A PETRI NET APPROACH FOR THE DESIGN OF DYNAMICALLY MODIFIABLE EMBEDDED SYSTEMS*

Carsten Rust, Franz Josef Rammig
University of Paderborn, Germany
car@c-lab.de, franz@upb.de

Abstract: A Petri net based approach for modeling dynamically modifiable embedded real-time systems is presented. The presented work contributes to the extension of a Petri net based design methodology for distributed embedded systems towards the handling of dynamically modifiable systems. Extensions to the underlying high-level Petri net model are introduced that allow for dynamic modifications of a net at run time.

1. INTRODUCTION AND RELATED WORK

To an increasing extent, embedded real-time systems these days are dynamically modifiable. As an example, consider an adaptive robot control, where components of the control software are changed at run time due to results of online learning algorithms. Another application scenario is a group of mobile robots that cooperatively solve a problem. Since robots may enter or leave the scenario or just change their location, the entire system is highly dynamic. Such systems gain increasing interest, e. g. when studying autonomic computing. Even in traditional application domains like automotive systems, dynamically modifying control systems are considered, for instance for the handling of so called fail-over situations, that is in error situations, where functionality has to be relocated. For the design of these systems, dynamically evolving subsystems – which imply a powerful basic model for specification – have to be considered together with basic controllers running under hard reliability constraints.

For the design of these systems, we propose to use a methodology based on high-level Petri nets as the underlying formal model [11]. We have chosen a

*This work was supported by the German Science Foundation (DFG) project SFB-376

high-level Petri net model for several reasons, for instance in order to benefit from the multitude of existing verification and analysis methods based on Petri nets. While Petri nets are well-established for the design of static systems, they lack support for dynamically modifiable systems. We propose an extension in such a way that an engineer is enabled to annotate transitions with transformation rules. A transformation rule specifies a modification of the system that is performed when the annotated transition fires. The basic concepts of our approach were first introduced in [10]. In [8] and [6], the extended design methodology and a tool for the simulation of dynamically modifiable systems were presented. In this paper, we concentrate on the formal Petri net model. We will define a self-modifying Petri net model as the extension of a hierarchical high-level Petri net model.

In the literature, dynamically modifiable Petri nets were often considered in the context of object-oriented Petri nets. An example are Object Petri nets introduced by Valk [13]. They support a two-stage modeling method: a main net called system net contains several object nets, which are instanciated via tokens of the system net. Transition firings in the system net, which lead to changes of its net marking, obviously can change the overall net. However, the dynamics is reflected in the marking of the net. No changes to the net structure are made. An early approach to self-modifying Petri nets was presented by Valk in the late seventies [12]. More recently, Badouel and Darondeau introduced Stratified Petri nets, a subset of Valks self-modifying nets. Both models are based on standard Petri nets without annotations. Modifications of a net are due to a simple mechanism switching edges on and off dependant on the current net marking. An approach for high-level nets which is based on similar ideas is presented in [1].

We propose a more generic approach, where modifications of the net structure at run time result from coupling a net model with graph transformation rules (productions), as they are known from graph grammars and high-level replacement systems respectively. Several other approaches for coupling Petri nets and graph transformation techniques can be found in literature (see for instance [2] for an overview). One example is the concept of net transformation systems [7]. Roughly speaking, a net transformation system is a graph grammar, where the generated graphs are Petri nets and the definition of productions is based on Petri net morphisms. Basically, we use very similar concepts. The characteristic feature of our approach is that the transformation system is integrated into the Petri net formalism by annotating transitions with productions. In the aforementioned approaches, graph transformations are applied to Petri nets at design time only. Our approach integrates them into the firing-rule.

In the following section, we will first give an informal brief overview of the hierarchical high-level Petri net model, that forms the basis for introducing dynamic modification. The specification of Petri net transformation rules

and their integration into the high-level Petri net model will then be defined formally in Sections 3 and 4. Finally, a small application example will be considered in Section 5.

2. BASIC PETRI NET MODEL

The basis for defininining dynamically modifiable Petri nets in the following section is a high-level form of Petri nets. Petri nets are bipartite directed graphs augmented by a marking and firing rules. The Petri net graph consists of a finite set of places *P,* a finite set of transitions *T,* directed edges from places to transitions and from transitions to places. Places model conditions. For this purpose they may be marked by tokens. Driven by specific firing rules, a transition can fire based on the local marking of those places it is directly connected with. By firing, the marking of these places is modified.

