

The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection^{*}

Sami Evangelista, Michael Westergaard and Lars Michael Kristensen

DAIMI, University of Aarhus, Denmark
{evangelista,mw,kristensen}@cs.au.dk

Abstract. The ComBack method is a memory reduction technique for explicit state space search algorithms. It enhances hash compaction with state reconstruction to resolve hash conflicts on-the-fly thereby ensuring full coverage of the state space. In this paper we provide two means to lower the run-time penalty induced by state reconstructions: a set of strategies to implement the caching method proposed in [18], and an extension through delayed duplicate detection that allows to group reconstructions together to save redundant work.

1 Introduction

Model checking is a formal method used to detect defects in system designs. It consists of a systematic exploration of the reachable states of the system whose behavior can be formally represented as a directed graph whose nodes are states and arcs are possible transitions from one state to another. This principle is simple, can be easily automated, and, in case of errors, a counter-example can be provided to the user.

However, even simple systems may have an astronomical or even infinite number of states. This state explosion problem is a severe obstacle to the application of model checking to industrial size systems. Numerous possibilities are available to alleviate, or at least delay, this phenomenon. One can for example exploit the redundancies in the system description that often induce symmetries [3], exploit the independence of some transitions to reduce the exploration of redundant interleavings [6], or encode the state graph using compact data structures such as binary decision diagrams [1].

Hash compaction [15,19] is a graph storage technique that reduces the amount of memory used to store states. It uses a hash function h to map each encountered state s into a fixed-size bit-vector $h(s)$ called the *compressed state descriptor* which is stored in memory as a representation of the state. The *full state descriptor* is not stored in memory. Thus, each discovered state is represented compactly using typically 32 or 64 bits. The disadvantage of hash compaction is that two different states may be mapped to the same compressed state descriptor

^{*} Supported by the Danish Research Council for Technology and Production.

which implies that the hash compaction method may not explore all reachable states. The probability of *hash collisions* can be reduced by using multiple hash functions [10,15], but the method still cannot guarantee full coverage of the state space. This is acceptable if the intent is to find errors, but not sufficient if the goal is to prove the correctness of a system specification.

The ComBack method [18] extends hash compaction with a backtracking mechanism that allows reconstruction of full state descriptors from compressed ones and thus resolve conflicts on-the-fly to guarantee full coverage of the state space. Its underlying principle is to store for any state a sequence of events that generated this state. Thus, when the search algorithm checks if it already visited a state s , it can reconstruct states mapped to the same hash value as s and compare them to it. Only if none of the states reconstructed is equal to s can the algorithm consider it as a new state.

This storage technique stores a small amount of information per state, typically between 16 and 24 bytes depending on the system being analyzed. Thus it is especially suited to industrial case studies for which the full state descriptor stored by a classical search algorithm can be very large (from 100 bytes to 10 kilo-bytes). This important reduction, however, has a time cost: a ComBack based algorithm will explore many more arcs in order to reconstruct states. As the graph is given implicitly, visiting an arc consists of applying a successor function that can be arbitrarily complex, especially for high-level languages such as Promela [8] or Colored Petri nets [9]. Experiments made in [18] report an increase in run-time ranging from 50% for the simplest examples to more than 600% for real-life protocols.

The goal of the work presented in this paper is to propose solutions to tackle this problem. Starting from the proposal of [18] to use a cache of full state descriptors to shorten sequences, we first propose different caching strategies. We also extend the ComBack method with delayed duplicate detection, a technique widely used by disk-based model checkers [16]. The principle is to delay the instant we check if a state has already been visited from the instant of its generation. Any state reached is put in a set of candidates and only occasionally is this set compared to the set of already visited states in order to identify new ones. The underlying idea of this operation is that comparing these two sets may be much cheaper than checking separately if each candidate has already been visited. Applied to the ComBack method, this results in saving the visit of transitions that are shared by different sequences. For instance if sequences $a.b.c$ and $a.b.d$ reconstruct respectively states s and s' we may group the reconstructions of s and s' in order to execute sequence $a.b$ only once instead of twice. This will result in the execution of 4 events instead of 6 events.

This article has the following structure. The basic elements of labeled transition systems and the ComBack method are recalled in Section 2. In Section 3, different caching strategies are proposed. An algorithm that combines the ComBack method with delayed duplicate detection is presented in Section 4. Section 5 reports on experiments made with the ASAP tool [12] which implements the techniques proposed in this paper. Finally, Section 6 concludes this paper.

2 Background

We give in this section the basic ingredients that are required for understanding the rest of this paper and provide a brief overview of the ComBack method [18].

2.1 Transition systems

As the methods proposed in this work are not linked to a specific formalism they will be developed in the framework of labeled transition systems that are the most low-level representation of concurrent systems.

Definition 1 (Labeled Transition System). *A labeled transition system is a tuple $\mathcal{S} = (S, E, T, s_0)$, where S is a finite set of **states**, E is a finite set of **events**, $T \subseteq S \times E \times S$ is the **transition relation**, and $s_0 \in S$ is the **initial state**.*

In the rest of this paper we assume that we are given a labeled transition system $\mathcal{S} = (S, E, T, s_0)$. Let $s, s' \in S$ be two states and $e \in E$ an event. If $(s, e, s') \in T$, then e is said to be *enabled* in s and the *occurrence* (execution) of e in s leads to the state s' . This is also written $s \xrightarrow{e} s'$. An *occurrence sequence* is an alternating sequence of states s_i and events e_i written $s_1 \xrightarrow{e_1} s_2 \cdots s_{n-1} \xrightarrow{e_{n-1}} s_n$ and satisfying $s_i \xrightarrow{e_i} s_{i+1}$ for $1 \leq i \leq n - 1$. For the sake of simplicity, we assume that events are *deterministic*¹, i.e., if $s \xrightarrow{e} s'$ and $s \xrightarrow{e} s''$ then $s' = s''$.

We use \rightarrow^* to denote the transitive and reflexive closure of T , i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{e_1} s_2 \cdots s_{n-1} \xrightarrow{e_{n-1}} s_n$, $n \geq 1$, with $s = s_1$ and $s' = s_n$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$. The *state space* of a system is the directed graph (V, E) where $V = \{s' \in S \mid s_0 \rightarrow^* s'\}$ is the set of nodes and $E = \{(s, e, s') \in T \mid s, s' \in V\}$ is the set of edges.

2.2 The ComBack method

A classical state space search algorithm (Algorithm 1) operates on a set of visited states \mathcal{V} and a queue of states to visit \mathcal{Q} . An iteration of the algorithm (lines 4–7) consists of removing a states from the queue, generating its successors and inserting the successor states that have not been visited so far both in the visited set and in the queue for a later exploration².

