Efficient Symbolic State-space Construction for Asynchronous Systems*

Gianfranco Ciardo[1], Gerald Lüttgen[2], and Radu Siminiceanu[1]

[1] Department of Computer Science, College of William and Mary, Williamsburg, VA 23187, USA, {ciardo,radu}@cs.wm.edu
[2] ICASE, NASA Langley Research Center, Hampton, VA 23681, USA, luettgen@icase.edu

Abstract. Many techniques for the verification of reactive systems rely on the analysis of their reachable state spaces. In this paper, a new algorithm for the symbolic generation of the state spaces of *asynchronous* system models, such as Petri nets, is developed. The algorithm is based on previous work that employs *Multi-valued Decision Diagrams* for efficiently storing sets of reachable states. In contrast to related approaches, however, it fully exploits *event locality*, supports intelligent *cache management*, and achieves faster convergence via advanced *iteration control*. The algorithm is implemented in the Petri net tool SMART, and runtime results show that it often performs significantly faster than existing state-space generators.

1 Introduction

Many state-of-the-art verification techniques rely on the *automated construction of the reachable state space* of the system under consideration. Unfortunately, state spaces of real-world systems are usually very large, sometimes too large to fit in a computer's memory. One contributing problem is the concurrency inherent in reactive systems, such as those specified by *Petri nets* [18]. Consequently, many research efforts in *state-exploration techniques* concentrated on the efficient exploration and storage of large state spaces. These may be categorized according to whether sets of states are stored explicitly or symbolically.

Explicit techniques represent state spaces by trees, hash tables, or graphs, where each state corresponds to an entity of the underlying data structure. Thus, the memory needed to store the state space of a system is linear in the number of the system's states, which in practice limits these techniques to fairly small systems having at most a few million states.

Symbolic techniques allow one to store reachability sets in sublinear space. They often use *Binary Decision Diagrams* (BDDs) as a data structure for efficiently representing Boolean functions [1], into which state spaces may be

* This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681, USA.

mapped. The advent of BDD-based techniques pushed the manageable sizes of state spaces to about 10^{20} states [4]. In the Petri net community, BDDs were applied by Pastor et al. [19, 20], Varpaaniemi et al. [23], and others for the generation of the reachability sets of Petri nets. Recently, symbolic state-space generation for Petri nets has been significantly improved [17] by considering *Multi-valued Decision Diagrams* (MDDs) [15] instead of BDDs. MDDs essentially represent integer functions and allow one to efficiently encode the state of an entire subnet of a Petri net using only a single integer variable, where the state spaces of the subnets are built by employing traditional techniques. Experimental results reported in [17] show that this approach enables the representation of even larger state spaces of size 10^{60} and even 10^{600} states for particularly regular nets. However, the time needed to generate some of these state spaces ranges from minutes for the *dining philosophers* [20], with 1000 philosophers, to hours for the *Kanban system* [7], with an initial token count of 75 tokens. Thus, state-space generation shifts from a *memory-bound* to a *time-bound* problem.

The objective of this paper is to improve on the time efficiency of symbolic state-space generation techniques for a particular class of systems, namely *asynchronous systems*. Our approach aims at exploiting the concept of *event locality* inherent in such systems. In Petri nets, for example, event locality means that only those sub-markings belonging to the subnets affected by a given transition need to be updated when the transition fires. Whereas event locality was investigated in explicit state-space generation techniques [6], it has been largely ignored in symbolic techniques. Only the MDD-based approach presented in [17] touches on event locality, but it exploits this concept only superficially. In particular, this approach does not support direct jumps to and from the part of the MDD corresponding to the sub-markings that need to be updated when a transition fires. The present paper develops a new algorithm for building the reachable state spaces of asynchronous systems. Like [17], it uses MDDs for representing state spaces; unlike [17], it fully exploits event locality. Moreover, it introduces an intelligent mechanism for *cache management* and also achieves faster convergence by firing events in a specific, predefined order. The new algorithm is implemented in the tool SMART [6]. When applied to a suite of well-known Petri net models, it proves to be significantly faster than the one presented in [17], while inducing only a small overhead regarding space efficiency. The algorithm can be employed immediately for verifying *safety properties*, such as the absence of deadlocks. Moreover, the developed MDD manipulation techniques may also provide a basis for implementing MDD-based model checkers [10].

2 Structured State Spaces and MDDs

We choose to specify finite-state asynchronous systems by Petri nets [18]; however, the concepts presented here are not limited to this choice. Thus, we interchangeably use the notions *net* and *system*, *subnet* and *sub-system*, *transition* and *event*, *marking* and *(global) state*, as well as *sub-marking* and *local state*.

Consider a Petri net with a finite set $\mathcal{P}$ of places, a finite set $\mathcal{E}$ of events, and an initial marking $s_0 \in \mathbb{N}^{|\mathcal{P}|}$. The semantics of Petri nets defines how the *firing* of an event e can move the net from some state s to another state s'. We denote the set of successor states reachable from state s via event e by $\mathcal{N}(e,s)$. If $\mathcal{N}(e,s) = \emptyset$, event e is *disabled* in s; otherwise, it is *enabled*. For Petri nets, $\mathcal{N}$ is essentially a simple encoding of the input and output arcs; thus, $\mathcal{N}(e,s)$ contains at most one element. For other formalisms, however, $\mathcal{N}(e,s)$ might contain several elements. We are interested in exploring the set $\mathcal{S}$ of reachable states of the considered net. $\mathcal{S}$ is formally defined as the smallest set that (i) contains s_0 and (ii) is closed under the "one-step reachability relation," i.e., if $s \in \mathcal{S}$, then $\mathcal{N}(e,s) \subseteq \mathcal{S}$, for any event $e \in \mathcal{E}$.

As in [17], our encoding of the state space of a Petri net requires us to partition the net into K subnets by splitting its set of places $\mathcal{P}$ into K subsets. This implies a partition of a *global state* s of the net into K *local states*, i.e., s has the form $(s_K, s_{K-1}, \ldots, s_1)$; the "backwards" numbering will prove to be a reasonable convention when representing global states using MDDs. The partition of $\mathcal{P}$ must satisfy a fundamental *product-form requirement* [8] which demands for function $\mathcal{N}$ to be written as the cross-product of K local functions, i.e., $\mathcal{N}(e,s) = \mathcal{N}_K(e,s_K) \times \mathcal{N}_{K-1}(e,s_{K-1}) \times \cdots \times \mathcal{N}_1(e,s_1)$, for all $e \in \mathcal{E}$ and $s \in \mathcal{S}$. Furthermore, in practice, each subnet should be small enough such that its reachable *local state space* $\mathcal{S}_k = \{s_{k,0}, s_{k,1}, \ldots, s_{k,N_k-1}\}$ can be efficiently computed by traditional techniques, where $N_k \in \mathbb{N}$ is the number of reachable states in subnet k. Note that this might require the explicit insertion of additional constraints to allow for the correct computation of $\mathcal{S}_k$ in isolation, or one may use a small superset of $\mathcal{S}_k$ obtained by employing *p-invariants* [18]. For all the examples we present, the computation of the local state spaces requires negligible time. Once $\mathcal{S}_k$ has been built, we can identify it with the set $\{0, 1, \ldots, N_k - 1\}$. Moreover, a set $\mathcal{S}$ of global states can then be encoded by the *characteristic function* $f_{\mathcal{S}} : \{0, \ldots, N_K - 1\} \times \cdots \times \{0, \ldots, N_1 - 1\} \longrightarrow \{0, 1\}$ defined by $f_{\mathcal{S}}(s_K, s_{K-1}, \ldots, s_1) = 1$ if and only if $(s_K, s_{K-1}, \ldots, s_1) \in \mathcal{S}$.

