Symbolic Methods for the State Space Exploration of GSPN Models

Ian Davies[1], William J. Knottenbelt[2], and Pieter S. Kritzinger[1]

[1] Department of Computer Science, University of Cape Town, Rondebosch 7700, South Africa; {idavies,psk}@cs.uct.ac.za

[2] Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, United Kingdom; wjk@doc.ic.ac.uk

Abstract. Generalised Stochastic Petri Nets (GSPNs) suffer from the same problem as any other state-transition modelling technique: it is difficult to represent sufficient states so that *general,* real life systems can be analysed. In this paper we use symbolic techniques to perform state space exploration for *unstructured* GSPNs. We present an algorithm for finding an encoding function which attempts to minimize the height of BDDs used to encode GSPN state spaces. This technique brings together and extends a spectrum of ad-hoc heuristics in a formal algorithm. We also develop a BDD state exploration algorithm which incorporates an adjustable memory threshold. Our results show the ability to encode over 10^8 states using just 13.7MB of memory.

1 Introduction

As communication systems evolve and become more complex, the need for software tools to analyse these systems grows. A popular way of analysing such systems is to model their behaviour using a high-level abstraction such as Generalised Stochastic Petri Nets (GSPNs) [1]. Qualitative and quantitative properties of the system can then be assessed by analysing the low-level state space and state graph underlying the model. However, even relatively simple models can suffer from the *state space explosion problem*, where the number of states reachable from the initial state becomes too large to store.

In order to cope with increasingly complex models we therefore require advanced techniques for constructing and storing state spaces and state graphs. Based upon earlier work in electronic circuit theory (e.g. [6,12,15]), this paper describes our technique to explore and store the state space and state graph of GSPN models, using Binary Decision Diagrams (BDDs) [5] and Multi-terminal BDDs (MTBDDs) [9] as data storage structures. The application of BDDs to modelling formalisms is an active research area and has been investigated by, amongst others, Pastor *et al* [7,8] and Hermanns *et al* [11]. More recently Ciardo *et al* have successfully applied another BDD variant, Multi-valued Decision Diagrams (MDDs), to construct structured Petri net model state spaces [2,10].

T. Field et al. (Eds.): TOOLS 2002, LNCS 2324, pp. 188–199, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The focus of the present study is twofold. Firstly, in Section 3, we bring together and extend a spectrum of known ad-hoc heuristics in a formal algorithm for finding an encoding function for BDDs used to encode GSPN state spaces. Secondly, in Section 4, we present a BDD state exploration algorithm for unstructured GSPNs which has an adjustable memory threshold. Experimental results are presented in Section 5.

2 Background Theory

A BDD is a rooted, directed, acyclic graph which contains a set of vertices. Each vertex is associated with an *index* that describes its height in the BDD and a unique *identifier* that distinguishes it from the other vertices. There are two types of vertices in a BDD: *non-terminal* vertices, each of which has two child vertices; and *terminal* vertices that have a Boolean value but no children. Each BDD vertex can have any number of parent vertices. The two children of a non-terminal vertex associated with the Boolean values true and false are called the *high* and *low* child respectively.

A BDD is *ordered* if and only if each non-terminal vertex has a lower index than its child vertices. Further, an ordered BDD is *reduced* if it contains no vertex v such that $low(v) = high(v)$, nor does it contain distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic [6]. Reduced, ordered BDDs are therefore *canonical* representations of Boolean functions. This means that two functions are equivalent if and only if the reduced, ordered BDDs for the two functions are isomorphic.

Boolean operators may be applied to BDD data-structure operands. In 1990, Brace *et al.* [14] devised the efficient *ite*-algorithm for manipulating Boolean functions based on the Boolean ternary operator *if-then-else*. We refer the interested reader to [14] for further details.