With regard to the definition of Petri net morphisms, we adopt the so-called algebraic notation for the formal description of Petri nets. Hence, a Petri net graph is a tuple $F = (P,T,pre,post)$ where $pre : T \rightarrow \mu(P)$ assigns a multiset of places (the preset) to each transition, while $post : T \rightarrow \mu(P)$ specifies the postset of each transition. Figure 1 a) shows an example net. Its formal definition is $N_1 = (\{p_5, p_6, p_7\}, \{t_8\}, \{(t_8, 3p_5 + 2p_6)\}, \{(t_8, p_7)\})$.

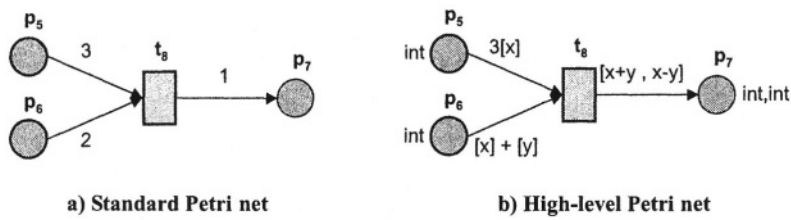

Figure 1. Petri net examples

In the case of high-level nets the tokens are typed individuals. The other net components are annotated accordingly: places with data types, transition in-edges with variable expressions, transitions with a guard and transition out-edges with term expressions, i. e. sums of functional expressions. Now a transition can fire only if the formal edge expressions can be unified with actually available tokens and this unification passes the guard expression of the transition. By firing, the input tokens are consumed and calculations associated with the transition out-edges are executed. That way new tokens are produced that are routed to output places of the transition. A simple high-level net is depicted in Figure 1 b). Different from this example, we usually annotate transition out-edges with variable expressions and transitions with corresponding variable assignments, since to our experience this representation has some advantages

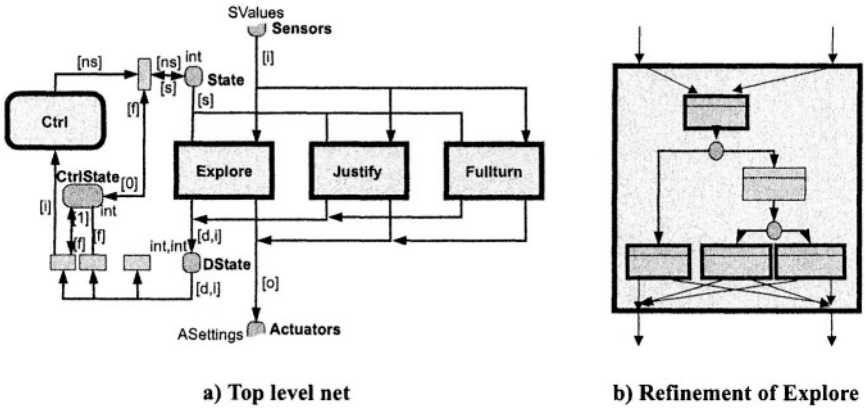

Figure 2. Hierarchical high-level Petri net

in practical applications. Obviously, both alternatives for annotating the outgoing edges of transitions are equivalent. The depicted notation was chosen, because the resulting formal definitions are more readable. In order to handle complex systems we added a hierarchy concept. As an example for a hierarchical specification, a small robot control is depicted in Figure 2 a). It contains three hierarchical transitions, each instantiating a distinct robot behavior which maps sensor values to according actuator settings. The components providing the sensor values and processing actuator values respectively are omitted from the figure. Furthermore, the net contains a hierarchical place instantiating a discrete control which is responsible for properly switching between the possible modes. The instantiation of the Explore-module is depicted in Figure 2 b).