Using hash compaction [19], items stored in the visited set are not actual state descriptors but compressed descriptors, typically a 32-bit integer, obtained through a hash function h . Algorithm 2 uses this technique. The few differences with Algorithm 1 have been underlined. This storage scheme is motivated by the observation that full state descriptors are often large for realistic systems,

¹ For an extension of the ComBack method to non-deterministic transition systems the reader may consult Section 5 of [18].

² We will use the term of *state expansion* to refer to this process.

Algorithm 1 A classical search algorithm.

```
1:  $\mathcal{V} \leftarrow \text{empty}$  ;  $\mathcal{V}.\text{insert}(s_0)$ 
2:  $\mathcal{Q} \leftarrow \text{empty}$  ;  $\mathcal{Q}.\text{enqueue}(s_0)$ 
3: while  $\mathcal{Q} \neq \text{empty}$  do
4:    $s \leftarrow \mathcal{Q}.\text{dequeue}()$ 
5:   for  $e, s' \mid (s, e, s') \in T$  do
6:     if  $s' \notin \mathcal{V}$  then
7:        $\mathcal{V}.\text{insert}(s')$  ;  $\mathcal{Q}.\text{insert}(s')$ 
```

Algorithm 2 A search algorithm based on hash compaction.

```
1:  $\mathcal{V} \leftarrow \text{empty}$  ;  $\mathcal{V}.\text{insert}(h(s_0))$ 
2:  $\mathcal{Q} \leftarrow \text{empty}$  ;  $\mathcal{Q}.\text{enqueue}(s_0)$ 
3: while  $\mathcal{Q} \neq \text{empty}$  do
4:    $s \leftarrow \mathcal{Q}.\text{dequeue}()$ 
5:   for  $e, s' \mid (s, e, s') \in T$  do
6:     if  $h(s') \notin \mathcal{V}$  then
7:        $\mathcal{V}.\text{insert}(h(s'))$  ;  $\mathcal{Q}.\text{insert}(s')$ 
```

i.e., typically between 100 bytes and 10 kilo-bytes, which drastically limits the size of state spaces that can be explored. Though hash compaction considerably reduces memory requirements, it comes at the cost of possibly missing some parts of the state space and potentially some errors. Indeed, as h may not be injective, two different states may erroneously be considered the same if they are mapped to the same hash value. Hence, hash compaction is preferably used at early stages of the development process for its ability to quickly discover errors rather than proving the correctness of the system.

The ComBack method extends hash compaction with a backtracking mechanism that allows it to retrieve actual states from compressed descriptors in order to resolve hash collisions on-the-fly and guarantee full coverage of the state space. This is achieved by modifying the hash compaction algorithm as follows:

1. A *state number* (integer), or identifier, is assigned to each visited state s .
2. A *state table* stores for each compressed state descriptor a *collision list* of state numbers for visited states mapped to this compressed state descriptor.
3. A *backedge table* is maintained which for each state number of a visited state s stores a *backedge* consisting of an event e and a state number of a visited predecessor s' such that $s' \xrightarrow{e} s$.

The key algorithm of the ComBack method is the insertion procedure that checks whether a state s is already in the visited set and inserts it into it if needed. Its principle can be illustrated with the help of Figure 1 which depicts a simple state space. Each ellipse represents a state. The hash value of each state is written in the right part of the ellipse. The state and backedge tables used to resolve hash conflicts have been depicted to the right of the figure for two different steps of the search. For the sake of clarity, we have also depicted on the state space the identifier of each state (the square next to the ellipse) and highlighted (using

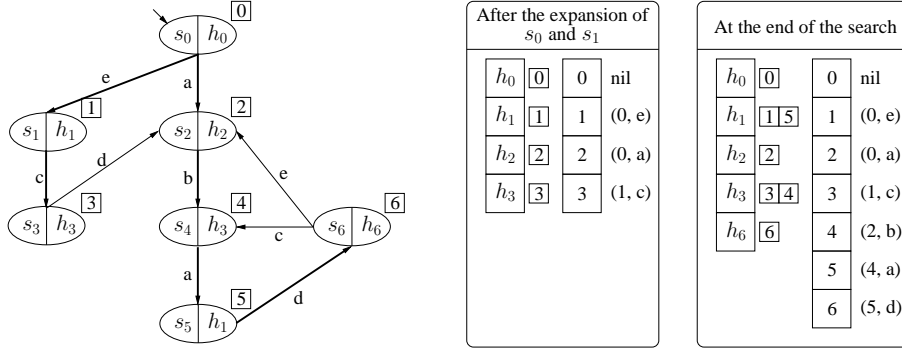


Fig. 1. A state space and the state and backedge tables at two stages.

thick arcs) the transitions that are used to backtrack to the initial state, i.e., the edges constituting the backedge table. Note that these identifiers also coincide with the expansion order of states.

After the expansion of s_0 and s_1 , the set of visited states is $\{s_0, s_1, s_2, s_3\}$. As no hash conflict was detected, a single state is associated in the state table (the left table of the first rounded box) with each hash value. In the backedge table (the right table of the first rounded box) a nil value is associated with state 0 (the initial state) as any backtracking will stop here. The table also indicates that the actual value of state 1 (s_1) is retrieved by executing event e on state 0 and so on for the other entries of the table. After the execution of event b on state s_2 we reach s_4 . Algorithm 2 would claim that s_4 has already been visited — since $h(s_3) = h(s_4)$ — and stop the search at this point, missing states s_5 and s_6 . Using the two tables the hash conflict between s_3 and s_4 can be handled as follows. The insertion procedure first looks in the state table if any state has already been mapped to $h(s_4) = h_3$ and finds out value 3. The comparison of state 3 (of which we do not have the actual state descriptor) to s_4 is first done by recursively following the pointers of the backedge table until the initial state is reached, i.e., 3 then 1 and then 0. Then the sequence of events associated with the entries of the table that have been met during the backtrack, i.e., e, c , is executed on the initial state³. Finally, a comparison between s_3 and s_4 indicates that s_4 is new. We therefore assign to s_4 a new identifier (4) insert it in the collision list of hash value h_3 and insert the entry $4 \rightarrow (2, b)$ in the backedge table.

This storage scheme is especially suited to systems exhibiting large state vectors as it allows to represent each state in the visited set with only a few bytes. The only elements of the state and backedge tables that are still dependent of

³ We will use the term of *state reconstruction* (or more simply *reconstruction*) to refer to this process, i.e., backtracking to the initial state and then executing a sequence of events to retrieve a full state descriptor. Sequence e, c will be called the *reconstructing sequence* of state 3.

the underlying model are the events stored to reconstruct states. In the case of Colored Petri Nets, this comprises a transition identifier and some instantiation values for its variables while for some modeling languages it may be sufficient to identify an event with a process identifier and the line of the executed statement. Still, a state rarely exceeds 16–24 bytes.

