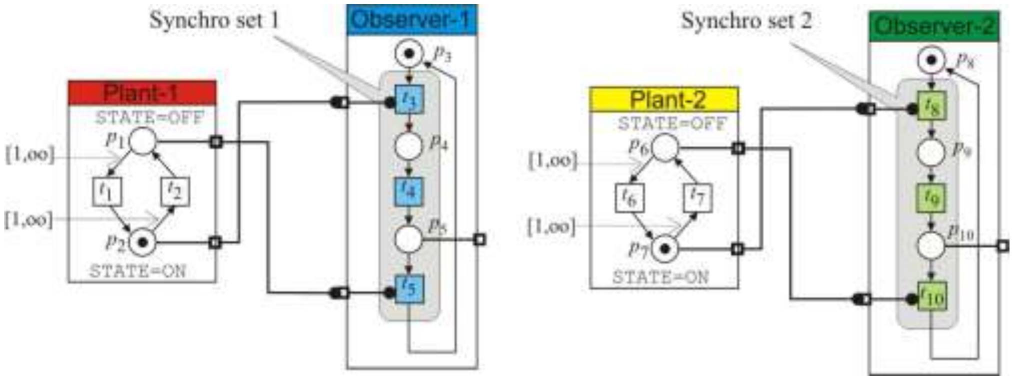


Figure 94 Synchro sets.

In the example in **Figure 94** we defined two synchro sets: S1={t3,t4,t5} and S2={t8,t9,t10} to model the allocation of the observers to separate devices. The reachability graph for this model is presented in Figure with initial state S1 equivalent to that shown in Figure 95 and with clock values equal to 0 at all places.

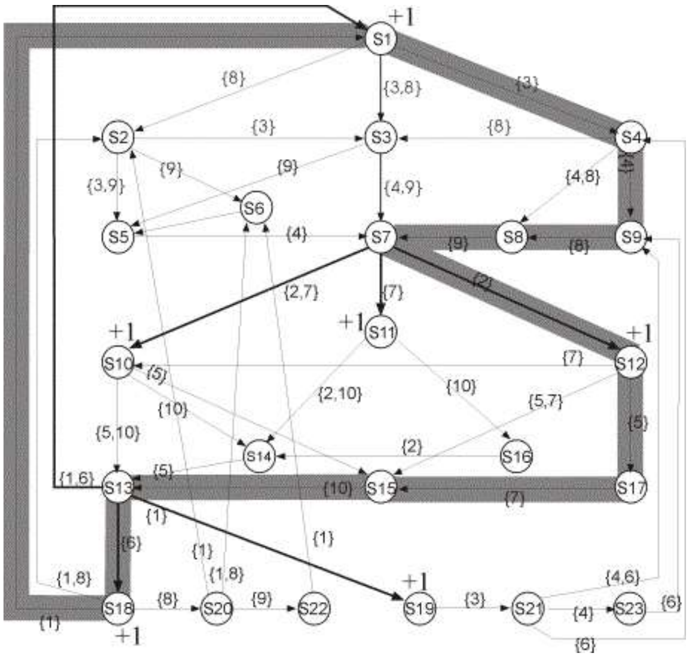


Figure 95 Reachability graph for the timed model with two synchro-sets (The “observers” reside in independent devices). The bold arcs have duration 1.

A path in the reachability graph corresponds to a particular scenario of system’s operation. Consider, for example, the path outlined in Figure 3. The state/time diagram of the model propagating along this path is shown in Figure 4. The behaviour along the path is as follows: the observer 1 starts computation in respond to occurrence of the marking in p2. It makes two computation steps before the observer 2 starts its execution in respond to another event (marking in p7). Since the results of computations may be used by other controllers, their sequence is important.

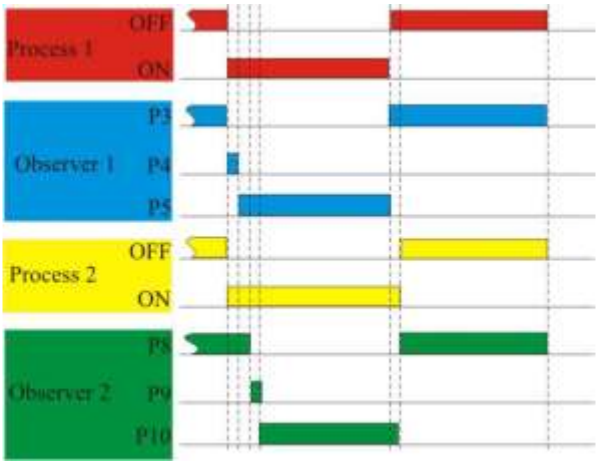


Figure 96 State/timing diagram of the outlined path in the reachability graph. (Small intervals represent states with zero-duration, larger intervals represent states with duration 1).

14.3 Using synchronous transitions

To use synchrosets in SESA one needs to create a file with ext '*.syn' and same name as the model file. An example of such a file is as follows:

```
Synchro for net 1:
  1: 188, 189;
  2: 173, 175;
@
```

This file defines two synchro sets, one with transitions t188 and t189, and the other with t173 and t175.

14.4 Synchro sets in ViVe

The model generator of ViVe automatically creates the *.syn file when assembles the flat model if the parameter "timed" is ON and if the model includes greedy transitions. All greedy transitions are included into the same synchroset.

To enable synchrosets when model checking with internal SESA, one needs to use the "Maximal steps" firing mode. In the "Single spontaneous" mode synchrosets are ignored.

You can modify manually the *.syn file in order to define another "layout" of synchro sets and then use either command line SESA or STARK.

14.5 Example of different firing rules application

We illustrate differences between available firing rules in both model-checkers using the following example.

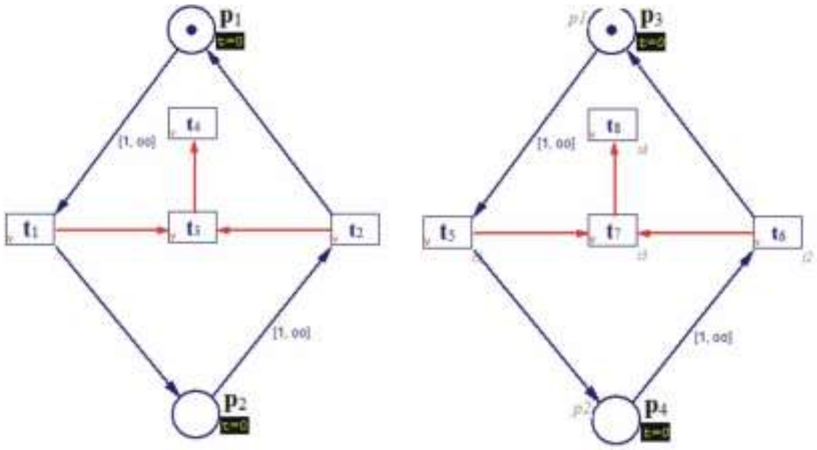
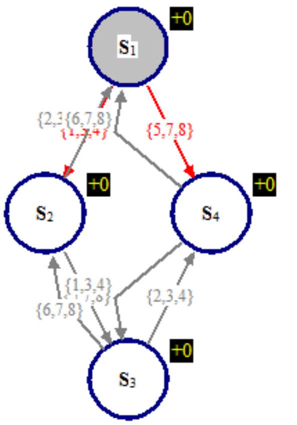
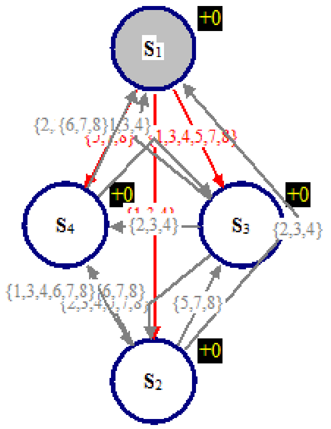


Figure 97. Test model of two concurrent processes.

14.5.1 Non-timed

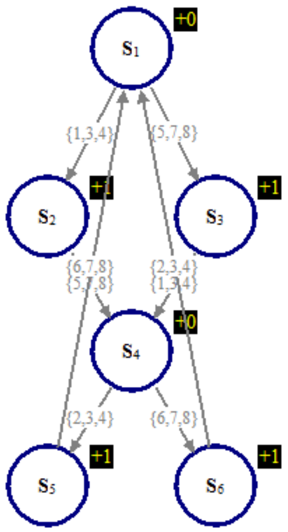
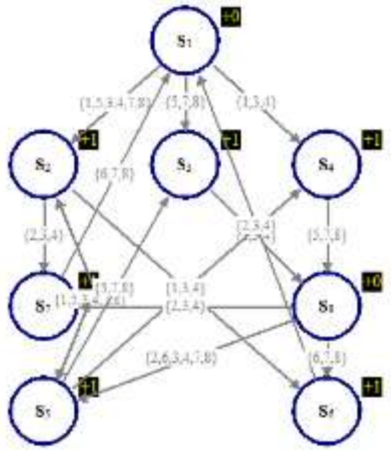
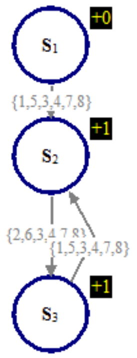
In the non-timed mode the timing on arcs is ignored.

STARK

Single spontaneous	Maximal steps
 S1 +0 {2,3{6,7,8} {1,3,4} {5,7,8} S2 +0 S4 +0 {1,3,4} {6,7,8} {2,3,4} S3 +0	 S1 +0 {2, {6,7,8}1,3,4} {1,3,4,5,7,8} S4 +0 {2,3,4} S3 +0 {2,3,4} {1,3,4,6,7,8}{6,7,8} {5,7,8} S2 +0

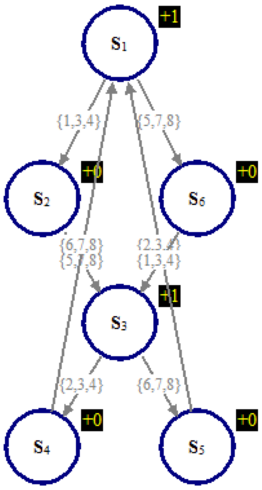
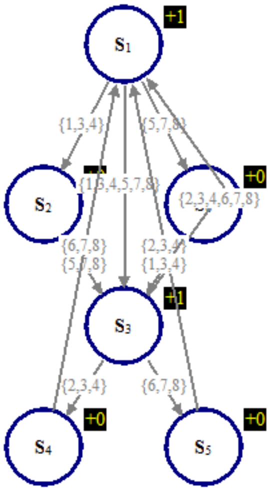
14.5.2 Timed

ViVe model checker

Single spontaneous	All combinations	Maximal
 S1 +0 {1,3,4} {5,7,8} S2 +1 S3 +1 {6,7,8} {5,7,8} {2,3,4} {1,3,4} S4 +0 {2,3,4} {6,7,8} S5 +1 S6 +1	 S1 +0 {1,5,3,4,7,8} {5,7,8} {1,3,4} S2 +1 S3 +1 S4 +1 {6,7,8} {2,3,4} {2,3,4} {5,7,8} S7 {5,7,8} {1,3,4} {2,3,4} S8 +0 {2,6,3,4,7,8} {6,7,8} S5 +1 S6 +1	 S1 +0 {1,5,3,4,7,8} S2 +1 {2,6,3,4,7,8} {1,5,3,4,7,8} S3 +1

STARK model-checker

Single spontaneous	Maximal steps

 s1
+1
{1,3,4}
{5,7,8}
s2
+0
s6
+0
{6,7,8}
{5,7,8}
{2,3,4}
{1,3,4}
s3
+1
{2,3,4}
{6,7,8}
s4
+0
s5
+0
	 s1
	+1
	{1,3,4}
	{5,7,8}
	{1,3,4,5,7,8}
	s2
	+0
	{2,3,4,6,7,8}
	{6,7,8}
	{5,7,8}
	{2,3,4}
	{1,3,4}
	s3
	+1
	{2,3,4}
	{6,7,8}
	s4
	+0
	s5
	+0

14.6 Model modification: synchronous transitions

Let us consider how the behaviour of the model would change if we constrain the behaviour of a single process by making one of the transitions synchronous as show in the Figure below.

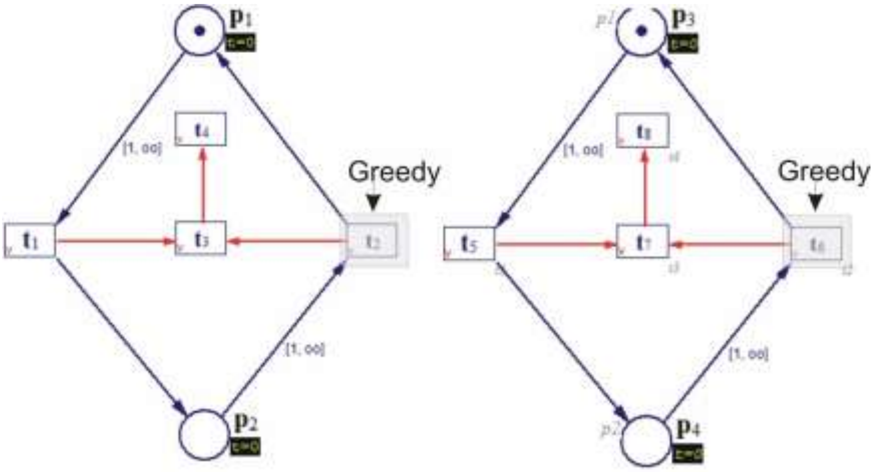
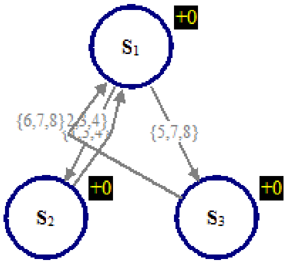
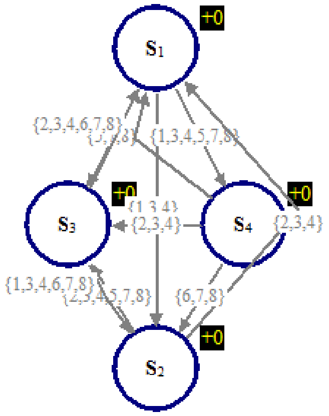


Figure 98. One transition is made synchronous instead of spontaneous in each model.

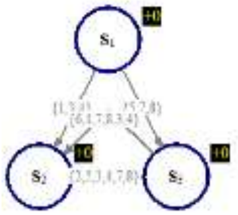
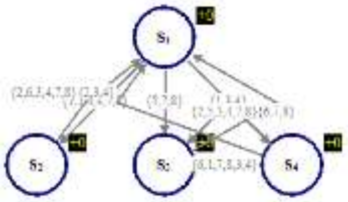
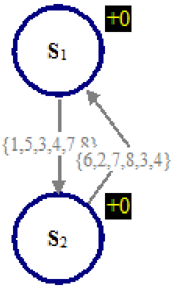
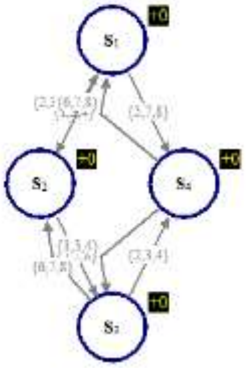
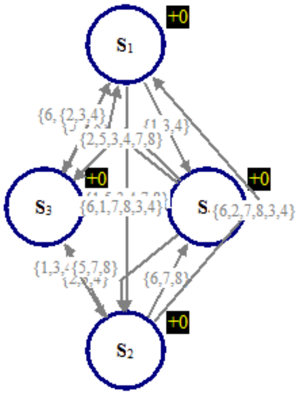
14.6.1 Non-timed

STARk:

Single spontaneous	Maximal steps

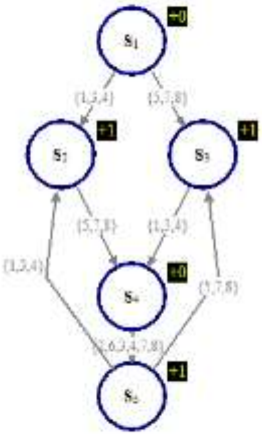
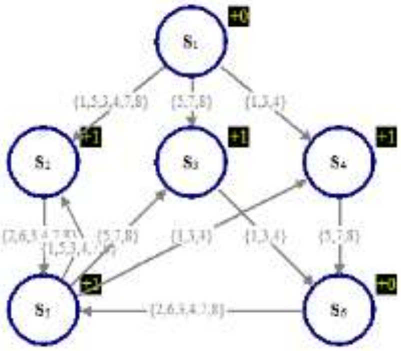
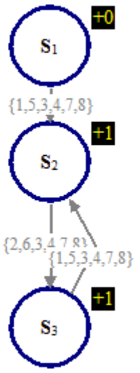
 S1
+0
{6,7,8}
{2,3,4}
{5,7,8}
S2
+0
S3
+0
	 S1
	+0
	{2,3,4,6,7,8}
	{1,3,4,5,7,8}
	S3
	+0
	{1,3,4}
	{2,3,4}
	S4
	+0
	{2,3,4}
	{1,3,4,6,7,8}
	{2,3,4,5,7,8}
	{6,7,8}
	S2
	+0

ViVe

Single spontaneous	All combinations	Maximal steps
All Greedy together 	All Greedy together  S1 +0 {5,7,8} {1,3,4} S2 +0 S3 +0 S4 +0	
Combinations of greedy  S1 +0 {2,7,8} S2 +0 S4 +0 {1,3,4} {2,3,4} S3 +0	Combinations of greedy  S1 +0 {6,{2,3,4} {1,3,4} {2,5,3,4,7,8} S3 +0 {6,1,7,8,3,4} S {6,2,7,8,3,4} +0 {1,3,4{5,7,8} {2,3,4} {6,7,8} S2 +0	

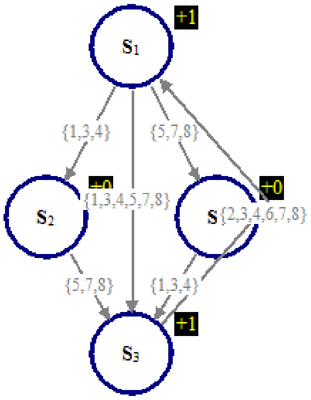
14.6.2 Timed

ViVe checker

Single spontaneous	All combinations	Maximal steps
 S₁ +0 {1,3,4} {5,7,8} S₂ +1 S₃ +1 {5,7,8} {1,3,4} {1,3,4} S₄ +0 {5,7,8} {2,6,3,4,7,8} S₅ +1	 S₁ +0 {1,5,3,4,7,8} {5,7,8} {1,3,4} S₂ +1 S₃ +1 S₄ +1 {2,6,3,4,7,8} {1,5,3,4,7,8} {5,7,8} {1,3,4} {1,3,4} {5,7,8} S₅ +2 {2,6,3,4,7,8} S₆ +0	 S₁ +0 {1,5,3,4,7,8} S₂ +1 {2,6,3,4,7,8} {1,5,3,4,7,8} S₃ +1

STARk

In STARk, in timed mode only the "Maximal steps" firing rule is applicable if the model includes synchronous transitions. These are interpreted as one single synchro set.

Single spontaneous	Maximal steps
- Not applicable	 S₁ +1 {1,3,4} {5,7,8} S₂ +0 {1,3,4,5,7,8} S +0 {2,3,4,6,7,8} {5,7,8} {1,3,4} S₃ +1

14.7 Modelling communicating processes

Communication can be modelled using the standard buffer approach as illustrated below.

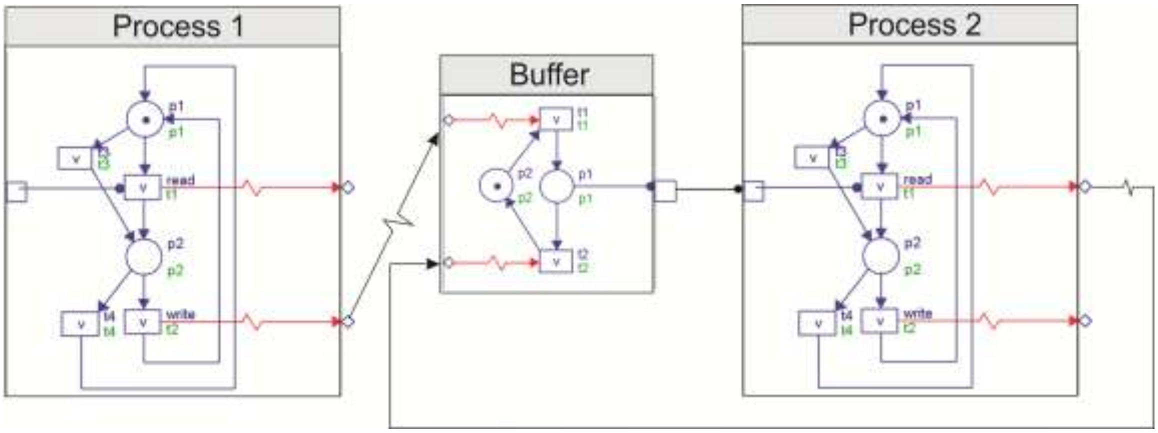


Figure 99. Model of two processes communicating via buffer of a unit capacity.