MTBDDs are an extension of BDDs proposed by Fujita *et al* [9]. They contend that MTBDDs are a superior data structure, in terms of memory utilisation, particularly for the representation of sparse matrices. In fact, the matrix need not be sparse. If several of the matrix elements have the same value then the isomorphism of MTBDD subgraphs may be exploited to gain further advantage over standard sparse matrix packages. MTBDDs represent functions of the form $f : \tilde{D} \rightarrow \mathbb{R}$ where $\tilde{D}$ is a arbitrary finite set. One can encode the members of $\tilde{D}$ using $\lceil \log_2 |\tilde{D}| \rceil$ Boolean variables. In the case where $\tilde{D}$ is the set of integers $\{0, ..., n-1\}$, then the MTBDD represents a vector of length n. In the case where $\tilde{D}$ is the finite set $\{0, ..., n-1\} \times \{0, ..., m-1\}$, then the MTBDD represents a matrix of dimension $n \times m$.

The individual positions of elements in a vector or matrix can be encoded by a unique combination of Boolean variables. MTBDDs are both *reduced* and *ordered* and form a canonical representation of the vector or matrix they encode.

Information about the length of the vector or dimension of the matrix is lost in the MTBDD representation, and this must be stored separately from the data structure. Readers interested in more background regarding MTBDDs are referred to [9], [18] and [11].

3 Encoding Functions

Central to using BDDs for GSPN state space exploration is the *encoding function*, which symbolically represents a state by its *characteristic function* encoded as a BDD. The encoding function determines the height of the BDD and thereby directly influences the time taken for BDD-based operations. In the context of state-space exploration, the encoding function must provide a *one-to-one* mapping from each marking (or state of the GSPN) onto $\{true, false\}^n$. Thus we need to determine how many bits to allocate for each place in the net in order that it can be identified uniquely in each possible marking.

This section presents a technique for encoding GSPN states in BDDs that combines and extends the heuristics found by Haverkort [4] *et al*, Pastor and Cortadella [8] and Hermanns *et al* [11] in a formal algorithm. For brevity we have a assumed a basic knowledge of Petri net invariant theory in the explanations that follow; interested readers should consult [17].

3.1 Using P-Invariants

A P-invariant describes a weighted sum of place markings which remains the same for any reachable marking of the Petri net. A P-invariant is characterised by an equation of the following form:

$$C_{p_1}p_1 + \ldots + C_{p_n}p_n = U \tag{1}$$

where p_i represents the marking on place i, C_{p_i} represents the coefficient (weight) of p_i in the invariant equation, and U is the upper bound imposed by the equation. An upper bound on p_k is thus given by (for non-zero C_{p_k}):

$$p_k \leq U/(C_{p_k}) \tag{2}$$

Rearranging Equation 1 we find (again for non-zero C_{p_k}):

$$p_k = (U - (C_{p_1}p_1 + ... + C_{p_{k-1}}p_{k-1} + C_{p_{k+1}}p_{k+1} + ... + C_{p_n}p_n))/C_{p_k} \tag{3}$$

Since, we can compute p_k given $\{p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_n\}$, it is only necessary to store $n - 1$ of the places in that equation. We can apply this procedure for each P-invariant equation.

The places of the GSPN can therefore be divided into two categories: those which must be stored, and those which can be computed. The challenge is to attempt to minimise the number of bits used for the stored places that are actually encoded in the BDD. Our heuristic algorithm for this is shown in Figure 1. The P-invariant equations are sorted in order of increasing right-hand-side. This way the equation with the smallest RHS will be used first and require the smallest number of bits for each of its stored places. This is only effective in cases where the place concerned appears in more than one invariant equation. In this case we use the minimum of the upper bounds. We also sort each equation's left-hand-side terms in order of decreasing coefficient so that the place added to the computed set would have required the highest number of bits to store.

program Invariant Encoding
begin
1 initialise set S of stored places $= \phi$
2 initialise set C of computed places $= \phi$
3 Sort invariant equations on RHS
4 **for each** equation **do**
5 Sort equation in order of decreasing coefficient
6 **for each** equation e in order **do**
7 **begin**
8 **for each** place p in order **do**
9 **begin**
10 **if** $(p \notin S) \wedge (p \notin C) \wedge (\neg(p$ is the last place in $e))$
11 add p to S
12 **if** p is the last place in e
13 insert p at end of C
14 **end**
15 **end**
end.

Fig. 1. Algorithm for selecting places to be encoded.

This is a *greedy* algorithm which may not result in the optimal encoding, but it improves on the technique of simply using P-invariants to remove one place per equation by greedily selecting the places required to be stored with the smallest number of bits first. Results using this technique are reported in Section 5.