However, the ComBack method is penalized by an (important) run-time increase due to the reconstruction mechanism. After a state s has been reached it will be reconstructed once for each following incoming arc, hence $in(s) - 1$ times where $in(s)$ denotes the in-degree of s . If we denote by $d(s)$ the length of the shortest path from s_0 to s , the number of event executions due to state reconstructions is lower bounded by:

$$\sum_{s \in S} (in(s) - 1) \cdot d(s)$$

Note that in Breadth-First Search (BFS) each sequence executed to reconstruct a state s is exactly of length $d(s)$ while it may be much longer in Depth-First Search (DFS). This is evidenced by some data of Table 1 in [18] showing that the ComBack method combined with DFS is in some cases much slower than with BFS while the converse is not true.

In addition, the time spent in reconstructing states depends, to a large extent, on the complexity of executing an event that ranges from trivial (e.g., for PT-nets) to high, e.g., for Promela or Colored Petri Nets for which executing an event may include the execution of embedded code.

3 Caching strategies

A cache mapping state identifiers to full descriptors is a good way to reduce the cost of state reconstructions. The purpose of such a cache is twofold. Firstly, the reconstruction of a state identified by i may be avoided if i is cached. Secondly, if a state has to be reconstructed we may stop backtracking as soon as we encounter a state belonging to the cache and thus execute a shorter reconstruction sequence from this state. As an example, consider the configuration of Fig. 1. Caching the mapping $1 \rightarrow s_1$ may be useful in two ways.

To avoid the reconstruction of state 1. A lookup in the cache directly returns state s_1 , which saves the backtrack to s_0 and the execution of event e .

For the reconstruction of state 3. During the backtrack to s_0 the algorithm finds out that state 1 is cached, retrieves its descriptor and only executes event c from s_1 to obtain s_3 , once again saving the execution of event e .

We now propose four strategies to implement this cache. We focus on strategies based on BFS as the traversal order it induces enables to take advantage of some typical characteristics of state spaces [13].

Random cache The simplest and easiest way is to implement a randomized cache. This gives us the first following strategy.

Strategy R: *When a new state is put in the visited set, it is inserted in the cache with probability p (1 if the cache is not full) and the state to replace (if needed) is randomly chosen.*

Fifo cache A common characteristics of state spaces is the high proportion of forward transitions⁴, typically around 80%. This has a significant consequence in BFS in which levels are processed one by one: most of the transitions outgoing from a state will lead to a new state or to a state that has been recently generated from the same level. Hence, a good strategy in BFS seems to be to use a fifo cache since when a new state at level $l + 1$ is reached from level l it is likely that one of the following states of level l will also reach it. If the cache is large enough to contain any level of the graph, only backward transitions will generate reconstructions as forward transitions will always result in a cache hit. This strategy can be implemented as follows.

Strategy F: *When a new state is put in the visited set, insert it unconditionally into the cache. If needed, remove the oldest state from the cache.*

Heuristic based cache Obviously, the benefit we can obtain from caching a state may largely differ from one state to another. For instance, it is pointless to cache a state s that does not have any successor state pointing to it in the backedge table as it will not shorten any reconstruction sequence, but only avoid the reconstruction of s .

To evaluate the interest of caching some state s we propose to use the following caching heuristic H .

$$H(s) = d(s) \cdot p(s) \text{ with } p(s) = \frac{r(s)}{L(d(s))}$$

where

- $d(s)$ is the distance of s to the initial state in the backedge table
- $r(s)$ is the number of states that reference s in the backedge table
- $L(n)$ is the number of states at level n , i.e., with a distance of n from the initial state

A cache hit is more interesting if it occurs early during the backtrack as it will shorten the sequence executed. Thus the benefit of caching a state s increases with its distance $d(s)$. Through rate $p(s)$ we evaluate the probability that s

⁴ If we define *level l* as the set of states that are reachable from s_0 in l steps (and not less), a transition that has its source in level l and its target in level $l + 1$ is called a *forward transition*. Any other transition is called a *backward transition*.

belongs to some reconstructing sequence. This one increases if many states point to s in the backedge table and decreases with the number of states on the same level as s . The distance of s could also be considered in the computation of $p(s)$ as s cannot appear in a reconstructing sequence of a length less than $d(s)$. Our choice is based on another typical characteristic of state spaces [17]: backward transitions are usually short in the sense that the levels of its destination and source are often close. Thus, in BFS, if a state has to be reconstructed, it is likely that the length of its reconstructing sequence is close to the current depth which is an upper bound of the length of a reconstructing sequence. Hence, assuming that the state space has this characteristic, the distance slightly impacts on $p(s)$.

Our third strategy is based on this heuristic.

Strategy H: *After all outgoing transitions of state s have been visited compute $H(s)$. Let s' be the state that minimizes H in the cache. If $H(s') < H(s)$ replace s' by s in the cache.*

Note that after the visit of s , all necessary information to compute $H(s)$ is available since all its successors have been generated and the BFS search order implies that $L(d(s))$ is known.

Other possibilities are available. In [5] a reduction technique also based on state reconstruction is proposed. The algorithm is parametrized by an integer k and only caches states at levels $0, k, 2 \cdot k, 3 \cdot k \dots$. The motivation of this strategy is to bound the length of reconstructing sequences to $k - 1$. As presented, the strategy in [5] does not bound the size of the cache but k could be dynamically increased to solve this problem.

Different strategies may also be combined. We can for example cache recently inserted states following strategy F and when a state leaves this cache it can be inserted into a second level cache maintained with strategy H. Thus we will keep some recently visited states in the cache and some old strategic states.

4 Combination with delayed duplicate detection

Duplicate detection consists of checking the presence of a newly generated state in the set of visited states. If the state has not been visited so far, it must be included in the set and later expanded. With delayed duplicate detection (DDD), this check is delayed from the instant of state generation by putting the state reached in a *candidate set* that contains potentially new states. In this scheme, duplicate detection consists of comparing the visited and candidate sets to identify new states. This is motivated by the fact that this comparison may be much cheaper than checking individually for the presence of each candidate in the visited set.

Algorithm 3 is a generic algorithm based on DDD. Besides the usual data structures we find a candidate set \mathcal{C} filled with states reached through event execution (lines 7–8). An iteration of the algorithm (lines 4–9) consists of expanding all queued states and inserting their successors in the candidate set.