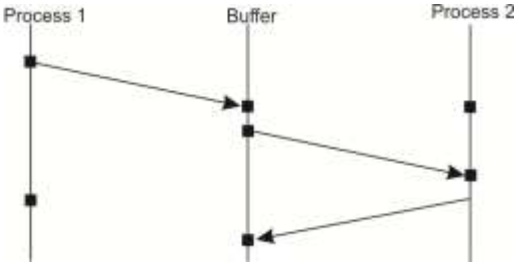


Figure 100. Message passing between process.

15 Example of a distributed system: two cylinders

Now let us consider a more complicated example of a system with distributed control. In the system of two cylinders in Figure 101 each cylinder pushes a workpiece to the destination hole. The process starts when the workpiece appears in front of the corresponding cylinder as indicated by sensors P1 and P2 respectively.

As it is clear from the Figure, cylinders can collide in the middle point, therefore the goal of controller design is to avoid such a situation.

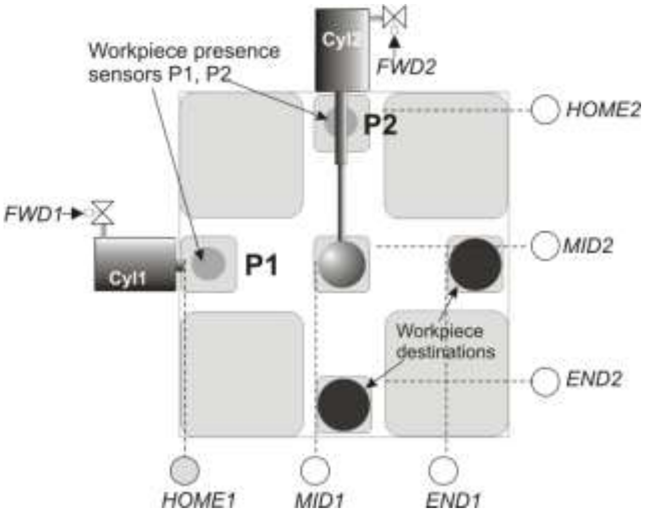


Figure 101. Two cylinders with a potential clash in the middle.

There are many possible ways to achieve the desired behavior, which can be done by designing a “central” controller of both cylinders, or a protocol ensuring that distributed controllers collaborate correctly. Distributed control is of interest for many practical reasons: imagine that control logic is “embedded” in each cylinder, so they can start working as soon as powered on.

15.1 Reusing original controllers

What if we take the individual cylinder controller introduced in Chapter 10 and let the cylinders to operate? A slight modification will be required to start the operation only on appearance of a workpiece. For that we introduced new input “*wps*” (workpiece sensor) as shown in **Figure 102**.

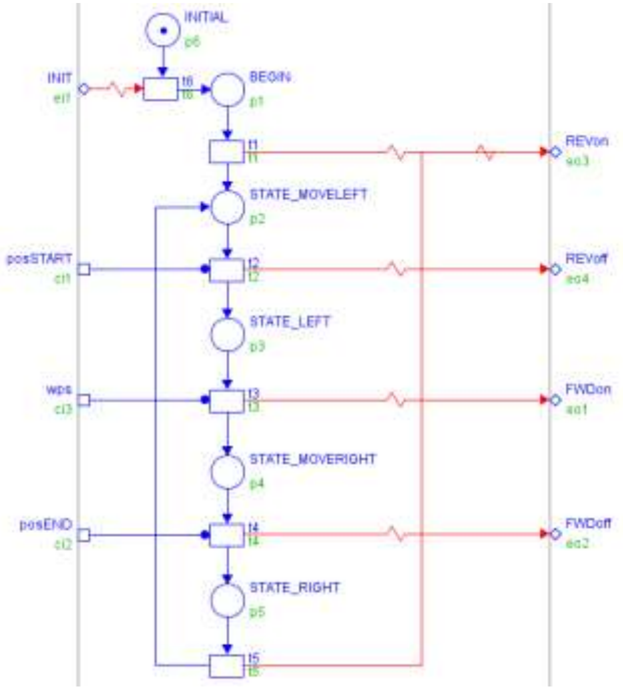


Figure 102. Modified cylinder controller with the added “wps” input.

15.2 Finding collision

The entire model of two cylinders is presented in Figure 97. The module “materials” models the position of workpieces. The workpieces can be pushed by the corresponding cylinders. The cylinders have an additional output “exc_mid” indicating that the cylinder is extended so that its tip exceeds the middle position. Cylinders collide if both exceed the middle position.

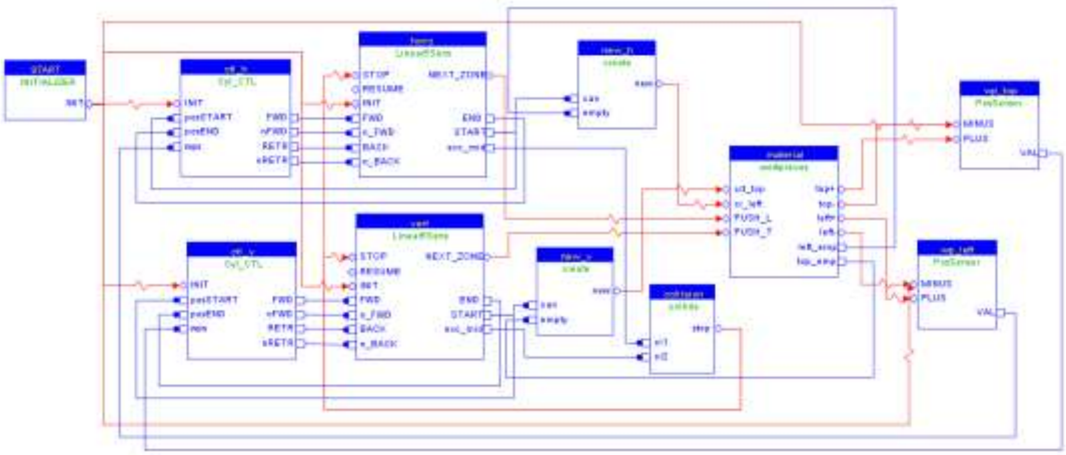


Figure 103. The model of two cylinders with a possible collision.

The role of the “collide” model (**Figure 104**) is to emit the “stop” signal after which both cylinders will move to the “Emergency STOP” state encoded by the place p4 in Figure 65. When ci1 and ci2 are TRUE, the model jumps to the p2 state and emits the event “stop”.

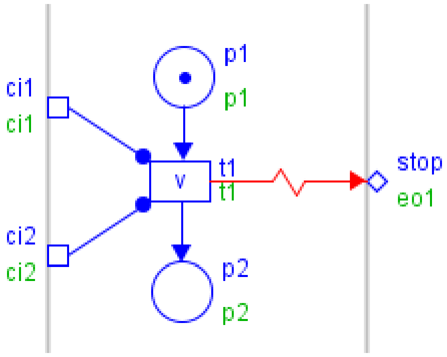


Figure 104. Model of collision.

The model will automatically enter the deadlock in case of a collision.

15.3 Block – permit protocol

Now, let us change the control logic so that one cylinder would allow the other to move only if it is not moving itself. For that we add an input “can_move” and output “permit” to the controller module types. The model is shown in

Figure 105.

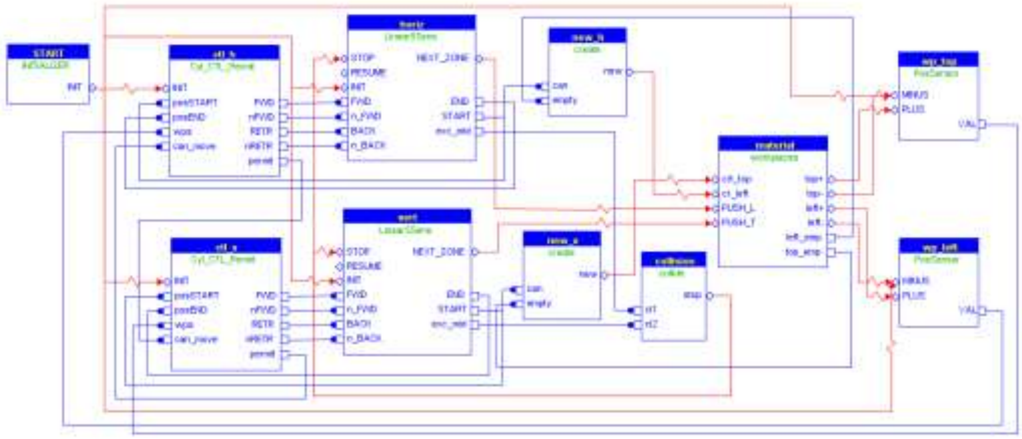


Figure 105. Model with controllers’ coordination.

If this model is checked with the “single spontaneous rule” it shows no deadlocks, implying that no collision of cylinders can occur.

15.4 Central controller

15.5 Exercises

1. Check the model of two cylinders with distributed coordinated control using the “all combinations” firing rule. Explain the results.

2. Develop a central controller module for two cylinders.

3. Develop a supervisor module for correct avoidance of collisions with minimum time losses on waiting.

16 Modelling Programmable Logic Controllers (PLCs)

16.1 System routines

Precise modelling of automation systems requires taking in account low level details of the control program execution in a PLC. The PLC programs are executed in a cyclic way. One cycle consists of the following phases: first the inputs are read, then the program logic is executed and then the outputs are written. Figure 26 depicts a NCES skeleton for a PLC model. Place p_1 holds a token representing the initial state of the PLC execution, if the PLC program is enabled (condition input to the t_1 transition) the cycle is started by the update of outputs and acquisition of input values. The firing of t1 transition generates these events. When a token is placed to *p2*, it resides there until a signal notifying about the change in the system enables transition *t2*. The monitoring of the changes in the systems is needed in order to not start a new PLC cycle unless something has changed in the system.

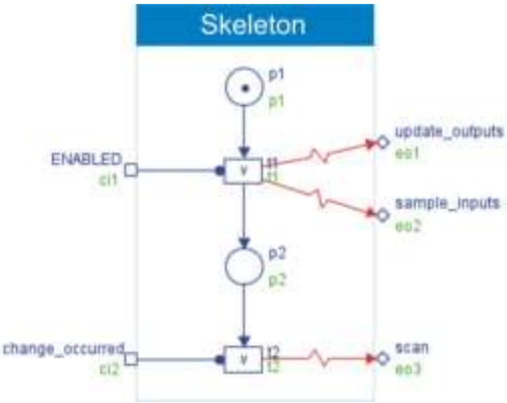


Figure 106. PLC model skeleton

The state of the NCES model is distributed and is defined by the marking of all places. Additionally to the marking, the state is characterized by the time stamp, e.g. the time the certain state (marking) is valid. Thus the two states representing the same marking but holding different times are different states.

A special module that monitors the change in the system has to be added to the model (Figure 107). The module has two event inputs for retrieving information about any change of the PLC program variables during the scan cycle. It does not make sense to run the model over the new scan-cycles if the markings in the model remain unchanged, i.e. when nothing would change during the next scan. The marking may change in the model of the plant or if the time dependent transition fires in a timer NCES module in the model of controller.

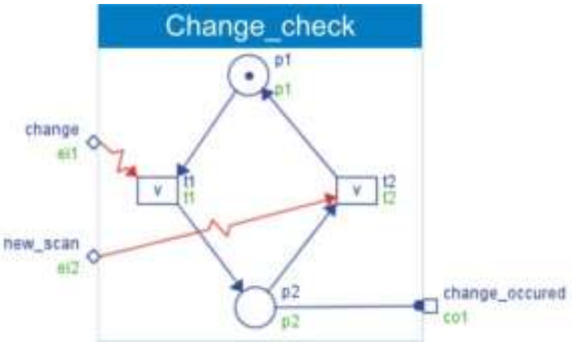
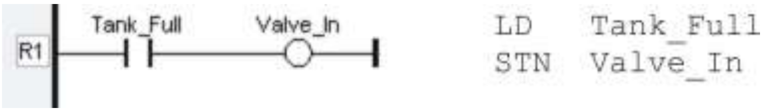


Figure 107. Change monitoring NCES module.

16.2 Ladder logic

Let us consider an example of ladder diagram in Figure 108. Textual representation of the first rung of the diagram is given on the right. The textual representation of LD resembles the Instruction List programming language.



```
LD   Tank_Full
STN  Valve_In
```

Figure 108 Tank control program

An LD instruction (that stands for *Load*) loads the value at *IX_Tank_Full* variable into accumulator. At the second row, the negated value of accumulator is stored to output variable *QX_Valve_In* that controls the valve. So, once the tank is filled, *IX_Tank_Full* becomes TRUE, the FALSE value is stored into the *QX_Valve_In*, that will close the valve.

In the given example it is just an evaluation of a single bit variable *IX_Tank_Full*. If the variable was not met in previous racks its model will be added to the PLC model. The PLC model skeleton is extended by the rack representation. In the current example, which has only one rack, *p3* place along with *t3* and *t4* transitions represent the rack 0. Figure 10 represents the PLC logic model. The whole PLC model consists of interconnected I/O modules (Figure 46), a change monitoring module (Figure 107), and a PLC logic model (Figure 109).

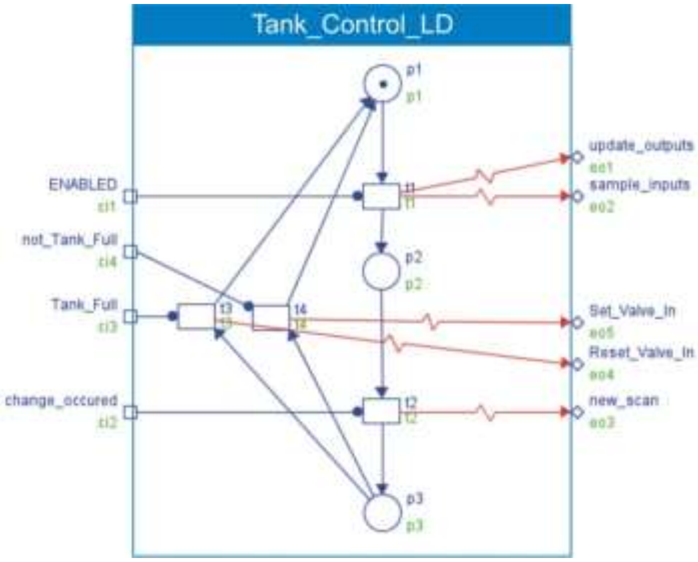


Figure 109. PLC logic model for tank control program.

17 Modelling of Complex Plants

In this Chapter we present more details on how the modelling of plant may benefit from the hierarchical model organisation and the reuse opportunities provided by the extended NCES. Thus, common modelling components may be reused in the same model and across different models.

Models of the plant and model of the controller are interconnected into the closed-loop providing the representation of entire system that consist of the controlled equipment and control device. The combined model is subject for making a judgement about modelled system properties by means of model-checking.

Depending on the required accuracy of modelling, 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 the complex models of equipment units, such as machine tools, other material processing and storage units, and the transportation means. The approach presented in this section extends the ideas of plant modelling of *(Hanisch et al, 1998), (Hanisch and Lüder, 2000).*

17.1 Process/Sensor model

Event arcs are able to express the variety of instantaneous actions, one of which is operation of sensor which detects the changes of the plant's state. Once transition t1 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 p2) and after the delay D – to the state with p3=1. Reading of the sensor usually comes to controller as a logic value modeled in our formalism as a condition signal. The sensor itself can have internal dynamics, e.g. delay D, as it is shown in the **Figure 110**, or an additional "malfunction" state (not shown in the Figure, but similar to the "exception state" considered earlier. Note that in the figure 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 results of modelling.

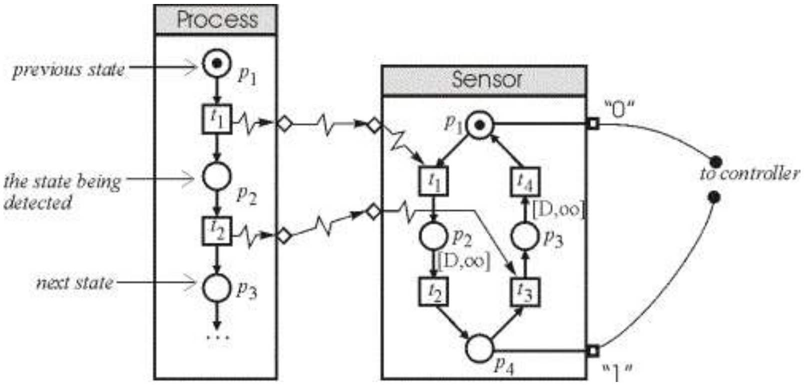


Figure 110 Model of sensing: sensor detects when the process comes to the observed state and with delay produces the required logic value (places p1,p4 of the module "Sensor").

17.2 Tank

Common modelling components may be reused in the same model and across different models. A common example of that is modelling of Boolean input and output variables that can be seen in Figure 111 that represents a plant model of the tank from Figure 39. The model of the plant has two Boolean variables corresponding to the valve and the level sensor. The valve is modelled by an input variable while the level sensor by an output variable. This is opposite to the model of the controller, where PLC program has a valve related variable as an output and the level sensor as an input.

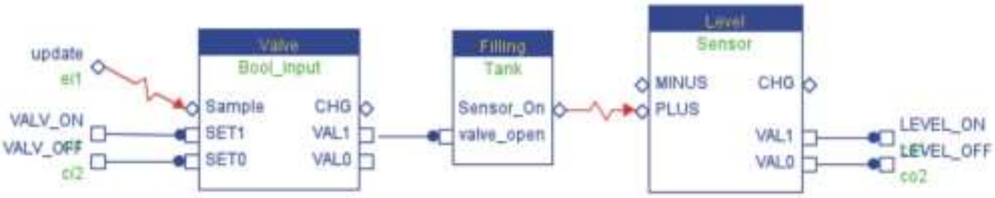


Figure 111 Model of the filling process –with valve input and level sensor output.

Figure 111 shows the model of the plant that embodies all the low level modules and is ready to be interconnected with the model of the controller. The filling process model of the tank has one condition input *valve_open* and one event input *turn_on_Sensor*. Figure 112 shows (a very simplistic) model of the process in detail. This is a trivial abstraction of the real filling process that has only two *states* (*p1* and *p2*) that may be seen as not filled (*p1*) and filled (*p2*). Once the incoming condition signal of the transition *t1* is TRUE for at least of 10000 time units (here 1 unit = 1ms) the *t1* fires generating output event that actually turns sensor on.

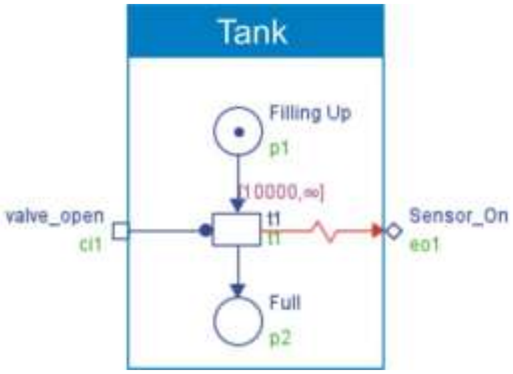


Figure 112. Model of the filling process

Models of the plant and model of the controller are interconnected into the closed-loop providing the representation of entire system that consist of the controlled equipment and control device. For the tank example, Figure 113 depicts the model of interconnected controller and plant. The combined model is subject for making a judgement about modelled system properties by means of model-checking.

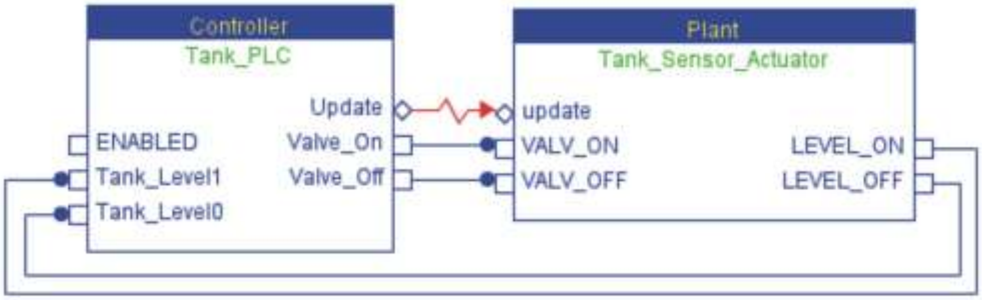


Figure 113. Closed-loop representation of the system at the highest level of hierarchy**.**

17.3 Conveyor

Let us consider the model of a conveyor shown in Figure 114, left side.