Multi-valued Decision Diagrams. *Multi-valued Decision Diagrams* [15], or MDDs, are data structures for representing integer functions of the form

$$f : \{0, \ldots, N_K - 1\} \times \cdots \times \{0, \ldots, N_1 - 1\} \longrightarrow \{0, \ldots, M - 1\}$$

where $K, M \in \mathbb{N}$ and $N_k \in \mathbb{N}$, for $K \geq k \geq 1$. When $M = 2$ and $N_k = 2$, for $K \geq k \geq 1$, function f is a Boolean function, and MDDs coincide with the better known *Binary Decision Diagrams* (BDDs) [1, 2]. We use the special case $M = 2$ to store the characteristic functions of the previous section.

Traditionally, integer functions are often encoded by *value tables* or *decision trees*. Figure 1, left-hand side, shows the decision tree of the minimum function *min(a,b,c)*, where the variables a, b, and c are taken from the set $\{0, 1, 2\}$. Hence, $K = 3$ and $N_1 = N_2 = N_3 = M = 3$. Each internal node, which is depicted by an oval, is labeled by a variable and has arcs directed towards its three children. The branch labeled with i corresponds to the case where the variable of the

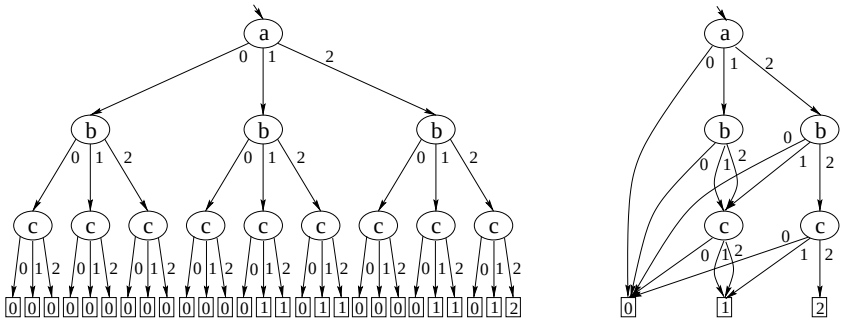

Fig. 1. Representation of *min(a,b,c)* as decision tree (left) and as MDD (right)

node under consideration is assigned value i. Moreover, all paths through the tree have the same *variable ordering*, which in our example is $a < b < c$. Leaf nodes, depicted by squares, are labeled by either 0, 1, or 2. Each path from the root to a leaf node corresponds to an assignment of the variables to values. The value of the leaf in a given path is the value of the function with respect to the assignment for this path.

An MDD is a representation of a decision tree as *directed acyclic graph*, where identical subtrees are merged. More precisely, MDDs are *reduced* decision trees which do not contain any *non-unique* or *redundant* node: a node is non-unique, if it is a replica of another node, and redundant, if all its children are identical. Together with a fixed variable ordering, these two requirements ensure that MDDs provide a *canonical* representation of integer functions [15]. Note that the elimination of redundant nodes implies that arcs can skip levels, e.g., the arc labeled with 0 connecting node a to leaf node 0 in Fig. 1, right-hand side, skips levels b and c. Hence, the value of the function is 0, whenever a is 0. For many functions, MDD representations can be exponentially more compact than their corresponding value tables or decision trees. However, the degree of compactness depends on the considered function and the chosen variable ordering.

Data Structures for MDDs. We organize MDD nodes in levels ranging from K at the top to 1 at the bottom. Additionally, there is the special level 0 which contains either or both leaf nodes corresponding to the values 0 and 1, indicating whether a state is reachable or not. The addresses of the nodes at a given level are stored within a hash table, to provide fast access to them and to simplify detection of non-unique nodes. Hence, we have K hash tables which together represent an MDD; we also refer to this data structure as *unique table*. Note that we could as well use a single unique table for representing MDDs, but this would require us to store the level for each node; furthermore, the level-wise organization of our data structures will prove very useful below. Each node at level k consists of an array of N_k node addresses, which contains the arcs to the children of the node. Since we enforce the reducedness property, we use the value of this array to compute the hash value of the node. In the following, we let

Table 1. *Union* operation on MDDs

```
Union(in p : mddAddr, in q : mddAddr) : mddAddr
 1. if p = ⟨0,1⟩ or q = ⟨0,1⟩ return ⟨0,1⟩;          • deal with the base cases first
 2. if p = ⟨0,0⟩ or p = q return q;
 3. if q = ⟨0,0⟩ return p;
 4. k ⇐ Max(p.lvl, q.lvl);                  • maximum of the levels of p and q
 5. if LookUpInUC(k, p, q, r) then return r;          • found in the union cache
 6. r ⇐ CreateNode(k);          • otherwise, the union needs to be computed in r
 7. for i = 0 to N_k − 1 do
 8.     if k > p.lvl then u ⇐ Union(p, q→dw[i]);      • p is at a lower level than q
 9.     else if k > q.lvl then u ⇐ Union(p→dw[i], q);  • q is at a lower level than p
10.     else u ⇐ Union(p→dw[i], q→dw[i]);          • p and q are at the same level
11.     SetArc(r, i, u);                              • make u the i-th child of r
12. r ⇐ CheckNode(r);                                 • store r in the unique table
13. InsertInUC(k, p, q, r);                 • record the result in the union cache
14. return r;
```

mddNode denote the type of nodes and *mddAddr* the type of addresses of nodes. For convenience, we write $\langle lvl, ind \rangle$ for the node q stored in the lvl-th unique table at position ind, and $q{\rightarrow}dw[i]$ for the i-th child of q. Finally, we use nodes $\langle 0,0 \rangle$ and $\langle 0,1 \rangle$ to indicate the Boolean values 0 and 1 at level 0, respectively.

The *Union* Operation on MDDs. An essential operation for generating reachable state spaces is the binary *union* on sets. Since in our context all sets are represented as MDDs, an algorithm is needed which takes two MDDs as parameters and returns a new MDD, representing the union of the sets encoded by its arguments. This algorithm, which is very similar to the one used in [17], is shown in Table 1. It recursively analyzes the argument MDDs when descending from the maximum level k of the argument MDDs to the lowest level 0 and builds the result MDD when finishing the recursions by ascending from level 0 to level k. Note that the maximum of the levels of the argument MDDs is the highest level the result MDD can have.

The base cases of the recursive function *Union* are handled in Lines 1–3, where the MDDs $\langle 0,0 \rangle$ and $\langle 0,1 \rangle$ encode the *empty set* and the *full set*, respectively. If $k > 0$, a cache – the so-called *union cache* – is used to check whether the union of the arguments p and q has been computed previously. If so, the result stored in the cache is returned. Otherwise, a new MDD node at level k is created, whose i-th child is determined by recursively building the union of the i-th child of p and the i-th child of q, for all $0 \leq i < N_k$ (cf. Lines 7–11). However, one needs to take care of the fact that some child might not be explicitly represented, namely if it is redundant (cf. Lines 8 and 9). Finally, to ensure that the resulting MDD is reduced, node r is checked by calling function *CheckNode*(r). If r is redundant, then *CheckNode* destroys r and returns r's child, and if r is equivalent to another node r' having the same children, then *CheckNode* destroys r and returns r'. Otherwise, *CheckNode* inserts node r in the unique table and returns it. Note that the algorithm in Table 1 can be easily adapted for computing other binary operations, such as intersection, by modifying Lines 1–3.