Algorithm 3 A generic search algorithm using delayed duplicate detection

```
1:  $\mathcal{V} \leftarrow \text{empty}$  ;  $\mathcal{V}.insert (s_0)$            10: proc duplicateDetection () is
2:  $\mathcal{Q} \leftarrow \text{empty}$  ;  $\mathcal{Q}.enqueue (s_0)$        11:    $new \leftarrow \mathcal{C} \setminus \mathcal{V}$ 
3: while  $\mathcal{Q} \neq \text{empty}$  do                       12:   for  $s \in new$  do
4:    $\mathcal{C} \leftarrow \text{empty}$                           13:      $\mathcal{V}.insert (s)$ 
5:   while  $\mathcal{Q} \neq \text{empty}$  do                       14:      $\mathcal{Q}.enqueue (s)$ 
6:      $s \leftarrow \mathcal{Q}.dequeue ()$ 
7:     for  $e, s' \mid (s, e, s') \in T$  do
8:        $\mathcal{C}.insert (s')$ 
9:     duplicateDetection ()
```

Once the queue is empty duplicate detection starts. We identify new states by removing visited states from candidate states (line 11). States remaining after this procedure are then put in the visited set and in the queue (lines 12–14).

The key point of this algorithm is the way the comparison at line 11 is conducted. In the disk-based algorithm of [16], the candidate set is kept in a memory hash table and visited states are stored sequentially in a file. New states are detected by reading states one by one from the file and deleting them from the table implementing the candidate set. States remaining in the table at the end of this process are therefore new. Hence, in this context, DDD replaces a large number of individual disk look-ups — that each would likely require to read a disk block — by a single file scan. It should be noted that duplicate detection may also be performed if the candidate set fills up, i.e., before an iteration (lines 4–9) of the algorithm has been completed.

4.1 Principle of the combination

The underlying idea of using DDD in the ComBack method is to group state reconstructions together to save the redundant execution of some events shared by different reconstruction sequences. This is illustrated by Fig. 2. The search algorithm first visits states s_0, s_1, s_2, s_3 and s_4 each mapped to a different compressed state descriptor. Later, state s is processed. It has two successors: s_4 (already met) and s_5 mapped to h_3 which is also the compressed state descriptor of s_3 . With the basic reconstruction mechanism we would have to first backtrack to s_0 , execute sequence $a.b.d$ to reconstruct s_4 and find out that e does not, from s , generate a new state, and then execute $a.b.c$ from s_0 to discover a conflict between s_5 and s_3 and hence that f generates a new state. Nevertheless, we observe some redundancies in these two reconstructions: as sequences $a.b.c$ and $a.b.d$ share a common prefix $a.b$, we could group the two reconstructions together so that $a.b$ is executed once for both s_3 and s_4 . This is where DDD can help us. As we visit s , we notice that its successors s_4 and s_5 are mapped to hash values already met. Hence, we put those in a candidate set and mark the identifiers of states that we have to reconstruct in order to check whether s_4 and s_5 are new or not, i.e., 3 and 4. Duplicate detection then consists of reconstructing marked states and to delete them from the

candidate set. This can be done by conducting a DFS starting from the initial state in search of marked states. However, as we do not want to reconstruct the whole search tree, we have to keep track of the subtree that we are interested in. Thus, we additionally store for each identifier the list of its successors in the backedge table that have to be visited. The DFS then prunes the tree by only visiting successors included in this list. On our example this will result in the following traversal order: s_0 , s_1 , s_2 , s_3 and finally s_4 .

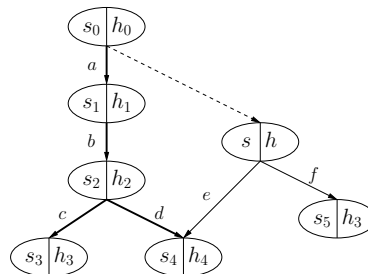


Fig. 2. The prefix $a.b$ of the reconstructing sequences of s_3 and s_4 can be shared.

4.2 The algorithm

We now propose Algorithm 4 that combines the ComBack method with DDD. As it is straightforward to extend the algorithm with a full state descriptor cache as discussed in Section 3 we only focus here on this combination.

The two main data structures in the algorithm are the queue \mathcal{Q} containing full descriptors of states to visit together with their identifiers and the visited set \mathcal{V} . The latter comprises three structures.

- As in the basic ComBack method, the *stateTable* maps compressed states to state identifiers. It is implemented as a set of pairs (h, id) where h is a hash signature and id is the identifier of a state mapped to h .
- *backedgeTable* maps each identifier id to a tuple $(id_{pred}, e, check, succs)$ where
 - id_{pred} and e are the identifier of the predecessor and the reconstructing event as in the basic ComBack method;
 - $check$ is a boolean specifying if the duplicate detection procedure must verify whether or not the state is in the candidate set;
 - $succs$ is the identifier list of its successors which must be generated during the next duplicate detection as previously explained.
- *candidates* is a set of triples (s, id_{pred}, e) where s is the full descriptor of a candidate state. In case duplicate detection reveals that s does not belong to the visited set, id_{pred} and e are the reconstruction information that will be associated with the state in *backedgeTable*.

The main procedure (lines 1–10) works basically as Algorithm 3. A notable difference is that procedure *insert* (see below) may return a two-valued answer:

NEW - if the state is surely new. In this case, the identifier assigned to the inserted state is also returned by the procedure. The state can be unconditionally inserted in the queue for a later expansion.

MAYBE - if we can not answer without performing duplicate detection.