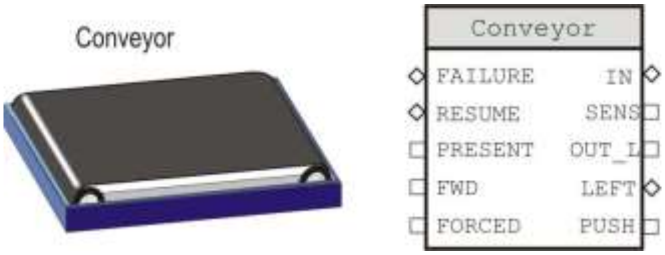


Figure 114. Conveyor and the NCES model interface

We will use two different types of conveyors– one capable of moving only in one direction, and another moving in both directions. The model of the more complex conveyor can be created based on the simple model using the mechanism of inheritance.

The interface of the model type "**Conveyor**" can be seen in Figure 114, right side. The model itself can be conceptually divided into three elements: Status, Position, and Sensor as shown in the class diagram in Figure 115, left. The Status element of type **MovingStatus** models the behaviour of the motor that drives the conveyor and converts the logic control signals into one of the states "Moving" or "Standing still" (that corresponds to the one-directional conveyor). Input "PRESENT" indicates if a pallet is present, and input "FORCED" is used to indicate the influence of a neighbour belt on the movement of the pallet. The output condition FW_ST is used by the model of belt position.

The structure of the model of the bi-directional conveyor is identical to that of the uni-directional one. The difference is in the module **Status** that has type **MovingStatus2D** that inherits the interface properties of the one-directional **MovingStatus** and extends them with one more input and output for the retracted movement. This is shown in Figure 115(right). All transporters are equipped with a single position sensor indicating the presence of the pallet (fully loaded on the conveyor).

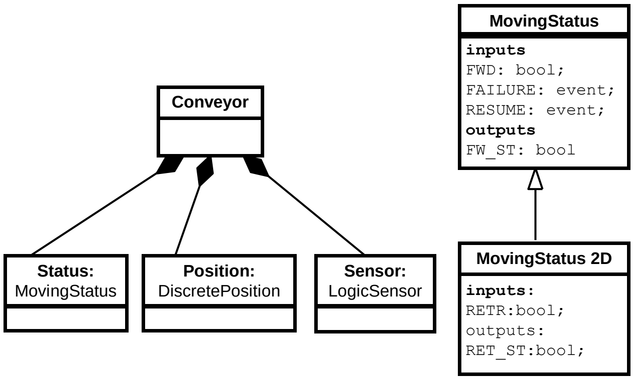


Figure 115. Model type definition of the conveyor and inheritance of the MovingStatus model types.

The condition and event flow connections between the sub-models constituting the model of the conveyor are represented in Figure 116.

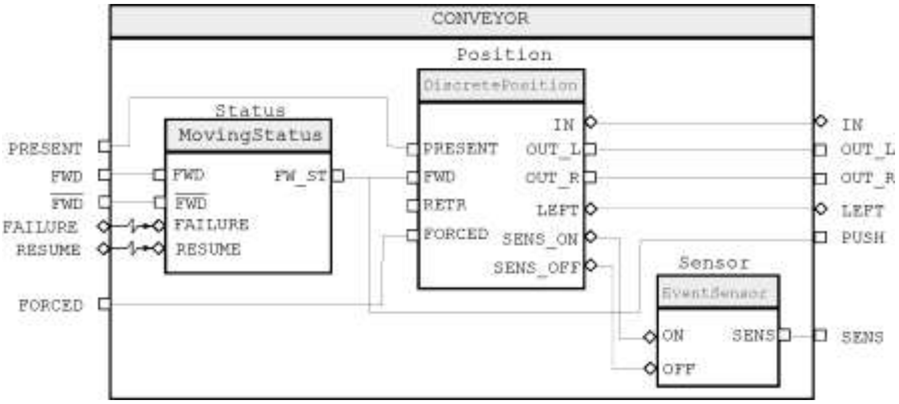


Figure 116**.** Modular view of the model of conveyor.

The basic models can be described further in form of NCES modules. Figure 117,A shows an implementation of the **MovingStatus** in NCES. The model receives the control signal FWD and transforms it into the state of the belt: place p_2 corresponds to the state “belt stands still”, place p_1 – belt moves and p_3 to the state indicating a failure. The belt moves when the control signal FWD is ON, and stops when the signal goes OFF (in the model the negation of the signal FWD goes on).

An occurrence of a failure is indicated by an external event that may come from the corresponding model. For example, that can be a nondeterministic model of failures. Note that the model is sensitive to failures only when the belt moves, i.e. when the place p_1 is marked. It is assumed that the failure can be fixed by an external interaction indicated by the event input RESUME.

The model **MovingStatus2D** for the bi-directional moving belt is shown in Figure 117,B. It models an additional state of backwards moving, and correspondingly has more transitions between the possible states.

The position of the pallet on the belt can be modelled with different precision. A qualitative model in Figure 118 distinguishes only 3 states of a pallet on the belt: no pallet, pallet on the belt with its front edge between the belt's ends, and pallet's front edge is beyond the right end of the belt.

A more precise modelling of the position can be done using the timed version of NCES. Let us assume that the belt is three units long and the pallet is two units long as shown in Figure 119. The speed of the belt is one unit of the length per second. Then it will take three seconds for a pallet to reach the right end of the belt and 2 more seconds to leave the belt completely.

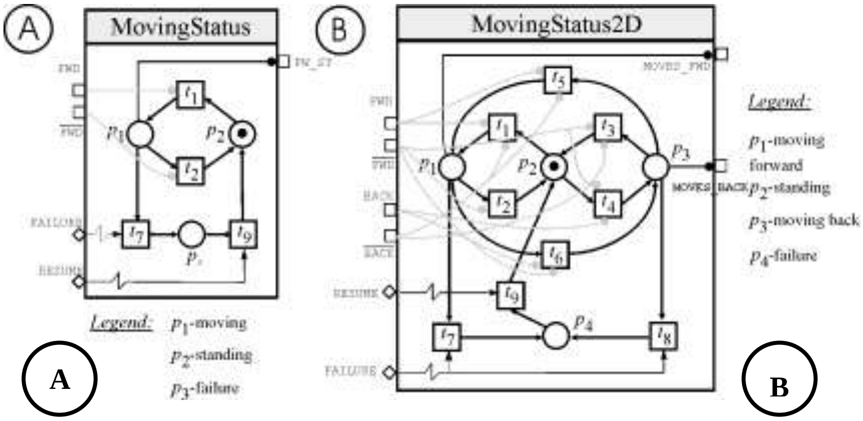


Figure 117. Models of the moving status for uni-directional and bi-directional belts.

Place p_1 corresponds to the state "No pallet". When a pallet appears (input condition "Present") and the state of the moving belt is "Moving forward" (indicated by the input condition FWD) then the transition t_1 occurs and the token goes to place p_2.

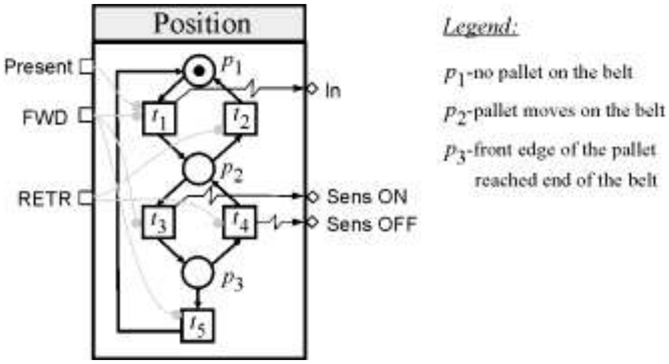

Position
Present
FWD
RETR
p_1
t_1
t_2
p_2
t_3
t_4
p_3
t_5
In
Sens ON
Sens OFF
Legend:
p_1-no pallet on the belt
p_2-pallet moves on the belt
p_3-front edge of the pallet reached end of the belt

Figure 118. A qualitative non-timed model of the pallet's position.

This place indicates the state “Front edge of the pallet is in the interval 1 of the conveyor”. Another reason to transfer to this state is the presence of the input condition “Forced”. This condition indicates that the pallet is pushed onto the belt by some external force that maybe another moving belt positioned backwards to this one. This option is modelled by transition t_{12}. In general, the moving in this case is slower than if driven by the own motor of the belt. The presented model, however, does not cover with enough precision the case when both forces are present simultaneously. Note that the transition from p_1 to p_2 (either via t_1 or t_{12}) is a qualitative one and does not take time (more precisely has zero delay).

The places p_2-p_4 correspond to the location of the pallet (again the front edge) in the intervals 1-3 respectively. A transition from interval i to interval i+1 occurs in either case “FWD” and “Present” or “Forced” and “Present”.

The latter, however, works only till less than the half of the pallet is on the belt – beyond this point the friction force would not let the pallet move driven only by the external force. The moving to the next interval takes 1000 ms if driven by the own motor of the belt or twice as long under the external force. The backward moving from interval i+1 to interval i occurs if the combination of input conditions “RETR” and “Present” are true. It also takes 1000ms under assumption that the speed of the moving belt in both directions is the same.

Arriving of the pallet to the 3rd interval is indicated by the sensor. This is modelled by two event outputs “Sens_ON” and “Sens_OFF” associated with firing of transitions t_8 and t_9 or t_{11}, respectively. The sensor goes off when either the front edge of the pallet moves backward to the interval 2, or when the back edge of the pallet leaves the belt in forward direction (and the pallet completely disappears from the belt).

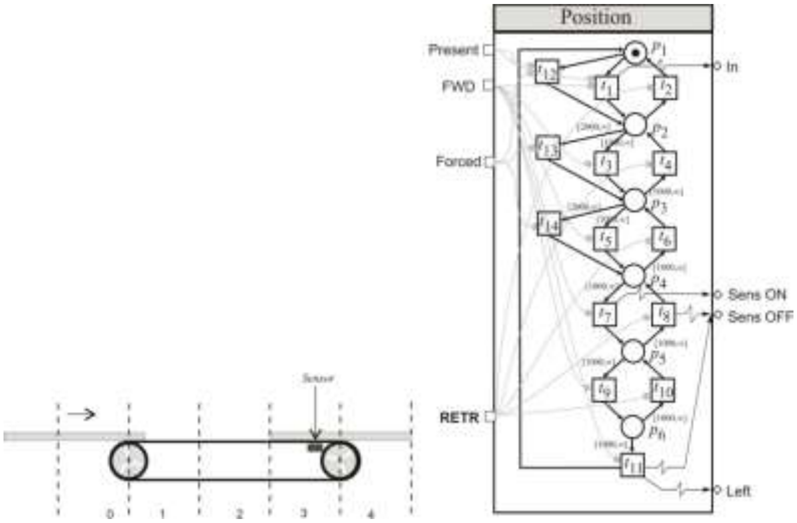


Figure 119. Model of the position of the pallet on the conveyor discretized on 3 intervals.

This model can represent the state of the pallet on the belt with better precision. However, it has other limitations. In particular, let us consider how the alternative kinds of movement are modelled. A place indicating a position (e.g. p3 indicating interval 2) has several outgoing arcs (p3-t4, p3-t5 and p3-t14) marked with non zero time delays ([1000,∞], [1000,∞], [2000,∞]). Transitions that are targets of these arcs have condition input signals that represent alternative control signals (RETR, FWD, Forced). Any of the transitions will fire when it is enabled by marking, conditions and time. It is important that all these conditions are mutually orthogonal (alternative) and they never change values within the minimum delay of the place (1000 time units in our case), otherwise the model will not work as intended.

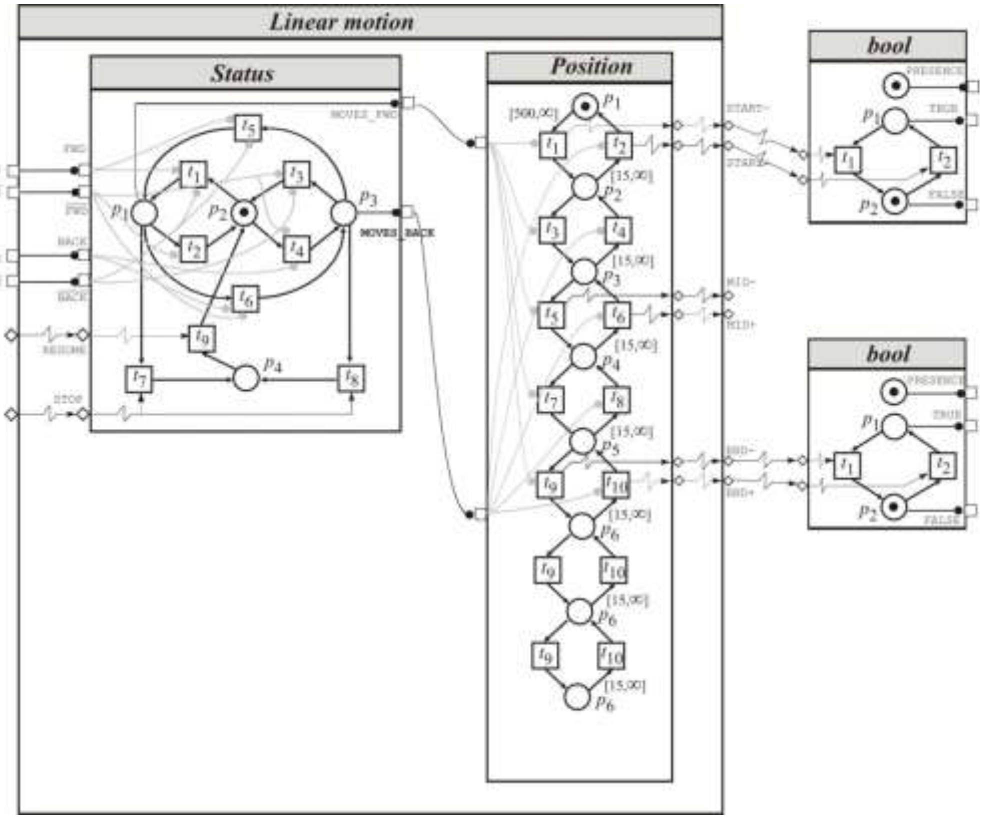


Figure 120. The complete model of the conveyor with sensors

17.4 Boring station

We illustrate the component-based system design and re-design with the help of a simple production cell “BORING STATION” as presented in Figure 121.

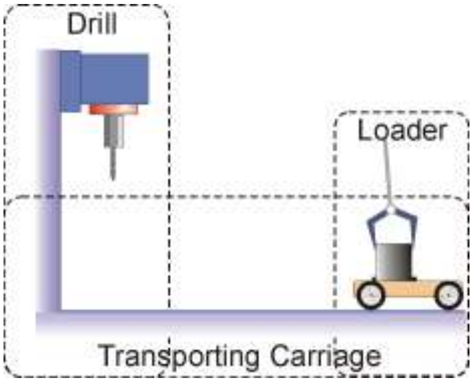


Figure 121. Structure of the production cell: a processing unit (drill), a transportation unit (carriage), and a logistics unit (loader).

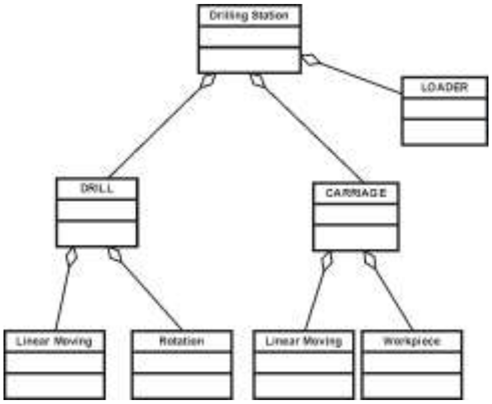
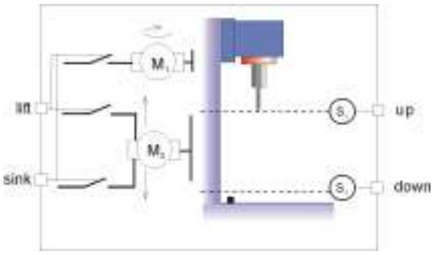


Figure 122. Structure of the boring station represented by UML class diagram.

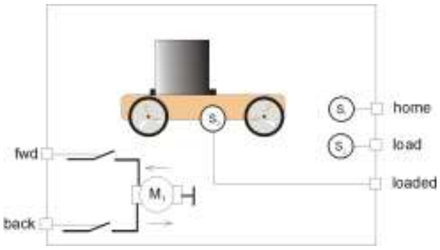
It consists of a boring machine (*drill*) and a *carriage*, which delivers work pieces to the home position of the drill. The loading/unloading of the carriage is performed by the *loader* in the *loading* position that is opposite to the home position. This example allows illustrate various phenomena arising in component-based industrial systems, e.g. concurrent operations in different components, or impacts of reconfigurations, such as substitution of one component by almost functionally equivalent one, having slight differences in interfaces, dynamic properties, etc.

Structural model of objects can be defined by means of UML class diagrams as exemplified in **Figure 122**. The drilling station is represented as an object, composed from 3 components: drill, carriage and loader. In the drill two processes are outlined: vertical linear movement and rotation of the spindle. The car is represented by its horizontal linear movement and by the load status: presence/absence of work piece on it, status of the work piece (blank, drilled). The loader's internal structure is not outlined in this model.

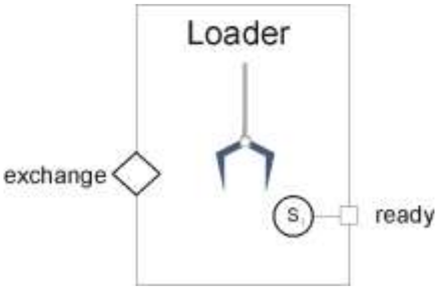
Sample constituent parts of the system are described in the following Table.



Drill: Spin motor M_1 rotates the bore of the drill. The step motor M_2 moves the spindle in vertical direction. The motor is controlled by two Boolean level signals: **LIFT** and **SINK**. These signals connected in parallel to the spin motor: thus the drill rotates always when the step motor moves the spindle. Position of the spindle is detected by two logic sensors: **UP** and **DOWN**.



Carriage: This type of carriage has two actuator signals moving it in two opposite directions. The sensor LOADED detects presence of the work piece on the carriage, and sensors HOME and LOAD detect the home and load positions.



Loader: The loader is independent and autonomous unit not controllable within our application. If loader is in the appropriate state (indicated by the output signal READY) it accepts only one pulse signal “EXCHANGE WORKPIECE” that starts the exchange procedure which consist in approaching of the loader “grip” to the workpiece, lifting, putting the workpiece to the storage, taking new blank workpiece and installing it onto the carriage.

17.4.1 S/E Net model of a Boring Station

Structure of control system of the boring station with centralized control is presented as in Figure 123. Modules representing plant and controller are interconnected via Boolean signals.

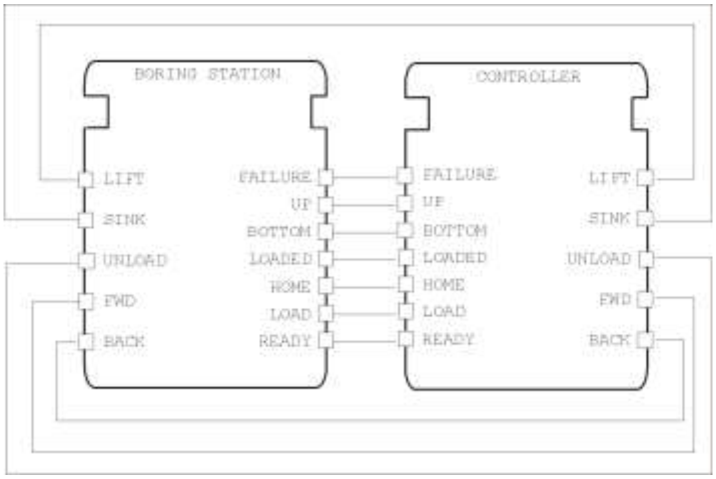


Figure 123. Closed-loop centralized control structure of the boring station.

Beyond the interface abstraction of the controller can be a control program, written in one of languages of IEC61131, or any other way defining outputs as functions of the inputs and internal states (e.g. state chart model, Boolean functions, etc.).

According to the structural description in Figure 122, where 3 constituent units are distinguished, internal structure of the model is presented in Figure 124 as a network of models of the units encapsulated into the module with the same interface as that of the plant in Figure 123.

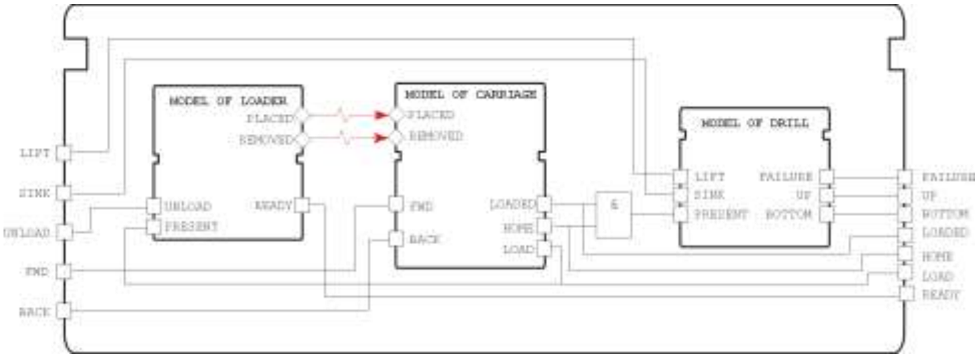


Figure 124. Model representing internal structure of the boring station.

