

Hardware Accelerators for Petri-net Analysis

Gy. Csertán, I. Majzik and A. Pataricza

Department of Measurement and Information Systems

Technical University of Budapest, H-1521 Budapest, Műegyetem rkp. 9

S. C. Allmaier and W. Hohl

Department of Computer Structures (IMMD3)

University of Erlangen, D-91058 Erlangen, Martensstr. 3

Abstract—For reachability analysis of Petri-nets an FPGA-based accelerator is proposed. Since the simple components of Petri-nets can be easily realized in high-density FPGAs, the complete problem can be mapped to silicon providing a solution environment faster than the traditional software-based simulators. Some classes of Petri-nets support the compositional analysis, this way the limited capacity of the FPGA does not prevent the investigation of real-life problems.

Keywords: Petri-nets, verification, hardware accelerators

I INTRODUCTION

As the complexity of today's computerized control systems increases, the efficient modeling and analysis of these systems becomes more and more time and resource consuming. Some methods, e.g. the *formal verification* of fundamental system properties (like absence of deadlock, freedom from starvation, unsafe states etc.) can be characterized by exponential complexity. Different abstract mathematical models, like dataflow networks, Petri-nets, process algebra share this very same key problem. Thus, although the computational power of modern computing equipment used for validation and verification increases rapidly, even accompanied by a radical drop in the price/performance ratio, the processing capacity is often insufficient. In the case of complex target systems, solving this problem either requires radical model simplifications or extremely long run times.

The traditional solutions to overcome the performance bottleneck can be grouped into two typical categories.

- In *multiprocessors* the task to be performed is distributed between different general purpose processing elements according to a structural or functional problem decomposition.
- In *co-processors* application dependent dedicated hardware subunits perform some elementary operations typically by about one order of magnitude faster, than a program running on the CPU itself.

In many cases, the large hardware overhead of a massively parallel system is unacceptable, thus either a mono-processor or a minor configuration of a multiprocessor extended by a co-processor per computing node is used.

This work was supported by the Hungarian NFS under the grant T15728, by the project MKM PFP 2839/98 and by the German-Hungarian intergovernmental research contract under the acronym ACCUSE.

The advent of high-speed and complex in-system programmable FPGA circuits opened new horizons for the use of dedicated hardware solutions by providing a similar circuit complexity, as earlier VLSI designs. The new field of reconfigurable (or in another terminology custom) computing supports the design of general-purpose co-processors. As both the structure and instruction set of these co-processors can be downloaded just before starting the program, they serve simply as a silicon compiled problem dedicated subroutine set for a variety of applications. The possibility of a run-time reconfiguration allows the use of adaptive algorithms from a pre-compiled library prepared in the algorithm development phase.

Another promising, but still unexplored possibility is the use of *accelerators*, which are well-proven solutions in hardware emulation. In accelerators some complete parts of the problem are realized on the FPGA, in contrary to the co-processor approach, where only some parts of the algorithm are mapped to silicon. This way, additionally to a part of the solution algorithm, some data structures corresponding to the actual data are realized on the FPGA as well. A potential way to reduce the synthesis related overhead is to adopt a similar philosophy as the standard cell based ASIC design approach does it: an application dependent basic cell library is defined and only the interconnection network is synthesized during the algorithm development phase. This way the majority of the synthesis task is accomplished already prior to the actual program run.

All of the above mentioned solution alternatives are candidates for the use in modeling and validation of digital systems. In this paper we propose a hardware accelerator for the reachability analysis of systems modeled using Petri-nets, one of the most popular modeling formalisms.

The paper is structured as follows. Section II introduces our target formalism, the Petri-nets, and the approaches of the solution of the reachability problem. In Section III the mapping of the Petri-net to silicon is presented, while in Section IV the structure and control algorithm of the accelerator is described, detailing also the advantages and limitations of the solutions. Section V presents the class of compositional Petri-nets where the accelerator can be efficiently used. Section VI describes our experimental implementation.

A Petri-net is a 4-tuple $N = \{P, T, F, m_0\}$ where $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions satisfying $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation (set of arcs) and $m_0 : P \mapsto \mathbb{N}$ is the initial state. The symbols $*t$, $t*$, $*p$, $p*$ denote the pre-set and post-set of a transition t or a place p .

