An Efficient Hash Table Based Approach to Avoid State Space Explosion in History Driven Quasi-Static Scheduling[*]

Antonio G. Lomeña†, Marisa López-Vallejo†, Yosinori Watanabe‡ and Alex Kondratyev‡

†Electronic Engineering Department
ETSIT, Technical University of Madrid
Madrid, Spain
{lomena, marisa}@die.upm.es

‡Cadence Berkeley Laboratories
Berkeley, USA
{watanabe,kalex}@cadence.com

Abstract

This paper presents an efficient hash table based method to optimally overcome a new variant of the state space explosion which appears during the quasi-static task scheduling of embedded, reactive systems.

Our application domain is targeted to one-processor software synthesis, and the scheduling process is based on Petri net reachability analysis to ensure cyclic, bounded and undeadlocked programs. To achieve greater flexibility, we employ a dynamic, history based criterion to prune the search space. This makes our synthesis approach different from most existing code generation techniques.

Our experimental results reveal a significant reduction in algorithmic complexity (both in memory storage and CPU time) obtained for medium and large size problems.

1. Introduction

During the last years, the use of design methodologies based on formal methods has been encouraged as a means to tackle the increasing complexity in the design of electronic systems. However, traditional formal verification methods such as model checking or reachability analysis have the drawback of requiring huge computing resources.

In this paper we address the problem of software synthesis for embedded, reactive systems, using Petri nets (PNs) as our underlying formal model and reachability analysis as a way to formally obtain a valid quasi-static task schedule that ensures cyclic, buffer bounded, undeadlocked programs [3]. To overcome the state space explosion problem that is inherent to the reachability analysis, we use a dynamic, history based criterion that prunes the state space of uninteresting states for our application domain. This improvement comes at the cost of introducing special properties to the search procedure. These properties preclude the application of some of the techniques used to reduce the effect of the state explosion. This fact makes our synthesis approach different from most previous code generation approaches and poses new challenges to the problem of state explosion. In the paper, we focus on eliminating the repeated states that appear during the state space search.

The paper is organized as follows. Next section reviews previous work related to the reduction of the state space explosion. Section 3 states the problem definition. In section 4 we formally characterize the situations that generate marking repetitions. Section 5 analyzes the performance of the proposed solution and presents the results obtained for several experiments. Finally, we draw some conclusions.

2. Related work

The detection of symmetries as a method to mitigate the state explosion problem has been previously studied in [9]. This approach has the drawback of its computation complexity, which is exponential with respect to the graph size.

The reduction theory tries to overcome this problem by performing a controlled simplification of the system specification. In [8] a series of transformations that preserve liveness, safety and boundedness in ordinary PNs are introduced. However the situations modeled by these rules are somewhat simple. In section 4.1 we will present a more general approach that considers more frequent situations. Based on this new framework, it will be easy to extend the approach of [8] to tackle problems of greater complexity.

Partial order methods are other approach to avoid the state repetitions that have been successfully employed, for example, in the formal verifier SPIN [1]. Specifically, the persistent set theory [6] allows to verify properties that only depend on the final state of the system and not on the history of traversed states. Unfortunately this method cannot be applied to our case, as we will see in section 3. Similarly, the theory of unfoldings [5] has been mainly developed for safe PNs while our application uses unbounded nets.

A different approach consists in storing the state space in a memory efficient manner. Implicit enumeration methods such as *binary decision diagrams, BDDs*, or *interval decision diagrams, IDDs*, aim at this goal [11]. However,

[*]This work has been funded by CICYT under project CORE (TIC2000-0583-C02-02).

the construction of these graphs tends to be time consuming and their performance highly depends on the specific problem. This makes them unattractive for our applications.

Hash tables [7] are another way to manage the storage of the reached states. We will use them in our work due to their simplicity and high efficiency.

3. Problem definition

Our synthesis problem belongs to the same class that was introduced in [3]. We deal with a system specification composed of concurrent processes. Each process may have input and output ports to communicate with other processes or with the environment. Ports communicating with the environment are called *primary ports*. Primary ports written by the environment are called *uncontrollable* since the arrival of events to them is not regulated by the system. The communication through ports occurs through unidirectional, point-to-point channels.