This structure may serve as the basis for description of model with distributed control, as it is shown in Figure 125. The model retains the same structure of signal connections, changing only content of the constituent modules: instead of models of uncontrolled behaviour they represent plant/controller closed-loop models. Input/output interfaces are left to allow manual interaction into process (all control signals are connected to corresponding parts of plant via switches, controlled by the parameter signal "AUTO/MANUAL").

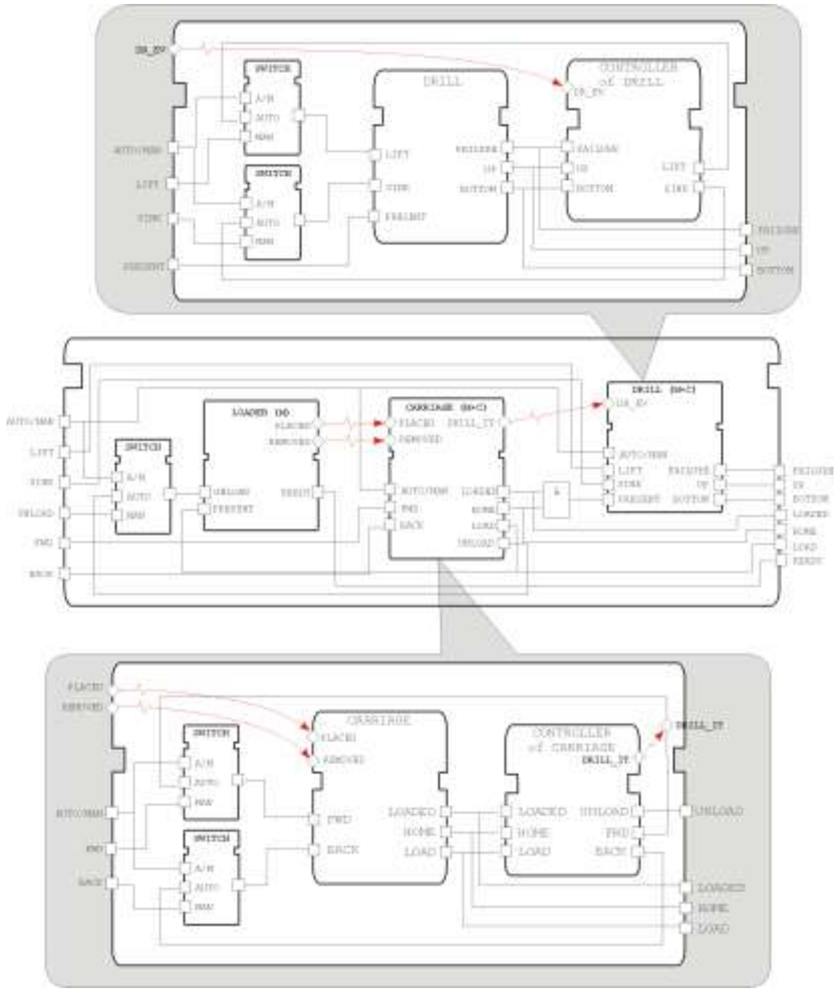


Figure 125. Modular model of the drilling station where components correspond to units with local control.

17.4.2 Controller

The controller of the Boring station is shown in Figure 126.

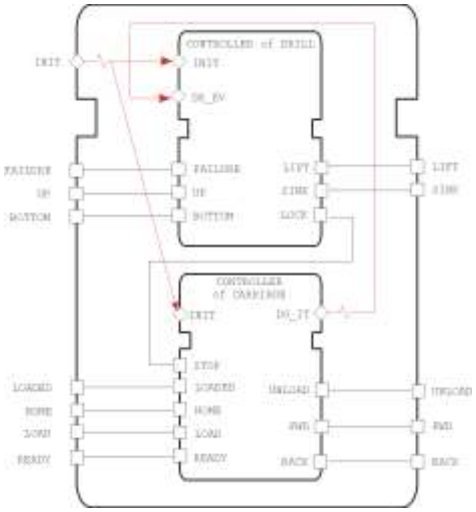
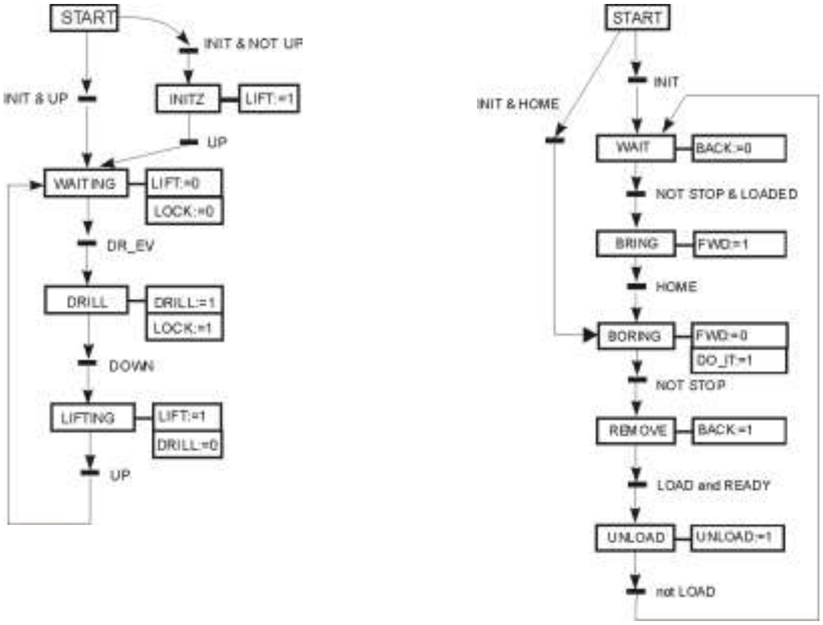


Figure 126. Modular controller of the object Drill/Carriage.

Sequential controllers of Drill and Carriage are presented in Figure 127.



a) b)

Figure 127. Controllers of a) Carriage and b) Drill.

17.5 Model of Drill

The next level of description concerns with structural and dynamic models of single constituent units, in this example CARRIAGE, DRILL and LOADER.

The drill comprises two functionalities: linear movement and rotation. Correspondingly its model can be decomposed onto two parts as shown in Figure 128: linear movement and rotation.

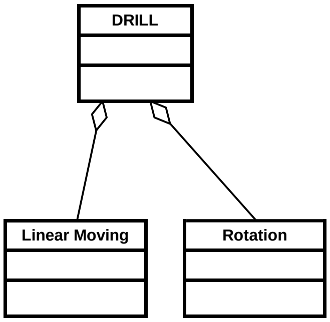


Figure 128. Structure of DRILL.

The two distinct functions are reflected in the structure of the block, representing the DRILL – it contains two blocks for the two mentioned processes, interconnected by signals according to the influence which they have to each other. As it was earlier defined, control signals LIFT and SINK serve also to switch the rotation. This is correspondingly reflected in the model: both signals are also connected to the switching input ON of the MOTOR. In turn, the flag “ROTATES” informs the model of linear movement about rotation status of the spindle. The need for this will be explained below.

Both processes (linear, rotation) are observed through corresponding sensors. This is represented in Figure 129: the model of linear movement is decomposed onto the dynamic model producing numeric coordinate, and two sensors, generating the values of sensors given the coordinate. Note that the FAILURE output of the DRILL is disconnected from the FAILURE output of the block representing LINEAR model, according to the description of the DRILL this information is unobservable by controller.

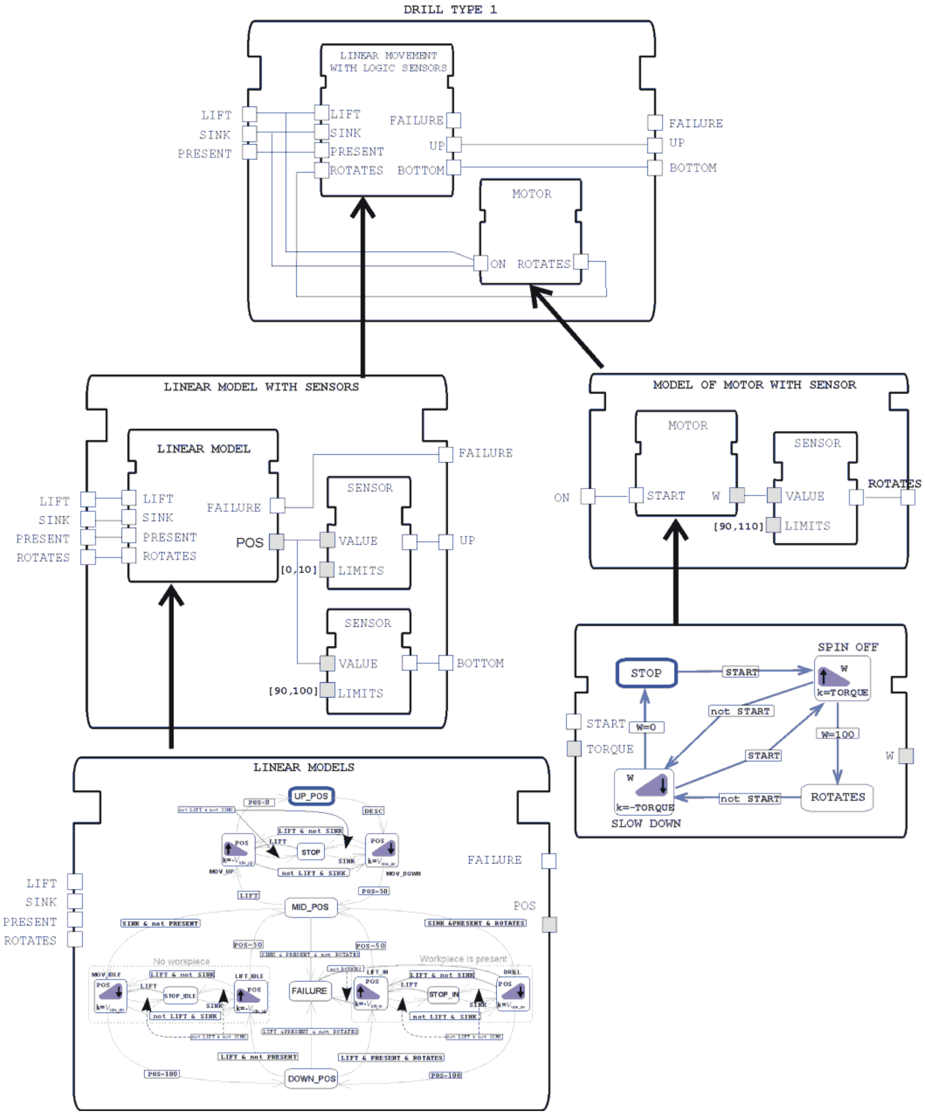


Figure 129. Structural model of DRILL encapsulating models of dynamics in form of state charts.

17.6 Variations

Two possible variations of drill’s type are shown below in order to illustrate their influence on the structure of modelling.

17.6.1 Enhanced Drill

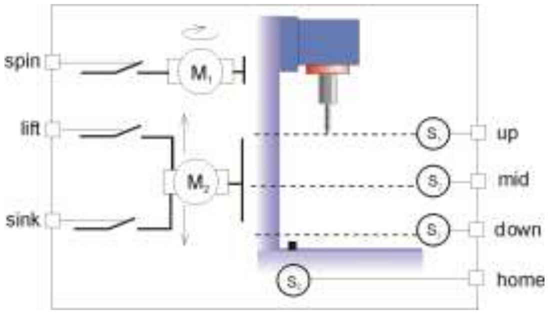


Figure 130. Enhanced drill has middle position sensor and separate control of the spin.

As opposed to the previously considered DRILL, in the enhanced drill the spin motor has a separate control signal SPIN independent from the step movement control signals. This potentially allows early spinning off the bore during the approach. Additional sensor of middle position is provided in order to optimize timing of the processing: this position corresponds to the upper edge of the work piece, spindle can approach the work piece while the carriage is approaching the home position. The sensor is ON whenever the spindle is below this position. Sensor HOME indicates presence of the carriage in the home position.

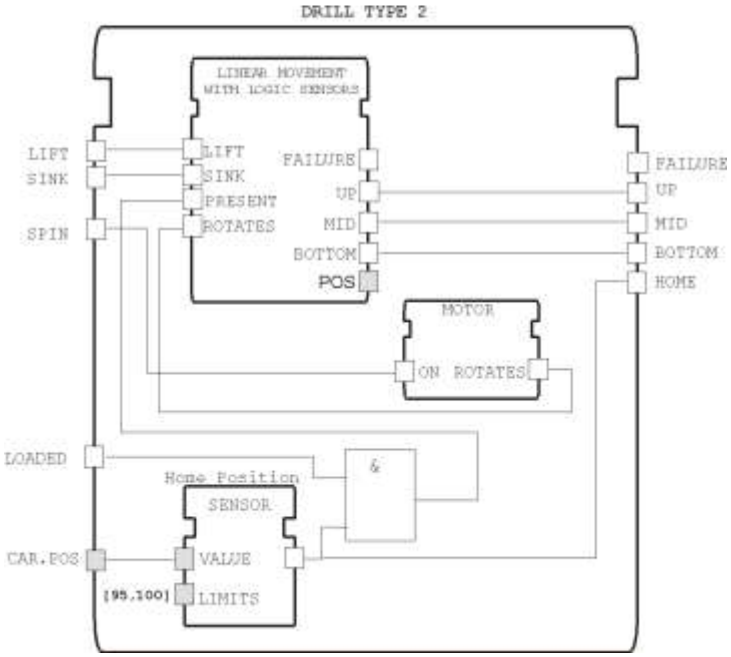


Figure 131. NCES model of the enhanced drill.

To produce the HOME signal the model needs the numeric coordinate of the carriage which can be provided by the model of carriage. This input is not used if the block represents interface to real drill.

17.6.2 Advanced Drill

In addition to the drill of type 2, a couple of extra logic sensors are provided: ROTATION that goes ON when the bore spins off fast enough to start drilling, and the light-screen sensor that issues the FAULT signal indicating presence of foreign bodies in the vicinity of the drill.

These add-ons allow for smarter control of the drill in order to save power, improve timing and secure better safety.

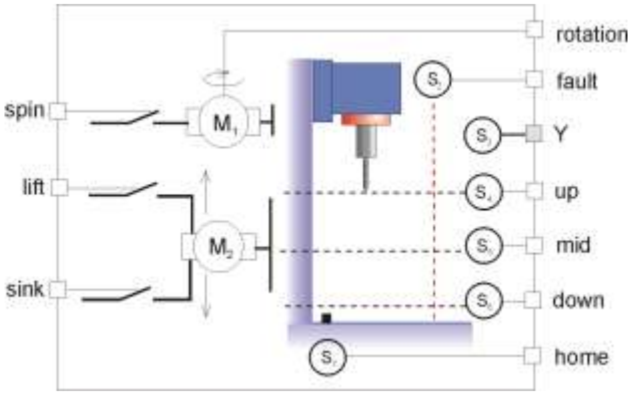


Figure 132 Advanced drill

Besides, the analog sensor Y provides the integer value in the interval from 0 to 100, indicating location of the spindle on the vertical axis (value 0 corresponds to the UP position, value 100 – to the DOWN position).

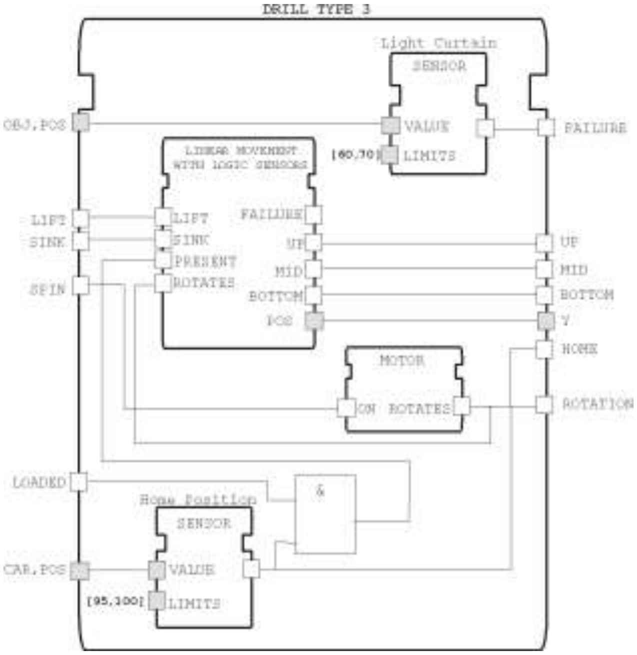


Figure 133. NCES model of the advanced drill.

Both models share the common model of linearly moving part of the drill with 3 position sensors. The model is presented in the following figure.

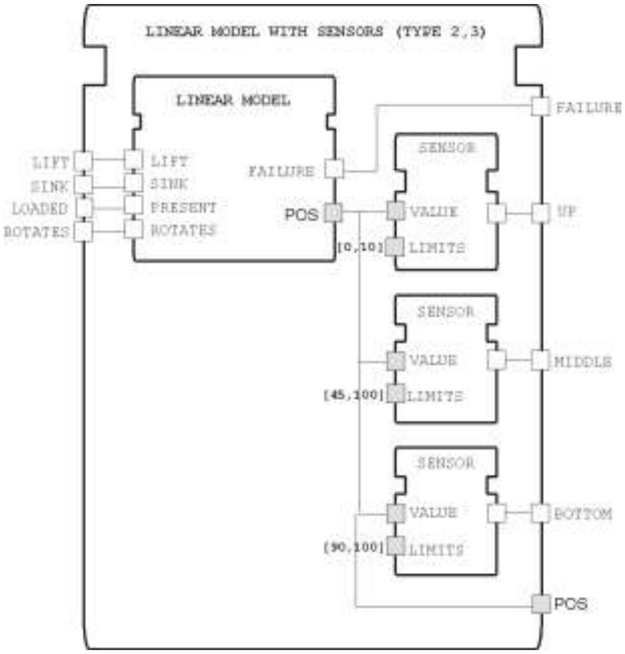


Figure 134. The model of linear movement with sensors.

This model refers to the same model of drill's dynamic as the one used in the similar model of the drill of type 1 (**Figure 129**). To be useful for both simulation and analysis purposes, the models shall exhibit dynamics of the corresponding object, as well as definition of states exhibiting erroneous behaviour. Thus, incorrect control signals shall drive the model to the erroneous states as that would happen with the real object. This approach will be illustrated below on examples.

17.7 Modelling dynamic and logic of processes

Even primitive dynamic processes such as linear movement of drill's head cannot be efficiently described by pure mathematical equations in presence of logic control signals. The model has to be hybrid, i.e. include both mathematical definition of the coordinate change, along with the logic model of state switching. For this purpose we develop Modular Dynamic State Charts (MDSC). These are customized UML State Charts having an explicit input/output interface of S/E systems and a customized set of state shapes, corresponding to particular dynamic properties of parameters.

The first part can be represented as the following S/E module:

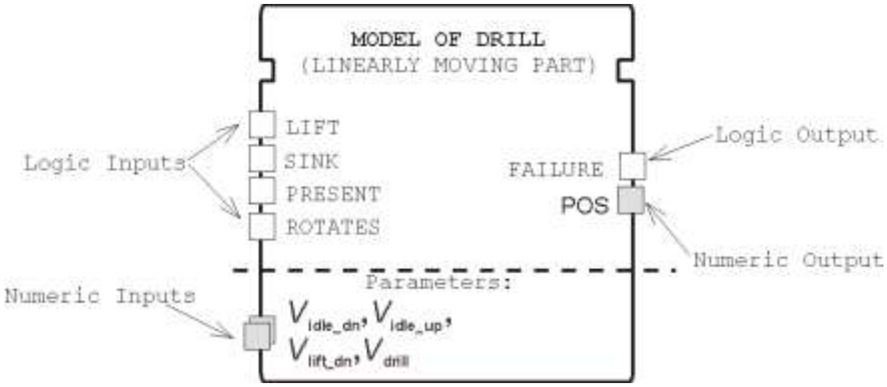


Figure 135. Parameterized module – model of DRILL.

The module's interface reflects the fact, that usually the control actions are transmitted to the plant by level Boolean signals (LIFT, SINK in this model). The model also needs some information about the external environment: the condition PRESENT stands for the workpiece status, and ROTATES informs model about the spinning status of the spindle. Depending on the values of these two conditions, the linear moving may have different speed in the lower part of the moving interval, e.g.: the drill cannot move down if the spindle does not rotate, but the workpiece is present. On the other hand, if no workpiece is present, rotation of the spindle does not influence vertical movement.