3.2 Partitions and BDD Ordering

Variable ordering can have a dramatic effect on the amount of memory and time needed to construct a BDD-based representation of the model's state space. In our experiments, we have observed state space searches completing in *less than half* the time when a good variable ordering is chosen. It is known to be an

NP-complete problem to *a priori* decide on the optimal ordering of variables [3]. However, good orderings can often be constructed using heuristics.

In our technique, each component place of a marking is binary encoded with the separate encodings being concatenated making a bit-vector describing the marking. The variables for the individual markings are stored with least significant bit first. We noted no difference between least significant bit versus most significant bit first. Therefore, in order to improve the encoding we look to the ordering of the places themselves. We have experimented with two different GSPN partitioning techniques.

First, we tried to cluster the places into logical partitions, based on the transitions of the GSPN. For each transition we construct a partition consisting of all places connected to it via incoming arcs. If any of those places are already elements of other partitions, then the partitions are combined. Once this operation is completed we are left with distinct sets of places which enable certain transition sets.

Breaking the net into partitions in this way turns out to be the close to the worst thing one can do. The aggregation of places enabling the same transitions does not lead to a suitable place ordering. One needs to *interleave* similarly affected places in the Petri net. This is true for the interleaving of row and column matrices in MTBDDs [11], and the same theory applies here. That is, finding two similar sub-nets and interleaving the corresponding places bit-vectors. To try to automate this process for Petri nets with no obvious structure, we grouped together those places whose number of incoming and outgoing arcs are equal. Our experience was that this resulted in a great improvement upon the previous technique, as discussed in Section 5.

4 State Space Exploration

Having found an efficient encoding method and having chosen a good variable ordering, we are in a position to conduct an efficient state space exploration. Typically symbolic state space exploration methods use breadth-first searches. The algorithm in Figure 2 shows how the set N of *new* states, initialised to the starting state, is used as a *frontier set*. That is, all those states whose successors have not been found. The states reachable in one-step from the states in N are determined by the function *findSuccessorStatesFrom* and stored in U, the *unexplored* state set. E accumulates all the *explored* states and, at each iteration, the frontier set is found by performing a set minus of U from E.

The most flexible part of this algorithm is the function *findSuccessorStatesFrom*. We can find all the states stored in N by a simple BDD traversal during which we maintain an array of values indicating whether the low child path or the high child path was taken, or whether we skipped a node, or number of nodes. If we do skip a number of nodes, then for each index, we need to set the corresponding

```
program Breadth-First Exploration
var
1       Set E, U, N = φ
begin
4       insert the initial state into N
5       while N ≠ φ do
        begin
6               let U = findSuccessorStatesFrom(N)
7               E = E ∪ N {add the newly explored states to the explored BDD}
8               N = U − E {find the frontier set}
        end
end.
```

Fig. 2. Breadth-First State Space Exploration Algorithm

path array value to a "don't care" value. Whenever we reach the true terminal, we use the path array to set the marking of the GSPN under analysis, which we store separately in memory. If the path array contains "don't care" values then all the permutations of the array are set one at a time.

For each permutation in N, we find and fire the enabled transitions. This process, encapsulated by the function *findAllSuccessorsOf*, changes the marking of the Petri net via transition firing. The new marking can then be inserted into the BDD denoted U (see Fig. 3) representing the set of states found at this depth of the state graph. These are the unexplored states, which will contribute to the frontier set in the next iteration of the state search algorithm.

The most time-consuming part of this algorithm is the collection of the successor states by the function *findSuccessorStatesFrom* from the algorithm in Figure 2. We could make use of the BDD *ite* (if-then-else) algorithm in [14] to insert each newly found successor state into the set of found successor states, but, in practice, this is very time consuming and impractical. A much more efficient way to add the new states is to realise that a new state simply means a new path in the BDD. This new path would simply be a worst-case $O(D)$ insertion, where D is the depth of the BDD. Unfortunately, since the BDDs are reduced we cannot simply add a new path to the existing BDD because arbitrarily adding new paths would violate this property.

