The ComBack Method – Extending Hash Compaction with Backtracking

Michael Westergaard, Lars Michael Kristensen*, Gerth Stølting Brodal, and Lars Arge

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,kris,gerth,large}@daimi.au.dk

Abstract. This paper presents the ComBack method for explicit state space exploration. The ComBack method extends the well-known hash compaction method such that full coverage of the state space is guaranteed. Each encountered state is mapped into a compressed state descriptor (hash value) as in hash compaction. The method additionally stores for each state an integer representing the identity of the state and a backedge to a predecessor state. This allows hash collisions to be resolved on-the-fly during state space exploration using backtracking to reconstruct the full state descriptors when required for comparison with newly encountered states. A prototype implementation of the ComBack method is used to evaluate the method on several example systems and compare its performance to related methods. The results show a reduction in memory usage at an acceptable cost in exploration time.

1 Introduction

Explicit state space exploration is one of the main approaches to verification of finite-state concurrent systems. The underlying idea is to enumerate all reachable states of the system under consideration, and it has been implemented in computer tools such as SPIN [9], Murϕ [10], CPN Tools [18], and LoLa [20].

The main drawback of verification methods based on state space exploration is the *state explosion problem* [25], and several reduction methods have been developed to alleviate this inherent complexity problem. For explicit state space exploration these include: methods that explore only a subset of the state space directed by the verification question [17, 24]; methods that delete states from memory during state space exploration [1, 3, 6]; methods that store states in a compact manner in memory [5, 8, 12]; and methods that use external storage to store the set of visited states [22]. Another approach is symbolic model checking using, e.g., binary decision diagrams [2] or multi-valued decision diagrams [13].

Of particular interest in the context of this paper is the *hash compaction method* [21, 27], a method to reduce the amount of memory used to store states.

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

Hash compaction uses a hash function H to map each encountered state s into a fixed-sized 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 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 [21], but the method still cannot guarantee full coverage of the state space. If the intent of state space exploration is to find (some) errors, this is acceptable. If, however, the goal is to prove the absence of errors, discarding parts of the state space is not acceptable, meaning that hash compaction is mainly suited for error detection.

The idea of the ComBack method is to augment the hash compaction method such that hash collisions can be resolved during state space exploration. This is achieved by assigning a unique *state number* to each visited state and by storing, for each compressed state descriptor, a list of state numbers that have been mapped to this compressed state descriptor. This information is stored in a *state table*. Furthermore, a *backedge table* stores a *backedge* for each visited state. A backedge for a state s consists of a transition t and a state number n, such that executing transition t in the predecessor state s' with state number n leads to s. The backedges stored in the backedge table determine a spanning tree rooted in the initial state for the partial state space currently explored. The backedge table makes it possible, given the state number of a visited state s, to *backtrack* to the initial state and thereby obtain a sequence of transitions (corresponding to a path in the state space) which, when executed from the initial state, leads to s, which makes it possible to reconstruct the full state descriptor of s.

A potential hash collision is detected whenever a newly generated state s is mapped to a compressed state descriptor $H(s)$ already stored in the state table. From the compressed state descriptor and the state table we obtain the list of visited state numbers mapped to this compressed state descriptor. Using the backedge table, the full state descriptor can be reconstructed for each of these states and compared to the newly generated state s. If none of the full state descriptors for the already stored state numbers is equal to the full state descriptor of s, then s has not been visited before, and a hash collision has been detected. The state s is therefore assigned a new state number which is appended to the list of state numbers for the given compressed state descriptor, and a backedge for s is inserted into the backedge table. Otherwise, s was identical to an already visited state and no action is required.

The rest of this paper is organised as follows. Section 2 introduces the basic notation and presents the hash compaction algorithm. Section 3 introduces the ComBack method using a small example, and Sect. 4 formally specifies the ComBack algorithm. Section 5 presents several variants of the basic ComBack algorithm, and Sect. 6 presents a prototype implementation together with experimental results obtained on a number of example systems. Finally, in Sect. 7,

we sum up the conclusions and discuss future work. The reader is assumed to be familiar with the basic ideas of explicit state space exploration.

2 Background

The ComBack method has been developed in the context of Coloured Petri nets (CP-nets or CPNs) [11], but applies to many other modelling languages for concurrent systems such as PT-nets [19], CCS [16], and CSP [7]. We therefore formulate the ComBack method in the context of (finite) labelled transition systems to make the presentation independent of a concrete modelling language.

Definition 1 (Labelled Transition System). *A labelled transition system (LTS) is a tuple $\mathcal{S} = (S, T, \Delta, s_I)$, where S is a finite set of **states**, T is a finite set of **transitions**, $\Delta \subseteq S \times T \times S$ is the **transition relation**, and $s_I \in S$ is the **initial state**.*

In the rest of this paper we assume that we are given a labelled transition system $\mathcal{S} = (S, T, \Delta, s_I)$. Let $s, s' \in S$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. An *occurrence sequence* is an alternating sequence of states s_i and transitions t_i written $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$ and satisfying $s_i \xrightarrow{t_i} s_{i+1}$ for $1 \leq i \leq n-1$. For the presentation of the ComBack method, we initially assume that transitions are *deterministic*, i.e., if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$. This holds for transitions in, e.g., PT-nets and CP-nets. In Sect. 5 we show how to extend the ComBack method to modelling languages with non-deterministic transitions.