Algorithm 4 The ComBack method extended with delayed duplicate detection

```
1:  $\mathcal{V} \leftarrow \text{empty}$  ;  $\mathcal{Q} \leftarrow \text{empty}$ 
2:  $n \leftarrow 0$  ;  $id \leftarrow \text{newState}(s_0, \text{nil}, \text{nil})$  ;  $\mathcal{Q}.\text{enqueue}(s_0, id)$ 
3: while  $\mathcal{Q} \neq \text{empty}$  do
4:    $\mathcal{V}.\text{candidates} \leftarrow \text{empty}$ 
5:   while  $\mathcal{Q} \neq \text{empty}$  do
6:      $(s, s_{id}) \leftarrow \mathcal{Q}.\text{dequeue}()$ 
7:     for  $e, s' \mid (s, e, s') \in T$  do
8:       if  $\text{insert}(s', s_{id}, e) = \text{NEW}(s'_{id})$  then  $\mathcal{Q}.\text{enqueue}(s', s'_{id})$ 
9:       if  $\mathcal{V}.\text{candidates}.\text{isFull}()$  then  $\text{duplicateDetection}()$ 
10:     $\text{duplicateDetection}()$ 


---


11: proc  $\text{newState}(s, id_{pred}, e)$  is
12:    $id \leftarrow n$  ;  $n \leftarrow n + 1$ 
13:    $\mathcal{V}.\text{stateTable}.\text{insert}(id, h(s))$ 
14:    $\mathcal{V}.\text{backedgeTable}.\text{insert}(id \rightarrow (id_{pred}, e, \text{false}, []))$ 
15:   return  $id$ 


---


16: proc  $\text{insert}(s, id_{pred}, e)$  is
17:    $ids \leftarrow \{id \mid (h(s), id) \in \mathcal{V}.\text{stateTable}\}$ 
18:   if  $ids = \emptyset$  then
19:      $id \leftarrow \text{newState}(s, id_{pred}, e)$ 
20:     return  $\text{NEW}(id)$ 
21:   else
22:      $\mathcal{V}.\text{candidates}.\text{insert}(s, id_{pred}, e)$ 
23:     for  $id$  in  $ids$  do
24:        $\mathcal{V}.\text{backedgeTable}.\text{setCheckBit}(id)$ 
25:        $\text{backtrack}(id)$ 
26:     return  $\text{MAYBE}$ 


---


27: proc  $\text{backtrack}(id)$  is
28:    $id_{pred} \leftarrow \mathcal{V}.\text{backedgeTable}.\text{getPredecessorId}(id)$ 
29:   if  $id_{pred} \neq \text{nil}$  then
30:     if  $id \notin \mathcal{V}.\text{backedgeTable}.\text{getSuccessorList}(id_{pred})$  then
31:        $\mathcal{V}.\text{backedgeTable}.\text{addSuccessor}(id_{pred}, id)$ 
32:        $\text{backtrack}(id_{pred})$ 


---


33: proc  $\text{duplicateDetection}()$  is
34:    $\text{dfs}(s_0, 0)$ 
35:   for  $(s, id_{pred}, e)$  in  $\mathcal{V}.\text{candidates}$  do
36:      $id \leftarrow \text{newState}(s, id_{pred}, e)$ 
37:      $\mathcal{Q}.\text{enqueue}(s, id)$ 
38:    $\mathcal{V}.\text{candidates} \leftarrow \text{empty}$ 


---


39: proc  $\text{dfs}(s, id)$  is
40:    $\text{check} \leftarrow \mathcal{V}.\text{backedgeTable}.\text{getCheckBit}(id)$ 
41:   if  $\text{check}$  then  $\mathcal{V}.\text{candidates}.\text{delete}(s)$ 
42:   for  $\text{succ}$  in  $\mathcal{V}.\text{backedgeTable}.\text{getSuccessorList}(id)$  do
43:      $e \leftarrow \mathcal{V}.\text{backedgeTable}.\text{getReconstructingEvent}(\text{succ})$ 
44:      $\text{dfs}(s.\text{exec}(e), \text{succ})$ 
45:    $\mathcal{V}.\text{backedgeTable}.\text{unsetCheckBit}(id)$ 
46:    $\mathcal{V}.\text{backedgeTable}.\text{clearSuccessorList}(id)$ 


---


```

Procedure *newState* inserts a new state to the visited set together with its reconstruction informations. It computes a new identifier for s , a state to insert, and update the *stateTable* and *backedgeTable* structures.

Procedure *insert* receives a state s , the identifier id_{pred} of one of its predecessors s' and the event used to generate s from s' . It first performs a lookup in the *stateTable* for identifiers of states mapped to the same hash value as s (line 17). If this search is unsuccessful (lines 18–20), this means that s has definitely not been visited before. It is unconditionally inserted in \mathcal{V} , and its identifier is returned by the procedure. Otherwise (lines 21–26), the answer requires the reconstruction of states whose identifiers belong to set ids . We thus save s in the candidate set for a later duplicate detection, set the check bit of all identifiers in ids to true so that the corresponding states will be checked against candidate states during the next duplicate detection and backtrack from these states.

The purpose of the *backtrack* procedure is, for a given state s with identifier id , to update the successor list of all the states on the path from s_0 to s in the backedge table so that s will be visited by the DFS performed during the next duplicate detection. The procedure stops as soon as a state with no predecessor is found, i.e., s_0 , or if id is already in the successor list of its predecessor, in which case this also holds for all its ancestors.

To illustrate this process, we have depicted in Fig. 3 the evolution of (a part of) the backedge table for the graph of Fig. 2. The four values specified for each state are respectively the identifier of the predecessor, the event used to reconstruct the state, the check bit (set to False or Tru \bar{e}), and the successor list. After the execution of event e from s we reach a state mapped to hash value h_4 already associated with state 4. We thus set the check bit of state 4 to true, backtrack from it and update the successor list of its ancestors 0, 1 and 2. The same treatment is performed for state 3 after the execution of f from s since the state thus reached and state 3 are mapped to the same hash value. The backtrack stops as we reach state 2 since it already belongs to the successor list of state 1.

Duplicate detection (lines 33–38) is conducted each time the candidate set is full (line 9), i.e., it reaches a certain peak size, or the queue is empty (line 10).

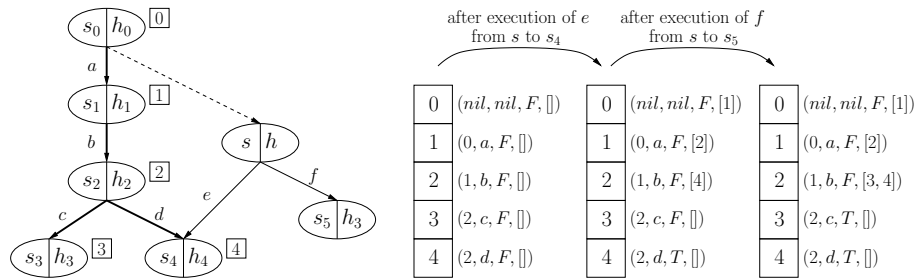


Fig. 3. Evolution of the backedge table after the execution of e and f from s

Using the successor lists constructed by the backtrack procedure, we initiate a depth-first search from s_0 (see procedure *dfs*). Each time a state with its check bit set to true is found (line 41) we delete it from the candidate set if needed. When a state leaves the stack we set its check bit to false and clear its successor list (lines 45–46). Once the search finishes (lines 35–37) any state remaining in the candidate set is new and can be inserted into the queue and the visited set.

4.3 Additional comments

We discuss several issues regarding the algorithm proposed in this section.