A state of a Petri-net is an assignment of nonnegative integers to each place. If a place p is assigned k then it can be referred to as p is marked with k tokens. A transition t is enabled if each place $p \in *t$ has at least one token. Firing of a transition removes one token from each place in its pre-set and puts a token to each place in its post-set. A state (often referred to as a marking) m is reachable from the initial state m_0 if there exists a sequence of firings that transforms m_0 into m . A Petri-net is said to be k -bounded if for all reachable states, the number of tokens in any place is less than or equal to k . An 1-bounded net is said to be *safe*.

A number of properties to be analyzed (e.g. safety and liveness) are effectively reducible or equivalent to the reachability problem, i.e. to check whether some arbitrary state(s) can be reached from a fixed initial state of the system.

[1] has shown exponential space lower bound for the reachability problem. In the literature, a number of methods are proposed for efficient reachability analysis, each having its advantages and drawbacks. The symmetry method [2] needs the help of the analyst. Stubborn sets [3] and incomplete reachability analysis [4] in its original form aim at only finding deadlocks. Symbolic model checking [5] uses binary decision diagrams (BDD) to analyze huge state spaces, however, it is not guaranteed that most distributed systems have a compact BDD-based representation.

In this paper we focus on a completely different approach by investigating how the performance bottleneck can be resolved by utilizing the advantages of custom computing, i.e. the application of complex, high-speed user-programmable and reconfigurable FPGA circuits which enable to map the complete problem to silicon [6].

III MAPPING THE PETRI-NET TO SILICON

Petri-nets (PN) are favorite candidates to be implemented in an accelerator since the very simple components of the PN (places and transitions) can easily be synthesized.

Places (Figure 1) are realized in the FPGA by the following elements: (i) a counter representing the number of tokens in the place, (ii) zero logic signaling if this counter (i.e. the number of tokens) is non-zero, (iii) enabling logic for counting up/down of the counter (i.e. changing the state).

Transitions (Figure 2) are realized by (i) a flip-flop which indicates if the transition has already fired in a given step, (ii) logic enabling the transition if input places contain the

necessary number of tokens, (iii) logic signaling if the transition is fireable i.e. it is enabled and selected to fire, (iv) daisy chain logic (discussed later).

A co-processor could realize the Petri-net and its computation by interpreting transition firings (state changes) stored in some tabular form in local memory, in a similar manner as a pure software simulator would do it, but using a silicon compiled interpreter. An accelerator provides a more efficient solution. It performs the very same task after a synthesis phase implementing the above realization of the PN, this way a speed as high as one transition firing per FPGA clock period can be reached by this direct emulation of the PN. The price of the higher speed is the run time needed for the synthesis of the PN. However, the few number of regular components (places and transitions) can be used as *precompiled macros* requiring only their interconnection (wiring) to be generated in each run.

IV STRUCTURE OF THE ACCELERATOR

The set of reachable states of a Petri-net forms its reachability set (RS). If it is accompanied by the transition relation, then a reachability graph (RG) is formed. The RS (or RG) can be computed by the following algorithm:

Algorithm 1 (Generation of the RS) Given a Petri-net in its initial state. The algorithm (Fig. 3) performs an exhaustive simulation (by a breadth-first-search). Function *ChooseRemove* selects a state and removes it from the set of found ones, *NewState* generates a successor state by firing one of the enabled transitions, *SearchInsert* examines whether the actual state is in the union of the completed and found states; if not, then adds the actual state to the set of found states. The reachability set is generated in *completed_states*.

The task of the accelerator is the generation of the RS of the PN by simulation as described in this Algorithm. Accordingly, it is structured as follows (Figure 4). The *PN simulator* implements the successor function in silicon. The *state storage* contains the set of found states. A new state reached by the simulator is examined. If it has not occurred yet, then it is inserted into the storage; otherwise a successor state is generated. The task of the *search/compare engine* is to compare the actual state in the simulator with the states in the state storage or to choose a new state from the found set if it is needed. This module (including the simulation control) is implemented as a traditional state machine.

IV.A The PN simulator

The PN simulator consists of the synthesized building blocks presented in Section III. Moreover, the simulation control is supported by connecting the transitions to form a daisy chain (DCT, Figure 5). In a given state, the enabled transitions will fire in this order. Since all enabled transitions will fire, their order is irrelevant from the point of view of the RS.