Besides support for easy modeling, another reason for presuming a hierarchical Petri net model as the basis of our dynamically reconfigurable model is that the structure induced by hierarchy may be used for defining scopes, to which transformation rules can be applied. In order to define a hierarchical structure, we assume a function s which assigns a parent node to each node of the net. A node acting as a parent node is called hierarchical node. The inverse function of s assigns to each hierarchical node the set of nodes which constitute the module instanciated by this node. In order to keep the definitions reasonably simple, we assume that each hierarchical Petri net can be realized by a flat net constituted by the basic, i. e. non-hierarchical, nodes of the net. This holds for many hierarchy concepts. In some cases however, extensions to the net model are necessary in order to facilitate this mapping. Since hierarchy is not in the focus of the presented work, we neglect these cases. Thus, in the following we consider flat Petri nets, which have a structure imposed by an original hierarchical definition.

3. RULES FOR DYNAMIC MODIFICATION

In the case of a static system, the entire system can be modeled in advance. To specify these systems, we propose to use the hierarchical high-level Petri net model outlined in the previous section. For dynamically modifiable systems however, only the generating system of a set of potentially resulting systems can be provided. A straightforward approach for describing the generating system is given by graph grammars, since a Petri net specification is strongly based on a graph, and graph grammars are a standard formalism for specifying graph manipulations. From the various existing approaches for defining graph grammars, we have chosen an algebraic approach, which was first introduced by Ehrig et. al. in the early seventies [4], Algebraic approaches typically make use of constructs from category theory in order to describe graph transformation rules (productions) and their semantics, i. e. the precondition for applying a production to a given graph as well as the effect to the graph. In order to apply category theory, categories of graphs are considered, where relationships between graphs are modeled by graph morphisms.

Hence, in order to define productions for Petri nets, we first have to define Petri net morphisms. At this point, a - merely technical - problem arises: Productions and likewise Petri net morphisms shall be formulated over high-level Petri nets as well as be part of the annotation of high-level Petri nets. In order to solve this cyclic dependency, we define morphisms for a generic model of annotated nets complying with standard high-level nets, but also with the dynamically modifiable nets we are aiming at. An annotated net combines a Petri net graph $F = (P,T,pre,post)$, as it was introduced in the previous section, with a tuple of functions $A = (A_P : P \rightarrow L_P, A_T : T \rightarrow L_T, A_{F_i} : T \rightarrow L_{F_i} \times P, A_{F_o} : T \rightarrow L_{F_o} \times P)$, where L_P, L_T, L_{F_i}, and L_{F_o} are languages. Each function assigns annotations to Petri net elements. Given two annotated Petri nets, a Petri net morphism is a tuple of functions $f : N \rightarrow M = (f_P : P^N \rightarrow P^M, f_T : T^N \rightarrow T^M, f_A : A^N \rightarrow A^M)$ which maps the places, transitions, and annotations of one net N to those of another net M. For f being a morphism, it is required that for all net components (i. e. places, transitions, and edges), the corresponding morphism component commutes with the annotation function. For the complete formal definition we refer to [9].

Having introduced Petri net morphisms, we are able to define productions for Petri net transformations in the usual way. Thereby we follow the double pushout approach which was introduced in [4]. A comprehensive tutorial can for instance be found in [3]. In the double pushout approach applied to Petri nets, each production p consists of two Petri net morphisms $l : K \rightarrow L$ and $r : K \rightarrow R$. Basically, p correlates an annotated Petri net L (the left side of the production) with an annotated Petri net R (the right side). Furthermore,

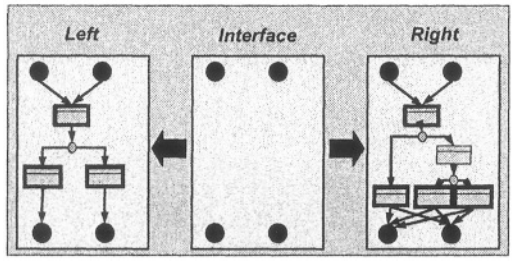

Figure 3. Rule for Dynamic Transformation of Petri nets

the production explicitly specifies the interface object *K,* typically a common subnet of *L* and *R.* An example for a production is depicted in Figure 3. The rule can be applied to a net *N,* if *N* contains the net on the left hand side of the rule *(Left).* When the rule is applied, the left hand side is replaced with the net on the right hand side *(Right).* The common interface object *(Interface)* must be part of the net *N* for application of a rule, but it remains unchanged during replacement. Hence, it specifies the interface of the modified net to the surrounding net.