Memory issues Our algorithm requires the storage of some additional information used to keep track of states that must be checked against the candidate set during duplicate detection. This comprises for each state a boolean value (the *check* bit) and a list of successors that must be visited. As any state may belong to the successor list of its predecessor in the backedge table, the memory overhead is theoretically one bit plus one integer per state. However, our experiments reveal (see Section 5) that even very small candidate sets show good performance. Therefore, successor lists are usually short and the extra memory consumption low. We did not find any model for which the algorithm of [18] terminated whereas ours did not due to a lack of memory.

Grouping reconstructions of queued states In [18] the possibility to reduce memory usage by storing identifiers instead of full state descriptors in the queue (Variant 4 in Section 5) was mentioned. This comes at the cost of an additional reconstruction per state required to get a description of the state that can be used to generate its successors. The principle of grouping state reconstructions can also be applied to the states waiting in the queue. The idea is to dequeue blocks of identifiers from the queue instead of individual ones and reconstruct those in a single step using a procedure similar to *dfs* given in Algorithm 4.

Compatibility with depth-first search A nice characteristic of the basic ComBack method is its total decoupling from the search algorithm thereby making it fully compatible with, e.g., LTL model checking [2,7]. Delaying detection from state generation makes an algorithm implicitly incompatible with a depth-first traversal where the state processed is always the most recent state generated. At first glance, the algorithm proposed in this section also belongs to that category. However, we can exploit the fact that the insertion procedure can decide if a state is new without actually putting it in the candidate set (if the hash value of the state has never been met before). The idea is that the search can progress as long as new states are met. If some state is then put in the candidate set the algorithm puts a marker on the stack to remember that a potentially new state lies here. Finally, when a state is popped from the stack, duplicate detection is performed if markers are present on top of the stack. If we

find out that some of the candidate states are new, the search can continue from these ones. This makes delayed detection compatible with depth-first search at the cost of performing additional detections, during the backtrack phase of the algorithm.

5 Experimental results

We report in this section the data we collected during several experiments with the proposed techniques. We used the ASAP verification tool [12] where we have implemented the algorithms described in this article. A nice characteristic of ASAP is its independence from the description language of the model. This allowed us to perform experimentations on DVE models taken from the BEEM database [14] and on CPN models taken from our own collection.

Experimenting with caching strategies In this first experiment we evaluated the different strategies proposed in Section 3. We picked out 102 instances of the BEEM database having from 100,000 to 10,000,000 states and run the ComBack algorithm of [18] using BFS with 6 caching strategies and 3 sizes of cache (100, 1,000 and 10,000 states). Out of these 6 strategies 3 are simple: R (Random, with a replacement probability $p = 0.5$), F (Fifo), H (Heuristic); and 3 are combinations⁵ of the first ones: F(20)-H(80), F(50)-H(50) and F(80)-H(20). We measured after each run the number of event executions that were due to state reconstructions. The results are summarized in table 1.

Strategy F performs well compared to R but it seems that its performance degrades (in comparison) as we allocate more states to the cache. This is also confirmed by the fact that the combination F-H seems to perform better for a large cache when the proportion of states allocated to the fifo sub-cache is low. Apparently with this strategy we quickly reach a limit where all (or most of) the forward transitions lead to a cached (or new) state and most backward transitions lead to a non cached state. Such a cache failure always implies backtracking to the initial state (the fifo strategy implies that if a state is not cached none of its ancestors in the backedge table is cached) which can be quite costly. Beyond this point, allocating more states to the cache is almost useless.

The performance of strategy H is poor for small caches but progresses well compared to strategy F. With this strategy, most transitions will be followed by a state reconstruction. However, our heuristic works rather well and reconstructing sequences are usually much shorter than with strategy F. Still, strategy H is usually outperformed by strategy F due to a high presence of forward transitions in state spaces [13]. To sum up, strategy F implies few reconstructions but long sequences and strategy H has the opposite characteristics.

From these observations it is not surprising to see that the best strategy is to maintain a small fraction of the cache with strategy F and the remainder with

⁵ F(X)-H(Y) denotes the combination where X% of the cache is allocated to a fifo sub-cache and Y% is allocated to a heuristic based sub-cache.

Table 1. Evaluation of caching strategies on 102 DVE instances.

Cache size	Strategy R	Strategy F	Strategy H	Strategy F(20)-H(80)	Strategy F(50)-H(50)	Strategy F(80)-H(20)
10^2	1.0	0.429	1.128	0.436	0.397	0.390
10^3	1.0	0.437	1.178	0.364	0.347	0.355
10^4	1.0	0.488	0.742	0.255	0.262	0.302
10^2	0	7	2	21	34	42
10^3	0	3	1	51	38	17
10^4	0	7	3	80	14	9

Top rows: Average on all instances of the number of event executions due to state reconstruction with this strategy reported to the same number obtained with strategy R. Bottom rows: Number of instances for which this strategy performed best.

strategy H, that is to keep a small number of recently visited states and many strategic states from previous levels that will help us shorten reconstructing sequences.

Out of these 102 instances we selected 4 instances that have some specific characteristics (`brp2.6`, `cambridge.6`, `firewire_tree.5` and `synapse.6`) and evaluated strategies F, H and F(20)-H(80) with different sizes of cache ranging from 1,000 to 10,000. Data collected are plotted on figure 4. On the x-axis are the different cache sizes used. For each run we recorded the number of event executions due to reconstructions and reported it to the same number obtained with strategy F. For instance, with `brp2.6` and a cache of 4,000 states, reconstructions generated approximately three times more event executions with strategy H than with strategy F. We also provide the characteristics of these graphs in terms of number of states and transitions, average degree, number of levels and number of forward transitions as a proportion of the overwhole number of transitions.

The graph of `firewire_tree.5` only has forward transitions, which is common for leader election protocols. Therefore, a sufficiently large fifo cache is the best solution. This is one of the few instances where increasing the cache size benefits strategy F more than H. Moreover its average degree is high, which leads to a huge number of reconstructions with strategy H. On the opposite side the graph of `cambridge.6` has a relatively large number of backward transitions. Increasing the fifo cache did not bring any substantial improvement: from 262,260,647 executions with a cache size of 1,000 it went down to 260,459,235 executions with a cache size of 10,000. Strategy H is especially interesting for `synapse.6` as its graph has a rather unusual property: a low fraction of its states have a high number of successors (from 13 to 18). These states are thus shared by many reconstructing sequences and, using our heuristic, they are systematically kept in the cache. Thus, strategy H always outperforms strategy F even for small caches. The out-degree distribution of the graph of `brp2.6` has the opposite characteristics: 49% of its states have 1 successor, 44% have 2 successors and the other states have 0 or 3 successors. Therefore, there is no state that is