Table 2. Iterative state-space generation

```
MDDgeneration(in m : array[1,...,K] of int) : mddAddr
 1. for k = 1 to K do ClearUT(k);                          • clear unique table
 2. q ⇐ SetInitial(m);          • build the MDD representing the initial state
 3. repeat                                  • start state-space exploration
 4.     for k = 1 to K do ClearUC(k);                      • clear union cache
 5.     for k = 1 to K do ClearFC(k);                      • clear firing cache
 6.     mddChanged ⇐ false;          • true if MDD changes in this iteration
 7.     foreach event e do Fire(e, q, mddChanged)   • fire e, add newly reached states
 8. until mddChanged = false;          • keep iterating until fixed point is reached
 9. return q;                  • return MDD representing the reachable state space
```

MDD-based State-space Construction. Table 2 shows a naive, iterative, and MDD-based algorithm to build the reachable state space of a system represented by a Petri net. As explained earlier, a global state $(s_K, s_{K-1}, \ldots, s_1)$ is stored over the K levels of the MDD, one substate per level. Recall that this requires us to partition Petri nets into subnets. While this can in principle be done automatically, it is still an open problem how to efficiently find "good" partitions, i.e., those that lead to small MDD representations of reachable state spaces; see [17] for a detailed discussion on partitioning.

The semantics of the Petri net under study is encoded in procedure *Fire* (cf. Table 2), which updates the MDD rooted at q according to the firing of event e by appropriately applying the *Union* operation. For efficiency reasons, it also makes use of another cache, which we refer to as *firing cache*. The procedure additionally updates a flag *mddChanged*, if the firing of e added any new reachable states. After first clearing the unique table, the initial marking m of the Petri net under consideration is stored as an MDD via procedure *SetInitial*. The algorithm then proceeds iteratively. In each iteration, every enabled transition is fired, and the potentially new states are added to the MDD. This is done until the MDD does not change, i.e., until no more reachable states are discovered. Finally, the root node q, representing the reachable state space of the Petri net, is returned.

3 The Concept of Event Locality

Our improvements for the MDD-based generation of reachable state spaces rely on the notion of *event locality*, which asynchronous systems inherently obey. Event locality is defined via the concept of *independence* of events from subnets. An event e is said to be *independent* of the k-th subnet of the net under consideration, or independent of level k, if $s_k = s'_k$ for all $s = (s_K, s_{K-1}, \ldots, s_1) \in \mathcal{S}$ and $s' = (s'_K, s'_{K-1}, \ldots, s'_1) \in \mathcal{N}(e, s)$. Otherwise, e depends on the k-th subnet, or on level k. If an event depends only on a single level k, it is called a *local event* for level k; otherwise, it is a *synchronizing event* [17]. We let $First(e)$ and $Last(e)$ denote the maximum and minimum levels on which e depends. Hence, e is independent of every level k satisfying $K \geq k > First(e)$ or $Last(e) > k \geq 1$, while e might or might not depend on levels strictly between $First(e)$ and $Last(e)$. For asynchronous systems in particular, the range of af-

fected levels is usually significantly smaller than K for most events e. We assume that all local events for level k are merged into a single *macro event* l_k satisfying $\mathcal{N}_k(l_k, s) =_{\text{df}} \bigcup_{e \in \mathcal{E}: First(e)=Last(e)=k} \mathcal{N}_k(e, s)$, for all $s \in \mathcal{S}$. This convention does not only simplify notation, but also improves the efficiency of our state-space generation algorithm.

Our aim is to define MDD manipulation algorithms that exploit the concept of event locality. Since an event e affects local states stored between levels $First(e)$ and $Last(e)$, firing e only causes updates of MDD nodes between these levels, plus possibly at levels higher than $First(e)$, but only when a node at level $First(e)$ becomes redundant, and possibly levels lower than $Last(e)$, but only until recursive *Union* calls stop creating new nodes. To benefit from this observation, we need to be able to access MDD nodes by "jumping in the middle" of an MDD, namely to level $First(e)$, rather than always having to start manipulating MDDs at their roots, as is done in traditional approaches and in [17]. This is the reason why we partition the unique table into a K-dimensional array of lists of nodes. However, two problems need to be addressed when accessing an MDD directly at some level $First(e)$.

Implicit Roots. When one wants to explore an MDD from level $First(e)$, all nodes at this level should play the role of root nodes. However, some of them might not be represented explicitly, since redundant nodes are not stored. This happens whenever there is a node p at a level higher than $First(e)$ pointing to a node q at a level k satisfying $First(e) > k \geq Last(e)$. This situation is illustrated in Fig. 2, left-hand side. Conceptually, we have to re-insert these "implicit roots" at level $First(e)$ when we explore and modify the MDD due to the firing of event e. There are two approaches for doing this. The first approach stores a bag (multiset) of *upstream arcs* in each node q, corresponding to the *downstream arcs* pointing to q. Hence, for each i such that $p{\rightarrow}dw[i] = q$, there is an occurrence of p in the bag of q's upstream arcs. Implicit roots can then be detected by scanning each node stored in the unique tables for levels $First(e) + 1$ through $Last(e)$ and by checking whether the node possesses one or more upstream arcs to a node at a level above $First(e)$. If so, an implicit root, i.e., a redundant node, is inserted at level $First(e)$. Note that at most one implicit root needs to be inserted per node, regardless of how many arcs reach it; in our example, the arcs from both p and p' are re-routed to the same new implicit root. These redundant nodes will be deleted after firing event e, if they are still redundant. Our second approach keeps all unique redundant nodes, so that downstream arcs in the resulting MDD exist only between subsequent levels. Then, the nodes at level $First(e)$ are exactly all the nodes from which we need to start exploring the underlying MDD when firing event e. Note that this slight variation of MDDs still possesses the fundamental property of being a *canonical* representation.

We refer to the two variants of our algorithm as *upstream-arcs approach* and *forwarding-arcs approach*. The latter approach, when compared to the former, eliminates the expensive search for implicit roots. However, both involve some memory penalty, the former for the storage of upstream arcs, which can in the worst case double the space requirements, and the latter because of the

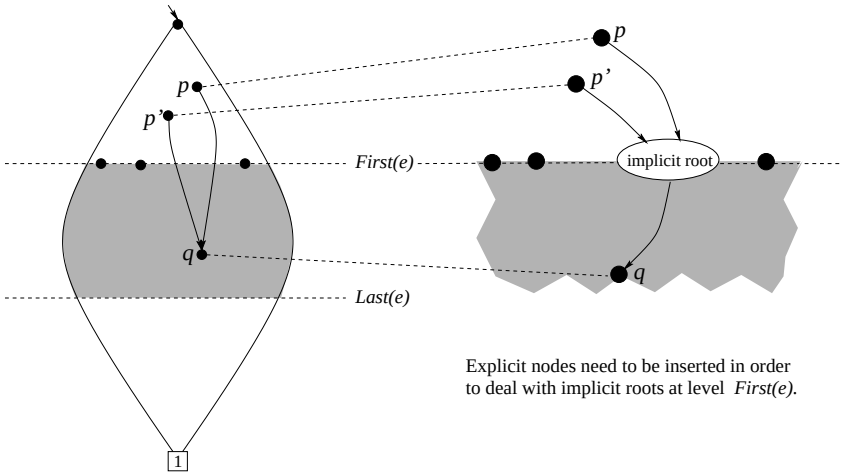

Fig. 2. Illustration of the problem of *implicit roots*

preservation of redundant nodes. We have implemented both approaches, and experimental results show that these memory overheads are compensated by a smaller peak number of MDD nodes when compared to [17] (cf. Sec. 6).

