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

Sami Evangelista[1], Michael Westergaard[1], and Lars M. Kristensen[2]

[1] Computer Science Department, Aarhus University, Denmark
{evangeli,mw}@cs.au.dk
[2] Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

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 [20], and an extension through delayed duplicate detection that allows to group reconstructions together to save redundant work.

Keywords: explicit state model checking, state explosion problem, state space reduction, hash compaction, delayed duplicate detection.

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. Its nodes are system states and its 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 is 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 for 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 [4], exploit the independence of some transitions to reduce the exploration of redundant interleavings [8], or encode the state graph using compact data structures such as binary decision diagrams [2].

Hash compaction [16,21] 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*

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

K. Jensen, J. Billington, and M. Koutny (Eds.): ToPNoC III, LNCS 5800, pp. 189–215, 2009.

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 which implies that the hash compaction method may not explore all reachable states. The probability of *hash collisions* can be reduced in several ways, e.g., by using multiple hash functions [12,16], but the method still cannot guarantee full coverage of the state space. Partial coverage of the state space 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 [20] extends hash compaction with a backtracking mechanism that allows reconstruction of full state descriptors from compressed ones and thus resolves conflicts on-the-fly to guarantee full coverage of the state space. Its underlying principle is to store for each state a sequence of events whose execution can generate this state. Thus, when the search algorithm checks if it already visited a state s, it can reconstruct full state descriptors for states mapped to the same hash value as s and compare them to s. Only if none of the states reconstructed is equal to s does the algorithm consider it as a new state.

The ComBack method 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 cost in time: 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 [10] or Coloured Petri nets [11]. Experiments made in [20] report an increase in run-time up 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. Firstly, starting from the proposal of [20] to use a cache of full state descriptors to shorten sequences, we first propose different caching strategies. Secondly, we extend the ComBack method with delayed duplicate detection, a technique widely used by disk-based model checkers [17]. 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 into 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 exploration 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 [19] which implements the techniques proposed in this paper. Finally, Section 6 concludes this paper.

2 Background

In this section we give the basic ingredients required for understanding the rest of this paper and provide a brief overview of the ComBack method [20].

2.1 Transition Systems

As the methods proposed in this paper are not linked to a specific formalism they will be developed in the framework of labeled transition systems, 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 set of* ***states****,* E *is a 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 state from the queue, generating its successors and inserting the successor states that have not been visited so far into both the visited set and the queue for later exploration. We use the term of *state expansion* to refer to this process.

[1] The ComBack method can be extended to support non-deterministic transition systems. The interested reader may consult Section 5 of [20] that describes such an extension.

Algorithm 1. A classical search algorithm.

```
1: V ← empty ; V.insert (s_0)
2: Q ← empty ; Q.enqueue (s_0)
3: while Q ≠ empty do
4:     s ← Q.dequeue ()
5:     for e, s' | (s, e, s') ∈ T do
6:         if s' ∉ V then
7:             V.insert (s') ; Q.insert (s')
```

Algorithm 2. A search algorithm based on hash compaction.

```
1: V ← empty ; V.insert (h(s_0))
2: Q ← empty ; Q.enqueue (s_0)
3: while Q ≠ empty do
4:     s ← Q.dequeue ()
5:     for e, s' | (s, e, s') ∈ T do
6:         if h(s') ∉ V then
7:             V.insert (h(s')) ; Q.insert (s')
```

Using hash compaction [21], items stored in the visited set are not actual state descriptors but compressed descriptors, typically 32-bit integers, 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, 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 therefore 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 the 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:

- A *state number*, or identifier, is assigned to each visited state s.
- A *state table* stores for each compressed state descriptor a *collision list* of state numbers for visited states mapped to this compressed state descriptor.
- 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 the visited set if needed. The insertion procedure can be illustrated with the help of Fig. 1, which

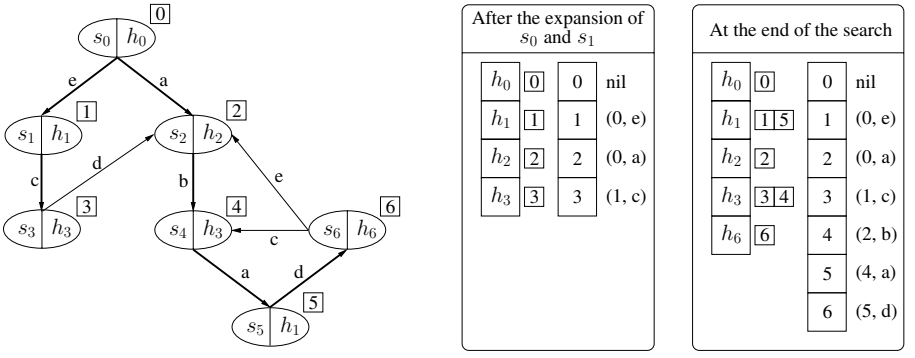


Fig. 1. A state space and the state and backedge tables at two stages

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 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 is detected, a single state is associated with each hash value in the state table (the left table of the first rounded box). 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—as $h(s_3) = h(s_4)$—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 value 3. The comparison of state 3 (of which we no longer 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 backtracking, i.e., $e.c$, is executed on the initial state. Finally, a comparison between s_3 and s_4 indicates that s_4 is new. We therefore assign a new identifier (4) to s_4, insert it in the collision list of hash value h_3 and insert the entry $4 \rightarrow (2, b)$ in the backedge table.

Throughout this article the term *state reconstruction* (or simply *reconstruction*) is used to refer to the process of backtracking to the initial state and then executing a sequence of events to retrieve a full state descriptor. The sequence executed will be called the *reconstructing sequence*.