The input of the first transition is connected to active level, the output of the last transition (i.e. the output of the daisy chain) is connected to the controller of simulation.

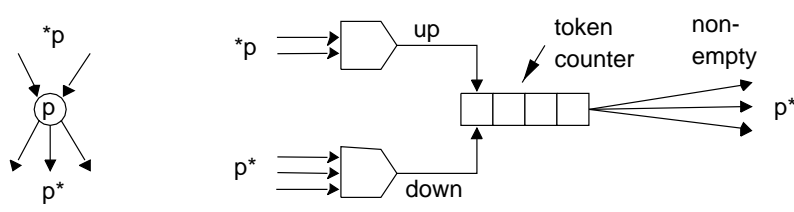


Fig. 1. Representation of a place

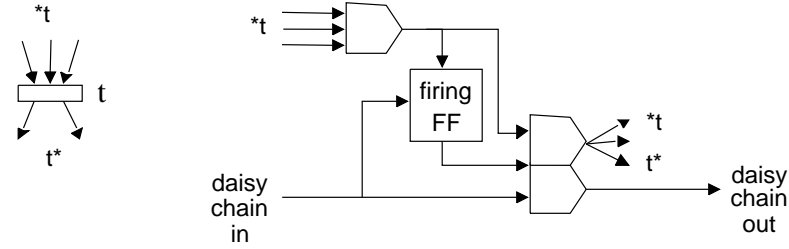


Fig. 2. Representation of a transition

```

completed_states =  $\emptyset$ 
found_states = {Initial_state}
while found_states  $\neq \emptyset$  do begin
    i = ChooseRemove(found_states);
    completed_states = completed_states  $\cup$  i
    for each transition t that is firable in i do begin
        j = NewState(i,t);
        SearchInsert(j, completed_states  $\cup$  found_states, found_states);
    end
done; done

```

Fig. 3. Algorithm of reachability set computation

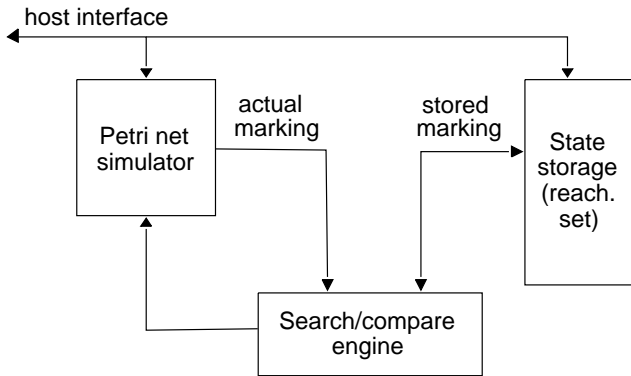


Fig. 4. General structure of the accelerator

IV.B The state storage and the search engine

Theoretically, the state storage and the search engine should be realized by a content addressable memory (CAM). However, since a state is a binary word consisting of a large number of bits (a few hundreds of places each represented by 4.8 bits), the efficient implementation is difficult. Thus we focussed on an alternative solution. A (hierarchical structure of) hash tables is built, and the set of states corresponding to the same hash code is stored dynamically in the state storage, in the form of linked lists. The advantage of this approach is, that the state storage is divided dynamically between the lists, no static splitting is needed.

A transition fires if it is enabled (enabling logic is active), it has not fired (firing flip-flop is inactive) and it has an active input signal in the daisy chain (it is the first in the chain among the transitions being enabled). In this case it does not propagate the active level for its successors in the chain until it has fired and the firing flip-flop is set. Transitions which are not firable propagate the level on their input to their output in the chain. Transitions which are enabled but are later in the chain can not fire before the preceding ones have fired.

The efficiency of the storage and the search engine can be improved further if the hash function is based on the structure of the net and can group together (some) successor states. Since the firing of a transition modifies the state in limited number of places, the difference between successor states is restricted to small parts of the binary word representing the state. In this way, only the first element should be stored in its full extent in the list corresponding to a given hash code, the next elements can be stored only as incremental differences. While doing the search, each element can be reproduced each after the other.