One of the main steps of our software synthesis methodology is the generation of a task schedule, verifying that (1) it is a cyclic schedule, (2) it does not deadlock and (3) the schedule requires bounded memory resources (in other words, the cyclic execution of the program makes use of a finite number of buffers). The system is specified in a high level language similar to C but modified to allow communication operations. Processes are described as sequential programs that are executed concurrently. Later this specification is compiled to its underlying Petri net model. The scheduling process is carried out by means of reachability analysis for that Petri net [8, 3].

Traditionally, each of the processes of the system specification will be separately compiled on the target architecture. On the contrary, our synthesis process builds a set of *tasks* from the functional processes that are present in the starting specification. Each task is associated to one uncontrollable input port from the environment and performs the operations required to react to an event of that port. The novel idea is that those tasks may differ from the user specified processes. The compiler applies its transformations on each of these tasks, therefore optimizing the code that must be executed in response to an external event.

Figure 1.A sketches this idea. It depicts two processes A and B, each of them reading data from its corresponding ports. Both processes communicate between them by means of the channel C.

However, the data that process B reads from port IN_B is processed independently of the data read from channel C. Hence, an efficient scheduling algorithm could reorder the source code to construct the threads 1 and 2 depicted in figure 1.B. In this way, architecture specific optimizations will be performed on these separate code segments or tasks.

Next sections introduce the fundamentals of the scheduling approach used in our software synthesis methodology.

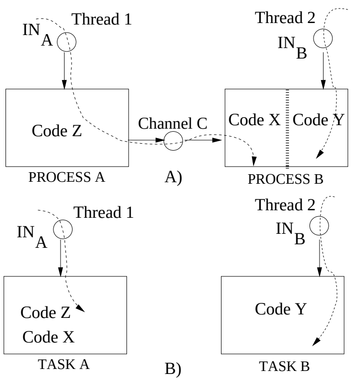

Figure 1. Code generation

3.1. Petri net fundamentals

A *Petri net, PN* is a tuple (P, T, F, M_0) where P and T are sets of nodes of *place* or *transition* kind respectively [3]. F is a function of $(P \times T) \cup (T \times P)$ to the set of non-negative integers. A *marking* M is a function of P to the set of non-negative integers. We call *number of tokens* of p in M, where p is a *place* of the net, to the value of the function M for p, that is, $M[p]$. If $M[p] > 0$ the place is said to be marked. Intuitively, the marking of all the places of the net constitutes the system state.

A PN can be represented as a bipartite directed graph, so that in case the function $F(u, v)$ is positive, there will exist an edge $[u, v]$ between such graph nodes u and v. The value $F(u, v)$ is the edge weight. A transition t is enabled in a marking M if $M[p] \geq F(p, t) \quad \forall p \in P$. In this case, the transition can be fired in the marking M, yielding a new marking M' given by $M'[p] = M[p] - F(p, t) + F(t, p) \ \ \forall p \in P$.

A marking M' is *reachable* from the *initial marking* M_0 if there exists a sequence of transitions that can be sequentially fired from M_o to produce M'. The set of markings reachable from M_0 is denoted $\mathcal{R}(M_0)$. The *reachability graph* of a PN is a directed graph in which $\mathcal{R}(M_0)$ is a set of nodes and each edge $[M, M']$ is a PN transition, t, such that the firing of t from marking M gives M'.

A transition t is called *source* if $F(p, t) = 0$ for all p of P. A pair of non source transitions t_i and t_j are in *equal conflict* if $F(p, t_i) = F(p, t_j) \ \ \forall p \in P$. An *equal conflict set, ECS* is a group of transitions that are in equal conflict.

A *place* is a *choice* if it has more than a successor transition. If all the successor transitions of a choice *place* belong to the same ECS, the *place* is called *equal choice place*. A PN is *Equal Choice* if all the choice *places* are *equal*.

A choice *place* is *unique* if in every marking of $\mathcal{R}(M_0)$ that *place* cannot have more than one enabled successor transition. A *unique-choice Petri net, UCPN,* is one in which all the choice *places* are either unique or equal. The PNs obtained after compiling our high-level specification present unique-choice ports.

3.2. Conditions for valid schedules

A schedule for a given PN is a directed graph where each node represents a marking of the PN and each edge joining two nodes stands for the firing of an enabled transition that leads from a marking to the other. A valid schedule has five properties. First, there is a unique root node, corresponding to the starting marking. Second, for every marking node v, the set of edges that start from v must correspond to the transitions of an enabled ECS. If this ECS is a set of source transitions, v is called an *await node*. Third, the nodes v and w linked by an edge t are such that the marking w is obtained after firing transition t from marking v. Fourth, each node has at least one path to one await node. Fifth, each await node is on at least one cycle.