This storage scheme is especially suited to systems exhibiting large state vectors as it represents each state in the visited set with only a few bytes. The only elements of the state and backedge tables that are still dependent on the underlying model are the events stored to reconstruct states. In the case of Coloured 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 incurs an (important) run-time penalty due to the reconstruction mechanism. After a state s has been reached it will be reconstructed once for each additional incoming arc of the state, 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). Evidence of this is shown in Table 1 of [20] 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 on reconstructing states depends, to a large extent, on the complexity of executing an event that ranges from trivial (e.g., for Place/Transition Nets) to high, e.g., for Promela or Coloured Petri Nets for which executing an event may include the execution of embedded code.

3 Caching Strategies

A cache that maps 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 again the configuration of Fig. 1. Caching the mapping $2 \rightarrow s_2$ may be useful in two ways:

- To avoid the reconstruction of state 2. A lookup in the cache directly returns state s_2, which saves the backtrack to s_0 and the execution of event a.
- For the reconstruction of state 4. During the backtrack to s_0 the algorithm finds out that state 2 is cached, retrieves its descriptor and only executes event b from s_2 to obtain s_4, once again saving the execution of event a.

We now propose four strategies to implement such a 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 [14].

Random cache. The simplest 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 chosen randomly.*

Fifo cache. A common characteristic of state spaces is the high proportion of forward transitions[2], 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 estimate the probability that s belongs to some reconstructing sequence. This increases if many states point to

[2] 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*.

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 [18]: 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 will only have a small impact 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.

Distance based cache. Heuristic H does not take into account the presence of already cached states. Yet it may be useless to cache a state with a high value of H if, for example, the state it points to in the backedge table is itself in the cache. The last strategy we propose is a slight variation of strategy H. It is also based on heuristic H but it is parameterized by an integer k that specifies the shortest possible sequence between two cached states.

> **Strategy D:** *Apply strategy H. Do not cache a state if, in the backedge table, one of its ancestors of degree k or less is already cached.*

Other possibilities are available. In [6] a reduction technique also based on state reconstruction is proposed. The algorithm is parameterized by an integer k and only caches states at levels $0, k, 2 \cdot k, 3 \cdot k \ldots$. The motivation of this strategy is to bound the length of reconstructing sequences to $k - 1$. As presented, the strategy in [6] 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 visited 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, i.e., states reached via event executions but not checked to be in the visited set. 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 *candidates* 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. 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 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 [17], 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 reading 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 to save the redundant execution of 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 state 4 and find out that e does not, from s, generate

Algorithm 3. A generic search algorithm using delayed duplicate detection

```
1: V ← empty ; V.insert (s0)                10: proc duplicateDetection () is
2: Q ← empty ; Q.enqueue (s0)               11:     new ← candidates \ V
3: while Q ≠ empty do                       12:     for s ∈ new do
4:     candidates ← empty                   13:         V.insert (s)
5:     while Q ≠ empty do                   14:         Q.enqueue (s)
6:         s ← Q.dequeue ()
7:         for e, s' | (s, e, s') ∈ T do
8:             candidates.insert (s')
9:     duplicateDetection ()
```

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 redundancy 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 sub-tree 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.

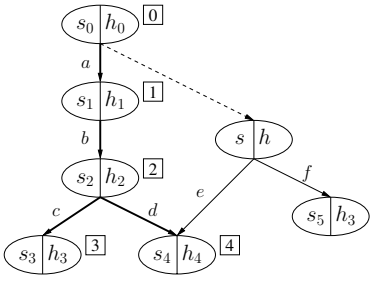


Fig. 2. The prefix $a.b$ of the reconstructing sequences of s_3 and s_4 can be shared

4.2 The Combined 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 (discussed in Section 3) we only focus on the basic combination here.

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: a *stateTable* as in the basic ComBack method, a *backedgeTable* mapping state numbers to predecessors and some auxiliary information used by the algorithm, and a *candidates* set consisting of states that may or may not be new states.

First, consider again the LTS from Fig. 2. In Fig. 3, we see the example annotated using the same notation used in Fig. 1 at the left. That is, each ellipse represents a state and the hash value of the state is written in the right part of the ellipse. The state number is shown in a square next to the ellipse. The search has investigated states s_0, s_1, s_2, s_3 and s_4, and is about to investigate s. We have not shown the state table, which simply maps h_i to i for $i = 0 \ldots 4$. We have just picked s from the queue, which is now empty. At the right of Fig. 3, we show the backedge table before executing any event from s, after executing

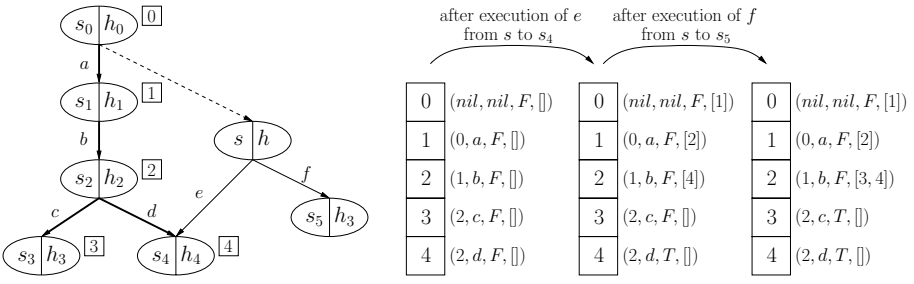


Fig. 3. Evolution of the backedge table after the execution of e and f from s

the event e, and after executing the event f. The first two components of the tuples in the backedge table are the state number of the predecessor and the event executed to reach the state as in Fig. 1. The third component is a boolean telling whether the state should be compared to the states in the candidate set. Before executing any events, this boolean is set to False for all states as we have just performed duplicate detection. When executing event e, we reach state s_4, which has hash value h_4 corresponding to state 4. We therefore add s_4 to the candidate set and set the boolean value to True for state 4 in the backedge table (the middle one in Fig. 3). The last component of the tuple in the backedge is a list of interesting successors, i.e., successor states that should be investigated when performing duplicate detection. We backtrack from state 4 to the initial state. For state number 2 we add the successor 4 as interesting, for state 1 we add state 2, and for the initial state 0 we add state 1 as can be seen in the middle backedge table in Fig. 3. After executing f from s, we obtain s_5, which has hash value h_3 corresponding to state 3. We add s_5 to the candidate set (which now consists of s_4 and s_5), and flags that state 3 should be compared against the candidate set. We also update the lists of interesting successors, adding 3 at state 2. As state number 2 is already marked as interesting successor of state 1, we stop backtracking at this point.