In general, the application of a production to a graph leading to another graph (in our case the application to a Petri net leading to another Petri net) is called a direct derivation. In algebraic approaches to graph grammars, direct derivations are defined by gluing constructions of graphs, that are formally characterized as pushouts, a standard construct from category theory. As the name suggests, a direct derivation step in the double pushout approach is modeled by two pushout diagrams. They are depicted in Figure 4. The first diagram (1) describes the deletion of all elements of *N* which have a pre-image in *L,* but none in *K.* The diagram contains the graphs *K, L, N,* and *C,* the latter being the graph resulting from the first step. In addition to the morphisms l and n, the diagram contains $k : K \rightarrow C$, which embeds *K* into C, and $l' : C \rightarrow K$. In terms of category theory, *N* is called the pushout object of l and k, while *C* is the pushout complement object of l and m. The second pushout diagram (2) describes the second step, where all elements of *R* are inserted that do not

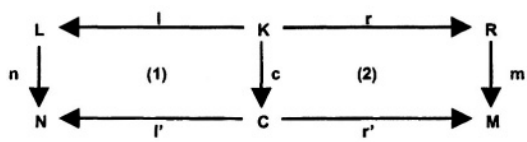

Figure 4. Diagram of a direct derivation

have a pre-image in *K*. In this diagram, the resulting graph *M* is the pushout object of r and k. Summarized in terms of category theory, given a production $p = (l : K \rightarrow L, r : K \rightarrow R)$, a Petri net *N* and a match $m : L \rightarrow N$, *p* is applied to *N* by first building the pushout complement object *C* of *l* and *m* and then building the pushout object of r and $k : K \rightarrow C$. We pass on a formal definition of pushouts and direct derivations and conclude this section by characterizing three objects from Figure 4 that are substantial for the application of a production $p = (l : K \rightarrow L, r : K \rightarrow R)$ to a given annotated Petri net *N*. These are the Petri net morphisms n (which embeds the left side of the rule into *N*), l'^{-1} (which removes the left side from *N*), and r'(which adds the right side). In the following we denote these objects by the tuple $D = (D_m, D_r, D_a)$.

4. INTEGRATION OF DYNAMIC MODIFICATION

With a formalism for describing net transformations, we now are able to define dynamically modifiable high-level Petri nets. They consist of a Petri net graph $F = (P,T,pre,post)$ as described in section 2 and a tuple of annotation functions $A_N = (A_P : P \rightarrow \mathcal{S}, A_I : T \rightarrow \mathcal{L}_d \times P, A_G : T \rightarrow \mathcal{L}_{Op}(bool), A_O : T \rightarrow \mathcal{L}_m \times P, A_T : T \rightarrow \mathcal{R}, A_M : P \rightarrow \mathcal{L}_{Const})$. A_P annotates places with sorts, i. e. with datatypes, A_I annotates transition in-edges with variable expressions. Transitions are annotated with a guard (by A_G) and with a transformation rule (by A_T). A_O annotates transition out-edges with sums of terms. Finally, the initial marking is specified by A_M. The definition of a signature including a set of sorts $\mathcal{S}$ as well as of the languages $\mathcal{L}_d$, $\mathcal{L}_{Op}(bool)$, $\mathcal{L}_m$, and $\mathcal{L}_{Const}$ is straightforward. $\mathcal{R}$ is the set of rules. Each element of $\mathcal{R}$ is a tuple $r = (p, v)$, where p is a production as described in the previous section, and v is a (hierarchical) Petri net node, the scope of r.