As a consequence of these properties, valid schedules are cyclic, bounded and undeadlocked.

3.3. Stopping conditions in the state exploration

The existence of source transitions in the PN produces an infinite reachability graph since the number of external events that can be generated is unlimited. Therefore, the design space that is explored must be pruned. Our implementation employs two types of stopping criteria: static and dynamic. The static criterion consists in stopping the search whenever the buffer size of a given port exceeds the bounds established by the designer. Besides, we employ a dynamic criterion based on two steps (see [3] for further details):

1. First, some static port bounds, called *place degrees*, are imposed. The degree of a place p is the maximum of **(a)** the number of tokens of p in the initial marking and **(b)** the maximum weight of the edges incoming to p, plus the maximum weight of the edges outgoing from p minus one

   A place of a PN is saturated when its token number exceeds the degree of the place.

2. During the reachability graph construction, a marking will be discarded if it is deemed as *irrelevant*. A marking w is irrelevant if there exists a predecessor, v, such that for every place p of the marking $M(v)$ of v, it is verified that **(1)** p has at least the same number of tokens in the marking $M(w)$ (and possibly more) and **(2)** if p has more tokens in $M(w)$ than in $M(v)$, then the number of tokens of p in $M(v)$ is equal or greater than the degree of p.

3.4. Scheduling algorithm

The scheduling algorithm is based on a recursive approach using two procedures (*proc1* and *proc2*) that alternatively call each other. Procedure *proc1* receives a node representing a marking and iteratively explores all the enabled ECSs of that node. Exploring an ECS is done by calling procedure *proc2*. This one iteratively explores all the transitions that compose that ECS. The exploration of a transition is done by computing the marking node (M_{new}) produced after firing it and calling *proc1* for that M_{new}.

The exploration of a path is stopped in procedure *proc2* when one of the following three criteria holds:

- Finding an *Entry Point (EP)* for the M_{new} (an EP is a predecessor of M_{new} with exactly the same marking).
- The M_{new} is irrelevant with regard to a predecessor.
- The marking of M_{new} exceeds the maximum number of tokens specified by the user for some of the ports.

The first criterion may lead to a valid schedule. If, after returning from the recursive calls we find that the schedule is not valid we would re-explore the path again. To exhaustively explore the subgraph that exists after a given node *v* all the ECSs enabled in *v* must be explored. This is controlled by procedure *proc1*.

The latter two criteria do not produce a valid schedule. Hence, after returning from the recursive call, a re-exploration of new paths will be done providing that there are still paths that have not been exhaustively explored.

4. Types of marking repetitions

A key property of PNs is their determinism. A given marking will always enable the same set of transitions. Hence, if two equal markings are obtained in different instants of the search, they will generate the same reachability graphs. The subgraph of the reachability graph induced by the second marking will be the same as that of the first.

Property (1) of a valid schedule (see section 3.2) forces that any time a new marking, M_i, is generated, the algorithm checks whether there exists some predecessor marking, M_j, such that $M_j = M_i$. This check is restricted to those markings *causally* related in the reachability graph. The detection of identical markings in conflict branches (those that are generated after the corresponding firings of the transitions that compose an ECS) would avoid the replication of scheduling subgraphs previously calculated.

According to their nature, we have identified two possible sources of repetitions. We explain them in detail in the following sections.

4.1. Structural sources of repetition

The property defining these repetitions is that they are produced by some characteristic structures of the PN that specifies the system. Once the reachability graph is developed, the repeated subgraphs are simultaneously present in the valid scheduling.