We can now perform duplicate detection by starting in the initial state. We see that we need to expand state 1, so we execute the event a. In state 1, we execute b to reach 2. In state 2, we must execute events to reach states 3 and 4. Executing c to reach state number 3, we obtain state s_3. As we have set the boolean to True, we attempt to remove s_3 from the candidate set. The candidate set did not contain s_3, so it still contains states s_4 and s_5. The successor list of state 3 is empty, so we execute d from state 2 to reach state 4. This also has the boolean set to True, so we remove s_4 from the candidate set, which now only contains the state s_5. Duplicate detection is then done, and we continue the search from s_5.

To sum up, the *stateTable* of the visited set $\mathcal{V}$ maps hash values to state numbers exactly like in the basic ComBack method, the *backedgeTable* maps a state number id to a tuple $(id_{pred}, e, check, succs)$ where

Algorithm 4. The ComBack method extended with delayed duplicate detection

```
 1: V ← empty ; Q ← empty
 2: n ← 0 ; id ← newState (s_0, nil, nil) ; Q.enqueue (s_0, id)
 3: while Q ≠ empty do
 4:     V.candidates ← empty
 5:     while Q ≠ empty do
 6:         (s, s_id) ← Q.dequeue ()
 7:         for e, s' | (s, e, s') ∈ T do
 8:             if insert (s', s_id, e)= NEW(s'_id) then Q.enqueue (s', s'_id)
 9:         if V.candidates.isFull () then duplicateDetection ()
10:     duplicateDetection ()
11: proc newState (s, id_pred, e) is
12:     id ← n ; n ← n + 1
13:     V.stateTable.insert (id, h(s))
14:     V.backedgeTable.insert (id → (id_pred, e, false, []))
15:     return id
16: proc insert (s, id_pred, e) is
17:     ids ← {id | (h(s), id) ∈ V.stateTable}
18:     if ids = ∅ then
19:         id ← newState (s, id_pred, e)
20:         return NEW(id)
21:     else
22:         V.candidates.insert (s, id_pred, e)
23:         for id in ids do
24:             V.backedgeTable.setCheckBit (id)
25:             backtrack (id)
26:         return MAYBE
27: proc backtrack (id) is
28:     id_pred ← V.backedgeTable.getPredecessorId (id)
29:     if id_pred ≠ nil then
30:         if id ∉ V.backedgeTable.getSuccessorList (id_pred) then
31:             V.backedgeTable.addSuccessor (id_pred, id)
32:             backtrack (id_pred)
33: proc duplicateDetection () is
34:     dfs (s_0, 0)
35:     for (s, id_pred, e) in V.candidates do
36:         id ← newState (s, id_pred, e)
37:         Q.enqueue (s, id)
38:     V.candidates ← empty
39: proc dfs (s, id) is
40:     check ← V.backedgeTable.getCheckBit (id)
41:     if check then V.candidates.delete (s)
42:     for succ in V.backedgeTable.getSuccessorList (id) do
43:         e ← V.backedgeTable.getReconstructingEvent (succ)
44:         dfs (s.exec (e), succ)
45:     V.backedgeTable.unsetCheckBit (id)
46:     V.backedgeTable.clearSuccessorList (id)
```

- 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.

The *candidates* set 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 comprise the reconstruction information that will be associated with the state in *backedgeTable*.

Consider again Algorithm 4 which shows the ComBack method extended with delayed duplicate detection. 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 cannot answer without performing duplicate detection.

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

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.

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). 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 below several issues regarding the proposed algorithm.

Memory issues. Our algorithm requires the storage of 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 memory performance. Therefore, successor lists are usually short and the extra memory consumption low. We did not find any model for which our algorithm aborted due to a lack of memory, but where the one of [20] did terminate.

Grouping reconstructions of queued states. In [20] 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 full descriptor for 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 [3,9]. Delaying detection from state generation makes an algorithm implicitly incompatible with a depth-first traversal in which 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 depth-first search algorithm.

5 Experimental Results

We report in this section the data we collected during several experiments with the proposed techniques[3]. We used the ASAP verification tool [19] where we

[3] Additional data on these experiments may be found in [7].

have implemented the algorithms described in this article. A nice characteristic of ASAP is its independence from the description language of the model. This allows us to perform experiments on CPN (Coloured Petri net) and DVE (the input language of the DiVinE verification tool [1]) models. All CPN models come from our own collection. DVE models were taken from the BEEM database [15] though we did not consider models belonging to the families "Planning and scheduling" and "Puzzles" as these are mostly very simple models, e.g., the towers of Hanoi, that have nothing to do with real life examples.

For CPN models, the hash function used is defined inductively on the state of the model. In CPNs, a state is a *marking* of a set of *places*. Each marking is a *multi-set* over a given *type*. We use a standard hash function for each type. We extend this hash function to multi-sets by using a *combinator function*, which takes two hash values and returns a new hash value. We extend the hash functions on markings of places to a hash function of the entire model by using the combinator function on the place hash functions. We proceed exactly the same way for DVE instances, except that the components of the system are now variables, channels, and process states rather than places. The performance of these functions in term of conflicts is usually very good. Typically with a 32 bit hash signature there is generally no collision for small instances (100,000 states or less) and for larger instances (e.g., up to 10^6–10^7 states), we can reasonably expect to cover more than 95% of the state space using a hash compaction based algorithm. The quality of this function is evidenced by Table 1 in [20] that reports the number of collisions for several CPN instances.