The model delivers two output values: numeric output POS represents vertical coordinate of the drill’s head, and logic value FAILURE is integral condition representing all sorts of incorrect or failure situations.

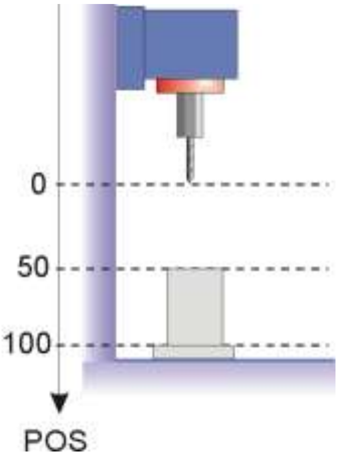


Figure 136. Drill's vertical movement axis.

As shown in **Figure 136** the coordinate variation limits are 0 and 100. The higher edge of the workpiece is assumed to have vertical coordinate 50. The state chart model of the linear progress of the drill is shown in Figure 137.

The dynamic state chart is built from states (rectangular shapes) and state transitions (arcs) marked with Boolean conditions. In the chart in the there are two types of states: fixed position states UP_POS (POS=0), MID_POS (POS=50), DOWN_POS (POS=100) and dynamic states with linear change of parameter POS as $POS=POS_{old}+kdt$, where the coefficient k is speed of moving, dt – time increment, POS_{old} is previously calculated value of the parameter.

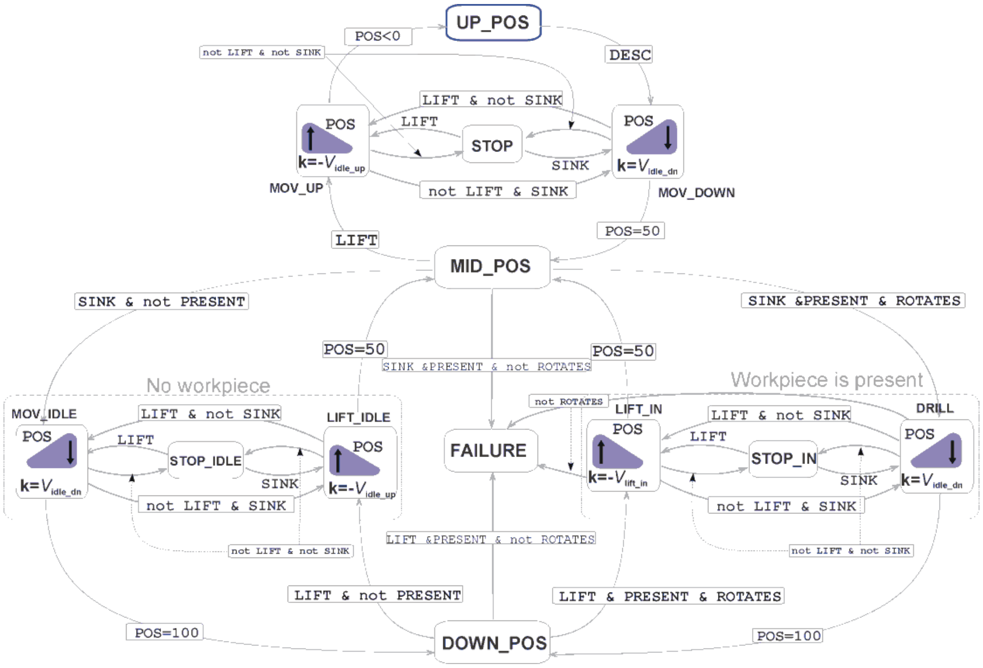


Figure 137. Modular Dynamic State Chart model of the linearly moving part of the drill.

The model describes the uncontrolled behaviour as follows. The spindle moves free in the upper part of the axis, no matter whether the workpiece present or not. When the middle position is reached and the control signal SINK remains ON, the spindle continues its moving downwards. Should the workpiece be in the home position, and the bore spins, then normal drilling goes on. If the drill does not rotate, then it just hits the blank workpiece and a failure occurs. If no workpiece is present, then the drill moves down idle, with the speed higher than that of drilling. The same applies to the moving upwards.

Note, that the presented model does not assume presence of the MIDDLE position sensor. It only generates the POS numerical value. Thus the model applies to all types of drills being in consideration.

17.8 Verification model in NCES

The NCES formalism has been especially tuned for the needs of heterogeneous modelling of systems combining synchronous and asynchronous behaviours. The modelling in form of place/transition nets allows efficient handling of distributed state models with concurrent synchronous/asynchronous behaviour.

The modular S/E interface that provides event and data inputs and outputs to the model, makes the models semantically equivalent to Condition/Event Automata introduced by Kowalewski and Chen. It also can be easier converted to the Net Condition/Event Systems. Thus the application schema is proposed, as illustrated in the following figure:

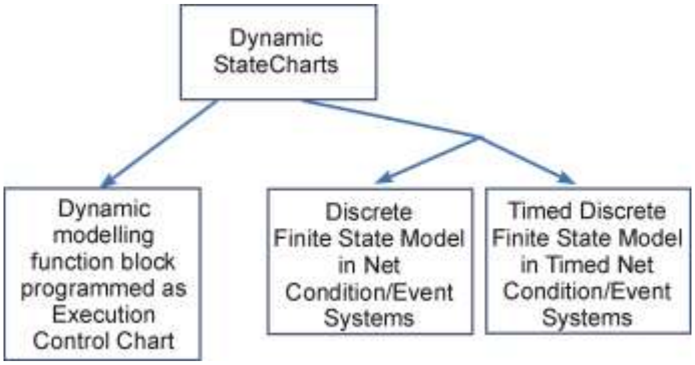

Dynamic StateCharts
Dynamic modelling function block programmed as Execution Control Chart
Discrete Finite State Model in Net Condition/Event Systems
Timed Discrete Finite State Model in Timed Net Condition/Event Systems

Figure 138. Development scenarios.

An initial description of the model is given in the intuitively clear form of Modular Dynamic State Charts. Then the equivalent simulation program can be automatically generated in the form of IEC61499 Execution Control Chart and algorithms to be encapsulated into a function block and further into a component definition as it was shown above.

17.9 Carriage

Model of uncontrolled behaviour of the carriage consists of two relatively independent models of: 1) linear movement and 2) load status. Placement/removal of workpieces onto the carriage is indicated by the corresponding events PLACED, REMOVED and is possible, according to the model, only in the load position of the carriage.

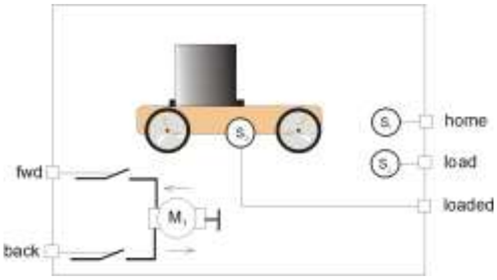


Figure 139. Carriage

Similarly to the model of drill, the model of linear movement consists of the hybrid dynamic model and two blocks representing logic position sensors.

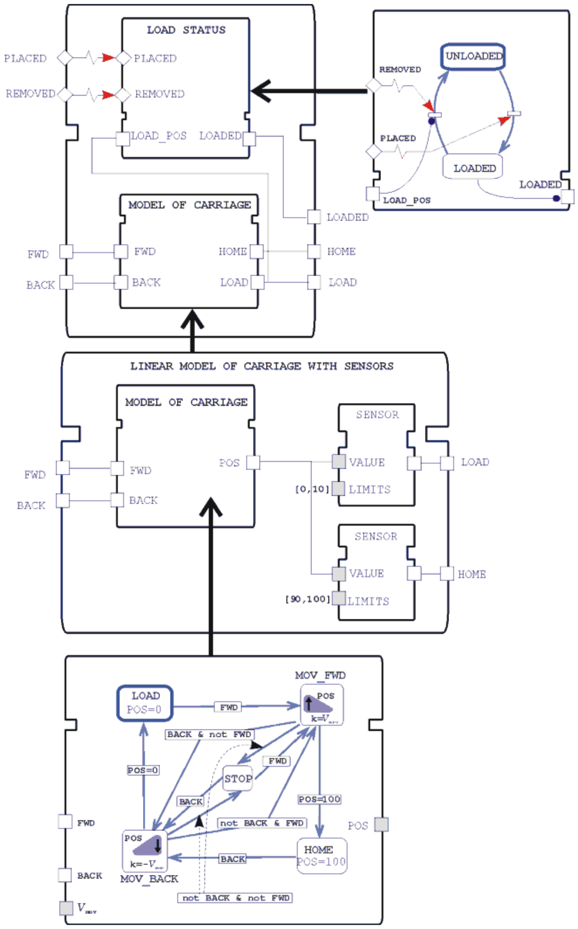


Figure 140. NCES model of the carriage

17.10 Loader

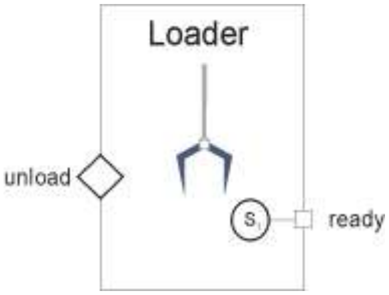


Figure 141 shows the model of overall behaviour of the loader, which provides mapping between input parameters (control signals) and output parameters. No internal structure of the loader is outlined, no information about its controller is available.

Note the differences in interfaces between the left figure representing input/output interface of the loader, and model on the right, which requires also information about presence of the workpiece. When the interconnected model is formed, this information can be provided by the model of object formerly possessing the workpiece.

Output event signals REMOVED, PLACED indicate the events when a workpiece correspondingly is grasped by the loader or released from it.

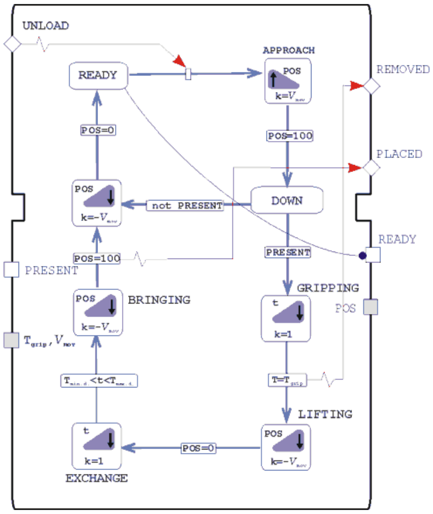


Figure 141. Uncontrolled behaviour of the loader.

17.11 Lifter

The automated lifter (product of Flexlink Automation Oy., FINLAND) as shown in Figure 142 is used in production of electronic components. The lifter can be controlled by two different controllers:

- OMRON PLC programmed in ladder logic and
- Nematron SoftPLC *(Lastra, 2000; Nematron, 2001)* programmed in Visual Flow Chart language.

Though both controllers achieve similar control goals, the internal logic of control algorithms and even the logic of program execution are completely different (cyclically scanned vs. sequential). However, both controllers eventually deal with the same object.

When the closed-loop plant-controller systems are validated, the model of the lifter can be reused over and over again in connection with models of controllers of different types.

The lifter consists of three transporters, one of which is mounted on a vertically moving platform driven by a step motor as schematically represented in Figure 142. The figure also shows sensors (B/S) and actuators (M) of the lifter described as follows.

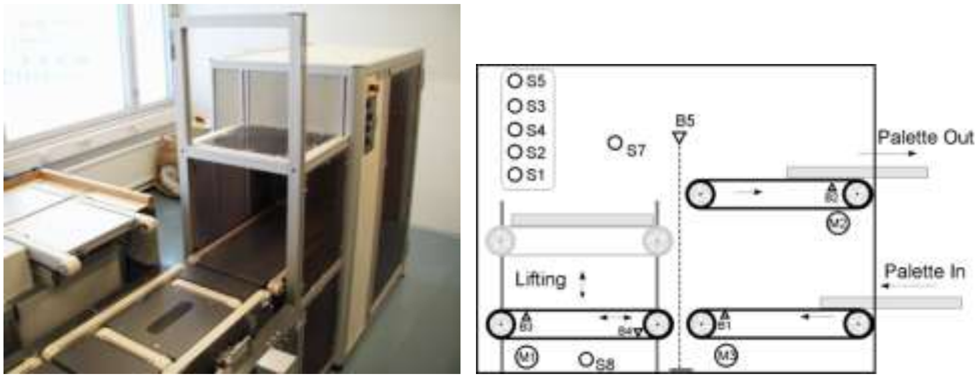


Figure 142. The lifter, its structure and operation sequence.

The lifter is composed of three conveyor elements. The pallet is received from the previous module at the lifter *lower terminal*, which is driven by motor M3 and is equipped with B1 sensor that may detect the presence of the pallet. The pallet may be conveyed from the lower terminal to the *sledge* conveyor that can move vertically between lower and *upper terminal* (or otherwise it is restricted with the two safety switches S7 and S8). The sledge has B3 sensor that detects a pallet and its belt is driven by motor M1. The upper terminal sensor is B2 and the motor denoted by M2. Besides the conveyors and their sensors and actuators, there is also an operator interface with switches (S1 - S5), B5 sensor, which is a safety sensor to detect an obstacle between sledge and terminals. The step motor and the rotary encoder that is used for vertically position the sledge are omitted in Figure 142. The figure does also not show the interface signals (SMEMA) that are used between the lifter and the previous/next module.

Each sensor and actuator has a unique name in mechanical/electrical blueprints and software code. The mechanical and electrical drawings with the general description of functionality form the logical point to start plant modelling.

The structure of the model type “Lifter” can be represented by means of UML class diagrams as shown in Figure 143.

The definition literally says that the object “Lifter” consists of 4 elements. The loading and unloading one-directional conveyors are identical but turned in opposite directions. The corresponding models are of type **Conveyor**. The vertically moving platform (an object of type **StepMotor**) has a moving belt that moves pallets in both directions (modelled as an object of type **Conveyor2D**).

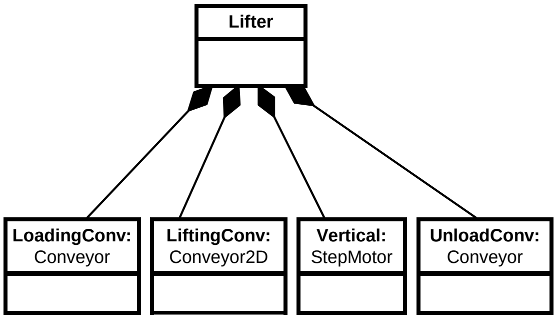


Figure 143. Definition of the model type (class) “Lifter” by means of UML class diagrams.

Note that the model in Figure 143 does not define an interface of the lifter, nor dependencies between its constituent parts. These dependencies can be reflected in modular models by event and condition connections between the corresponding modules as exemplified in Figure 144.

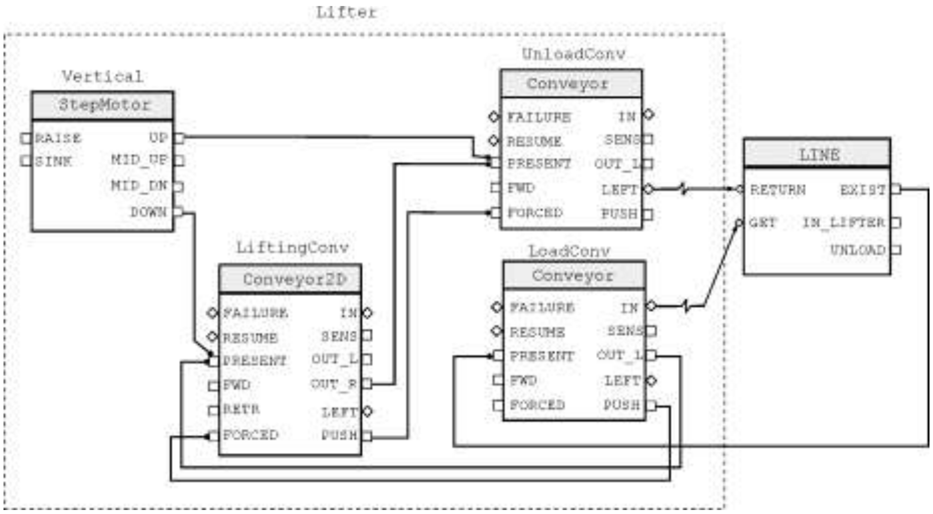


Figure 144. A model of Lifter represented as a network of NCES modules.

17.12 Examples of specifications of Lifter's behaviour[2]

Specifications are the formally expressed properties of system's behaviour. Table 2 provides some examples of the formalization of specification of system requirements for the Lifter object, whose description is provided in Chapter 15 (sections 15.3 and 15.11).

The first column in the table gives a logical proposition formula and expresses the mapping of the local labels in the NCES modules to the global S/E Net label (given in parenthesises). The second column provides a description of formula arguments given in the first column. The last column contains the *case* description in a natural language. The long names of arguments in the formulae are due to the hierarchy of the modules and the places coming at the lowest level. For instance, "Controller._M1DIVIDECW.p4" is interpreted as place p4 at M1DIVIDECW module (represents the motor of the sledge run clockwise) in the controller module.

[2] This Section uses the material developed by Andrei Lobov from Tampere University of Technology. It was published in our common paper (*Hanisch et al, 2006*),

Table 2. Examples of specifications

	#	**Formula**	**Description of arguments**	**Case description**
MAY LEAD TOSAFETY	1.	Controller._M1DIVIDECW.p4 (p213) AND Controller. _M1DIVIDECCW.p4 (p249)	P4 in _M1DIVIDECW – sledge motor running to download the pallet p4 in _M1DIVIDECCW – sledge motor running to unload the pallet	The processes of sledge loading and unloading should never happen at the same time that in terms of the models means that both places should never hold tokens simultaneously
	2.	Plant.Vertical.Vertical.Position.p2 (p472) AND (Controller._M1DIVIDECW.p4 OR Controller. _M1DIVIDECCW.p4)	Plant.Vertical.Vertical. Position.p2 – lift is in the middle of its journey.	It never should happen that a lifter is in the middle of its vertical move while sledge is loading or unloading.
	3.	Plant.Vertical.Vertical.Position.p3 (p473) AND Controller. _M1DIVIDECW.p4	Plant.Vertical.Vertical. Position.p3– lift is in the upper position	It never should happen that the lift is in the upper position while the sledge is loading
	4.	Plant.Vertical.Vertical.Position.p1 (p471) AND Controller. _M1DIVIDECCW.p4	Plant.Vertical.Vertical. Position.p1– lift is in the lower terminal position	It never should happen that the lift is in the lower terminal position and the sledge is unloading
CHECKPOINTS	5.	Plant.Low_Conv.Sensor.p2 (p503) Plant.Sledge_Conv.Sensor. p2 (p515) Plant.Up_Conv.Sensor.p2 (p486)	Plant.Low_Conv.Sensor.p2 – Low lifter terminal sensor detects a pallet Plant.Sledge_Conv.Sensor.p2 – Sledge sensor detects a pallet Plant.Up_Conv.Sensor. p2 – Upper lifter terminal sensor detects a pallet	All three states have to be found in the model. The pallet has visited all the conveyors.

The requirements specifications given in

Table 2 were simplified from the real ones for illustrative purposes. Let's consider verification of each formula in more details:

1. 'p213 AND p249' when evaluated in Visual Verifier fulfils in no states. That means the controller never turns the motor of the sledge to run into both directions, which could have lead to the physical damage of the motor.

2. Checking of the second formula "*p472 AND (p213 OR p249)*" gives a set of states for which it is true. Thus there are states where the lifter is in the middle of its vertical move and the sledge motor is running in either one direction or another. The next step in analysis is to identify the reason. The first step is to define in what direction the motor is running (loading – p213, unloading – p249) or both. This is can be identified by two separate formulae: "**p472 AND p213**" and "**p472 AND p249**". Checking both formulae has given the result that only "**p472 AND p213**" is TRUE and has a number of states in the reachability graph. Furthermore, the direction of motion may be defined by "**p205 AND p472 AND p213**", where p205 represents upward motion. The formula is false if there is p221 (downward motion) instead of p205. The direction of the vertical and conveyor belt motion is therefore identified. Now, we know that the motor of the sledge runs at the lower terminal level to retrieve the pallet from the terminal. The next step is to find out where the pallet is located. There are several possibilities:

    a. Plant.Sledge_Conv.Sensor.p2 (p515) – on the sledge;
    b. Plant.Sledge_Conv.Position.p10 (p529) – the pallet is not on the sledge;
    c. Plant.Low_Conv.Sensor.p2 (p503) – the pallet is at the lower terminal;

   We checked the formula "**p472 AND p213 AND p205 AND p515**" and it is fulfilled in no states. This means that the sensor does not detect the pallet. Checking the "**p205 AND p472 AND p213 AND p529**" formula finds the same states in the reachability graph as the initial formula "**p205 AND p472 AND p213**", which means there is no pallet on the sledge at all. Formula "**p205 AND p472 AND p213 AND p529 AND p503**" again fulfils in the same states.