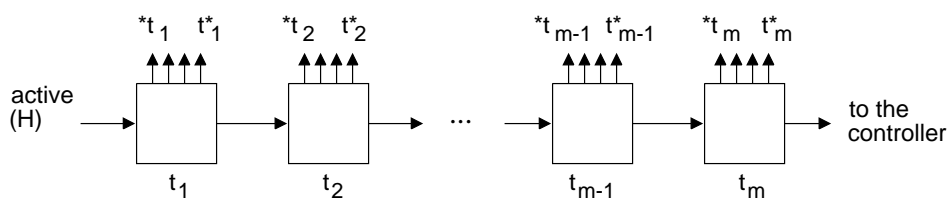


Fig. 5. Daisy chain of transitions (DCT)

IV.C The simulation control

The simulation control can be implemented either in breadth-first-search (BFS) or in depth-first-search (DFS) manner.

In a BFS approach, all successors of a given state are generated and stored in the state storage before a next state is set. In a given state, the transitions are examined. If there is a firable transition then it fires. The resulting state is searched in the RS; if it is not found then it is stored. Then the previous state is restored in the simulator and, if exists, the next firable transition fires and the procedure is repeated. In this way, all enabled transitions in a given state fire, in the order determined by the DCT. After all enabled transitions fired, the examination of the actual (repeatedly restored) state is completed. This fact is indicated in the state storage by a flag associated with the state (C flag). A new state, which is not completed yet, is chosen from the state storage, and its successor states are examined as described. The algorithm is sketched as follows:

Algorithm 2 (BFS based generation of the RS)

Given the Petri-net in its initial state in the simulator (the initial state is loaded into the counters corresponding to the places, the firing flip-flops of the transitions are reset).

- If the output of the DCT is inactive then there is a firable transition. By providing a clock signal, the firable transition in the DCT fires, which steps the counters of the places corresponding to the transition and sets the firing flip-flop of the transition. A new state is reached.

The new state is read from the simulator and searched in the state storage. If it is not found then it is stored without an active C flag. (If the RG is also to be built then a pointer is stored which identifies its predecessor state, i.e. the previous one.)

The previous state of the Petri-net is restored in the simulator and the algorithm restarts (the flip-flops of the DCT are not changed after the last firing).

- If the output of the DCT is active then there is no firable transition. The actual state has no further successors, its C flag in the state storage is activated. A new state without an active C flag is chosen from the state storage.
 - If there is such state then it is loaded into the simulator. The firing flip-flops in the DCT are reset. The algorithm restarts.
 - If none is found then the generation of the reachability set is completed, the algorithm exits.

In comparison with the above BFS, the DFS approach has the advantage that the original state needs not be restored after each simulation step, thus the speed of the simulation increases. However, it requires a modification of the hardware structure of the simulator. If a transition fired then this fact has to be remembered (returning to this state it should not fire again). To do this, a firing counter should be assigned to each state, marking the transition which last fired. The input of the DCT is not wired to an active level, since only that part of the DCT is enabled which is located after the last fired transition. The logic implementing this modification is quite complex [7], this way the BFS was implemented.

IV.D Advantages and limitations

The performance of the accelerator is significantly higher than that of a software simulator in the subtasks of (i) selection of the transition to fire, (ii) update of the marking in input/output places of a transition and (iii) search and compare in the reachability set. The reconfigurable FPGA has the advantage that different initial states can be set without reprogramming the device.

The speed of the accelerator can be estimated in clock cycles of the simulator. It is assumed that (i) the PN contains N places each having a bound of 16 tokens (ii), in a single cycle 32 bits are accessed in the simulator and 64 bits in the state storage, (iii) the number of elements in a sub-list of the state storage is S . The phases of a simulation step are presented in Table I. Accordingly, one step requires about $N(S + 5)/16$ clock cycles (e.g. if $N = 100$ and $S = 10$ then about 100 clock cycles).

Phase	Clock cycles
Firing of a firable transition, generating the new state	1
Reading the new state from the simulator	$4N/32$
Accessing the (hash) pointer table	1
Searching in the sub-list of the state storage	$< S(4N/64)$
Restoring the original state in the simulator	$< 4N/32$
Storing the actual state into the state storage	$4N/64$

TABLE I
PHASES OF A SIMULATION STEP

Main limitations of the approach (and their partial solutions) are as follows:

- Only a bounded PN can be analyzed. Unbounded nets should be examined by approximate analysis or by introduction of more sophisticated formalisms like ω -states.
- The size of the PN is limited by the FPGA. This problem can be solved by model decomposition. Since reachability analysis in general Petri-nets is not compositional, candidate nets are restricted to PNs built in a top-down manner by predefined refinement rules or special net classes like Superposed Generalized Stochastic Petri Nets (SGSPN, [8]) which provide decomposition by definition.
- The size of the RS is limited by the available state storage of the accelerator. The implementation of a more efficient state storage (e.g. custom memory architecture, BDD-based state storage) and a corresponding search engine is a task of our future research.