In-place Updates. Once all explicit and implicit nodes at level $First(e)$ are detected, one can update the MDD to reflect that the firing of event e may lead to new, reachable states. Our routine *Fire* implementing this update is described in Sec. 5. It relies on the *Union* operation, as presented in Table 1, i.e., new MDD nodes are created and appropriately inserted, as needed. However, there is one important difference with respect to existing approaches. Our *Fire* operation stops creating new MDD-nodes as soon as it reaches level $First(e)$ when backtracking from recursive calls. At this level our algorithm just links the new sub-MDDs at the appropriate positions in the original MDD, in accordance with event locality. The only difficulty with the in-place update of a node p arises when it becomes redundant or non-unique. In the former case, p must be deleted and its incoming arcs be re-directed to its unique child node q. In the latter case, p must be deleted and its incoming arcs be re-directed to replica node q.

In the upstream-arcs approach, either operation can be easily accomplished since p knows its parents. In the forwarding-arcs approach we keep redundant nodes; thus, we eliminate p only if it becomes non-unique. Instead of scanning all nodes in level $First(e)+1$ to search for arcs to p, which is a costly operation, we mark p as deleted and set a forwarding arc from p to q. The next time a node accesses p, it will update its own pointer to p, so that it points to q instead. Since node q itself might be marked as deleted later on, forwarding chains of nodes can arise. In our implementation, the nodes in these chains are deleted after all events at level $First(e)$ have been fired and before nodes at the next higher level are explored.

It is important to note that, although these *in-place updates* change the meaning of MDD-nodes at higher levels, they do not jeopardize the correctness of our algorithm; this is due to the property of event locality. Rather than performing *in-place updates*, existing approaches reported in the literature create an MDD encoding the set of global states reachable from the current states in the state space by firing event e. This is a K-level MDD, i.e., it is expensive to build compared to our sub-MDD, especially when MDDs are tall and the effect of e is restricted to a small range of levels.

Summarizing, it is the notion of event locality that allows us to drastically improve on the time efficiency of MDD-based state-space generation techniques. Exploiting locality, we can jump in and out of the "middle" of MDDs, thereby exploring only those levels that are affected by the event under investigation. While the approach reported in [17] also exploits locality, it just considers some simplifications and improvements of MDD manipulations in the case of local events. However, it does not support localized modifications of MDDs, neither for synchronizing nor for local events.

4 Improving Cache Management and Iteration Control

The concept of event locality also paves the road towards significant improvements in *cache management* and *iteration control*.

Intelligent Cache Management. The technique of in-place updates allows us to enhance the efficiency of the union cache. In related work, including [17], the lifetime of the contents of the union cache cannot span more than one iteration, since the root of any MDD is deleted and re-created whenever additional reachable states are incorporated in the MDD.

In contrast, in our approach the "wave" of changes towards the root, caused by firing an event e, is stopped at level $First(e)$, where only a pointer is updated. This permits some union cache entries to be reused over several iterations until the referred nodes are either changed or deleted. For this purpose, MDD nodes in our implementation have two status bits attached, namely a *cached* flag and a *dirty* flag. Instead of thoroughly cleaning up the union cache after each iteration, we can now perform a *selective purging* according to the above flags. If an MDD node associated with a union cache entry is not deleted and if the copies present in the cache are not stale, the result may be kept in the union cache. Experimental studies show us that the rate of reusability of union cache entries averages about 10% and that the overall performance of our algorithm can be improved by up to 13% when employing this idea.

Additionally, we devise a second optimization technique for the union cache, which is based on *prediction*. Our prediction relies on the fact that if $Union(p, q)$ returns r, then also $Union(p, r)$ and $Union(q, r)$ will return r. Thus, these two additional results can be memorized in the cache, immediately after storing the entry for $Union(p, q)$. Experiments indicate that this heuristics accelerates our algorithm by up to 12%. The reason for such a significant improvement is the

following. Assume we are exploring the firing of event e in node p at level k, and assume $j \in \mathcal{N}_k(e,i)$. Then, the set of states encoded by the MDD rooted at $p{\rightarrow}dw[i]$ needs to be added to the set of states encoded by the MDD rooted at $p{\rightarrow}dw[j]$. Let r be the result of $Union(p{\rightarrow}dw[i], p{\rightarrow}dw[j])$, which becomes the new value of $p \rightarrow dw[j]$. In the next iteration, and assuming that p has not been deleted, we explore event e in node p again and, consequently, find out that e is enabled in local state i. Hence, we need to perform the update $p \rightarrow dw[j] \Leftarrow Union(p \rightarrow dw[i], p \rightarrow dw[j])$ again. However, if p has not changed, $Union(p \rightarrow dw[i], p \rightarrow dw[j])$ is identical to $Union(p \rightarrow dw[i], r) = r$. By having cached r in the previous iteration, we can avoid computing this union.

Advanced Iteration Control. Event locality also allows us to reduce the number of iterations needed for generating state spaces. Existing MDD-based algorithms for Petri nets [17, 20] fire events in some arbitrary order within each iteration, as indicated in Line 7 of function *MDDgeneration* in Table 2. In our version of *MDDgeneration*, however, we presort events according to function $First(\cdot)$. Our algorithm then starts at level 1 and searches for the states that can be reached from the initial state by firing all events e satisfying $First(e) = 1$ and $Last(e) \geq 1$, i.e., the macro event l_1. When reaching level k, the algorithm finds the states that can be added to the current state space by firing all events e satisfying $First(e) = k$ and $Last(e) \geq 1$, i.e., the local macro event l_k at level k and all synchronizing events that affect only level k and some levels below. Moreover, in our implementation we repeatedly fire each event as long as it is enabled and as long as firing it adds new states. This specific sequence of firing events is essential for the correctness and efficiency of the implementation of our cache management. By working from the bottom to the top levels we can clear the union and firing caches selectively, thus, extending the lifetime of cache entries. Moreover, the access pattern to the caches is more regular. Our firing sequence also enables delayed node deletion which allows for the efficient collection and removal of non-unique and disconnected nodes.

In [17], repeatedly firing events is only applied for local events which are relatively inexpensive to process, while synchronizing events are still fired once and in no particular order. We stress that while the new iteration control means that our iterations are potentially more expensive than those in [17], they are also potentially fewer. More precisely, our algorithm generates state spaces in at most as many iterations as the maximum *synchronizing distance* of any reachable state s, which is defined in [17] as the minimal number of synchronizing events required to reach s from the initial state. We stress that the advanced iteration control we use implies a much finer management of MDD nodes than the one resulting from the use of breadth-first, mixed breadth-first/depth-first BDDs [25], or other techniques for reducing the intermediate sizes of BDDs [21].