This section uses several abbreviations. Table 1 lists all abbreviations used in this section and it provides a short description of all selected models. Each instance name consists of the concatenation of a model name followed by an instance number (i.e., an instantiation of the model parameters). For instance `firewire_tree.5` is the 5^{th} instance (in the BEEM database) of the model named `firewire_tree`. The instantiation values of parameters are not relevant for this study.

5.1 Experiment 1: Evaluation of Caching Strategies

In this first experiment we evaluated the different strategies proposed in Section 3. We selected 143 DVE instances having from 10,000 to 10,000,000 states and ran the ComBack algorithm of [20] using BFS with 10 caching strategies and 4 sizes of cache expressed as a fraction of the state space size: { 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1} }. Of these 10 strategies 4 are simple: R (with a replacement probability $p = 0.5$), F, H, D (with a minimal distance $k = 5$ between cached states); and 6 are combinations of the first ones: F_{20}-H_{80}, F_{50}-H_{50}, F_{80}-H_{20}, F_{20}-D_{80}, F_{50}-D_{50} and F_{80}-D_{20}. We measured after each run the number of event executions that were due to state reconstructions. The results are summarized in Table 2. On the top four rows we give, for each strategy, the average over all instances of the number of event executions due to state reconstruction with this strategy divided by the same number obtained with strategy R. The bottom four rows report for each strategy the number of instances for which it performed best.

Table 1. List of abbreviations and short description of models used in Section 5

Abbreviations	
BFS	Breadth-First Search
CPN	Colored Petri Net
DDD	Delayed Duplicate Detection
DVE	Input language of the DiVinE toolset
Caching strategies	
D	Distance based caching strategy
F	Fifo caching strategy
H	Heuristic based caching strategy
R	Random caching strategy
F_X-D_Y	Combination of caching strategies F and D where X% (resp. Y%) of the cache is allocated to a Fifo sub-cache (resp. Distance based sub-cache)
F_X-H_Y	Combination of caching strategies F and H where X% (resp. Y%) of the cache is allocated to a Fifo sub-cache (resp. Heuristic based sub-cache)
Introduced in Experiment 2	
Std.	Execution time using a standard storage method, i.e., without any reduction technique, or using hash compaction if the classical search ran out of memory
DDD(X)	Characterize a search using the ComBack method with delayed duplicate detection. X denotes the size of the candidate set as a fraction of the cache size.
T↑	Denote, for a search using the ComBack method, the increase of the execution time as the ratio $\frac{\text{execution time of this run}}{\text{execution time with a standard search}}$.
E↑	Denote, for a search using the ComBack method, the increase of the number of event executions as the ratio $\frac{\text{event executions during this run}}{\text{transitions of the graph}}$.
Introduced in Experiment 3	
S	Caching strategy used
F_C	Proportion of full state descriptors allocated to the cache
F_{CS}	Proportion of full state descriptors allocated to the candidate set
F_Q	Proportion of full state descriptors allocated to the queue

Models	
CPN models	
`dymo`	Dynamic MANET on-demand routing protocol (from [5])
`erdp`	Edge router discovery protocol (from [13])
`protocol`	Simplified stop and wait protocol
`telephones`	Telecommunication service
DVE models	
`brp`	Bounded retransmission protocol
`brp2`	Timed version of the bounded retransmission protocol
`cambridge`	Cambridge ring protocol
`firewire_link`	Layer link protocol of the firewire protocol (IEEE 1394)
`firewire_tree`	Tree identification protocol of the firewire protocol (IEEE 1394)
`needham`	Needham-Schroeder public key authentication protocol
`synapse`	Synapse cache coherence protocol

These latter are only given for information and our interpretations will be based on the values of the top rows.

Strategy F performs well compared to strategy R but its performance degrades as we allocate more states to the cache. This is also confirmed by the fact that combinations F-H and F-D seem 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) 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. We even see that strategy F is largely outperformed by strategy R for the largest cache size we experimented with.

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 for small cache sizes due to a high presence of forward transitions in state spaces [14]. To sum up, strategy F implies few reconstructions but long reconstructing sequences and strategy H has the opposite characteristics.

As expected, strategy D improves strategy H although only slightly. It prevents the algorithm from caching two states that are linked by too short a path in the backedge table. However, for the largest cache size we experimented with, the algorithm could not use all the memory allocated to the cache due to this restriction. This is the only case where strategy D performed worse than H.

From all these observations it is not surprising to see that the best strategy is to always keep a small fifo cache and allocate remaining memory to a second level cache maintained with strategy D, that is, to keep a small number of recently visited states to limit the number of reconstructions and many strategic states from previous levels that will help us shorten reconstructing sequences.

Note that for a large cache holding 10% of the state space, the strategy used impacts less than for small caches. This is evidenced by the fact that the values observed for strategy H and D approach 1 and more generally that all values of line 10^{-2} are smaller than those of line 10^{-1}.

Out of these 143 instances we selected 4 instances that have some specific characteristics (`brp2.6`, `cambridge.6`, `firewire_tree.4` and `synapse.6`) and evaluated strategies F, D and F_{20}-D_{80} with different sizes of cache ranging from 1,000 to 10,000. The collected data is plotted in Fig. 4. On the x-axis is the different cache sizes used. For each run we recorded the number of event executions due to reconstructions and compared it to the same number obtained with strategy F. For instance, with `brp2.6` and a cache of 2,000 states, reconstructions generated approximately three times more event executions with strategy D than with strategy F. We also provide the characteristics of these graphs in terms of number of states and transitions, average degree, i.e., average number