   This situation may be interpreted as follows: *The pallet is stuck at the lower terminal and has not been transmitted to the sledge. After some timeout for receiving the pallet and without getting it, the lifter starts upward motion while the sledge conveyor continues running.*

   Further investigation shows that the low terminal motor is running as well (Plant.Low_Conv.Status.p1 (p499)), but the pallet remains at the lower terminal (the formula "**p205 AND p472 AND p213 AND p499 AND p510 AND p503**" gives the same states in the reachability graph). Furthermore, this situation is not found for the sledge in the upper terminal position (Plant.Vertical.Vertical.p3 (p473): checking of the following formula "**p205 AND p473 AND p213**" gives no states found).

This error reveals an uncontrollable object's property when nothing can be done by controller to resolve it. If this situation were to occur with the real lifter the operating personnel would be required to resolve it and reset the lifter.

However, the reason why the controller commands to move up while the loading operation of the sledge is not complete is interesting, but not the primary goal. The primary goal is the conclusion that *there were no states found in which the pallet has been successfully loaded onto the sledge (p515), the lifter is half way (p472) driving up (p205) and the sledge motor is running (p213) ("**p515 AND p472 AND p205 AND p213**" checking gives no states found).*

This situation is one of such type which would not be detected by the common testing.

4. The next formula represents the situation when the sledge motor is running to download the pallet while the lifter is at the upper terminal level where the pallet should be unloaded "**p473 AND p213**". Checking this simple request gives no states found in reachability graph. It is therefore possible to conclude that the sledge conveyor belt will not run to the wrong direction at the upper terminal level.
5. Next formula describes a situation opposite to the previous one: '**p471 AND p249**'. The sledge conveyor is running to unload the pallet at the lower terminal level. Checking of the formula also returns a false result meaning that no such states exist in the reachability graph.

Places p503, p515 and p486 model TRUE value of the pallet sensors of the low conveyor, sledge conveyor and upper conveyor respectively. Checking if any of these places ever holds a token gives an affirmative answer. In this example, we may highlight one of the advantages in applying CTL. The CTL formula '**E**[**E**[**EF** m(p503)=1 **U EF** m(p515)=1] **U EF** m(p486)=1]' represents the case when a path exists in the reachability graph where first the low terminal sensor detects a pallet, then the sledge terminal sensor detects a pallet and finally the upper terminal sensor detects a pallet. This is an example of a checkpoint rule, proving which we may conclude that the *lifter is able to transfer a pallet through it*.

18 Multi-level model design pattern

18.1 Hierarchies in models

Hierarchical representation of behavior has been addressed in Harel Statecharts and in hierarchical Petri nets.

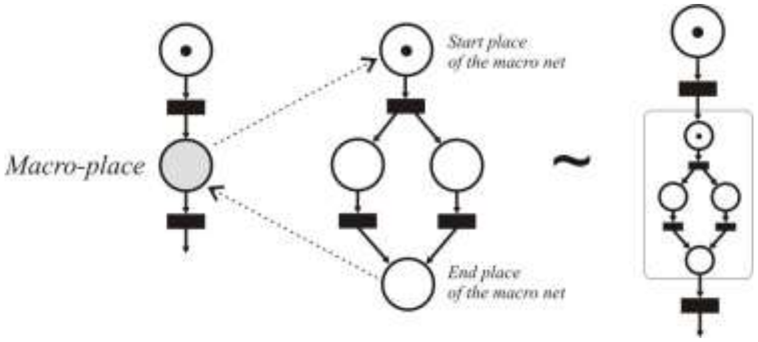


Figure 145 Petri net with hierarchical states and equivalent semantics

In NCES similar behavior can be modeled as follows:

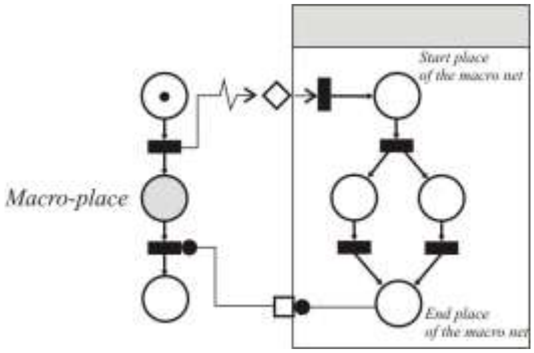


Figure 146 Implementation of the 'hierarchy' in NCES.

18.2 Motivation

A piece of equipment with complex internal dynamic behavior can be seen from the outside as a simple one with respect to the material flow on the factory shop level. However properties and conditions of its primitive material-flow relevant functionality (take one pallet – give it away) may strongly depend on the internal behavior.

The modeling of such units asks to take in account this particular feature and represent the multiple facets of the behavior as necessary.

The general idea of the suggested modeling approach is schematically illustrated in Figure 147. Both internal and external models of an equipment unit are represented by NCES modules.

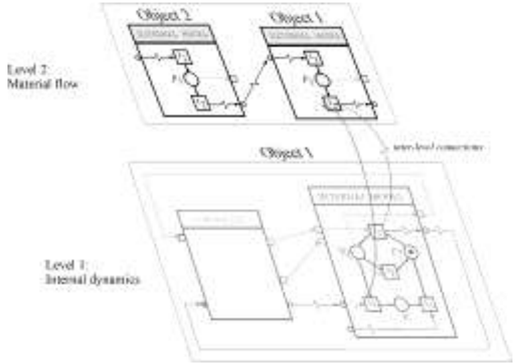


Figure 147. The idea of the two-level pattern of modelling.

Mutual influence between internal and external levels of modeling is defined by means of event and condition arcs that may connect places and transitions of both modules in both directions, i.e. from level 1 to level 2 and from level 2 to level 1. The inputs and outputs of the internal model may be connected in closed-loop with the model of controller, while inputs and outputs of the external model serve for connection with external models of other objects.

Thus, this chapter suggests a specific application-oriented pattern of using NCES.

18.3 Notation of the two-level modules

In the following we are introducing notation which is intended to simplify representation of the hierarchically built multi-domain models. The notation however does not imply any new semantics as compared with NCES, as the mapping from it to the NCES will be introduced.

The two-level modules are structures that destined to encapsulate models of both internal dynamic behavior of an object along with its externally observed behavior of interest.

Figure 148 shows the corresponding graphical notation of a module for the two-level formalism. The module consists of the head containing the external model, the body, containing the internal module, and event and data interconnections between them. Both external and internal sections may have event and data inputs and outputs, and can be further specified as networks of modules.

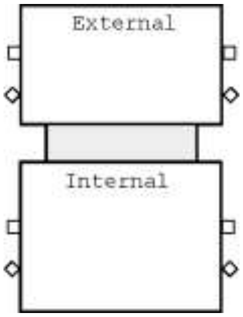


Figure 148. A two-level module.

A two-level module with an empty EXTERNAL part makes a usual NCES module (single level). The EXTERNAL part of a double-level module can be specified via a network of single-level modules. The INTERNAL part can be specified by a network of two-level NCES modules. This is illustrated in Figure 149.

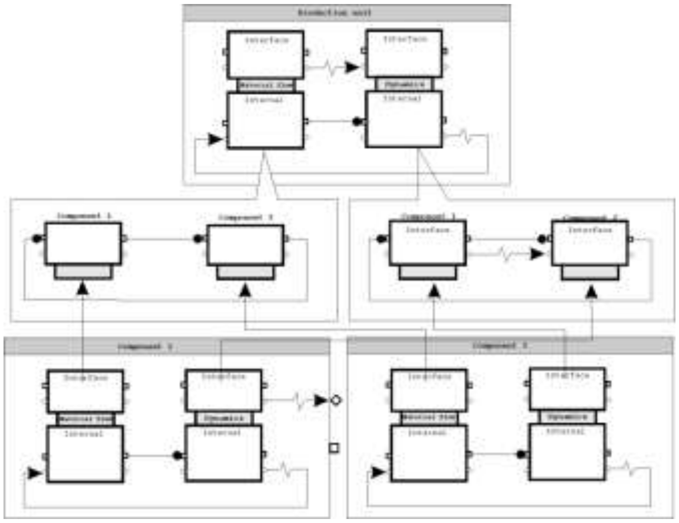


Figure 149. Composition of two-level models into a composite model.

This example shows that the suggested encapsulation pattern can be used for defining of hierarchical models of an arbitrary complexity.

The multi-domain model is a network of interconnected modules whose inputs and outputs are divided on two groups: one for interconnection with other facets ("internal" IOs) and the other for interfacing their domain counterparts in other models (interface IOs).

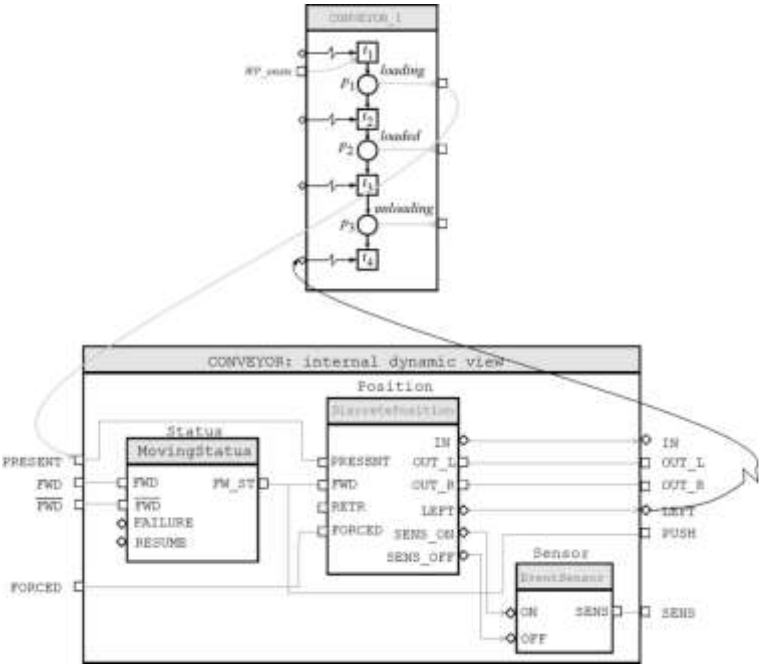


Figure 150. A two-level model of a conveyor belt.

We illustrate the application of the two-level modeling pattern when the Lifter is a part of a more complex automated machinery system that consists of several storage buffers and transportation units, as shown in the example in Figure 151.

Goals of the modeling are:

- simulation and observation of the material flow relevant properties, e.g. average loading of buffers, absence of deadlocks, etc.
- checking correctness of the distributed control

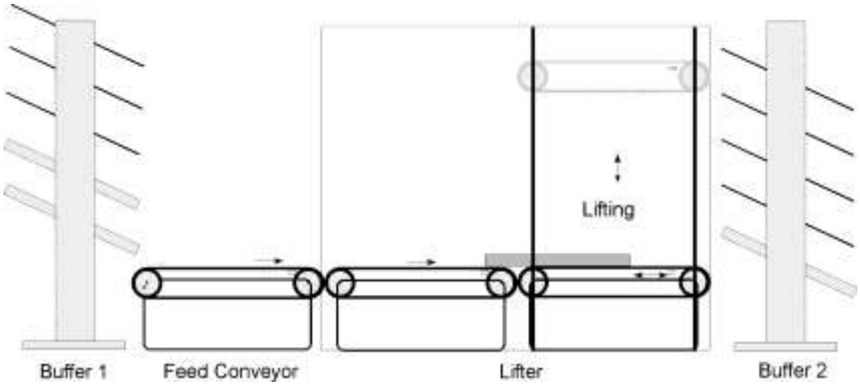


Figure 151. An automated storage and transportation system built from modular machines.

The process going on in the object can be seen from several perspectives:

Level 1: 4 pallets in the shop

Level 2: 2 pallets in buffer 1, 1 pallet in the lifter, 1 pallet in buffer 2

Level 3: In lifter: the pallet is being transferred from the entry conveyor to the lifting conveyor

Level 4: position of the pallet is 4/5 on the lifting conveyor

19 Specifications using Timing Diagrams

Control engineers are not familiar with the languages commonly used for formal specification, such as temporal logic. Therefore the engineers would benefit from user-friendlier means of specifying the desired or forbidden behaviour of a model.

Inspired by the timing diagram specification languages developed in the domain of digital systems design (e.g. by K. Fisler [45], N. Amla et al., [46], R. Schlör [47]), a graphical language for describing the dependency of interface signal changes was proposed in [49].

In this Chapter we proceed with the issues that are specific for application of timing diagrams for specification and verification purposes of some classes of industrial automation systems. Visualising the behaviour of discrete-state models using diagrams is quite helpful. In Figure 152 one sees the waveform diagram representing values of some model parameters along a certain path in the reachability graph (the model was introduced earlier in **Figure 19**).

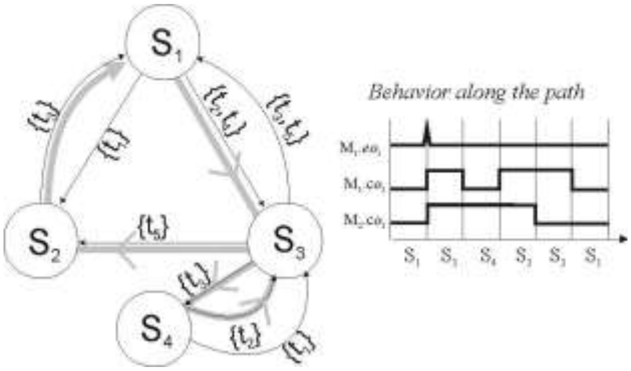


Figure 152. Reachability graph describing the complete behaviour of the model from Figure 19 and timing diagram in one of the trajectories.

This Chapter suggests two procedures for translation of visual specifications that differ slightly depending on whether the verified module has inputs.

19.1 Timing Diagrams for specification

The idea of using timing diagrams for specification is to draw a specification graphically and then ask the model checker the question: *If the inputs behave like it is shown in the input diagram, would the outputs behave like in the output diagram?*

However, a single timing diagram describes only a single scenario. Sometimes it is desirable to define a class of input scenarios with certain properties and then check if certain output patterns are observed among all or any trajectories in the reachability graph. The idea is illustrated in **Figure 153**. The diagram consists of two parts: the upper (if) part presents the “input” part of guaranteed signals and the lower part is the “conjecture” to prove. In this example there is a conditional restriction added between the rising edge of $M_1.co_1$ (event e_2) and the falling edge of $M_2.co_1$ (event e_3). The restriction says that e_3 occurs after e_2. Note that the signal $M_1.co_1$ belongs to both parts. In the “input” part it is specified by a single waveform change that is simultaneous with the event $M_1.eo_1$. The waveform of the same signal in the “output” diagram is more complicated.

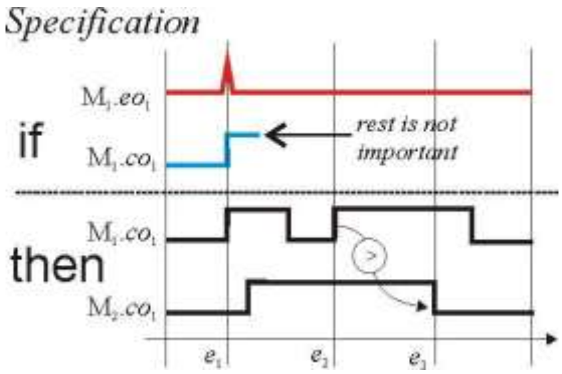


Figure 153. Timing diagram specification

Comparing the “then” part of the specification with the timing diagram of real behaviour in Figure 152 one can see that the specification holds in the given path. The problem is to implement such a check automatically using model checkers.

19.1.1 Definitions

The use of Timing Diagrams (TD) as a method of formal specification requires formal definition of its graphical notation and its semantics.

Diagrams are represented by sequences of signals' value changes. Given the subsets $E \subseteq E^{in} \cup E^{out}$ and $C \subseteq C^{in} \cup C^{out}$, a specification for a signal set $A = E \cup C$ is described as a tuple $S = (A, f, g)$, where $f = f_e \cup f_c$ defines sequences of specification values. The mapping $f_e : E \to \Sigma_e^*$ ($\Sigma_e = \{noevent, maybe, always\}$) specifies sequences for event inputs and outputs, while $f_c : C \to \Sigma_c^*$ with $\Sigma_c = \{zero, any, stable, one\}$ defines values for condition signals.

The partial function $g : f(A) \times \mathrm{N} \times f(A) \times \mathrm{N} \to (>,=,\neq)$ assigns an ordering operator (precedence, simultaneity or non-simultaneity) between signal changes from different signals in such a way that $g(a_i,m,a_j,n)$ indicates an ordering restriction between the m-th signal change of a_i and the n-th signal change of a_j. We assume the signals value changes at the beginning of the diagram to be simultaneous across all signals. If the ordering operator for a pair of changes of different signals is not defined, the horizontal position of the changes won't imply any implicit ordering.

Consider the example in Figure 154.

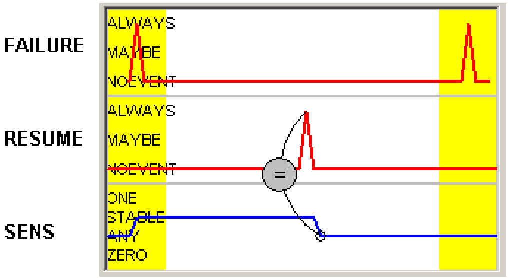


Figure 154. Specification including two event inputs, one condition output and a simultaneity operator.

The semantics of the diagram is as follows: when the set of levels specified at the beginning of the diagram is achieved, it is required that the sequence of changes of the signals does not violate the partial ordering specified in the diagram, until a final state is reached.

19.1.2 *Specified Signals*

In order to describe specifications of NCES models, TDs must provide different representations for event and condition signals. Thus, we define the following possibilities of specification:

- in the case of a condition signal, the specification might assume four possible levels: *zero*, corresponding to a logical zero; *any*, representing the situation where the signal may take any logical value; *stable*, which also means undefined, however assuming that the signal remains at a single level; or *one*, corresponding to the logical one;

- event signals are specified in two possible levels: *no event*, in the case where the occurrence of the event is forbidden, and *maybe*, meaning that the event might occur. It is also possible to specify an obligatory occurrence of the event *signal (always)*, but in this case only as a single impulse, because of the instantaneous nature of an event signal.

We define a *diagram event* as: any level change specified at a condition signal; a level change from *no event* to *maybe* or vice-versa, at an event-signal; or a specification of an obligatory occurrence of an event (*always* peak at an event signal).

19.1.3 Event Ordering in Different Signals

If a *partial ordering semantics is* assumed, no prior ordering of events on different signals is implicit. In other words, although each signal presents an ordering of its events, two events of different signals may occur at any sequence, except when special operators explicitly define their sequence. On the other hand, it is also possible to assume that the ordering of all events is defined through their position at the visual description. In this case, we are talking about a *strict* or *sequential ordering*.

Although more intuitive, adopting a sequential ordering would limit the representational capabilities of a diagram. Therefore, we adopt a partial ordering semantics for the TD language. In this case, the same TD represents a set of possible behaviours of the system, each one represented by a different event chain on the modelled system. Each chain is called *scenario,* and the set of scenarios defined by the diagram is named *diagram language*.

In Figure 155 (a) we observe the specification of two signals $\mathbf{s_1}$ and $\mathbf{s_2}$. If we have adopted as our convention a sequential ordering semantics, only one scenario would compose the diagram language: $\mathbf{s_2^+s_1^-s_2^-}$. As the temporal dependence among events from different signals is not predefined (assumed partial ordering semantics) the same figure represents a TD with the following scenarios: $\mathbf{(s_2^+,s_1^-)s_2^-}$; $\mathbf{s_2^+(s_1^-,s_2^-)}$; $\mathbf{s_1^-s_2^+s_2^-}$ and $\mathbf{s_2^+s_2^-s_1^-}$. Figure 155(b) indicates the timing diagram that, based on the adopted semantics, accepts as its only scenario $\mathbf{s_2^+s_1^-s_2^-}$, by introducing operators that indicate the obligatory ordering among events from different signals. The meaning of these operators will be stated in the next section.

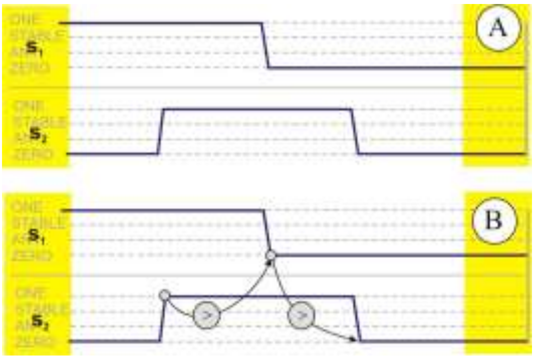


Figure 155. Temporally independent signals (a) and event ordering (b).

In order to constrain the ordering of two events from different signals, we define the following precedence operators:

≠ : events are not allowed to occur simultaneously;
= : events must be simultaneous;
> : event from the first signal must occur prior to the event from the second signal.