5 Details of the New Algorithm

We now present some details of our new algorithm and argue for its correctness; we refer the reader to [5] for the complete pseudo-code.

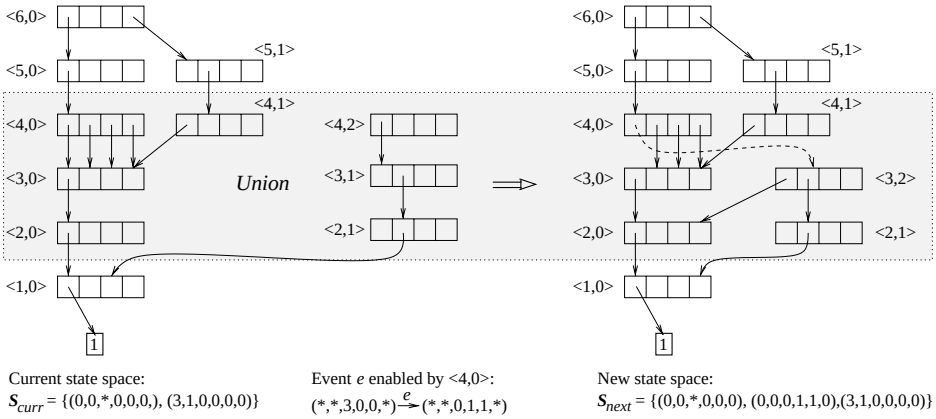

Fig. 3. Example of an MDD-modification in response to *firing an event*

Illustration of MDD-based Firing of Events. At each iteration of our algorithm, enabled events are fired to discover additional reachable states which are then added to the MDD representing the currently-known portion of the reachability set. Function $Fire(e, \cdot, \cdot)$ implements this behavior with respect to event e. Figure 3 illustrates how *Fire* works: the example net is partitioned into six subnets, each of them having four possible local states, numbered from 0 to 3. Hence, our MDD has six levels, and each MDD node has four downstream arcs; here, we do not draw node $\langle 0, 0\rangle$, nor any arc to it. Let the current state space, depicted on the left in Fig. 3, be $\mathcal{S}_{curr} = \{(0, 0, *, 0, 0, 0), (3, 1, 0, 0, 0, 0)\}$, where "$*$" stands for any local state. Assume further that event e is enabled in every state of the form $(*, *, 3, 0, 0, *)$ and that the new state reached when firing e is $(*, *, 0, 1, 1, *)$, i.e., $First(e) = 4$ and $Last(e) = 2$. Hence, if the net is in a global state described by local state 3 at level 4 and local state 0 at levels 3 and 2, event e can fire and the local states of the affected subnets are updated to 0, 1, and 1, respectively.

Exploiting event locality, our search for enabling sequences starts directly at level $First(e) = 4$. The sub-MDDs rooted at this level are searched to match the enabling pattern of e. At level 4, only the MDD rooted at $\langle 4, 0\rangle$ contains such a pattern, along the path $\langle 4, 0\rangle \xrightarrow{3} \langle 3, 0\rangle \xrightarrow{0} \langle 2, 0\rangle \xrightarrow{0} \langle 1, 0\rangle$. Then, our algorithm generates a new MDD rooted at node $\langle 4, 2\rangle$, representing the set of substates for levels 4 through 1 that can be reached from $\langle 4, 0\rangle$ via e. This MDD is depicted in Fig. 3 in the middle. Note that only nodes at levels $First(e)$ through $Last(e)$ might have to be created, since those below $Last(e)$ can simply be linked to existing nodes, such as node $\langle 1, 0\rangle$ in our example. Indeed, in our implementation even node $\langle 4, 2\rangle$ is actually not allocated, since we explore it one child at a time. This MDD corresponds to all states of the form $(\alpha, 0, 1, 1, \beta)$, where α is any substate leading to node $\langle 4, 0\rangle$ and where β is a substate reachable from the 0-th arc of node $\langle 2, 0\rangle$. In our example, α and β can only be the substates $(0, 0)$ and (0), respectively. In other words, the set of states to be added by

firing e in node $\langle 4,0\rangle$ is $\mathcal{S}_{add} = \{(0,0,0,1,1,0)\}$. Finally, the 0-th downstream arc of node $\langle 4,0\rangle$ is updated to point to the result of the union of the MDDs rooted at nodes $\langle 3,0\rangle$ and $\langle 3,1\rangle$, which is stored in an MDD rooted at the new node $\langle 3,2\rangle$, as depicted on the right in Fig. 3. Hence, the resulting state space $\mathcal{S}_{next}$ is $\{(0,0,*,0,0,0), (0,0,0,1,1,0), (3,1,0,0,0,0)\}$. Observe that our version of *Fire*(e) is more efficient than the one in [17] since it exploits the locality of e and, thus, operates on smaller MDDs. This is important as the complexity of the *Union* operation is proportional to the sizes of its operand MDDs.

Further Implementation Details. MDD nodes store not only the addresses of their children, but also Boolean flags for garbage collection and intelligent cache management, as well as information specific to the upstream-arcs approach and to the forwarding-arcs approach.

In our implementation, nodes are stored using one *heap array* per MDD level. The pages of the heap array are created only upon request and accommodate dynamic deletion and creation of nodes. Therefore, existing nodes may not be stored contiguously in memory. For fast retrieval we maintain a doubly-linked list of nodes. Upon deletion, a node is moved to the back of the list, thereby, allowing for garbage collection (but not garbage removal) in constant time.

The unique table, the union cache, and the firing cache are organized as arrays of hash tables, i.e., one hash table per level. For the unique table, the hash key of a node is determined using the values in its *dw*-array. For the union cache, the addresses of the two MDD nodes involved in the union are used to determine the hash key. Together with the *cached* and *dirty* flags, this allows us to reuse union cache entries across iterations without danger of accessing stale values. Finally, the hash key for firing cache entries is determined using only the address of the MDD node to which the firing operation is applied. Note that the identity of the event is implicit, since the firing cache is cleared when moving from one event to the next. The alternative approach, i.e., allowing the co-existence of entries referring to different events, would require a larger cache with a key based on a pair of MDD node and event. However, this would not bring enough benefits as the major cost of processing the event firing lies in the *Union* operations, and these can indeed be cached across operations.

For the upstream-arcs approach, MDD nodes include the addresses of their parents, which we store in a bag. Our implementation uses a dynamic data structure for bags rather than a static data structure, since the number of parents of a node is not known in advance and may be very large, in the range of several thousand nodes. While this memory overhead is still acceptable, the approach also puts a burden on time efficiency, since each update of a downstream arc must be reflected by an update of the corresponding upstream arc. Moreover, the bag of some node q only stores the addresses of parents p, as well as the number of indexes i such that $p{\rightarrow}dw[i] = q$, but not the indexes themselves. Thus, a linear search in $p{\rightarrow}dw$ must be performed to find these indexes. The alternative of storing these indexes in q would require even more memory overhead.

Regarding the forwarding-arcs approach, time efficiency is improved by allowing redundant nodes to be represented explicitly. As a consequence, MDD nodes

do not need to store bags of parents' addresses, but simply a counter indicating the number of incoming arcs [17]. When this counter reaches zero, it indicates that the node has become disconnected and can be deleted. Experiments show that the memory overhead of this approach, due to the storage of redundant nodes and the delayed deletion of non-unique nodes, is about the same as the memory overhead of the upstream-arcs approach. However, the forwarding-arcs approach is more time-efficient, as confirmed by the results in Sec. 6.