Table 2. Evaluation of caching strategies on 143 DVE instances

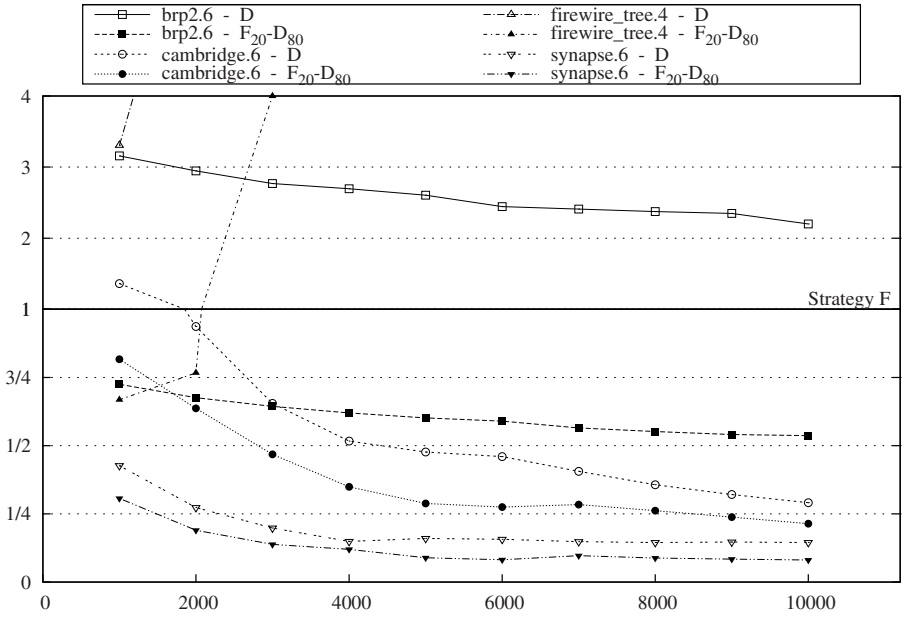
Cache size	Strategy									
	R	F	H	F_{20} H_{80}	F_{50} H_{50}	F_{80} H_{20}	D	F_{20} D_{80}	F_{50} D_{50}	F_{80} D_{20}
10^{-4}	1.000	0.397	1.123	0.365	0.339	0.339	1.057	0.356	**0.332**	0.337
10^{-3}	1.000	0.452	0.940	0.292	0.293	0.317	0.763	**0.272**	0.278	0.308
10^{-2}	1.000	0.587	0.479	0.187	0.208	0.261	0.385	**0.162**	0.188	0.247
10^{-1}	1.000	2.090	0.670	0.258	0.348	0.538	0.771	**0.229**	0.280	0.472
10^{-4}	3	**55**	3	9	18	28	2	18	21	15
10^{-3}	0	12	3	25	26	19	2	**50**	22	13
10^{-2}	0	4	2	42	12	6	4	**85**	9	8
10^{-1}	0	0	4	53	10	6	0	**66**	19	14

of transitions per state, number of levels and number of forward transitions as a proportion of the overwhole number of transitions.

The graph of `firewire_tree.4` 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 D. Moreover, its average degree is rather high, which leads to a huge number of reconstructions with strategy D. On the contrary 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 D 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 D 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 other states have 0 or 3 successors. Therefore, there is no state that is really interesting to keep in the cache. This is evidenced by the fact that the progressions of distance based strategies (relative to strategy F) are not so good. It goes from 3.157 to 2.200 for strategy D and from 0.725 to 0.537 for strategy F_{20}-D_{80}.

5.2 Experiment 2: Evaluation of Delayed Duplicate Detection

To experiment with delayed detection we picked out 63 DVE instances having from 1,000,000 to 60,000,000 states and 12 CPN instances having from 100,000 to 5,000,000 states. The ComBack method was especially helpful for the `dymo` and `erdp` models which are models of two industrial protocols—a routing protocol [5] and an edge router discovery protocol [13]—and have large descriptors (1,000–5,000 bytes).



Instance	States	Transitions	Deg.	Levels	Forward tr.
`brp2.6`	5,742,313	9,058,624	1.57	571	92.6%
`cambridge.6`	3,354,295	9,483,191	2.83	259	66.8%
`firewire_tree.4`	169,992	630,811	3.71	118	100%
`synapse.6`	625,175	1,190,486	1.90	70	74.9%

Fig. 4. Evolution of strategies F, D and F_{20}-D_{80} on some selected instances

Table 3 summarizes our observations. We only report data for 6 instances of each family but still provide the average increases on all instances (see Table 4(a)). We used caching strategies F and F_{20}-D_{80} with a cache size corresponding to 1% of the state space for DVE instances and 0.1% for CPN instances. This choice is motivated by the observation that state vectors of Coloured Petri Nets (100–10,000 bytes) are usually much larger than those of DVE models (10–500 bytes). For each instance we performed 6 tests: one with a standard storage method, i.e., full state descriptors are kept in the visited set, (column Std.), one with the ComBack method without delaying detection (column No DDD) and four with delayed detection enabled with different candidate set sizes expressed as a fraction as the memory given to the cache (columns DDD(0.1), DDD(0.2), DDD(0.5) and DDD(1)). For CPN instances, we kept identifiers in the queue rather than full state descriptors—as described in [20], Variant 4 of Section 5. Most of the instances studied have rather large levels (typically more than 10% of the state space) which prevented us from keeping full state descriptors. The optimization described in Section 4.3 that consists of grouping the reconstruction of queued identifiers was turned on and each block of states reconstructed