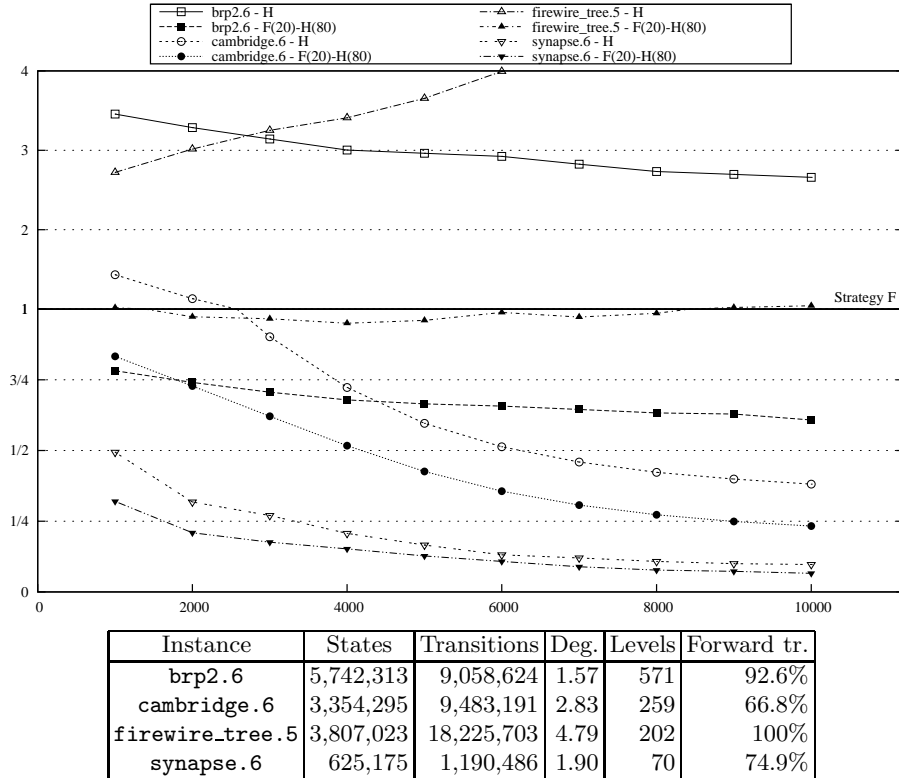


Fig. 4. Evolution of strategies F, H and F(20)-H(80) on some selected instances.

really interesting to keep in the cache. This is evidenced by the fact that the relative progressions of heuristic based strategies are not so good. It goes from 3.456 to 2.660 for strategy H and from 0.782 to 0.608 for strategy F(20)-H(80).

Experimenting with delayed duplicate detection (DDD) To experiment with delayed detection we picked out 94 DVE instances from the BEEM database (all instances having between 500,000 and 50,000,000 states) and 2 CPN instances from our own database. The ComBack method was especially helpful for nets `dymo` and `erdp` that model two industrial protocols — a routing protocol [4] and an edge router discovery protocol [11] — and have rather large descriptors (1,000 - 5,000 bytes).

Table 2 summarizes our observations. Due to a lack of space we only report the data for some DVE instances but still provide the average on all instances. We used caching strategy F(20)-H(80), as it is apparently the best we proposed, with a cache size of 10,000. For each instance we performed 4 tests: one with a standard storage method, i.e., full state descriptors are kept in the visited

Table 2. Evaluation of delayed duplicate detection on DVE and CPN instances.

Std. T	Type of items in the queue	ComBack					
		No DDD		DDD(10^2)		DDD(10^3)	
		T↑	E↑	T↑	E↑	T↑	E↑
DVE instances							
brp.5*		17,740,267 states			36,903,290 transitions		
23.1	SD	11.02	21.19	8.08	7.56	9.01	7.10
	ID	35.03	69.12	14.57	17.48	15.74	15.37
cambridge.7		11,465,015 states			54,850,496 transitions		
195	SD	14.12	35.88	2.63	4.40	2.30	3.49
	ID	18.54	44.50	3.22	5.92	2.96	5.03
iprotocol.5*		31,071,582 states			104,572,634 transitions		
63.1	SD	8.74	11.71	5.07	2.97	4.93	2.65
	ID	21.12	30.81	6.72	5.44	6.63	4.91
pgm_protocol.8		3,069,390 states			7,125,121 transitions		
18.45	SD	2.07	2.64	1.65	1.48	1.59	1.35
	ID	16.87	42.04	4.60	8.05	4.42	7.31
rether.6		5,919,694 states			7,822,384 transitions		
13.0	SD	3.86	7.23	3.16	3.89	3.17	3.64
	ID	24.38	75.16	9.11	19.99	9.33	19.18
synapse.6		625,175 states			1,190,486 transitions		
1.4	SD	2.42	1.74	2.50	1.45	2.42	1.43
	ID	3.50	3.54	3.57	3.01	3.57	3.01
Average on 94 instances							
	SD	4.92	7.06	3.92	3.44	4.02	3.06
	ID	10.11	18.69	5.49	6.31	5.61	5.69
CPN instances							
dymo.6*		1,256,773 states			7,377,095 transitions		
2,115	SD	2.97	2.88	1.65	1.42	1.72	1.37
	ID	4.12	4.39	1.93	1.94	1.93	1.85
erdp.3*		2,344,208 states			18,739,842 transitions		
5,425	SD	3.99	6.17	2.19	2.69	2.10	2.28
	ID	4.37	7.50	2.15	3.15	1.92	2.70
Average on 2 instances							
	SD	3.48	4.52	1.92	2.05	1.91	1.82
	ID	4.24	5.94	2.04	2.54	1.92	2.27

*: standard search out of memory. Time in column Std. is approximated.
Type of items in the queue: SD (full state descriptor) or ID (state identifier).

set, (column `Std.`), one with the `ComBack` method without delaying detection (column `ComBack - No DDD`) and two with delayed detection enabled with a candidate set of size 100 and 1,000 (columns `ComBack - DDD(102)` and `ComBack - DDD(103)`). Each test using the `ComBack` method actually comprises two runs: one keeping full state descriptors in the queue (line `SD`) and one keeping only identifiers in the queue (line `ID`) — as described in [18], Variant 4 of Section 5. For this second run, we used the optimization described in Section 4.3 that consists of grouping the reconstruction of queued identifiers. Each block of identifiers dequeued to be reconstructed had the same size as the candidate set. Hence, when DDD was not used this optimization was turned off. In column `Std. - T` we provide the execution time in seconds using a standard search algorithm. In columns `T↑` we measure the run-time increase (compared to the standard search) as the ratio $\frac{\text{execution time of this run}}{T \text{ (with standard search)}}$ and in column `E↑` the increase of the number of event executions as the ratio $\frac{\text{event executions during this run}}{\text{transitions of the graph}}$. Hence, a value of 1 in this column means that we executed exactly the same number of events as the basic algorithm and that no state reconstruction occurred. Some runs using standard storage ran out of memory. This is indicated by a `*`. For these, we provide the time obtained with hash compaction as a lower approximation.