Correctness of the Algorithm. Here, we informally argue for the correctness of our algorithm since the formal proof is quite lengthy and, thus, omitted. Our comments concern three main features of the algorithm: (i) in-place updates, (ii) iteration control, and (iii) cache management.

The correctness of performing *in-place updates* is implied by the notion of event locality, i.e., by the asynchronous semantics of Petri nets. Formally, consider a snapshot of our algorithm where the MDD under construction currently encodes some state set $\mathcal{S}^* \subseteq \mathcal{S}$ and where event e will be fired next. Assume further that e is enabled in some $s = (s_K, s_{K-1}, \ldots, s_1) \in \mathcal{S}^*$ and that $s' = (s'_K, s'_{K-1}, \ldots, s'_1)$ is the state reached by firing it. Due to event locality we know that $s_k = s'_k$, for all k satisfying $K \geq k > \mathit{First}(e)$ or $\mathit{Last}(e) > k \geq 1$. Further, we may conclude $r' =_{\text{df}} (r_K, \ldots, r_{\mathit{First}(e)+1}, s'_{\mathit{First}(e)}, \ldots, s'_{\mathit{Last}(e)}, r_{\mathit{Last}(e)-1}, \ldots, r_1) \in \mathcal{S}$, for all $r = (r_K, r_{K-1}, \ldots, r_1) \in \mathcal{S}^*$. Hence, one may simultaneously add all these states r' to the MDD for $\mathcal{S}^*$ as is done by our in-place updates.

The exact ordering in which events are fired by the *iteration control* does not influence which MDD is returned by our algorithm. Unless an event is ignored forever during the iterations, which is not the case with our iteration control, the algorithm computes the unique least fixed point of the next-state function [11], i.e., the unique MDD encoding the reachable state space of the net under consideration. Additionally, it is obvious that our algorithm terminates for finite-state systems, since each iteration of the body of the algorithm – except the last one in which termination is detected – adds new states to the reachability set.

The correctness of our *cache management* could be hard to establish, as it is closely intertwined with iteration control and the implementation of *Fire*. For now, however, we adopt a conservative cache purging protocol to ensure that no stale entries can be accessed. Advanced protocols that achieve an even higher "hit ratio" in the union and firing caches will be the subject of further study.

6 Experimental Studies

In this section we present several performance results regarding the two variants of our algorithm and compare them with the approach most closely related to ours, namely the one reported in [17]. The variants of our algorithm are implemented in the Petri net tool SMART [6]. We apply the tool to the four Petri net models also considered in [17], i.e., the *dining philosophers*, the *slotted-ring system*, the *flexible manufacturing system* (FMS), and the *Kanban system*. The former two models, originally taken from [20], are composed of N identical *safe*

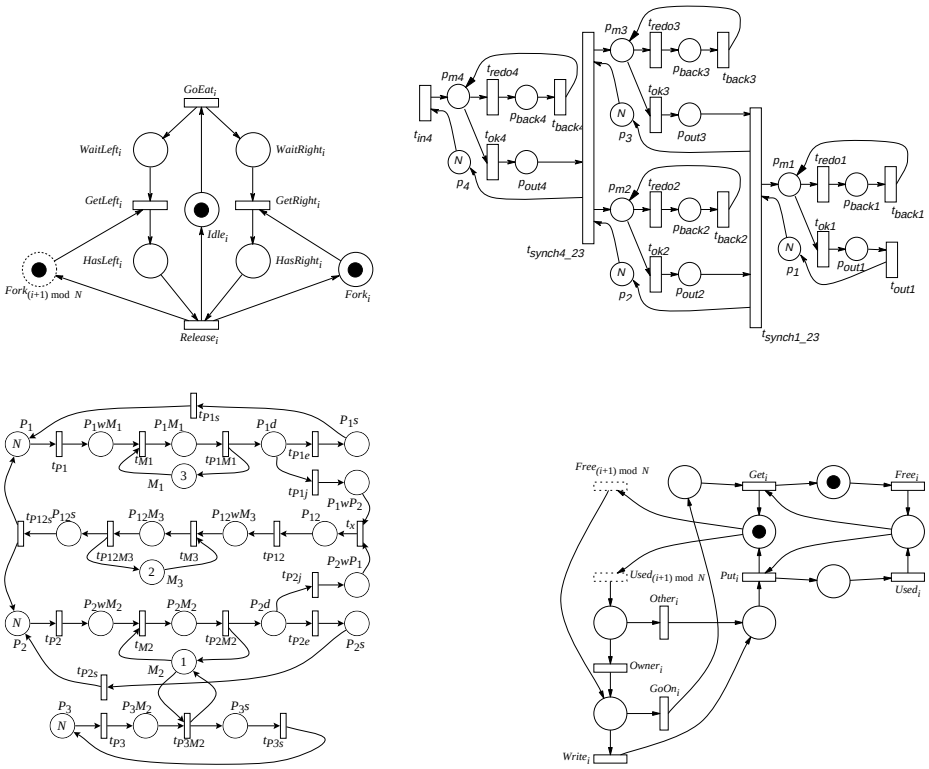

Fig. 4. Petri nets used in our experiments: dining philosophers (upper left), Kanban system (upper right), FMS (lower left), and slotted ring (lower right)

subnets, i.e., each place contains at most one token at a time. The latter two models, originally taken from [7], have a fixed number of places and transitions, but are parameterized by the number N of initial tokens in certain places. The Petri nets for these systems are depicted in Fig. 4. To use MDDs, we adopt the "best" partitions found in [17]: we consider two philosophers per level and one subnet per level for the slotted-ring protocol, while we split the FMS and the Kanban system into 19 subnets (each place in a separate subnet except for $\{P_1M_1, M_1\}$, $\{P_{12}M_3, M_3\}$, and $\{P_2M_2, M_2\}$) and 4 subnets ($\{p_{mX}, p_{backX}, p_{outX}, p_X\}$ for $X = 1, 2, 3, 4$), respectively.

Table 3 presents several results for the two variants of our new algorithm, as well as the best-known existing algorithm [17], obtained when running SMART on a 500 MHz Intel Pentium II workstation with 512 MB of memory and 512 KB cache. For each model and choice of N we give the size of the state space and the final number of MDD nodes, which is of course independent of the algorithm used. Then, for each algorithm we give the peak number of MDD nodes allocated during execution and the corresponding memory consumption (in KBytes), the number of iterations, and the CPU time (in seconds). The number of iterations

for the upstream-arcs and forwarding-arcs approaches coincide. The peak number of nodes and the memory consumption reported for our approach refer to the forwarding-arcs approach. For the upstream-arcs approach, the peak number of nodes reported should be decreased by one for the FMS and the Kanban system, while the memory consumption should be increased by 6–8% for the dining philosophers, 31–33% for the slotted-ring net, 41–45% for the FMS, and 1–5% for the Kanban system. This implies that, even without introducing redundant nodes, essentially all arcs already connect nodes between adjacent levels. Thus, in our examples, the only memory overhead in the forwarding arcs approach is due to postponed node deletion.