19.1.4 Specification of Finite Behaviour

The TD represents a finite behaviour that must be satisfied by the model. The satisfaction of a TD is evaluated from the moment when all specified signals are in their initial levels and some of them execute an initial transition, as indicated at the beginning of the diagram. The verification process ends when all signals achieve their final state, indicated in the end of the diagram. The initial part of the diagram, denominated *precondition*, corresponds to a condition, whose satisfaction by the model indicates that we must start comparing the model's behaviour with the remaining part of the TD. The comparison ends up when the final part of the diagram, called *postcondition*, is reached. Both pre- and postcondition are highlighted at the diagram (**Figure 156**).

When a TD specifies a finite behaviour, different interpretations are possible:

Existence of a scenario (from the diagram language): here we require that at least one of the specified scenarios will occur at the model. In other words, there is a path at the state tree of the model, where the precondition is satisfied and the behaviour of the model does not contradict the specification.

Existence of all scenarios: the existence of each scenario must be tested inside the state space of the model.

Generality of a single scenario: here a single scenario from the set of scenarios specified at the diagram, must be present in every path, indicating a situation that has to occur in the future, regardless of which path is taken by the model.

Generality of the diagram's language: the behaviour specified by the diagram will eventually occur, no matter which scenario, in each path from the reachability graph of the model. Note that, in this case, the existence of a path with no occurrence of the precondition would already be a counter-example.

Satisfaction of a single scenario: every satisfaction of the precondition must be followed by the satisfaction of the same scenario, among those that are possible according to the specification. This corresponds to an assume-guarantee clause, where the precondition plays the role of an assumption that, when fulfilled, guarantees the occurrence of a given sequence of events.

Satisfaction of the diagram: the specified behavior must not be contradicted, which means that every occurrence of the precondition at the model leads to a behaviour that is accepted by the diagram language. As a particular case, a model that presents no occurrence of a given precondition satisfies every specification starting with this precondition. The following topics will be based on this interpretation of the TD.

19.1.5 Specification of infinite behaviour

The timing diagram could also correspond to a specification to be satisfied from the time when the precondition occurs, without the need to specify a postcondition. In this case, the final state specified at the diagram would correspond to a restriction that must not be violated.

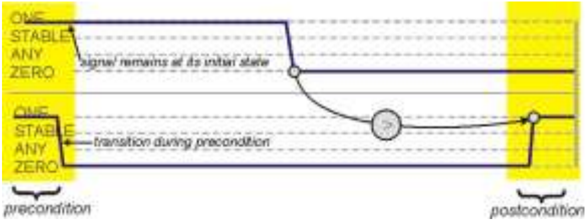

ONE
STABLE
ANY
ZERO
signal remains at its initial state
ONE
STABLE
ANY
ZERO
transition during precondition
>
precondition
postcondition

Figure 156. Pre- and postcondition.

The absence of a specification for the precondition could indicate that the initial state of the model should comply with the levels specified at the beginning of the diagram. Although these two approaches also present a practical appeal, the absence of postcondition or precondition will not be issued in the work, as a matter of simplicity.

In order to allow the translation of the timing diagram into a formal model, some requirements have to be done in respect to the events presented in each signal. Diagrams satisfying the requirements are said to be *feasible*.

19.2 NCES Model of Timing Diagrams

When verifying autonomous NCES models without inputs, each signal specification is translated into a NCES supervisor module comprising two basic submodules: an **event generator** creates sequences of transitions, one for each change of level specified for the signal. Each transition stimulates, through an event arc, the corresponding event input of a **signal generator**, which causes the output of the signal generator to recreate the signal according to the input stimulated. Ordering operators are translated into special places and transitions that create interdependency of event generators.

The verified module is then connected through event arcs to the event generators of the corresponding signals, in such a way that every change of signal in the first is reported to the latter. Along with the translation of the specification into NCES modules, a set of automatically generated temporal-logic statements is created. The composite module is then model-checked against these statements to verify if each transition at the supervisor always fires whenever the corresponding transition at the verified module is fired.

The graphical specification also provides automatic test possibilities for input/output behaviour or non-autonomous NCES modules. In this case, the NCES supervisor modules that describe input signals are used for generating the specified sequences of input signal changes, while the output signals are again verified as described before. The components of the NCES model of the timing diagram are detailed in the following sections.

19.2.1 Event Generator

The main part of the NCES model for the specification is called *event generator* and consists of a set of parallel *processes* (sequences of transitions and places), started simultaneously by the firing of a transition denoted $\mathbf{t_{start}}$. Each process is responsible for reproducing the behavior specified for one signal. Events on the signals are translated into transitions at the processes.

For each signal **i**, there is a place $\mathbf{p_{notstart,i}}$ which is a preplace of $\mathbf{t_{start}}$ and postplace of the last transition of the corresponding process. The transition $\mathbf{t_{start}}$ indicates the beginning of the timing diagram. The situation where the diagram language is not being executed corresponds to the marking $\mathbf{p_{notstart,i}}$=1 for every signal **i**.

In the case that at least a signal **j** has the marking $\mathbf{p_{notstart,j}}$=0, the marking $\mathbf{p_{notstart,i}}$=1 for a signal **i** indicates that this signal has already achieved the last level specified at the diagram.

The precedence relationships among events of different signals are mapped to special interconnections among the corresponding processes, as shall be detailed in the following section.

19.2.2 Signal Generation Module

For each specified signal, we create a signal generator module which reproduces, at its output, the possible values for the signal, according to the level specification stimulated at its input. Each event on the timing diagram (modelled by the firing of a transition at the event generator) stimulates, by an event arc, the corresponding change at the signal generator, which guarantees that the NCES module, resulting from the combination of the event generator with the signal generators, will reproduce at its output the diagram language. The idea is illustrated in **Figure 157**. A signal generator module is assigned to each condition signal included in the specification. The module hasfour event inputs, corresponding to the four possible specification levels, and two condition outputs, indicating the two possible values assumed by the condition signal (*zero* or *one*).

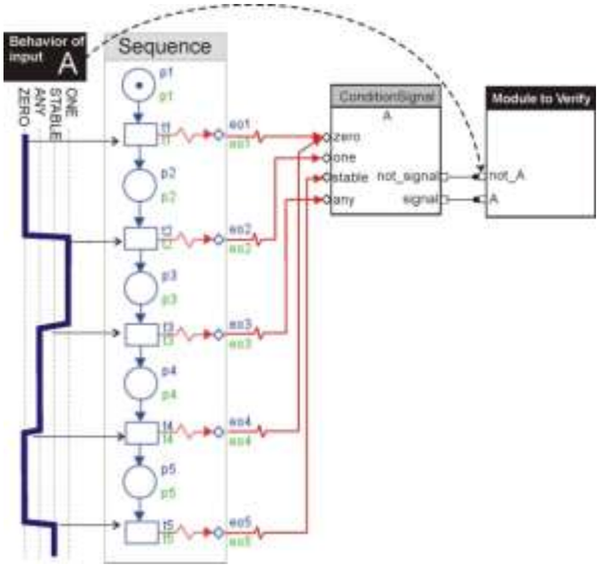


Figure 157. Translation of a single specification for a condition output, and linking to the verified model.

Figure 158 shows the structure of a signal generator for a condition signal.

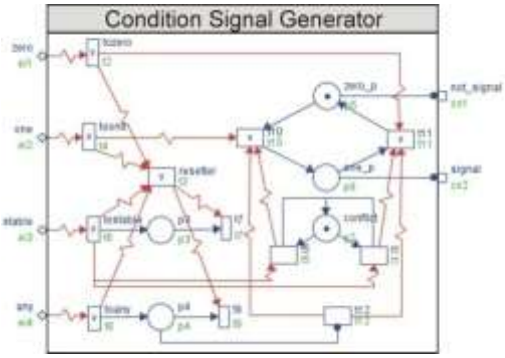


Figure 158. Generator of condition signals.

The transitions **tozero**, **toone**, **tostable** and **toany** receive event arcs, respectively, from the **zero**, **one**, **stable** and **any** event inputs.

Firing one of these transition means that the corresponding signal has changed its specification level to, respectively, *zero*, *any*, *stable* or *one* – in other words, a diagram event has occurred. The condition outputs **not_signal** and **signal** are linked to the internal places **zero_p** and **one_p**. The remaining transitions and places implement the desired non-deterministic behaviour - after the firing of **tostable** and **toany**, the marking of places **zero_p** and **one_p** should be non-deterministic, and may change randomly in the latter case, until another input event is stimulated.

Figure 159 presents the internal structure of a signal generator for an event signal.

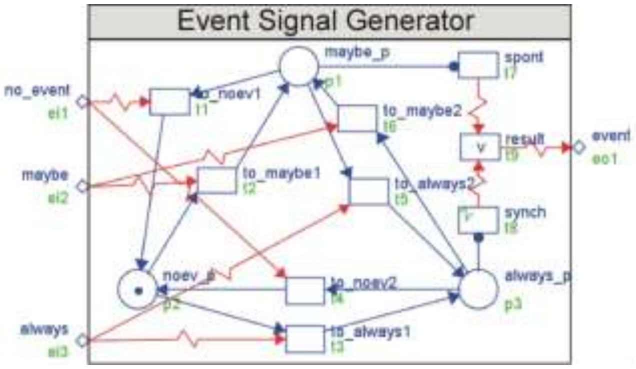


Figure 159. Generator of event signals.

Event signals are represented by modules with three event inputs, corresponding to the three possible specification values, and an event output, whose firing corresponds to the generation of the event. Internally, this generation corresponds to the firing of the **result** transition.

The transitions **to_noev#** (1 and 2), **to_maybe#** (1 and 2) and **to_always#** (1 and 2) are fired by stimulating the **no_event**, **maybe** and **always** inputs respectively. Every diagram event leads to the firing of at least one of these transitions – actually, an *always* peak at the specification, followed by the specification of a new level, implies that both the **result** and the transition that leads to the new level specification (**to_noev#** or **to_maybe#**) will be enforced to fire.

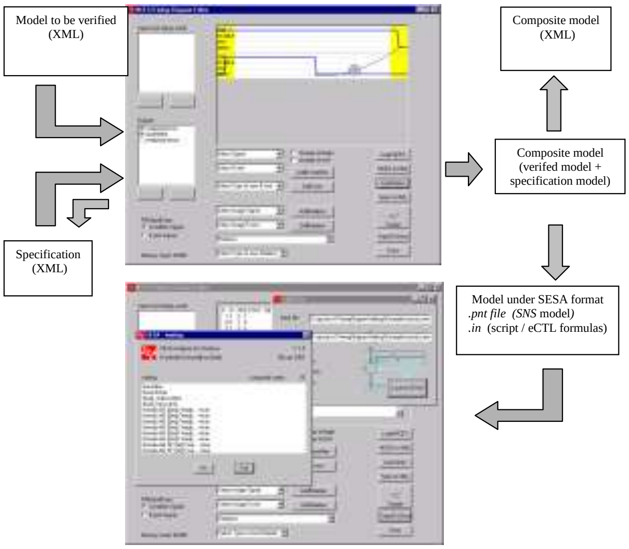

Model to be verified (XML)
Specification (XML)
Composite model (XML)
Composite model (verifed model + specification model)
Model under SESA format
.pnt file (SNS model)
.in (script / eCTL formulas)

Figure 160. User interface of the TDE tool and file formats adopted for data storage.

19.3 Program Implementation

The Timing Diagram Editor (TDE) is an application developed with the aims of providing the following functionalities:

- create, edit, save and load specifications of function blocks whose internal logic is specified by means of a NCES. These specifications are generated and visualized graphically as timing diagrams, while each signal at the timing diagram may be of one of the following types: event signals and condition signals; the signal levels allowed for each type of signals that were presented above.
- translate the combination of a function block and the behaviour specified for it into a composite finite state model (NCES) and temporal propositions written in the eCTL [51] format, in such a way that the composite model, and consequently the original function block, can be verified formally with the aid of the SESA tool [52]. If all the generated eCTL propositions evaluate to true

with regard to the composite model, we conclude that the behaviour of the original model satisfies the specification.

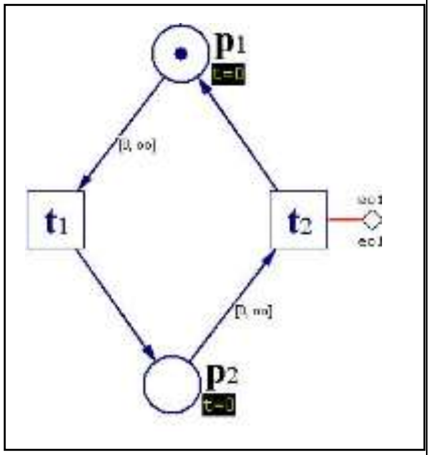
- The TDE tool uses XML as a storage format for both timing diagrams and NCES models and converts them to the input formats of the SESA model checker as illustrated in Figure 160.

Annex 1: XML format of Condition/Event Nets

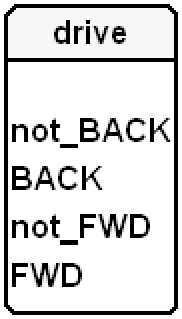
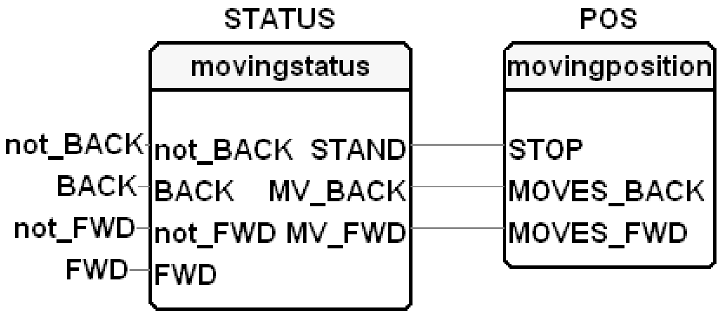
Example of a basic module made in TNCES editor

XML

```
<!DOCTYPE NetConditionEventSystem>
<?TNCES-Editor Version="1.06.06 (eps)"?>
<FBType X="65" Y="251" Num="0" LocNum="0" Name="spont_eo" Comment="" Width="30.0" Height="35.0" >
   <InterfaceList>
      <EventOutputs>
         <Event X="193" Y="140" Num="1" LocNum="1" Name="eo1" Comment="_"/>
      </EventOutputs>
   </InterfaceList>
   <SNS LeftPageBorder="70.0" RightPageBorder="770.0">
      <place X="52" Y="157" Diameter="6" Num="1" LocNum="1" Name="p1" Mark="1" Clock="0" Capacity="1" Comment="_"/>
      <place X="51" Y="123" Diameter="6" Num="2" LocNum="2" Name="p2" Mark="0" Clock="0" Capacity="1" Comment="_"/>
      <trans X="39" Y="140" Width="6" Height="6" Num="1" LocNum="1" Name="t1" Type="AND" TransInscription="_" SwitchMode="s" Comment="_"/>
      <trans X="64" Y="140" Width="6" Height="6" Num="2" LocNum="2" Name="t2" Type="AND" TransInscription="_" SwitchMode="s" Comment="_"/>
      <arc StartPoint="p1" EndPoint="t1" ArcWeight="1" TimeValue="" Comment="_">
         <Point Num="1" X="52" Y="157"/>
         <Point Num="2" X="39" Y="140"/>
      </arc>
      <arc StartPoint="t1" EndPoint="p2" ArcWeight="1" TimeValue="" Comment="_">
         <Point Num="1" X="39" Y="140"/>
         <Point Num="2" X="51" Y="123"/>
      </arc>
      <arc StartPoint="p2" EndPoint="t2" ArcWeight="1" TimeValue="" Comment="_">
         <Point Num="1" X="51" Y="123"/>
         <Point Num="2" X="64" Y="140"/>
      </arc>
      <arc StartPoint="t2" EndPoint="p1" ArcWeight="1" TimeValue="" Comment="_">
         <Point Num="1" X="64" Y="140"/>
         <Point Num="2" X="52" Y="157"/>
      </arc>
      <evarc StartPoint="t2" EndPoint="eo1" Comment="_" EventPos="1">
         <Point Num="1" X="64" Y="140"/>
         <Point Num="2" X="193" Y="140"/>
      </evarc>
   </SNS>
</FBType>
```



XML of a "composite" NCES block

Interface	Content
	 STATUS POS movingstatus movingposition not_BACK BACK not_FWD FWD not_BACK BACK not_FWD FWD STAND MV_BACK MV_FWD STOP MOVES_BACK MOVES_FWD

```
<FBType Name="drive" Comment="Composite Function Block" >
  <InterfaceList>
    <InputVars>
      <VarDeclaration Name="not_BACK" Type="BOOL" />
      <VarDeclaration Name="BACK" Type="BOOL" />
      <VarDeclaration Name="not_FWD" Type="BOOL" />
      <VarDeclaration Name="FWD" Type="BOOL" />
    </InputVars>
  </InterfaceList>
  <FBNetwork >
    <FB Name="STATUS" Type="movingstatus" x="447.0588" y="241.1765" />
    <FB Name="POS" Type="movingposition" x="1294.1177" y="241.1765" />
    <DataConnections>
      <Connection Source="BACK" Destination="STATUS.BACK" dx1="317.6471" />
      <Connection Source="FWD" Destination="STATUS.FWD" dx1="335.2941" />
      <Connection Source="not_BACK" Destination="STATUS.not_BACK" dx1="252.9412" />
      <Connection Source="not_FWD" Destination="STATUS.not_FWD" dx1="270.5882" />
      <Connection Source="STATUS.STAND" Destination="POS.STOP" dx1="129.4118" />
      <Connection Source="STATUS.MV_BACK" Destination="POS.MOVES_BACK" dx1="205.8824" />
      <Connection Source="STATUS.MV_FWD" Destination="POS.MOVES_FWD" dx1="158.8235" />
    </DataConnections>
  </FBNetwork>
</FBType>
```

Annex 2: More formal definition of Condition/Event Nets

19.4 NCES definition

NCES is a place-transition net formally represented by a tuple:

$$NCES = (P, T, F, CN, EN, C^{in}, E^{in}, C^{out}, E^{out}, B_c, B_e, C_s, D_t, m_0)$$

Where:

P is a non-empty finite set of places,

T is a non-empty finite set of transitions, disjoint with P,

F is a subset of $(P \times T) \cup (T \times P)$ - the set of flow arcs.

CN is the set of condition arcs $CN \subseteq (P \times T)$.

EN is the set of event arcs $EN \subseteq (T \times T)$.

C$^{\text{in}}$ is the set of condition inputs.

E$^{\text{in}}$ are the event inputs set,

C$^{\text{out}}$ and *E*$^{\text{out}}$ are conditions and events outputs.

B_c is the set of NCES module condition inputs arcs $B_c \subseteq (C^{in} \times T)$,

B_e is the set of event input arcs $B_e \subseteq (E^{in} \times T)$.

Cs is the set of condition output arcs $Cs \subseteq (P \times E^{out})$,

Dt is the set of event output arcs $Dt \subseteq (T \times E^{out})$, and

m$_0$: $P \to \{0,1\}$ is the initial marking.

19.5 C/E Net definition

19.5.1 Set theoretical definition

Timed C/E Net = (P,T,F,V,B,W,S,M,m0,eft,lft)

where

P is a non-empty finite set of *places*;

T is a non-empty finite set of *transitions* disjoint with *P*;

F is the set of *flow arcs*, where $F \subseteq (P \times T) \cup (T \times P)$;

V maps a weight to every flow arc and $V : F \rightarrow \mathbb{N}$;

B is the set of *condition arcs*, which carry condition signals and $B \subseteq P \times T$;

W maps a weight to every condition arc and $W : B \rightarrow \mathbb{N}$;

S is the set of irreflexive *event arcs*, which convey event signals and $S \subseteq T \times T$;

M maps a event-processing mode (AND or OR) to every transition, $M : T \rightarrow \{\boxed{\wedge}, \boxed{\vee}\}$;

m_0: $P \rightarrow \mathbb{N}_0$ is the initial marking of *SNS*, where for each place $p \in P$, there are $n_p \in \mathbb{N}_0$ tokens;

eft maps the *earliest firing time* to every pre-arc $[p, t] \in F$, *eft*: $F \cap (P \times T) \rightarrow \mathbb{N}_0$; and, *lft* maps the *latest firing time* to every pre-arc $[p, t] \in F$, *lft*: $F \cap (P \times T) \rightarrow \mathbb{N}_0 \cup \{\omega\}$, where $\omega \in \mathbb{N}$ and $0 \leq eft(p, t) \leq lft(p, t) \leq \omega$. The interval $[eft(p, t), lft(p, t)]$ is called the *permeability interval*.