We first observe that DDD is indeed useful to save the redundant exploration of transitions during reconstruction even with small candidate sets. Typically we can reduce the number of events executed by a factor of 3 or even more if we group the reconstruction of queued identifiers. It seems that, using BFS, states generated successively are “not so far” in the graph so their reconstructing sequences are quite similar, which allows many sharings.

However, this reduction does not always impact on the time saved as we could expect. Indeed DDD is much more interesting for CPN models than DVE models. If we consider for example the average made on the 94 DVE instances with our optimization disabled we divided the number of events executed by more than 2 ($7.06 \rightarrow 3.44$) whereas the average time slightly decreased ($4.92 \rightarrow 3.92$). The reason is that executing an event is much faster for DVE models than for CPN models. Events are typically really simple in the DVE language, e.g., a variable incrementation, whereas they can be quite complex with CPNs and include the execution of some embedded code. Therefore, the only fact of maintaining the candidate set or successors lists has a non negligible impact for DVE models which means that DDD reduces time only if the number of executions decreases in a significant way, e.g., instance `cambridge.7`.

Grouping the reconstruction of queued states can save a lot of executions, especially for long graphs like those of `brp.5` and `cambridge.7`. It should be noted that by storing identifiers in the queue we obtain an algorithm that bounds the number of full state descriptors kept in memory. Hence, we can theoretically consume less memory compared to an algorithm based on hash compaction which has to store full descriptors in the queue. This was indeed the case for nets `dymo.6` and `erdp.3`. Both have rather wide graphs: their largest levels contains approximately 10% of the state space and so contained the queue as it reached its peek size.

6 Conclusion

The ComBack method has been designed to explicitly store large state spaces of models with large state descriptors. The important reduction factor it may provide is however counterbalanced by an increase in run-time due to the on-the-fly reconstructions of states. We proposed in this work two ways to tackle this problem. First, some strategies have been devised in order to efficiently maintain a full state descriptor cache, used to perform less reconstructions and shorten the length of reconstructing sequences. Second, we combined the method with delayed duplicate detection that allows to group reconstructions and save the execution of events that are shared by multiple sequences. We have implemented these two extensions in ASAP and performed an extensive experimentation on both DVE models from the BEEM database and CPN models from our own collection. These experiments validated our proposals on many models. Compared to a random replacement strategy, a combination of our strategies could, on an average made on a hundred of DVE instances, decrease the number of transitions visited by a factor of four. We also saw that delaying duplicate detection is efficient even with very small candidate sets. In the best cases, we could even approach the execution time of a hash compaction based algorithm. Moreover, by storing identifiers instead of full descriptors in the queue we bound the number of full state descriptors that reside in memory. Hence, our data structures can theoretically consume less memory during the search than hash compaction structures. We experienced this situation on several instances.

In this work, we mainly focused on caching strategies for breadth-first search. BFS is helpful to find short error-traces for safety properties, but not if we are interested in the verification of linear time properties that is inherently based on depth-first search. The design of strategies for other types of search is thus a future research topic. In addition, the combination with delayed duplicate detection opens the way to an efficient multi-threaded algorithm based on the ComBack method. The underlying principle would be to have some threads exploring the state space and visiting states while others are responsible for performing duplicate detection. We are currently working on such an algorithm.

References

1. J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of Logic In Computer Science*, pages 428–439, 1990.
2. J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of Formal Methods*, Volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, 1999.
3. E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in Systems Design*, 9(1-2):105–131, 1996.
4. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and initial validation of the dymo routing protocol for mobile ad-hoc networks. In *Proceedings of Application and Theory of Petri Nets*, Volume 5062 of *Lecture Notes in Computer Science*, pages 152–170. Springer-Verlag, 2008.

5. S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proceedings of SPIN – Software Model Checking*, Volume 3639 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2005.
6. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, Volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
7. P. Godefroid and G.J. Holzmann. On the verification of temporal properties. In *Proceedings of Protocol Specification, Testing and Verification*, Volume C-16 of *IFIP Transactions*, pages 109–124. North-Holland, 1993.
8. G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
9. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 1-3*. Springer-Verlag, 1992-1997.
10. W. Knottenbelt, M. Mestern, P. Harrison, and P. Kritzing. Probability, parallelism and the state space exploration problem. In *Proceedings of Modelling, Techniques and Tools*, Volume 1469 of *Lecture Notes in Computer Science*, pages 165–179. Springer-Verlag, 1998.
11. L.M. Kristensen and K. Jensen. Specification and validation of an edge router discovery protocol for mobile ad-hoc networks. In *Proceedings of Integration of Software Specification Techniques for Applications in Engineering*, Volume 3147 of *Lecture Notes in Computer Science*, pages 248–269. Springer-Verlag, 2004.
12. L.M. Kristensen and M. Westergaard. The ASCoVeCo state space analysis platform. In *Proceedings of Practical Use of Coloured Petri Nets and the CPN Tools*, Volume 584 of *DAIMI-PB*, pages 1–6, 2007. Available at: <http://www.daimi.au.dk/~ascoveco/asap.html>.
13. R. Pelánek. Typical structural properties of state spaces. In *Proceedings of SPIN – Software Model Checking*, Volume 2989 of *Lecture Notes in Computer Science*, pages 5–22. Springer-Verlag, 2004.
14. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proceedings of SPIN – Software Model Checking*, Volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer-Verlag, 2007.
15. U. Stern and D.L. Dill. Improved probabilistic verification by hash compaction. In *Proceedings of Correct Hardware Design and Verification Methods*, Volume 987 of *Lecture Notes in Computer Science*, pages 206–224. Springer-Verlag, 1995.
16. U. Stern and D.L. Dill. Using magnetic disk instead of main memory in the Mur ϕ verifier. In *Proceedings of Computer Aided Verification*, Volume 1427 of *Lecture Notes in Computer Science*, pages 172–183. Springer-Verlag, 1998.
17. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting transition locality in automatic verification. In *Proceedings of Correct Hardware Design and Verification Methods*, Volume 2144 of *Lecture Notes in Computer Science*, pages 259–274. Springer-Verlag, 2001.
18. M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The ComBack method - extending hash compaction with backtracking. In *Proceedings of Application and Theory of Petri Nets*, Volume 4546 of *Lecture Notes in Computer Science*, pages 445–464. Springer-Verlag, 2007.
19. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proceedings of Computer Aided Verification*, Volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.