For the models we ran, our new approach is up to one order of magnitude faster and with few exceptions uses fewer MDD nodes than the one in [17]. The improvement mainly arises from the structural changes made to the core routine *Fire*, which reflect the notion of event locality. Other improvements – most importantly our cache optimizations – contribute in average about 7–13%, and up to 22% in total, to the overall improvement in time efficiency. A comparison between the run-times for the new algorithm and the ones for the algorithm in [17] indicates an increase factor in speed ranging from approximately constant for the Kanban and FMS nets to what appears to be almost linear (in N) for the slotted-ring model and the dining philosophers. Moreover, the forwarding-arcs approach is slightly faster than the upstream-arcs approach, except for the Kanban system on which we comment below. Since both variants of our new algorithm require significantly fewer peak MDD nodes, where the Kanban system is again an exception, our memory penalty is more than compensated.

The two models whose parameter N affects the height of the MDD, namely the dining philosophers and the slotted-ring model, provide a good testbed for our ideas since they give rise to tall MDDs with a high degree of event locality. For these models, the CPU times are up to 15 times faster than the ones for [17], and the gap widens as we continue to scale-up the nets. The main reason for this is that the number of explored nodes per event fired is more contained in our approach, compared to [17]. When MDD heights are small, such as for the FMS and the Kanban system, our algorithm is still faster than the one in [17], but the difference is not as impressive due to our book-keeping overhead.

The memory consumption figures for the Kanban system are poor compared to the ones for our other examples, although the number of iterations is reduced from $2 \cdot N + 1$ to 4 due to our advanced iteration control, and the solution time is still better than the one of [17]. There are several reasons for this. First, splitting the Kanban net into only four subnets leads to an MDD with a small depth, but a very large breadth, i.e., extremely large nodes. Clearly, any attempt to exploit locality in this case cannot have much pay-off. Second, our garbage collection policy in the forwarding-arcs approach contributes to the proliferation of deleted nodes which are not truly destroyed until the end of the iteration. Combined with the reduced number of iterations in our approach, the garbage collection bin grows very rapidly. Usually, late node deletion is beneficial, since doing garbage collection in bulk reduces the number of times nodes are scanned

Table 3. Results: (A) dining philosophers, (B) slotted ring, (C) FMS, (D) Kanban

			Approach in [17]				Our new approach				
N	$\|\mathcal{S}\|$	final	peak	mem.	#	time	peak	mem.	#	time (sec.)	
		nodes	nodes	(KB)	it.	(sec.)	nodes	(KB)	it.	upstr.	fwd.
A 10	$1.86{\cdot}10^{6}$	17	45	3	2	0.03	28	4	2	0.02	0.01
50	$2.23{\cdot}10^{31}$	37	285	22	2	0.74	168	26	2	0.11	0.10
100	$4.97{\cdot}10^{62}$	197	585	58	2	3.04	343	54	2	0.30	0.28
200	$2.47{\cdot}10^{125}$	397	1,185	129	2	12.23	693	109	2	1.00	0.90
300	$1.23{\cdot}10^{188}$	597	1,785	198	2	28.10	1,043	164	2	2.16	2.12
400	$6.10{\cdot}10^{250}$	797	2,385	265	2	54.16	1,393	219	2	3.95	3.80
500	$3.03{\cdot}10^{313}$	997	2,985	333	2	81.83	1,743	274	2	6.33	6.02
600	$1.51{\cdot}10^{376}$	1,197	3,585	400	2	125.74	2,093	329	2	9.19	8.77
700	$7.48{\cdot}10^{438}$	1,397	4,185	468	2	181.61	2,443	384	2	12.56	12.03
800	$3.72{\cdot}10^{501}$	1,597	4,785	535	2	247.97	2,793	439	2	16.43	15.79
900	$1.85{\cdot}10^{564}$	1,797	5,385	602	2	305.16	3,143	493	2	20.67	19.88
1,000	$9.18{\cdot}10^{626}$	1,997	5,985	669	2	386.26	3,493	548	2	25.51	24.59
B 10	$8.29{\cdot}10^{9}$	60	691	39	7	1.25	509	41	7	0.65	0.56
20	$2.73{\cdot}10^{20}$	220	4,546	263	12	28.64	3,197	259	12	11.04	8.75
30	$1.04{\cdot}10^{31}$	480	15,101	973	17	212.62	10,433	845	17	70.25	53.53
40	$4.16{\cdot}10^{41}$	840	37,066	2,149	22	935.75	25,374	2,055	22	282.02	210.52
50	$1.72{\cdot}10^{52}$	1,300	76,308	5,342	27	3,036.88	47,806	4,208	27	861.17	635.15
C 5	$2.90{\cdot}10^{6}$	149	433	21	10	0.48	240	10	10	0.22	0.18
10	$2.50{\cdot}10^{9}$	354	1,038	66	15	2.08	600	34	15	0.82	0.67
15	$2.17{\cdot}10^{11}$	634	1,868	145	20	5.63	1,110	78	20	2.31	1.79
20	$6.03{\cdot}10^{12}$	989	2,923	267	25	11.97	1,770	149	25	5.23	3.89
25	$8.54{\cdot}10^{13}$	1,419	4,203	441	30	23.86	2,580	253	30	10.59	7.42
50	$4.24{\cdot}10^{17}$	4,694	13,978	2,410	55	213.89	8,880	1,469	55	168.23	65.64
75	$6.98{\cdot}10^{19}$	9,844	29,378	7,035	80	863.62	18,930	4,399	80	538.29	265.03
100	$2.70{\cdot}10^{21}$	16,869	50,403	15,439	105	2,362.63	32,730	9,792	105	1,529.50	740.82
D 5	$2.55{\cdot}10^{6}$	7	47	3	11	0.07	56	14	4	0.05	0.04
10	$1.01{\cdot}10^{9}$	12	87	22	21	1.10	156	182	4	0.54	0.48
15	$4.70{\cdot}10^{10}$	17	127	86	31	6.34	306	1,005	4	3.07	2.73
20	$8.05{\cdot}10^{11}$	22	167	238	41	22.87	506	3,595	4	11.67	10.45
25	$7.68{\cdot}10^{12}$	27	207	541	51	63.70	756	9,923	4	33.88	30.37
30	$4.99{\cdot}10^{13}$	32	247	1,068	61	150.28	1,056	23,068	4	85.59	76.74
40	$9.94{\cdot}10^{14}$	42	327	3,823	81	583.01	1,806	89,189	4	378.73	343.72
50	$1.04{\cdot}10^{16}$	52	407	9,510	101	1,703.69	2,756	258,306	4	1,221.40	1,106.75

for removal. However, in case of the Kanban system we see how this can backfire. It is worth noting that using a finer and not particularly "good" partition of the Kanban net, with one place per level, drastically changes the results, as shown in Table 4. We only need to scale-up the model to $N = 20$ to see an improvement of about a factor 70 with respect to [17]. This observation testifies to the suitability of our algorithm (mostly) in cases when a good partitioning cannot be found automatically or by hand, e.g., due to insufficient heuristics.