19.5.2 State of C/EN model

C/EN places bear integer clocks whose values are denoted as u: $P \rightarrow \mathbb{N}_0$, where for each place $p \in P$, the clock reading in the place is denoted as $u_p \in \mathbb{N}_0$;

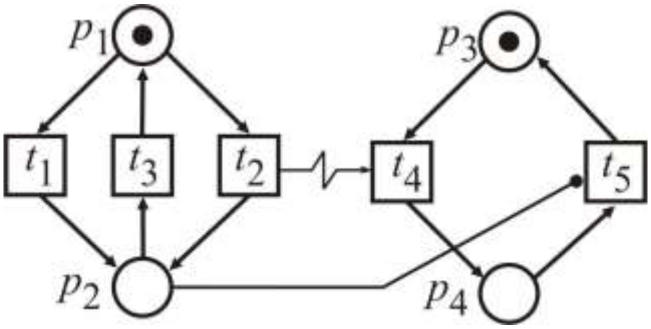


Figure 161. C/E Net

All clocks have zero value at the initial state of the model. The clock of a place resets to zero anytime marking of the place changes.

A *state* in timed *C/EN* is defined as a pair $z=[m, u]$, where m is a marking of P and u is the P-vector of the clock positions and $u(p) > 0 \rightarrow m(p) > 0$.

A state of C/E net model is determined by a) m – vector of marking of its places, i.e. allocation of tokens across the places; and b) u – vector of clock values:

Evolution of a C/E net consists in changing its states. A state change (also called *state transition*) can consist in changing net’s marking, or changing values of clocks (elapsing of time).

In every state there could be some *enabled* net transitions. If there are no enabled transitions then the clocks count (increment they value by 1) in all marked places and the C/E net transitions to a new state. Otherwise, i.e. if there are some enabled transitions, then it is said that one or several enabled transitions *fire* that leads to the change of marking as explained by the firing rules. The set of simultaneously firing transitions is called *step*. In a given state there could be several different steps ready to fire, meaning that a state of C/E net can have several successor states.

19.5.3 Firing rules

Let *St* denote the set of incoming event arcs of transition *t*: $St \coloneqq \{t' \mid [t', t] \in S\}$. If *St* is empty, which indicates that no incoming event arc is associated with transition *t*, then *t* is spontaneous, otherwise it is forced. Firing of a forced transition is caused by firing of some other transition connected to it by an event arc. Both are included in the same step, i.e. fire simultaneously. Enabled spontaneous transitions can fire regardless of other transitions.

For example, the transition *t4* in Figure 161 is forced and other transitions are spontaneous. Accordingly, the transition set *T* in can be subdivided on two disjoint sets: $T = Spont \cup Forc$, where

- *Spont* is the set of all spontaneous transitions of the *C/EN*, and
- *Forc* denotes the set of all forced transitions of the *C/EN*.

For any transition *t*, there can be three kinds of markings: the marking on incoming flow arc t^-, the marking on outgoing flow arc t^+, and the marking on incoming condition arc $\hat{t}$, defined as follows:

$$t^-(p) := \begin{cases} V(p,t), & \text{if } [p,t] \in F \\ 0, & \text{else} \end{cases}$$

$$t^+(p) := \begin{cases} V(t,p), & \text{if } [t,p] \in F \\ 0, & \text{else} \end{cases}$$

$$\hat{t}(p) := \begin{cases} W(p,t), & \text{if } [p,t] \in B \\ 0, & \text{else} \end{cases}$$

For any subset $s \subseteq T$, the marking s^- and s^+ denote the sum of markings t^- and t^+ respectively, and $\hat{s}$ represents the union of markings $\hat{t}$ for $t \subseteq s$.

The firing of a spontaneous transition is determined by the three factors listed below:

1. *Token concession*: A transition is said to have a *token concession* or is *token-enabled* when all the flow arcs from its pre-places are enabled. More specifically, a flow arc is enabled when the token number in its source place is not less than its weight, i.e. $m(p) \geq V(p, t)$. For example, given the marking m, transition t is token-enabled if $t^- \leq m$. Transitions which have no pre-places are always marking-enabled.
2. *Permeability interval*: The permeability interval defines the time constraints applied to the input flow arcs of transitions. A transition t: $\exists\ (p, t) \in F$ is *time-enabled* only when clocks of all its pre-places have a time $u(p)$ within permeability interval of the corresponding place-transition arc: $eft(p, t) \leq u(p) \leq lft(p, t)$.
3. *Incoming condition signals*: A spontaneous transition may have incoming condition arcs. It is considered *condition-enabled* when all the condition signals on its incoming condition arcs are true, i.e. $\hat{t} \leq m$.

A spontaneous transition is eligible to fire only when it is token-enabled, time-enabled, and condition-enabled.

19.5.4 Step and state transitions

C/EN models are *executed in steps,* meaning that for each state transition there is a unique set of concurrently firing transitions $s \subseteq T$. A state is *dead* if no further step is enabled or will be enabled by elapsing time. For non-dead states, the *delay* $D(m,u)$ denotes the minimum amount of elapsed time before a step is enabled.

A step is referred as *executable at the state* $[m, u]$ if all of its constituent transitions fire after $D(m,u)$. The execution of an executable step s at state $[m, u]$ is accomplished by first elapsing $D(m,u)$ amount of time and then firing s.

The new state $[m', u']$ led by the execution of step s is determined by:

$$m' = m - s^- + s^+, \text{ and}$$

$$u'(p) := \begin{cases} u(p) + D(m,u), & \text{if } m(p) > 0 \wedge m'^{(p)} > 0 \wedge p \notin (Fs \cup sF), \\ 0, & \text{else} \end{cases}$$

Subsequent step executions from the initial state construct the *reachability graph* of the *C/EN* model, which illustrates the relationship of all realizable states within the state space. The reachability graph of a timed *C/EN* can be represented as a 3-tuple:

$$RG = (Z, R, z_0),$$

where Z is a finite set of reachable states, R is a finite set of state transitions, and z_0 is the initial state $[m_0, u_0]$.

For any subsequent states $[m_i, u_i]$ and $[m_{i+1}, u_{i+1}] \in Z$, there is a state transition $\tau \in R$, such that $[m_{i+1}, u_{i+1}]$ is reachable from $[m_i, u_i]$ via state transition τ. This state transition is also denoted as $[m,u] \xrightarrow{\tau} [m',u']$.

The step s causing a state transition τ is defined by the mapping $\sigma: R \rightarrow T^*$, i.e. $s = \sigma(\tau)$.

Annex 3. CTL syntax of SESA

CHARACTERS

```
digit = "0123456789" .
letter = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_" .
```

TOKENS

```
number = digit {digit} .

name = '"' {letter | digit} '"' | "'" {letter | digit} "'".

add = '+' .

less = '>>' .

equal = '=' .

unequal = '#' | "" | "!=" .

less_equal = ">>=" .

not = "NOT" | '-' | '!' .

and = "AND" | '&' | '^' .

or = "OR" | 'V' | '|' | 'v' .

impl = "IMPL" | "->" .

equiv = "EQUIV" | "" .

infinity = "oo" | "o" .

true = "TRUE" | 'T' .

false = "FALSE" | 'F' .
```

PRODUCTIONS

```
ctl
  = formula  EOF
formula
```

```
  =
    impl_expr
.

impl_expr
  =
  ( equiv_expr
  [ impl equiv_expr
  ]
  )
.

equiv_expr
  =
  ( or_expr
  { equiv or_expr
  }
  )
.

or_expr
  =
  ( and_expr
  { or and_expr
  }
  )
.

and_expr
  =
  ( factor
  { and factor
  }
  )
.

factor
  =
  ( true
  | false
  | predicate
  | not factor
  | '(' formula  ')'
  | ( 'E'
    | 'A'
    ) [ transition_formula  ]
    ( '[' formula
      ( 'U' [ interval  ]
      | 'B'
      ) formula  ']'
    | 'X' [ interval  ] factor
    | 'F' [ interval  ] factor
    | 'G' factor
    )
  )
.

transition_formula
  =
    transition_impl_expr
.

transition_impl_expr
  =
  ( transition_equiv_expr
  [ impl transition_equiv_expr
  ]
  )
.

transition_equiv_expr
  =
  ( transition_or_expr
  { equiv transition_or_expr
  }
  )
.

transition_or_expr
  =
  ( transition_and_expr
  { or transition_and_expr
  }
  )
.

transition_and_expr
  =
  ( transition_factor
  { and transition_factor
  }
  )
.

transition_factor
  =
  ( true
  | false
  | [ 't' ] node
  | not transition_factor
  | '(' transition_formula  ')'
  )
.
```

```
interval
  = '[' number
    ','
    ( number
    | infinity
    ) ']'
.

predicate
  =
  ( atomic_pred
  | def_pred
  )
.

def_pred
  =
  ( 'P'
    ( number
    | name
    )
  )
.

atomic_pred
  =
  ( atomic_term
  { condition  atomic_term
  }
  )
.

atomic_term
  =
  ( atomic_factor
  { add atomic_factor
  }
  )
.

atomic_factor
  =
  ( variable
  | constant
  )
.

condition
  = less
  | greater
  | equal
  | unequal
  | less_equal
  | greater_equal
.

constant
  =
  ( number
  | infinity
  )
.

variable
  =
  ( marking
  | clock
  )
.

marking >
  = 'm' '(' [ 'p' ] node  ')'
.

clock >
  = 'u' '(' [ 'p' ] node  ')'
.

node >
  = ( number
      | name
    )
  [ '.' (
    number
    | name
  ) ]
.


END ctl.
PRODUCTIONS
```

Annex 4: Command line SESA parameters

Command line options start with "-". Some options can have different names for the same purpose most of them can abbreviated (characters in [] can be omitted). If the <filename> argument to "-command" and "-options" is "-", then the default names (COMMAND.sna and OPTIONS.sna) are used.

If the last argument has no leading "-", then it is interpreted as a name of a .pnt or .cnt file (please include the extension of the file).

Ordering of "-reset", "-command", and "-options" is relevant and resets the influence of previous command line options.

-help	show option summary
-b[lack] -pnt -ptn -toktyp=b[lack]	use only black tokens:
-c[olour] -cnt -cpn -toktyp=c[olour]	use coloured tokens:
-notim[es] -time=no -arctim[ed] -time=yes -time=arcs -tim[ed]/[es]	use arctimes or not:
-nopr[iorities] -pr[iorities]	Use priorities or not:

-nogr[eedy] -gr[eedy]	Use greedy transitions or not:
-nosy[nc] -sy[nc]	Use synchronisation sets or not:
-max[imal] -fmod=m[aximal] -fmod=n[ormal] -s[ingle] -fmod=s[ingle] -red[uced] -fmod=r[educed]	Determine the firing mode:
-stubborn -symmetric -diamond	Apply different reduction techniques:
-names -named -nonames	Write place/transition names in the output or not:
-pre[fix] <prefix>	Prefix for file names for options, commands and session results (set before file name options):
-def[ault] -reset	Reset to default options (same as starting with -nooptions):
-noopt[ions] -opt[ions] <filename>	Ignore OPTIONS.sna or load options from file:
-nocom[mand] -nocmd -com[mand] <filename>	Ignore COMMAND.sna or load commands from file:

-cmd <filename>	
-se[ssion] <filename>	Save session results in file:

References

1. Clarke, E., E.A. Emerson and A.P. Sista.: Automatic verification of finite state concurrent systems using temporal logic., ACM Trans. on Programming Languages and Systems, vol. 8, 1986, pp. 244-263
2. J.S. Ostroff. *Temporal Logic for real-time systems*, Wiley, London, 1989.
3. 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.
4. C.A. Petri: „Kommunikation mit Automaten“, 1962: Dissertation, University of Bonn
5. Harald Störrle: *Models of Software Architecture - Design and Analysis with UML and Petri-Nets*, Books on Demand GmbH, ISBN 3-8311-1330-0
6. Robert-Christoph Riemann: *Modelling of Concurrent Systems: Structural and Semantical Methods in the High Level Petri Net Calculus*, Herbert Utz Verlag, ISBN 3-89675-629-X
7. Kurt Jensen: *Coloured Petri Nets*, Springer Verlag, ISBN 3-540-62867-3
8. James Lyle Peterson: *Petri Net Theory and the Modeling of Systems*, Prentice Hall, ISBN 0136619835
9. Wolfgang Reisig: *A Primer in Petri Net Design*, Springer-Verlag, ISBN 3-540-52044-9
10. Mengchu Zhou, Frank Dicesare: *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer Academic Publishers, ISBN 0792392892
11. Mengchu Zhou: *Modeling, Simulation, & Control of Flexible Manufacturing Systems: A Petri Net Approach*, World Scientific Publishing Company, ISBN 981023029X
12. Jörg Desel and Gabriel Juhás, "What is a Petri Net? -- Informal Answers for the Informed Reader", Hartmut Ehrig et al. (Eds.): Unifying Petri Nets, LNCS 2128, pp. 1-25, 2001.
13. Alur, R., C. Courcoubeitis and D.L. Dill, Model checking for real-times. In Proc 5th Annual IEEE Symposium on Logics in Computer Science, Philadephia, 1990.
14. Aygalinc, P. and J.P. Denat, Validation of functional Grafcet models and performance evaluation of the associated systems using Petri Nets., Automatic Control Production Systems A.P.I.I.,1993, 27,81-93
15. De Loor, P.J. Zaytoon and G. Villerman-Lecolier.: Abstraction and heuristics for the validation Grafcet controlled systems, European Journal of Automation, 1997, 31, 561-580

16. Hanisch, H.-M., Thieme, J., et al (1997). *Modelling of PLC Behaviour by Means of Timed Net Condition/Event Systems*, 6th International Conference on Emerging Technologies and Factory Automation, Los Angeles, USA

17. Heiner M. and Menzel T., Instruction list verification using a Petri net semantics, IEEE International Conference on Systems, Man, and Cybernetics, vol.1, October 1998, pp. 716 -721

18. Bani Younis, M., Frey, G. (2003). *Formalization of Existing PLC programs: A Survey,* In Proc. Of Computing Engineering in Systems Applications, Lille, France

19. Rausch M. and Hanisch H.-M.: *Net condition/event systems with multiple condition outputs*. Symposium on Emerging Technologies and Factory Automation, Paris, France, October 1995, Proc., Vol.1, pp. 592-600, INRA/IEEE

20. H.-M. Hanisch, T. Pannier, D. Peter, S. Roch, and P. Starke: *Modelling and verification of a modular lever crossing controller design*, Automatisierungstechnik, 48, 2000.

21. Analysing Signal-Nets with SESA: http://www.informatik.hu-berlin.de/lehrstuehle/automaten/sesa/, 2004

22. V. Vyatkin, H.-M. Hanisch: *A modelling approach for verification of IEC1499 function blocks using Net Condition/Event Systems*, IEEE conference on Emerging Technologies in Factory Automation (ETFA'99), Proc., pp. 261-270, Barcelona, Spain, September, 1999

23. *IEC61499 - Function Blocks for Industrial Process Measurement and Control Systems,* International Standard, International Electrotechnical Commission, Tech. Comm. 65, Working group 6, Geneva, 2005

24. Vyatkin V., Hanisch H.-M. *Verification of Distributed Control Systems in Intelligent Manufacturing,* Journal of Intelligent Manufacturing, special issue on Internet Based Modelling in Intelligent Manufacturing, *vol.14, N.1, 2003, pp.123-136*

25. J. Thieme: *Symbolische Erreichbarskeitanalyse und automatische Implementierung struktuirter, zeitbewerter Steuerungsmodelle,* Dissertation zur Erlagung des Grades Dr.-Ing., Berlin: Logos Verl., 2002

26. A. Lobov, J. LM Lastra, R. Tuokko, V. Vyatkin: *Modelling and Verification of PLC-based Systems Programmed with Ladder Diagrams*, INCOM’2004, Proc., Salvador, Brazil, April, 2004

27. A. Lobov, J. L. Martinez Lastra, R. Tuokko, V. Vyatkin: Methodology for Mo1delling Visual Flowchart Control Programs using Net Condition/Event Systems Formalism in Distributed Environments, IEEE Conference on Emerging Technologies in Factory Automation (ETFA'03), Proc., Lisbon, September, 2003

28. Vyatkin V., Hanisch H.-M., Pfeiffer T., "*Modular typed formalism for systematic modelling of automation systems*", 1st IEEE Conference on Industrial Informatics (INDIN'03), Proc., Banff, Canada, August 2003

29. V. Vyatkin, H.-M Hanisch, G. Bouzon: Op*en Object-oriented validation framework for modular industrial automation systems*, INCOM'2004, Proc., Salvador, Brazil, April, 2004

30. H.-M. Hanisch and A. Lüder: *Modular Modelling of Closed-Loop Systems*, Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Berlin, Germany, October 21-22, 1999, Proc., pp. 103-126

31. Starke P.H., Hanisch H.-M., Analysing of Signal/Event Nets, In Proc. 6th IEEE International Conference on Emerging Technologies and Factory Automation ETFA-97, Los Angeles, USA, pages 253-257, September 1997.

32. P. Starke, S. Roch, K. Schmidt, H.-M. Hanisch, A. Lüder: Analysing signal-event systems, Technical report**,** *Humboldt Universitat zu Berlin*, Institut für Informatik, http://www.informatik.hu-berlin.de/lehrstuehle/automaten/tools/, July, 2004

33. V. Vyatkin, H.-M. Hanisch, P. Starke, and S. Roch: *Formalisms for verification of discrete control applications on example of IEC1499 function blocks*, Conference "Verteilte Automatisierung" (Distributed Automation), Proc., pp. 72-79, Magdeburg, Germany, March 2000

34. M. Bonfè and C. Fantuzzi: *Design and Verification of Industrial Logic Controllers with UML and Statecharts*, submitted to the IEEE Conference on Control Application 2003, June 23-35, Istanbul, Turkey

35. K. Thramboulidis: *Using UML for the Development of Distributed Industrial Process Measurement and Control Systems*, IEEE Conference on Control Applications (CCA), September 2001, Mexico.

36. Hanisch H.-M., Lüder A., Thieme J., A Modular Plant Modelling Technique and Related Controller Synthesis Problems. IEEE International Conference on Systems, Man, and Cybernetics, October 1998, vol.1, pp. 686 –691

37. Hanisch, H.-M. and A. Lüder: Modular modelling of closed-loop systems, Colloquium on Petri Net Technologies for Modelling Communication Based Systems. Proc., pp.103—126, Berlin, Germany, 2000

38. International Standard IEC 1131-3, Programmable Controllers - Part 3, International Electrotechnical Commission, 1993, Geneva, Switzerland

39. Nematron Corp., OpenControl: About open architecture, http://www.nematron.com/OpenControl/oc_architecture.shtml, September 2001

40. Lastra, Jose L.M., Evaluation of New Open Control Systems for Light Assembly Applications. M.Sc. Thesis. Tampere University of Technology, 2000

41. FBDK - Function Block Development Kit at www.holobloc.org, visited in June, 2005

42. Lobov, A., *An Approach to the Formal Verification of Automated Manufacturing Systems with Programmable Control,* M.Sc. Thesis, Tampere University of Technology, April 2004, thesis related material: http://www.pe.tut.fi/movida/LobovThesis/

43. K. Thramboulidis: *Development of Distributed Industrial Control Applications: The CORFU Framework*, 4th IEEE International Workshop on Factory Communication Systems, August 2002, Vasteras, Sweden

44. K. Takatsuka and S. Tomita: *On modelling and an algorithm for verifying behaviour of discrete parallel production system, PSE2002ASIA*

45. S.Kowalewski, P.Herrmann, S.Engell, R.Huuk, H.Krumm, Y.Lakhnech, B.Lukoschus, and H.Treseler: *Approaches to the formal verification of hybrid systems.* Automatisierungstechnik, 2:66--73, 2001.

46. Fisler, K.: *Timing diagrams: Formalization and algorithmic verification*. Journal of Logic, Language, and Information, 8(7), July 1999.

47. Amla, N., Emerson, E., Kurshan, R., and Namjoshi, K: *Model checking of synchronous timing diagram,*. Conference on Formal Methods in Computer Aided Design, Proc., Nov. 2000

48. Schlör, R., Allara, A. and Comai, S.: *System Verification using User-Friendly Interfaces*. In *Design, Automation and Test in Europe,* pp. 167-172. IEEE Computer Soc. Press, 1999

49. Vyatkin, V. and Hanisch, H.-M.: *Application of Visual Specifications for Verification of Distributed Controllers*, Proc. of IEEE Systems, Man, and Cybernetic Conf, pp. 646-651, Tucson, 2001

50. G. Bouzon, V. Vyatkin, H.-M. Hanisch: *Timing Diagram Specifications in Modular Modelling of Industrial Automation Systems*, IFAC World Congress, Prague, 2005

51. Roch, S.: Extended Computation Tree Logic, in Proc. of Workshop on Concurrency, Specification and Programming, Berlin, 2000

52. P. Starke, S. Roch, K. Schmidt, H.-M. Hanisch, and A. Lüder, Analysing Signal-Event Systems, Technical report, Humbold,[Online]:http://www.ece.auckland.ac.nz/~vyatkin/tools/modelchekers.html