Annotating a transition t_{mod} of a Petri net *N* with a rule $r = (p, v)$ specifies that, during firing of t_{mod}, the production p is applied to the subnet instantiated by v. If several matches of the production are feasible, one of them is chosen non-deterministically. For defining the semantics of a dynamically modifiable net formally, the definition of net markings as well as the transition firing rule have to be extended. As usual, markings assign tokens to the places of a net, whereby the sort of each token must fit to that of the place. For dynamically modifiable nets, the notion of a Petri net marking is extended towards a Petri net configuration consisting of a marking and a Petri net. Petri net configurations are modified by firing of transitions. For enabling a transition firing, a match of the transition's transformation rule has to be found in the current net as well as a consistent substitution of the transition's variables by values of the current marking. Hence, a transition t, a tuple $D= (D_m, D_r, D_a)$ characterizing a direct derivation, and a variable binding *B* are combined to form a

transition step $S = (t, D, B)$. B must be a consistent substitution of the transition's variables, for which the transition guard is true. If a transformation rule is specified for the transition, the transition guard as well as the out-edge annotations may include references to the rule components, for instance in order to specify additional conditions. Therefor, the set of transition variables includes the places and transitions of the rule's left side. Accordingly, a consistent substitution B being part of a transition step $S = (t, D, B)$ must assign values to these variables, which comply with D.

The semantics of a transition step S is defined in terms of its incremental effects describing the effect of the demarking process of a transition and of the marking process respectively. For dynamically modifiable nets, the incremental effects of a transition step are twofold. We have to define the effects of S on the current net $E_N^-(S)$ and $E_N^+(S)$ as well as the effects on the current marking $E_M^-(S)$ and $E_M^+(S)$. Given a transition step $S = (t, D, B)$, where $D = (D_m, D_r, D_a)$, and a configuration $C = (N,M)$, the effects on the current net N are given directly by the two functions D_r and D_a applied to N. From these functions, the two nodesets $E_V^-(S)$ and $E_V^+(S)$ can be derived. $E_V^-(S)$ contains all nodes of N that are removed by D_r, while $E_V^+(S)$ contains all nodes added by D_a. As in static nets, the negative effect of a transition step results from evaluating the in-edge annotations of t with the substitution B. Similarly, the out-edge annotations are evaluated for generating the positive incremental effect $E_M^+(S)$. In addition, modifications to the net must be taken into account. The positive effect $E_M^+(S)$ has to be restricted to the marking of those places, that remain in the net after modification, i. e. the elements of $E_V^-(S)$ are not marked. Places contained in $E_V^+(S)$, i. e. places created by the transition step, are assigned their initial marking.

Based on these definitions, the firing rule of dynamically modifiable high-level Petri nets can be defined expectedly. Let t be a transition with the transformation rule $A_T(t) = (p, v)$, where $p = (l : K \rightarrow L, r : K \rightarrow R)$. Let $D = (D_m, D_r, D_a)$ be a direct derivation, $S = (t, D, B)$ a transition step, and $C_1 = (N_1, M_1)$ a configuration. S is enabled, i. e. t can fire, in C_1, if the following conditions hold. (1) The transition t is an element of N_1. (2) The scope of the transformation rule v is a hierarchical node of N_1. (3) v instantiates $D_m(L)$. (4) The negative incremental effect $E_M^-(S)$ is included in the current marking M_1. If S is enabled in C_1, it may fire leading to a configuration $C_2 = (N_2, M_2)$ where N_2 results from applying D_r and D_a to N_1 and M_2 results from subtracting $E_M^-(S)$ from M_1 and adding $E_M^+(S)$.

5. APPLICATION

Using the Petri net transformation features introduced in the previous sections, a more concise specification of the robot control presented in Figure 2 is

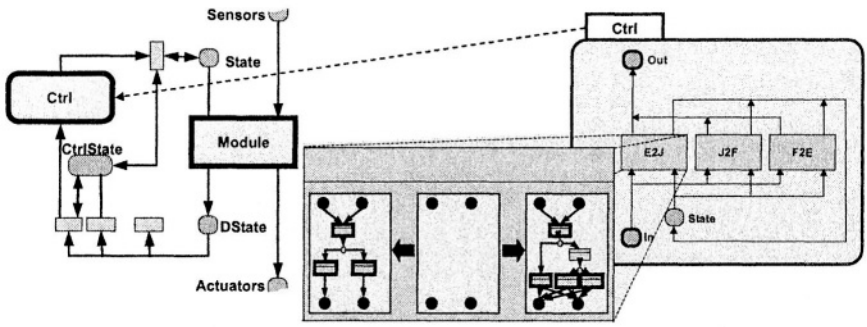

Figure 5. Petri Net with dynamic modification

feasible. An excerpt of the revised specification is depicted in Figure 5. The hierarchical transitions *Explore, Justify,* and *Fullturn* specifying the behaviour in each possible mode of the robot have been folded into one transition *Module*. The mode switching transitions of the discrete control *Ctrl* are annotated with transformation rules changing the refinement of *Module* as exemplarily shown for one transition in Figure 5. This specification is more compact than the original one. Beyond that, it is more flexible, since transformation rules determining the refinement can depend on run time data.