Table 3. Evaluation of DDD on some DVE and CPN instances

	Std.	ComBack									
	Strat.	No DDD		DDD(0.1)		DDD(0.2)		DDD(0.5)		DDD(1)	
		T↑	E↑	T↑	E↑	T↑	E↑	T↑	E↑	T↑	E↑
CPN instances	`dymo.2`*	*4,196,714 states, 29,227,638 transitions*									
	15,073 F	8.75	17.83	1.31	1.94	1.36	1.90	1.40	1.87	1.45	1.86
	F_{20}-D_{80}	4.97	9.05	1.88	2.97	1.92	2.87	2.00	2.85	2.02	2.75
	`dymo.6`	*1,256,773 states, 7,377,095 transitions*									
	1,429 F	11.64	15.17	1.91	2.00	1.84	1.91	1.80	1.85	1.79	1.82
	F_{20}-D_{80}	8.98	10.51	2.41	2.57	2.32	2.49	2.24	2.37	2.14	2.24
	`erdp.2`*	*4,277,126 states, 30,503,876 transitions*									
	6,324 F	10.08	26.58	1.85	3.16	1.89	2.97	1.84	2.77	1.83	2.65
	F_{20}-D_{80}	4.84	10.80	1.75	3.16	1.79	3.01	1.90	2.82	1.89	2.71
	`erdp.3`*	*2,344,208 states, 18,739,842 transitions*									
	3,784 F	9.99	22.13	2.07	3.17	1.98	2.97	1.89	2.78	1.83	2.66
	F_{20}-D_{80}	5.44	10.04	2.14	3.23	2.06	3.06	1.98	2.87	1.92	2.75
	`protocol.3`*	*2,130,381 states, 11,584,421 transitions*									
	424 F	34.17	45.85	5.32	6.95	4.71	6.08	4.16	5.25	3.80	4.77
	F_{20}-D_{80}	18.20	25.89	5.12	6.73	4.61	6.02	4.13	5.29	3.82	4.82
	`telephones.2`	*1,004,967 states, 11,474,892 transitions*									
	894 F	3.90	8.06	1.76	2.78	1.64	2.54	1.53	2.30	1.45	2.13
	F_{20}-D_{80}	2.90	5.32	1.85	2.93	1.77	2.76	1.68	2.58	1.60	2.39
DVE instances	`brp.4`	*12,068,447 states, 25,085,950 transitions*									
	16.4 F	9.37	23.02	4.42	3.39	3.76	2.74	2.89	1.87	2.51	1.48
	F_{20}-D_{80}	7.27	10.80	5.14	3.04	4.57	2.55	3.65	1.85	3.22	1.53
	`brp2.6`	*5,742,313 states, 9,058,624 transitions*									
	6.5 F	8.20	16.79	7.86	5.64	6.68	5.16	6.62	5.26	6.41	5.27
	F_{20}-D_{80}	5.59	7.06	8.55	5.31	6.03	3.88	7.16	4.85	7.07	4.91
	`cambridge.7`	*11,465,015 states, 54,850,496 transitions*									
	197 F	23.12	65.43	1.92	3.12	1.90	3.05	1.88	2.97	1.83	2.83
	F_{20}-D_{80}	2.83	5.63	1.79	2.40	1.79	2.38	1.78	2.36	1.75	2.30
	`firewire_link.5`*	*18,553,032 states, 59,782,059 transitions*									
	358 F	3.34	3.54	1.30	1.18	1.26	1.14	1.23	1.11	1.22	1.10
	F_{20}-D_{80}	1.45	1.32	1.32	1.15	1.29	1.12	1.26	1.10	1.26	1.09
	`needham.4`	*6,525,019 states, 22,203,081 transitions*									
	20.2 F	2.29	2.41	1.73	1.29	1.74	1.29	1.75	1.28	1.75	1.27
	F_{20}-D_{80}	2.08	1.72	1.98	1.33	2.00	1.33	2.01	1.33	2.01	1.32
	`synapse.7`	*10,198,141 states, 19,893,297 transitions*									
	24.4 F	9.11	13.53	2.02	1.42	2.02	1.41	2.01	1.39	2.00	1.36
	F_{20}-D_{80}	2.43	1.94	2.21	1.31	2.21	1.31	2.22	1.31	2.23	1.30

Parameters for CPN instances	cache size 0.1% of the state space items in the queue state identifiers
Parameters for DVE instances	cache size 1% of the state space items in the queue full state descriptors

from the queue had the same size as the candidate set. Hence, when DDD was not used this optimization was turned off. In column Std. 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 $\frac{\text{execution time of this run}}{\text{T (with standard search)}}$ 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 for E↑ 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 of out memory. This is indicated by a $\star$ following the instance name. For these, we provide the time obtained with hash compaction as a lower approximation.

We notice that DDD is indeed useful to save the redundant exploration of transitions during reconstruction, and this, even with small candidate sets. Typically, the number of executions is reduced by a factor of 3–5 or even more if we keep identifiers in the queue and group their reconstruction. It seems that, using BFS, states generated successively are "not so distant" in the graph so their reconstructing sequences are quite similar, which allows many sharings. This especially holds during the reconstruction of queued states. Two states reconstructed this way often have the same predecessor in the backedge table (since they have been generated from that state) or at least a common ancestor of low degree. Hence, the average decrease of executions is more sensible for CPNs.

However, although DDD saves many redundant operations in state reconstructions, this does not always impact on the time saved as we would have expected. Indeed DDD is much more interesting when analyzing CPN instances rather than DVE instances. If we consider for example the average increase in the number of events and in the run-time of the 63 DVE instances with caching strategy F_{20}-D_{80} (see Table 4(a)) we could reduce the number of events executed by more than a factor of 2 ($4.00 \rightarrow 1.63$) whereas the average time ($3.45 \rightarrow 2.65$) did not decrease in a comparable way. The reason is that executing an event is much faster for DVE models than for CPN models. Events are typically simple in the DVE language, e.g., a variable incrementation, whereas they can be quite complex with CPNs and include the execution of code embedded in the transition. Therefore, the 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., for instance `brp.4`.

We saw in the previous experiment that model characteristics largely impact on the performance of caching strategies and, hence, that a significant increase in the size of the cache did not necessarily lead to the expected decrease of event executions. This assertion is less valid when duplicate detection is used as we can see from Table 4(b). This one gives, for each configuration (caching strategy and candidate set size), the distribution of values in column E↑ for all 63 DVE instances (rather than the 6 selected). We notice that, by using a very small candidate set (0.1% of the state space), the number of instances in the last range, i.e., with $E\uparrow > 4$, is drastically reduced and, more generally, that allocating more states to the candidate set always brings some improvement.