We developed a new technique to avoid this problem. If the BDD is *un*reduced, we *can* simply add the nodes to build the new path from root to *true* terminal. This reduces the time complexity of adding a new state to $O(D)$. Unfortunately, unreduced BDDs lose their compactness. Parent nodes are still re-used in the unreduced BDDs, but isomorphic child sub-graphs are not re-used, thus leading to a blow-up in memory usage. We therefore cannot leave BDDs in their unreduced form indefinitely.

Our technique makes use of two BDDs for the successor state computation. The first is an ordinary reduced BDD, U, which simply stores the states found

```
procedure findSuccessorStatesFrom(BDD : U, BDD : T, Vertex : v, array of integers : path)
begin
1       if (value(v) == 0) return
2       elseif (value(v) == 1)
        begin
3               foreach (combination of path) do
4                findAllSuccessorsOf(path, T)
5               if (|T| >maxSize)
6                U = U ∪ T
7                T = false
        end
8       else
        begin
9               path[index(v)] = false
10              fillInDontCares(index(v), index(low(v)), path)
11              findSuccessorStatesFrom(U, T, low(v), path)
12              path[index(v)] = true
13              fillInDontCares(index(v), index(high(v)), path)
14              findSuccessorStatesFrom(U, T, high(v), path)
        end
end.
```

Fig. 3. Our algorithm for incremental reduction during state space exploration.

to that point. The second is an *unreduced* temporary BDD denoted T. The temporary BDD has states added to it by simply inserting the new paths to the *true* terminal. This is done until the temporary BDD reaches a user-defined upper-bound for the number of nodes used.

Once this upper-bound is reached, the *ite* algorithm is used to *union* T, the unreduced graph, with U, the reduced graph. Since the *ite* algorithm always returns a reduced BDD, this process has the effect of reducing the size of the BDD needed to store the newly explored states. Since the reduced BDD is normally several orders of magnitude smaller than its unreduced counterpart, this algorithm effectively allows the user to place an upper-bound on the amount of memory used for the exploration. In this way the maximum amount of memory required during the exploration is limited and the user can ensure that the problem will fit in the available memory. Naturally, the lower this limit the less time efficient the algorithm becomes.

The algorithm just described and shown in Figure 3 finds all the successors of all the states in N, some of which may have been already explored. In order to ensure that we do not re-explore them, they are removed at the next step in the algorithm in the *set-minus* operation on line 8. We implemented an enhancement to this. For each state found, we first check whether it is in the BDD for the explored states, which involves a worst-case $O(D)$ comparison. This is more efficient than re-inserting and then removing the states; in our experiments, this generally lead to a time improvement of approximately 25%.

A final enhancement we made was in the detection of enabled transitions. As described in Section 3, some place values are stored in the BDD, while others are computed. Transitions have places from either category in their input set. As the BDD is traversed, place values are determined in the order in which they are stored from root to terminal node. Some transitions may rely only on places stored near the root of the BDD, and other transitions may rely on computed place values derived from places stored near the BDD root. Our algorithm for detecting enabled transitions is applied at every recursive step of the function *findSuccessorStatesFrom*. When the recursion reaches the base case, a much smaller subset of the transitions only need be tested to see if they are enabled. This enhancement typically led to an average 70% reduction in the state space exploration time compared with our previous results.

During the exploration, all state-to-state rates are recorded in an MTBDD matrix. States are added in exactly the same way as above, also incrementally reducing the MTBDD used to store the states when it becomes too large.

Note that it is more common in symbolic state space search algorithms to find all the successor states of the frontier set in one computation using the transition relation BDD in the predicate transformation. For highly structured models this is an extremely efficient approach, but for more non-structured examples the state space sizes that can be explored are dramatically reduced. This is because the BDD depth for the transition relation is double that of BDD needed to store the state space. The transition relation BDD is thus normally a very large BDD and each predicate transformation takes an impractically long time.

We conclude this section with a short discussion of how vanishing state elimination is incorporated into our algorithm. So-called "vanishing" states occur in GSPN state spaces because enabled immediate transitions lead to states with a sojourn time of zero. We eliminate these states on-the-fly using a standard technique that makes use of a vanishing state depth-first stack [13]. The alternative approach of removing vanishing states at the end of a traversal involves a matrix inversion operation which, except in some specialised cases, is inefficient in MTBDD data structures [9]. This can be attributed to the fact that inversions typically involve a large amount of *fill-in*.