Next, we will introduce several definitions that will help us to characterize the origin of the repetition sources:

Definition 1 (Equivalent edge set, EES) *Two edge sets are equivalent if a bijective function can be established between the two sets, such that for every edge of one set there exists one edge in the other set having the same weight.*

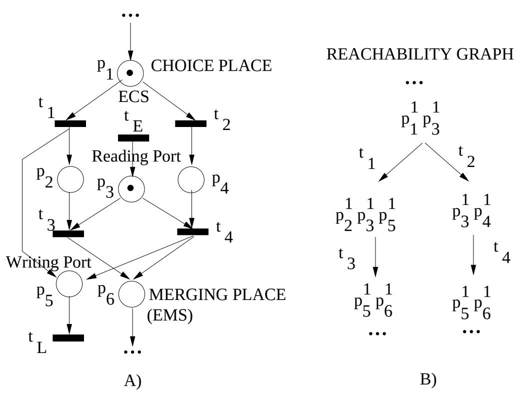

Figure 2. Example of a structure that generates marking repetitions (a) and its associated reachability graph (b)

Definition 2 (Equal merging set, EMS) *A set of places in a PN constitutes an EMS if **(a)** every place is connected to the same predecessor transitions, and **(b)** the set of edges that every place uses for those connections is equivalent to the set used by any other place of the EMS.*

Definition 3 (Alternative paths, AP) *Two paths are alternative when **(a)** both paths access to the same ports and **(b)** the set of edges that access to a given port through a path is equivalent to the set of edges that the other path uses to access that port.*

We are interested in repeated markings appearing on conflict branches. Since the reachability graph begins with a unique root node, such branches must have a common predecessor. The branching under this predecessor corresponds to the decision to fire a specific transition of an ECS. However, the existence of a choice place is not enough to produce a repetition. It is necessary that the branches in conflict generate equal markings. A PN structure that can exhibit such behavior is that constituted by:

- An ECS, which starts a series of execution paths.
- An EMS, on which the previous paths converge.
- The paths generated between the ECS and the EMS must be *alternative paths*.

Figure 2.A shows an example where there exists an ECS that allows to choose between firing t_1 or t_2. As can be seen, both paths are alternative, since they access the same ports and with the same weights. Figure 2.B shows the reachability graph corresponding to the net of figure 2.A. Each marking has been denoted with the *places* that contain tokens in that marking, specifying the number of tokens contained within each *place* as a super-index.

A problem with the structures we have defined so far, is that we cannot guarantee that every alternative path can be traversed when there are reading and writing ports that communicate with other processes. To ensure this, we will introduce the concept of equivalent paths.

Definition 4 (Ports depending on a writing port) *Given a writing port, p_e, belonging to process P_i, the set of ports that depend on p_e is formed by all the ports of P_i such that the transition that accesses to each of these ports is a successor of the transition that writes on p_e.*

Intuitively, the firing of a transition that writes on the port p_e of P_i affects the number of *tokens* that the depending ports can contain, since the writing access on p_e can change the evolution of processes different to P_i. These processes can vary the marking on ports dependent on p_e. We will introduce now a new definition to deal with this situation.

Definition 5 (Equivalent paths, EP) *Given two alternative paths C_1 and C_2, both paths are equivalent when for each writing port that is present in the paths, the set of ports depending on p_e is coincident in both paths.*

Given an initial marking, the condition of equivalent paths ensures that if one of the paths can be traversed, the other path will also be able to be traversed. This is due to the fact that for each writing port the depending ports are the same in both paths. Hence, there is no possibility that the writing operation performed on a port alters the marking of successor ports of one path and not of the other.

The previous concepts lead to the following theorem:

Theorem 1 *Given a PN composed of a set of equivalent paths starting from an ECS and ending by means of an EMS, the reachability tree will have repeated subgraphs in the branches corresponding to the firing of that ECS.*

The knowledge of the causes that produce the marking repetitions allows to develop ad-hoc techniques to avoid them. Thus, we could devise an algorithm to detect the structures defined by theorem 1. This technique would be an intermediate approach between the detection of general symmetries and the application of reduction rules. In this paper, though, we will employ a more general and less time consuming approach based on hash tables.

Conditionals are a possible structure that would produce structural marking repetitions when they verify the conditions of theorem 1. A simple example is shown in figure 3, where the last parameter of each port access indicates the number of tokens to read or write.

4.2. Dynamic sources of repetition

In this case, the generation of repeated markings is due to the search method employed. These repetitions appear in searches performed in depth-first order. The search explores all the enabled transition firing permutations, even though these lead to the same marking. Unlike the previous case, the repetitions do not belong to the same schedule, but to different scheduling solutions that are tried during the schedule construction.