Table 4. Summary of data for Experiment 2

(a) Average on all instances of time and event executions increase

Strat.	No DDD		DDD(0.1)		DDD(0.2)		DDD(0.5)		DDD(1)	
	T↑	E↑	T↑	E↑	T↑	E↑	T↑	E↑	T↑	E↑
Average on 63 DVE instances										
F	7.15	13.40	2.70	2.12	2.59	1.99	2.43	1.81	2.36	1.66
F_{20}-D_{80}	3.45	4.00	2.92	1.96	2.81	1.86	2.71	1.74	2.65	1.63
Average on 12 CPN instances										
F	16.25	24.03	4.05	4.98	3.45	4.18	2.97	3.52	2.71	3.20
F_{20}-D_{80}	9.59	13.61	3.77	4.81	3.36	4.24	3.01	3.72	2.83	3.44

(b) Distribution of the values in column E↑ for the 63 DVE instances

	Strat. F					Strat. F_{20}-D_{80}				
	Candidate set size					Candidate set size				
Range of E↑	0	0.1	0.2	0.5	1	0	0.1	0.2	0.5	1
1 – 1.25	3	9	9	13	14	3	8	8	11	14
1.25 – 1.50	2	8	8	7	15	6	9	10	12	17
1.50 – 1.75	0	0	1	6	9	6	5	7	9	9
1.75 – 2	1	11	15	18	15	2	15	15	18	13
2 – 4	5	33	28	18	9	24	24	22	12	9
> 4	52	2	2	1	1	22	2	1	1	1

With the largest candidate set size (1% of the state space), duplicate detection generated less event executions than the exploration algorithm, i.e., $\mathsf{E}\uparrow < 2$, for 53 instances out of the 63 selected, regardless the caching strategy. Instances `cambridge.7` and `brp2.6` (see Table 3) belong to the 10 instances that do not have this property. To sum up, increasing the candidate set size is (almost) always useful; and the benefit of DDD is hence more predictable than the benefit of state caching (which depends to a large extent on the characteristics of model).

A somewhat negative observation is that the relative advantage of strategy F_{20}-D_{80} previously observed in Experiment 1 is lost when DDD is used. We indeed observe that numbers reported in Table 4(a) are roughly the same for both strategies: the caching strategy impacts less when delayed detection is used. For DVE instances the algorithm executed slightly more events with strategy F but was also faster. This is due to the time overhead induced by the distance based strategy: each time a state is likely to enter the cache we have to verify in the backedge table that none of its ancestors of degree k (5 in our case) is itself cached. For CPN instances, strategy F generally lead to fewer event executions. Since we stored identifiers instead of full state descriptors in the queue, a fifo cache was more appropriate: by keeping states of the last BFS levels, the probability that an unprocessed state that has to be reconstructed is cached greatly increases. Hence, the grouped reconstruction of queued states occurs less frequently.

These results are best explained in light of the observations made in the first experiment. The goal of strategies F and D are indeed different. The first one

reduces the number of reconstructions while the second one reduces the number of events executed per reconstruction. Thus, delayed duplicate detection and caching strategy D work at the same level and may be redundant whereas DDD is fully complementary to strategy F. Hence, it is not surprising that DDD brings better results when combined with a fifo caching strategy.

5.3 Experiment 3: Optimal Use of Available Memory

In this last experiment we look at a more practical problem. We basically want to know how to tune the different parameters of the ComBack algorithm in order to guarantee a good reduction of the execution time, whatever the input model. This is of interest since the previous experiments stressed that huge variations can be observed for one model when using different combinations of the various extensions we proposed.

First, let us note that by keeping state identifiers in the queue we bound the number of full state descriptors used by the algorithm. Each of these descriptors may be kept in three places: in the queue (if we group the reconstruction of states waiting to be processed), in the cache or in the candidate set. Now, let us suppose that the user has an idea of the available memory and the full state descriptor size[4]. From this user supplied information, we can roughly decide on a maximal number of full state descriptors that may reside in memory. Since there is an obvious conflict between the data structures used by the algorithm, i.e., a full state descriptor may only be kept in one of these, we have to decide which portion should be allocated to the queue, to the cache and to the candidate set. Hence, the problem considered is the following. Given a maximal number of full state descriptors F given to the algorithm, find the configuration that minimizes the extra time introduced by the ComBack method. By configuration we mean here a tuple (S, F_C, F_{CS}, F_Q) where S is a caching strategy and F_C, F_{CS} and F_Q are respectively the proportion of the F states allocated to the cache, to the candidate set and to the queue.

To try to answer this question we performed numerous runs using the different instances and parameters listed by Table 5(a) (naturally, under the constraint that $F_C + F_{CS} + F_Q = 1$). We voluntarily selected small values for parameter F and large state spaces (with respect to F) to simulate verification runs where the state space is large, e.g., 10^9 states, and the state descriptor is so large that only a small fraction of them can be kept in main memory. We did not experiment with values greater than 0.4 for parameter F_Q as beyond this bound the performance of the algorithm quickly degraded. Table 5(b) summarizes our results. The performance reported in column $\mathsf{E}\uparrow$ is the average over all instances of event executions increase (see Table 1). Note that the table has been sorted according to the values reported in that column. For the sake of clarity we only report the performance of a few configurations including the ones that performed the best and the worst. We can make the following observations regarding this data.

[4] Otherwise, a partial search exploring a sample of the state space can provide a rather good approximation of this size.