Table 4. Results for the Kanban net with 16 levels (one place per level)

			Approach in [17]				Our new approach				
N	$\|S\|$	final	peak	mem.	#	time	peak	mem.	#	time (sec.)	
		nodes	nodes	(KB)	it.	(sec.)	nodes	(KB)	it.	upstr.	fwd.
1	$1.60 \cdot 10^2$	32	112	31	5	0.79	56	58	6	0.20	0.19
2	$4.60 \cdot 10^3$	51	271	81	7	2,46	110	124	6	0.44	0.31
3	$5.84 \cdot 10^4$	73	521	231	10	6.56	221	230	6	0.62	0.54
4	$4.54 \cdot 10^5$	98	895	428	13	14.21	373	389	6	1.00	0.82
5	$2.55 \cdot 10^6$	126	1,414	705	16	25.46	587	613	6	1.56	1.28
8	$1.34 \cdot 10^8$	228	4,015	2,099	25	113.01	1,833	1,914	6	4.49	3.70
10	$1.01 \cdot 10^9$	311	6,838	3,629	31	237.11	3,402	3,552	6	8.01	6.56
15	$4.70 \cdot 10^{10}$	571	19,061	10,185	46	1,032.12	11,507	12,013	6	24.93	20.12
20	$8.05 \cdot 10^{11}$	906	40,741	21,902	61	3,340.99	29,137	30,419	6	62.34	48.31

7 Related Work

A variety of approaches for the generation of reachable state spaces of synchronous and asynchronous systems have been suggested in the literature, where state spaces are represented either in an *explicit* or in a *symbolic* fashion.

Explicit state-space generation techniques construct the reachable state space of the system under consideration by successively iterating its next-state function (see, e.g., [3, 7]). To achieve space efficiency, numerous techniques have been introduced, out of which *multi-level data structures* and *merging common bitvectors* deserve special mentioning. Multi-level data structures exploit the structure of the underlying system representation, e.g., the approach reported in [7] is based on a decomposition of a Petri net into subnets. As the name suggests, merging common bitvectors aims at compressing the storage needed for each state – a bitvector – by merging common sub-bitvectors [3, 14]; indeed, the result is somewhat analogous to the one obtained using BDDs. While explicit methods still require space linear in the number of states, they usually possess some advantages for numerical state-space analyses [16].

To avoid the problem of state-space explosion when building the explicit state space of concurrent, asynchronous systems, researchers developed three key techniques: (i) *compositional minimization techniques* build the state space of a concurrent system stepwise, i.e., parallel component by parallel component, and minimize the state space of each intermediate system according to a behavioral congruence or an interface specification [13]; (ii) *Partial-order techniques* exploit the fact that several traces of an asynchronous system may be equivalent with respect to the properties of interest [12, 22, 24]; thus, it is sufficient to explore only a single trace of each equivalence class; (iii) *techniques exploiting symmetries in systems* – such as those with repeated sub-systems – can be used to avoid the explicit construction of symmetric subgraphs of the overall state spaces [9].

Symbolic state-space generation techniques have largely concentrated on (synchronous) hardware systems rather than on (asynchronous) software systems [1,

4, 10, 15]. For safe Petri nets, Pastor et al. [20] developed a BDD-based algorithm for the generation of the reachability sets by encoding each place of a net as a Boolean variable. The algorithm is capable of generating state spaces of very large nets within hours; similar techniques were also implemented by Varpaaniemi et al. [23] in the Petri net tool PROD. In recent work, Pastor and Cortadella introduced a more efficient encoding of nets by exploiting place invariants [19]. However, the underlying logic is still based on Boolean variables. In contrast, our work uses a more general version of decision diagrams, namely MDDs [15, 17], by which the amount of information carried in each single node of a decision diagram can be increased. In particular, MDDs allow for a straightforward encoding of arbitrary, i.e., not necessarily safe, Petri nets. Since we have already compared our approach to related MDD-based techniques in the previous sections, we refrain from a repetition of the issues here.

8 Conclusions and Future Work

This paper presented a novel algorithm for constructing the reachable state spaces of asynchronous systems. As in previous work [17], state spaces are symbolically represented via Multi-valued Decision Diagrams (MDDs). In contrast to previous work, our algorithm fully exploits event locality in asynchronous systems, integrates an intelligent cache management, and achieves faster convergence via an advanced iteration control. Experimental results for examples well-known in the Petri net community show that our algorithm is often significantly faster than the one introduced in [17], with no or usually neglectable decrease in space efficiency. To the best of our knowledge, our algorithm is the first symbolic algorithm taking advantage of event locality.

Regarding future work, we would like to further explore various approaches to iteration control and to partitioning, as well as the mutual influences between partitioning and variable ordering. Moreover, we plan to employ our MDD manipulation algorithms as a basis for implementing a model checker in SMART, such that not only *safety properties* but also *liveness properties* can be automatically checked by the tool. Finally, we intend to parallelize our algorithms for shared-memory and distributed-memory architectures.

Acknowledgments. We would like to thank A.S. Miner and the anonymous referees for their valuable comments and suggestions.

References

[1] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.

[2] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):393–418, 1992.

[3] P. Buchholz. Hierarchical structuring of superposed GSPNs. In *PNPM '97*, pp. 81–90. IEEE Computer Society Press, 1997.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *I&C*, 98(2):142–170, 1992.

[5] G. Ciardo, G.Lüttgen, and R. Siminiceanu. Efficient state-space construction for asynchronous systems. Techn. Rep. 99-50, Institute for Computer Applications in Science and Engineering (ICASE), 1999.

[6] G. Ciardo and A.S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *IPDS '96*, p. 60. IEEE Computer Society Press, 1996.

[7] G. Ciardo and A.S. Miner. Storage alternatives for large structured state spaces. In *Tools '97*, vol. 1245 of *LNCS*, pp. 44–57. Springer-Verlag, 1997.

[8] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Techn. Rep. 96-35, Institute for Computer Applications in Science and Engineering (ICASE), 1996.

[9] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in model checking. In *CAV '93*, vol. 697 of *LNCS*, pp. 450–462. Springer-Verlag, 1993.

[10] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] A. Geser, J. Knoop, G. Lüttgen, B. Steffen, and O. Rüthing. Chaotic fixed point iterations. Techn. Rep. MIP-9403, Univ. of Passau, 1994.

[12] P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems – An Approach to the State-explosion Problem*, vol. 1032 of *LNCS*. Springer-Verlag, 1996.

[13] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *FAC*, 8(5):607–616, 1996.

[14] G. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[15] T. Kam, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[16] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. on Software Engineering*, 22(4):615–628, 1996.

[17] A.S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *ICATPN '99*, vol. 1639 of *LNCS*, pp. 6–25. Springer-Verlag, 1999.

[18] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, 1989.

[19] E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *DATE '98*, pp. 790–795. IEEE Computer Society Press, 1998.

[20] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri net analysis using Boolean manipulation. In *ICATPN '94*, vol. 815 of *LNCS*, pp. 416–435. Springer-Verlag, 1994.

[21] K. Ravi and F. Somenzi. High-density reachability analysis. In *ICCAD '95*, pp. 154–158. IEEE Computer Society Press, 1995.

[22] A. Valmari. A stubborn attack on the state explosion problem. In *CAV '90*, pp. 25–42. AMS, 1990.

[23] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Techn. Rep. B13, Helsinki Univ. of Technology, 1995.

[24] F. Vernadat, P. Azema, and F. Michel. Covering step graph. In *ICATPN '96*, vol. 1091 of *LNCS*, pp. 516–535. Springer-Verlag, 1996.

[25] B. Yang and D.R. O'Hallaron. Parallel breadth-first BDD construction. In *PPOPP '97*, vol. 32, 7 of *ACM SIGPLAN Notices*, pp. 145–156, 1997.