The sleep set theory, first proposed by Godefroid in [6], has been successfully applied to eliminate this kind of repetitions. However, the sleep set theory assumes that a state

```
PROCESS proc1(In_DPORT start,
  In_DPORT in, Out_DPORT out) {

  int N, x, y, z;
  while (1) {
    READ_DATA(start, &N, 1);
    if (N>1000) {
      READ_DATA(in, &x, 1);
      y = 10 * x;}
    else {
      READ_DATA(in, &z, 1);
      y = z*z;}
    WRITE_DATA(out, &y, 1);}}
```

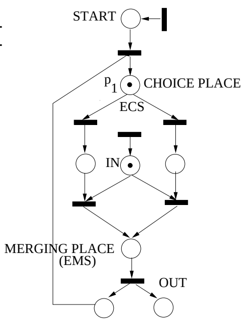

Figure 3. FlowC code and PN for a source of structural repetitions

that did not lead to a successful depth-first search will always fail when reached again in a new search. The irrelevance condition makes this assumption no longer valid, hence precluding a direct application of this technique. How to efficiently adapt the sleep set theory to take into account the history dependence in our search method still constitutes an open problem.

5. Characterization of the implemented solution

Due to the special constraints that our synthesis application presents, the approach we have finally implemented to avoid the repetition of states is based on hash tables. Hash tables are a much less complex method to store old reached states than implicit representation methods. But, more importantly, their performance is much less problem dependent than that of BDDs or IDDs.

The main problem of hash tables is the existence of collisions. When the table is finite, two or more indexes can be mapped to the same table location, originating a collision. Several solutions can be applied, depending on the kind of verification to be done. Since we need to store all the generated states, we must discard the use of recent methods such as the *bit-state hashing* [4] used in the SPIN tool [1] or *hash compaction* [10], in which collisions produce the loss of states. In order to tackle the collisions we will employ hash tables with subtables implemented by means of *chained lists* [7].

5.1 New scheduling algorithm

In our application, the concept of predecessor node plays a key role. When a new marking is generated it is important to know whether it is irrelevant, what forces to keep information about its predecessors. Thus, it is essential to maintain the scheduling graph structure. In other words, it is not sufficient to keep a set of previously reached states. On the contrary, the order relations among the markings must be stored. Hence, we still keep the current portion of the reachability graph that constitutes a valid schedule.

The scheduling algorithm after including the hash table remains basically the same. The only difference lies in the stopping conditions presented in section 3.3. Now, a new stopping criterion must be considered. The search will also be stopped whenever a new marking node *v* has the same marking as a previously reached node *w*. The reason for this is that the schedule from *v* will be identical to that from *w*. Hence node *v* will be merged to node *w*.

In the next sections we will devise a process model that will allow us to assess the amount of repeated states. Then we will perform some experiments to show the practical benefits, in terms of algorithmic complexity reduction, obtained after the incorporation of the hash tables.

5.2. Modeling the number of repeated states

Our implementation is based on a hash table with a chaining scheme to handle the occurrence of collisions. We apply a hash function based on the *xor* logic function to the key string that defines each marking. This function is inspired on that appearing in [2]. When the hash table is filled to a 75% of its capacity, it is enlarged one order of magnitude. The amount of repetitions produced by the cyclic execution of a process, P_o, can be assessed analyzing each single connected component (SSC) of P_o. For the moment, we will assume there is only a conditional inside the SCC. If we call $p_E^{P_o}$ the place that starts the cyclic execution of P_o (*entry point* of P_o), we can consider that the conditional partitions the SCC in the following sections:

Predecessor path (L_{pred}): from $p_E^{P_o}$ to the choice place.

Successor path (L_{succ}): from the merging place to $p_E^{P_o}$.

Branch paths (L_{branch}): those places belonging to each of the branches of the conditional. Given a conditional formed by N_B branches, the places belonging to a branch path will not belong to any of the other branch paths of the same conditional.

Assuming that the application execution starts in process P_o with the marking of $p_E^{P_o}$, M_0, the number of repeated states will be equal to the number of states produced by:

1. Traversing the successor path, plus
2. Scheduling a cycle of each of the processes reading from ports that were written during step 1 or during the traversal of the paths L_{branch}. The entry point for one of these cycles is the port used by the process to communicate with P_o

After performing steps 1 and 2, the schedule will have produced a cycle with respect to the starting marking M_0. These concepts are summarized in the following equation:

$$S_{P_o} = L_{pred} + N_B \times L_{succ} + \sum_{i=1}^{N_B} L_{branch_i} \qquad (1)$$

where S_{P_o} is the number of states contained in the cyclic schedule of process P_o and N_B is the number of conditional branches. The number of repeated states of this schedule is equal to $(N_B - 1) \times L_{succ}$.