V THE COMPOSITIONAL METHOD

To overcome the problem of size limitation a compositional solution is proposed. The net is decomposed into sub-nets that are called partitions. The reachability analysis of the sub-nets can be done independently, and the final reachability graph is composed from the sub-graphs. Since the analysis of sub-nets can be done independently, there is no obstacle to execute the analysis runs one after the other on the same FPGA accelerator board. It is also possible to execute the different tasks on different FPGA processing units at the same time, if one has multiple accelerator boards.

The selected method is based on Superposed Generalized Stochastic Petri Nets (SGSPN) for the relative simplicity of the decomposing and composing algorithms and due to their nice property that their components are GSPNs. Superposed Generalized Stochastic Petri Nets [8] is a class of Petri-nets which enables compositional reachability as well as performance analysis of the net.

In our environment, concentrating on reachability problems only, we introduce the following restrictions. First, we assume that the net consist of timed transitions only (which means that the priorities between timed and immediate transitions are ignored). Second, we restrict our investigation to models without inhibitor arcs. Moreover, we assume that every arc has a multiplicity of one. This way, for the experimental version of the accelerator, we restrict the model to be a “Superposed Stochastic Petri Net”. Since we use the top-level algorithms developed for SGSPNs, and these restriction will be removed in the next versions (by modifying the hardware structure of the accelerator, i.e. including two daisy chains for timed and immediate transitions, including extra logic for inhibitor arcs and multiple arcs), we use the SGSPN notation in the following.

During analysis the reachability sets of components are explored in complete isolation (assuming that synchronized transitions are not disabled by other components) and sub-

```

1: decompose SGSPN into N GSPNs
2: forall i ∈ N
3:   Qi=reachability analysis of GSPNi
4:   compose Q from Qi's
5:   produce RG from Q

```

Fig. 6. Reachability Analysis Algorithm for SGSPNs

sequently the overall reachability set is composed. Accordingly, the reachability analysis algorithm (see Figure 6 for SGSPNs in general consists of the following steps:

- decompose the SGSPN into N GSPNs along of synchronized transitions
- solve the reachability analysis of each GSPN separately, and store the results in the state-transition matrices Q_i
- compose the state-transition matrix Q of the SGSPN from the sub-matrices Q_i ([9])
- create the reachability graph of the SGSPN from the state-transition matrix Q
- optionally timing analysis of SGSPN can be solved like in [8]

V.A The decomposition task

Unfortunately, nothing about the how-to of decomposition is presented in the mentioned bibliography. Later articles also does not provide information about the decomposition task. Therefore, an own method is used to partition of the SGSPN.

It relies on a simple spatial partitioning method. This is done in the hope that the designer of the Petri net drew the net in such a way, that corresponding / neighbor places and transitions are stored subsequently in the GSPN file. In this case these places and transitions will be part of the same sub-net, and “communication” between sub-nets can be minimized. Note that this minimal communication criteria could be a good starting point for optimization of the partitioning.

Of course manual assisted partitioning would be also possible, but often analysis of the net is not done by the designer of the net, if ever the net was not created automatically. The steps of decomposition are the following:

1. Places are partitioned into N parts by spatial partitioning. N is defined by the user, since implementations details of the accelerator card are still missing. Therefore, N can not be computed automatically based on the size of the Petri nets the accelerator card can accept.
2. Transitions are checked upon their input and output places:
 - (a) If all input and output places of the transition belong to the same partition, the transition is called local and it will belong to that same partition.
 - (b) Otherwise the transitions belongs to multiple partitions, and is a synchronization transitions. (Except the case, when the transition is an immediate one that must not be a synchronization transition. In

this case repartitioning of the places is necessary. Fortunately in our current work only timed transitions are assumed.)

3. Transitions are partitioned according to the results of step 2, and TS the set of synchronization transitions is built.
4. Finally the arcs are partitioned according to the partitioning of places and transitions.
5. After partitioning the sub-nets are written into file in a user defined Petri net tool format.