5 Experimental Results

The results below are based on our own implementation of a BDD package, using hashing to improve the time taken for each run of the *ite* algorithm.

Invariant-Based Encoding Function: The height of the BDD directly impacts on the time taken of any BDD-based manipulation and the encoding function determines the height of the BDD. We compared our invariant-based GSPN encoding function with a naive binary encoding, where the number of tokens on every place is converted to its unsigned binary form. The smallest improvement

upon this technique was a 70% reduction in the height of the corresponding BDD. The greatest improvement we found reduced the BDD height to just 20% of its original size.

Table 1. Comparative performance of classical and BDD storage methods. Dashed entries indicate that the process ran out out memory on a SPARC Station 5 with 256MB total memory.

No. of Phil.	No. of states	Classical	Binary		Invariant	
			Peak	Final	Peak	Final
5	81	256 KB	27 KB	12 KB	5 KB	1 KB
7	477	296 KB	198 KB	33 KB	29 KB	2.4 KB
10	6725	800 KB	-	-	398 KB	3.7 KB
12	39201	3.3 MB	-		2.2 MB	4.6 KB
15	551,613	44.9 MB	-	-	13.2 MB	3.3 KB
17	3,215,041	-	-	-	13.7 MB	9.1 KB
21	110 Million	-	-	-	13.7 MB	11.7 KB

State Space Exploration: Using BDDs for this purpose lead to a remarkable saving in space, allowing us to represent systems with as many as 10^8 states on a single workstation using 13.7MB of memory. While our BDD-based approach is slower for smaller state spaces, the speed of our implementation became comparable to a published classical technique (Knottenbelt *et al* [20]) at about the million-state mark on a single workstation. We are able to explore approximately 11 million states, and save the state to state rates for performance analysis in an MTBDD, in 6 hours on a single workstation using 4.5MB of memory. This is compared with a tool (DNAmaca [13]) using a probabilistic hashing-based technique which fails after 20 hours having explored only 5 million states in nearly 800MB of memory.

Table 1 shows results from the scalable Petri net model of the Dining Philosophers problem. It demonstrates how the use of BDDs in state space explorations, in combination with our computational invariant encoding technique, improves on the memory usage of both classical storage techniques as well as a naive binary encoding technique.

Partitions and Variable Ordering: We found that memory and time efficiency can be improved by ensuring that the Boolean variables encoding the state vectors are ordered felicitously by exploiting the structure of the GSPN. We found that the most efficient variable ordering technique *interleaved* the places of similar sub-areas in the GSPN.

The graph in Figure 4 shows some interesting general trends we have found in the course of our experimentation. Each bar corresponds to the number of states per second the technique allowed, given a particular model. For example, the leftmost bar shows that approximately 2 800 states per second were found using a random ordering on the Kanban [16] model with 3 kanbans. For each technique,

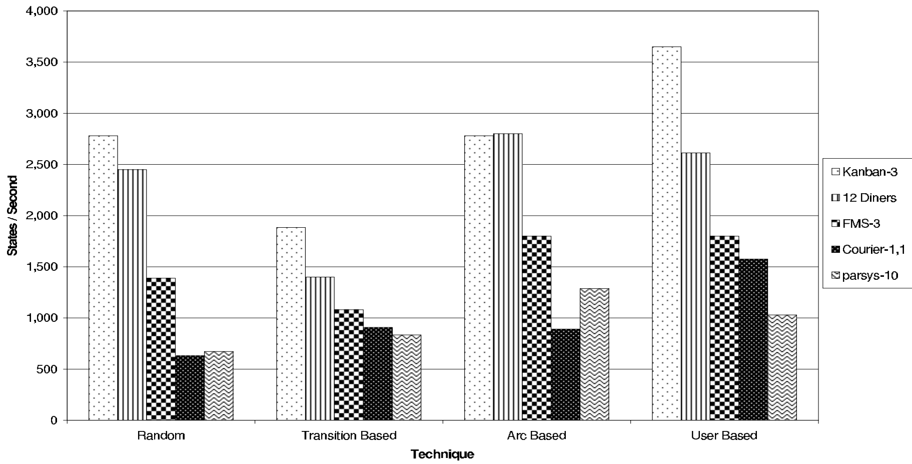

Fig. 4. A graph showing the relative performance of different variable ordering techniques in states per second.