	System 1			System 2			System 3			System 4			System 5		
	WO	W	I%	WO	W	I%	WO	W	I%	WO	W	I%	WO	W	I%
Num. places	10			26			34			34			34		
Num. transitions	10			24			29			30			30		
Num. states	19	14		309	257		694	356		2056	702		2987	1016	
Num. repeated states	5	0		52	0		338	0		1354	0		1971	0	
CPU time (sec.)	≈ 0	0.01	-	0.25	0.21	16	1.03	0.6	42	3.11	1.03	67	5.43	1.74	68
Memory (KB)	49.7	60.23	-21	331	337	-2	614.4	542	12	1204	843	30	1687	1132	33

Table 1. Scheduling results

If there were more than one conditional in the SCC, the equation (1) should be applied to each of the subsections in which those conditionals would divide the SCC. If P_o were composed of several SCCs, the equation (1) should be applied to all of them. Also, as the different paths are traversed it may be necessary either to provide or consume the necessary tokens of the ports accessed during the traversal. As we are seeking cyclic schedules, the processes that are involved in this token transactions must return to their original state. This means that a cycle of these processes must also be executed. All the produced states would add to the total number of computed repeated states.

5.3. Experimental results

Our scheme has been tested with several practical examples. Table 1 offers the scheduling results. Columns labelled with *WO* were obtained without employing the hash table. Columns tagged with *W* show results with the hash table. The columns labelled with *I* show the percentage improvement both in memory and CPU time obtained when hash tables are used. All the examples are variations of a producer-consumer system targeted to multi-media applications such as video decoding. The processes communicate through FIFO buffers. System 1 is composed of two processes with a producing/consuming rate (*pcr*) equal to two. System 2 incorporates a controller to better manage the synchronization among processes, and the *pcr* is equal to 48 in this case. System 3 has the same *pcr* but includes some additional processes that perform filtering functions on the data the producer sends to the consumer. All these examples contain a unique conditional, composed of two equivalent branches, that generates structural repetitions. System 4 is equivalent to System 3 but including two conditionals in the SCC, each one with two equivalent branches. Finally, System 5 is identical to System 4 except that its *pcr* is equal to 70.

In System 1 the memory overhead introduced by the hash table is larger than the reduction obtained by avoiding the state repetition. This is so because the small size of this example produces few states to explore. However, the rest of examples show that as the number of conditionals increases and as the difference between the producing and the consuming rates raises, the benefits in CPU time and consumed memory obtained when using hash tables soar up.

6. Conclusions

In this paper we have addressed the problem of reducing the number of repeated states that are generated during quasi-static task scheduling in a software synthesis methodology for one-processor, embedded, reactive systems.

We have studied and formally characterized the main causes of the state repetitions. Two prime sources of repetitions were found, with structural and dynamical origin and we explored different solutions to prevent their occurrence. Most importantly, we have shown that a scheme based on the use of efficient hash tables provides the best way to completely eliminate the structural sources of repetitions.

The experimental results we have obtained demonstrate the usefulness of our technique, yielding outstanding reductions in the CPU time and memory consumed by the scheduling algorithm for large-scale, real world problems.

References

[1] http://spinroot.com/spin/whatispin.html.

[2] R. S. A. V. Aho and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] J. Cortadella et al. Task generation and compile-time scheduling for mixed data-control embedded software. In *Design Automation Conference*, pages 489–494, 2000.

[4] J. Eckerle and T. Lais. New methods for sequential hashing with supertrace. Technical report, Institut fur Informatik, Universitat Freiburg, Germany, 1998.

[5] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *TACAS'96*, 1996.

[6] P. Godefroid. *Partial Order Methods for the Verifi cation of Concurrent Systems. An Approach to the State-Explosion Problem*. PhD thesis, Universite de Liege, 1995.

[7] R. Jain. A comparison of hashing schemes for address lookup in computer networks. *IEEE Trans. on Communications*, 4(3):1570–1573, October 1992.

[8] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, April 1989.

[9] K. Schmidt. Integrating low level symmetries into reachability analysis. In *TACAS*, 2000.

[10] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *GI/ITG/GME Workshop*, 1996.

[11] K. Strehl et al. *FunState*– an internal design representation for codesign. *IEEE Trans. on VLSI Systems*, August 2001.