– First, for each of the three tables, the top part is occupied by configurations using the three techniques (i.e., for which $F_C>0$, $F_{CS}>0$ and $F_Q>0$) while the bottom part is almost only composed of configurations for which one of these parameters is set to 0. Hence, none of these techniques should be disabled.
– All other things being equal, strategy F_{20}-D_{80} is outperformed by strategies F and F_{80}-D_{20}. It is therefore preferable to allocate more states to a fifo cache although a small distance based cache can bring some improvements compared to a pure fifo cache.
– It seems that the best way to proceed is, whatever the value of F, to attribute the majority of the memory to the cache (50–60%), a small amount to group the reconstruction of queued states (10–20%) and the remainder to the candidate set for delayed duplicate detection (20–40%). Although this

Table 5. Summary of data for Experiment 3

(a) Parameters used for experimentation

Parameters	$F \in \{ 100, 1{,}000, 10{,}000 \}$ $F_C \in \{ 0, 0.1, 0.2, \ldots, 1 \}$ $F_{CS} \in \{ 0, 0.1, 0.2, \ldots, 1 \}$ $F_Q \in \{ 0, 0.1, 0.2, 0.3, 0.4 \}$ $S \in \{ F, F_{20}$-D_{80}, F_{80}-$D_{20} \}$
Selected instances	all DVE instances with $[100 \cdot F - 1,000 \cdot F]$ states

(b) Results

F=10^2 (58 instances)

Configuration				E↑
S	F_C	F_{CS}	F_Q	
F_{80}-D_{20}	0.5	0.3	0.2	6.51
F	0.4	0.3	0.3	6.64
F_{20}-D_{80}	0.9	0	0.1	8.91
F_{20}-D_{80}	0.5	0.5	0	13.7
F_{80}-D_{20}	0.8	0.2	0	16.5
F	1	0	0	25.9
-	0	1	0	30.9

F=10^3 (46 instances)

Configuration				E↑
S	F_C	F_{CS}	F_Q	
F_{80}-D_{20}	0.6	0.2	0.2	5.45
F	0.4	0.3	0.3	5.54
F_{20}-D_{80}	0.9	0	0.1	9.51
F_{20}-D_{80}	0.8	0.2	0	13.3
F_{80}-D_{20}	0.8	0.2	0	18.4
F	1	0	0	34.6
-	0	1	0	74.4

F=10^4 (41 instances)

Configuration				E↑
S	F_C	F_{CS}	F_Q	
F_{80}-D_{20}	0.6	0.3	0.1	3.59
F	0.4	0.3	0.3	3.72
F_{80}-D_{20}	0.8	0.2	0	6.58
F_{20}-D_{80}	0.9	0	0.1	7.42
F_{80}-D_{20}	0.1	0.9	0	13.7
-	0	1	0	20.7
F	1	0	0	29.5

information is not provided by the table, these are also the configurations for which the standard deviation is the lowest, meaning that the performance of the algorithm is more predictable and depend less on the specific characteristics of the state space.

6 Conclusion

The ComBack method has been designed to explicitly store large state spaces of models with complex state descriptors. The important memory reduction factor it may provide is however counterbalanced by an increase in run-time due to the on-the-fly reconstruction of states. We proposed in this work two ways to tackle this problem. First, 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 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 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 over a hundred of DVE instances, decrease the number of transitions visited by a factor of five. We also observed that delayed 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 occasions.

In this work, we mainly focused on caching strategies for a 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, which is inherently based on a depth-first search. The design of strategies for other types of searches 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.

Acknowledgments. We thank the anonymous reviewers for their detailed comments that helped us to improve this article.

References

1. Barnat, J., Brim, L., Cerná, I., Moravec, P., Rockai, P., Simecek, P.: DiVinE - A Tool for Distributed Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)

2. Burch, J.R., Clarke, E.M., Dill, D.L., Hwang, L.J., McMillan, K.: Symbolic Model Checking: 10^{20} States and Beyond. In: LICS 1990, pp. 428–439 (1990)
3. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
4. Emerson, E.A., Sistla, A.P.: Symmetry and Model Checking. Formal Methods in Systems Design 9(1-2), 105–131 (1996)
5. Espensen, K.L., Kjeldsen, M.K., Kristensen, L.M.: Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 152–170. Springer, Heidelberg (2008)
6. Evangelista, S., Pradat-Peyre, J.-F.: Memory Efficient State Space Storage in Explicit Software Model Checking. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 43–57. Springer, Heidelberg (2005)
7. Evangelista, S., Westergaard, M., Kristensen, L.M.: The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. Technical report, DAIMI, Aarhus University, Denmark (2008), `http://www.cs.au.dk/~evangeli/doc/comback-extensions.pdf`
8. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
9. Godefroid, P., Holzmann, G.J.: On the Verification of Temporal Properties. In: Danthine, A.A.S., Leduc, G., Wolper, P. (eds.) PSTV 1993. IFIP Transactions, vol. C-16, pp. 109–124. North-Holland, Amsterdam (1993)
10. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
11. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
12. Knottenbelt, W., Mestern, M., Harrison, P., Kritzinger, P.: Probability, Parallelism and the State Space Exploration Problem. In: Puigjaner, R., Savino, N.N., Serra, B. (eds.) TOOLS 1998. LNCS, vol. 1469, pp. 165–179. Springer, Heidelberg (1998)
13. Kristensen, L.M., Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 248–269. Springer, Heidelberg (2004)
14. Pelánek, R.: Typical Structural Properties of State Spaces. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 5–22. Springer, Heidelberg (2004)
15. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007), `http://anna.fi.muni.cz/models/`
16. Stern, U., Dill, D.L.: Improved Probabilistic Verification by Hash Compaction. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)
17. Stern, U., Dill, D.L.: Using Magnetic Disk Instead of Main Memory in the Murϕ Verifier. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
18. Tronci, E., Della Penna, G., Intrigila, B., Venturini Zilli, M.: Exploiting Transition Locality in Automatic Verification. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 259–274. Springer, Heidelberg (2001)

19. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009)
20. Westergaard, M., Kristensen, L.M., Brodal, G.S., Arge, L.: The ComBack Method – Extending Hash Compaction with Backtracking. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 445–464. Springer, Heidelberg (2007)
21. Wolper, P., Leroy, D.: Reliable Hashing without Collision Detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)