The presented application example is comparatively small. An application of our approach to a larger robot scenario, a small robot contest called 'Capture the Flag', is described in [5]. In both cases the extended high-level Petri net model has proven useful for the specification of dynamically modifiable systems. For an evaluation of the system behavior, we provide an execution platform [6], which allows to simulate the execution of a dynamically modifiable high-level Petri net on the simplified model of the target hardware. The execution platform will also serve as a basis for the automated implementation of dynamically modifiable Petri nets. For the realization of the above described applications, it was necessary to transform the dynamic Petri net models into equivalent static nets, since a direct implementation of truly dynamic behavior on the available small microcontrollers was not feasible. However, with regard to more powerful backends, we made first experiments with a direct implementation of the dynamic nets in Java, whose results were very promising. Compared to the implementation of an equivalent static net, the time for executing a modifiable net increased by a factor of 1,5 only.

6. CONCLUSION

We have presented an extension of our high-level Petri net model in order to capture dynamically modifiable embedded systems, for instance adaptive robot

controls. In order to achieve a formal definition of dynamically modifiable Petri nets, the existing high-level Petri net model was coupled with transformation rules as they are known from graph grammars. In our approach, these productions annotate transitions. The firing rule for transitions was modified accordingly.

REFERENCES

[1] E. Badouel and J. Oliver. Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes. In *Proc. of a workshop within the 19th Int'l Conf. on Applications and Theory of Petri Nets,* 1998.

[2] B. Braatz, K. Ehrig, K. Hoffmann, J. Padberg, and M. Urbasek. Application of Graph Transformation Techniques to the Area of Petri Nets. In *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002),* pages 35–44, Grenoble, France, 2002.

[3] H. Ehrig, M. Korff, and M. Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of 4th International Workshop on Graph-Grammars and Their Application to Computer Science,* volume 532 of *Lecture Notes in Computer Science,* Bremen, Germany, Mar. 1990. Springer.

[4] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory,* pages 167–180. IEEE, Oct. 1973.

[5] M. Koch and C. Rust. Design of intelligent mechatronical systems with high-level petri nets, *submitted to:* IEEE/ASME Transactions on Mechatronics, Sept. 2004.

[6] W. Y. Liu, C. Rust, and F. Stappert. A simulation platform for petri net models of dynamically modifiable embedded systems. In *The European Simulation and Modeling Conference (ESMC 2003),* Naples, Italy, Oct. 2003.

[7] J. Padberg, H. Ehrig, and L. Ribiero. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science,* 5:217–256, 1995.

[8] F. J. Rammig and C. Rust. Modeling of dynamically modifiable embedded real-time systems. In *9th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003F),* Capri, Italy, Oct. 2003.

[9] C. Rust. A High-Level Petri Net Model for the Design of Dynamically Modifiable Systems. *Internal Report,* URL: http://wwwhni.uni-paderborn.de/eps/uni/publications/, May 2004.

[10] C. Rust, F. Stappert, and R. Bernhardi-Grisson. Petri Net Based Design of Reconfigurable Embedded Real-Time Systems, In *Distributed And Parallel Embedded Systems.* Kluwer Academic Publishers, 2002.

[11] C. Rust, J. Tacken, and C. Böke. Pr/T–Net based Seamless Design of Embedded Real-Time Systems. In *Applications and Theory of Petri Nets 2001,* LNCS 2075, pages 343–362. Springer Verlag, 2001.

[12] R. Valk. Self-modifying nets, a natural extension of petri nets. *Lecture Notes in Computer Science: Automata, Languages and Programming,* 62:464–476, 1978.

[13] R. Valk. Petri nets as token objects, an introduction to elementary object nets. In J. D. und M. Silva, editor, *Applications and Theory of Petri Nets 1998,* LNCS 1420, pages 1–25. Springer Verlag, 1998.