the bars corresponding to each model are in order of decreasing regular structure. The most structured model is the Kanban model and the least structured is the parsys [19] GSPN. In general, there is a decrease in states per second as the regularity of the model structure decreases. This confirms the intuition that, generally, symbolic methods are less efficient on less regular GSPNs. The graphs also show that a user-chosen ordering method was better than the others in almost every case. However, our automated technique comes close to achieve similar throughputs as the user-chosen ordering. This shows that interleaving the places of individual subnets yields a better ordering than clustering places into partitions based on the transitions to which they are connected.

6 Conclusion

We have reported on our experience with symbolic methods for GSPN state space exploration. In this work we are not constrained to solving only regularly structured models, any model structure is suitable. We presented our GSPN encoding technique which brings together in a formal algorithm several heuristics found in existing work. We found that memory and time efficiency can be improved drastically by a well-chosen variable ordering, which, in our experience, involves *interleaving* the places of similar, but not necessarily the same, subnets in the GSPN. We presented results showing how our techniques improve over the memory and time efficiency of probabilistic hashing-based methods for large models on a single workstation.

BDD-based techniques move the focus from the search for space efficient methods, to a search for time efficient ones. In the future we plan to develop more

efficient MTBDD-based matrix manipulation algorithms so that the good memory efficiency of MTBDDs can be complemented with good time-savings. Furthermore, we intend to investigate the possibility of distributing our techniques over a network of workstations.

References

1. M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of Generalised Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.
2. A.S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. *LNCS 1639, Proc. 20th Intl. Conf. of the Application and Theory of Petri Nets*, pages 6–25, 1999.
3. B. Bollig and I. Wegener. Improving the variable ordering of OBBDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.
4. B.R. Haverkort, A. Bell, H.C. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. *Proc. 8th Intl Conf. Petri Nets and Perf. Models (PNPM'99)*, pages 12–21, 1999.
5. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computer Science*, C-35(8):677–691, August 1986.
6. R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
7. E. Pastor and J. Cortadella. Petri net analysis using boolean function manipulation. *Technical Report, University of Zaragoza*, 1993.
8. E. Pastor and J. Cortadella. Efficient Encoding Schemes for Symbolic Analysis of Petri nets. *Proc. Design, Automation and Test in Europe*, pages 790–795, 1998.
9. M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representations. *Formal Methods in System Design*, 10(2/3):149–169, 1997.
10. G. Ciardo, G. Luettgen, R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. *ICASE Report No. 99-50*, December 1999.
11. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. *In Proc. 3rd Int. Workshop on the Num. Solution of Markov Chains*, pages 188–207, 1999.
12. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Proc. 5th IEEE Symp. on Logic in Computer Science*, pages 428–439, 1990.
13. W. Knottenbelt. Generalised Markovian Analysis of Timed Transition Systems. Master's thesis, University of Cape Town, June 1996.
14. K.S. Brace, R.L. Rudell, R.E. Bryant. Efficient implementation of a BDD package. *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.
15. O. Coudert, C. Berthet, J-C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. *Applied Formal Methods For Correct VLSI Design*, 1:111–128, 1990.
16. P. Buchholz, G. Ciardo, S. Donatelli and P. Kemper. Kronecker operations and sparse matrices with applications to the solution of Markov models. *ICASE Report No. 97-66*, December 1997.
17. P.S. Kritzinger and F. Bause. *Stochastic Petri Nets: An Introduction to the Theory.* Verlag Vieweg, Wiesbaden, Germany, 1995.

18. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hactel, E. Macii, A. Pardo, F. Somenzi. Algebraic Decision Diagrams and their Applications. *In Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 188–191, 1993.
19. S. Caselli, G. Conte, P. Marenzoni. Parallel State Space Exploration for GSPN Models. *LNCS 935, Proc. 16th Intl Conference on the Application and Theory of Petri Nets*, pages 181–200, June 1995.
20. W.J. Knottenbelt, P.G. Harrison, M.A. Mestern, P.S. Kritzinger. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Performance Evaluation*, pages 127–148, February 2000.