V.B The composition task

Composition of the reachability tree from sub-reachability trees. The input of the task are the state-transition matrices Q^i of the sub-nets. These state-transition matrices should be (if they are not already) split into parts Q_i^i describing local transitions and Q_t^i describing the effects of synchronizing transitions. The state transition matrix Q is then composed by using tensor addition \oplus and tensor multiplication \otimes :

$$Q = \bigoplus_{i=0}^{N-1} Q_i^i + \sum_{t \in TS} w(t) \bigotimes_{i=0}^{N-1} Q_t^i$$

VI EXPERIMENTAL IMPLEMENTATION OF THE ACCELERATOR

An experimental version of the accelerator was built to estimate the time required to synthesize the net and the size of the Petri-net which could fit into the FPGA. The search engine and the state storage were designed to be as simple as possible and the class of Petri-nets was restricted to safe nets.

The evaluation board is the VCC H.O.T.Works DS, a complete PCI-based programming and development system built upon the Xilinx XC6200 chip family. Its main features include XC6216 on-the-fly reconfigurable FPGA with 64x64 logic cells (each containing a flip-flop and/or an arbitrary two-input logic gate), 128Kx32bit on-board SRAM and programmable clock generator. The accelerator was built as a single-chip one, i.e. both the Petri-net and the search/compare engine were implemented in the available FPGA and the on-board memory was used as state storage.

The accelerator was designed by using a VHDL elaborator and the XACT-Step6000 place and route (P&R) software provided for the evaluation board. Our first experiments showed that the main bottleneck of the technology is that the synthesis tools are not able to handle efficiently the regular structure of the PN to be mapped to the FPGA. This way, in order to avoid the P&R process to be an order of magnitude slower than the simulation itself, specialized P&R tools have to be developed.

VII CONCLUSION

In this paper we have studied the application of an FPGA-based accelerator for reachability analysis of Petri-nets. The experiments showed that up-to-date high-density, high-speed FPGA circuits are able to simulate

Petri-nets of reasonable size at encouraging speed. However, in order to reduce the time required for synthesis, specific tools are required which are optimized to the problem domain, i.e. to the components and structure of Petri-nets.

REFERENCES

- [1] E. Cardoza, R. Lipton, and A. Meyer, "Exponential space complete problems for Petri nets and commutative semi-groups", in *Proc. 8th annual ACM Symposium on Theory of Computing*, 1976, pp. 50-54.
- [2] P. Huber, A. M. Jensen, I. O. Jensen, and K. Jensen, "Towards reachability trees for high-level Petri-nets", Technical report PD-174, DAIMI, Dept. of Computer Science, Aarhus University, 1985.
- [3] A. Valmari, "Stubborn sets for reduced state space generation", in *Supplement to Proc. 10th Int. Conference on Application and Theory of Petri-nets*, Bonn, 1989, pp. 1-22.
- [4] G. J. Holzmann, "On limits and possibilities of automated protocol analysis", in *Proc. 7th Protocol Specification, Testing and Verification*, 1987.
- [5] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia, "Petri net analysis using Boolean manipulation", in *Proc. 5th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain*, 1994.
- [6] Gy. Cseran, I. Majzik, A. Pataricza, and S. C. Allmaier, "Reachability Analysis of Petri-nets by FPGA Based Accelerators", in *Proceedings of Design and Diagnostics of Electronic Circuits and Systems Workshop, DDECS98, Szczyrk, Poland*, September 1998, pp. 307-312.
- [7] I. Majzik, A. Pataricza, and S. C. Allmaier, "Support of formal verification by FPGA based accelerators", Internal report, Dept. of Computer Structures (IMMD3), University of Erlangen-Nuremberg, Erlangen, Germany, 1997.
- [8] S. Donatelli, "Superposed Generalized Stochastic Petri Nets: Definition and efficient solution", in *Proc. 15th Int. Conf. on Application and Theory of Petri Nets 1994, Zaragoza, Spain*, 1994, pp. 258-277, Springer Verlag, LNCS-815.
- [9] P. Kemper, "Reachability analysis based on structured representations", in *Proc. 17th Int. Conf. on Application and Theory of Petri Nets, Osaka, Japan, June 24-28, 1996*, J. Billington and W. Reisig, Eds. 1996, pp. 269-288, Springer Verlag.