We use $\rightarrow^*$ to denote the transitive and reflexive closure of Δ, i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_{n-1} \xrightarrow{t_{n-1}} s_n$, $n \geq 1$, with $s = s_1$ and $s' = s_n$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$, and $\mathsf{reach}(s) = \{\, s' \in S \mid s \rightarrow^* s' \,\}$ denotes the set of states reachable from s. The *state space* of a system is the directed graph (V, E) where $V = \mathsf{reach}(s_I)$ is the set of nodes and $E = \{(s, t, s') \in \Delta \mid s, s' \in V\}$ is the set of edges.

The standard algorithm for explicit state space exploration relies on two data structures: a *state table* storing the states that have been discovered until now, and a *waiting set* containing the states for which successor states have not yet been calculated. The state table can be implemented as a hash table, and the waiting set can be implemented, e.g., as a stack or a fifo-queue if depth-first or breadth-first exploration is desired. The state table and the waiting set are initialised to contain the initial state and the algorithm terminates when the waiting set is empty, at which point the state table contains the reachable states.

The basic idea of the *hash compaction method* [21,27] is to use a *hash function* H mapping from states S into the set of bit-strings of some fixed length. Instead of storing the *full state descriptor* in the state table for each visited state s, only the *compressed state descriptor* (hash value) $H(s)$ is stored. The waiting set still stores full state descriptors. Algorithm 1 gives the basic hash compaction

Algorithm 1 Basic Hash Compaction Algorithm

```
1: StateTable.Init(); StateTable.Insert(H(s_I))
2: WaitingSet.Init(); WaitingSet.Insert(s_I)
3:
4: while ¬ WaitingSet.Empty() do
5:    s ← WaitingSet.Select()
6:    for all t, s' such that (s, t, s') ∈ Δ do
7:       if ¬ StateTable.Contains(H(s')) then
8:          StateTable.Insert(H(s'))
9:          WaitingSet.Insert(s')
```

algorithm [27]. The state table and the waiting set are initialised in lines 1–2 with the compressed and full state descriptors for the initial state s_I, respectively. The algorithm then executes a while-loop (lines 4-9) until the waiting set is empty. In each iteration of the while loop, a state s is selected and removed from the waiting set (line 5) and each of the successor states s' of s are calculated and examined (lines 6-9). If the compressed state descriptor $H(s')$ for s' is not in the state table, then s' has not been visited before, and $H(s')$ is added to the state table and s' is added to the waiting set. If the compressed state descriptor $H(s')$ for s' is already in the state table, the assumption of the hash compaction method is that s' has already been visited. The advantage of the hash compaction method is that the number of bytes stored per state is heavily reduced compared to storing the full state descriptor, which can be several hundreds of bytes for complex systems. The disadvantage is that the method cannot guarantee full coverage of the state space.

Figure 1 shows an example state space which will also be used when introducing the ComBack method in the next section. Figure 1(left) shows the full state space consisting of the states $s_1, s_2, \ldots, s_6$. The initial state is s_1. The compressed state descriptors h_1, h_2, h_3, h_4 have been written to the upper right of each state. As an example, it can be seen that the states s_3, s_5, and s_6 are mapped to the same compressed state descriptor h_3. Figure 1(right) shows the part of the state space explored by the hash compaction method. The hash compaction method will consider the states s_3, s_5, and s_6 to be the same state since they are mapped to the same compressed state descriptor h_3. As a result, the hash compaction method does not explore the full state space.

Several improvements have been developed for the basic hash compaction method to reduce the probability of not exploring the full state space [21]. None of these improvements guarantee full coverage of the state space. For the purpose of this paper it therefore suffices to consider the basic hash compaction algorithm.

3 The ComBack Method

The basic idea of the ComBack method is similar to that of the hash compaction method: instead of storing the full state descriptors, a hash function is used to

calculate a compressed state descriptor. When using hash compaction, the main problem is *hash collisions*, i.e., that states with different full state descriptors (such as s_3, s_5, and s_6 in Fig. 1) are mapped to the same compressed state descriptor. The ComBack method addresses this problem by comparing the full state descriptors whenever a new state is generated for which the compressed state descriptor is already stored in the state table. This is, however, done without storing the full state descriptors for the states in the state table. Instead the full state descriptors of states in the state table are reconstructed on-demand using *backtracking* to resolve hash collisions. The reconstruction of full state descriptors using backtracking is achieved by augmenting the hash compaction algorithm in the following ways:

1. A *state number* $N(s)$ (integer) is assigned to each visited state s.
2. The 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 $N(s)$ of a visited state s stores a *backedge* consisting of a transition t and a state number $N(s')$ of a visited state s' such that $s' \xrightarrow{t} s$.

The augmented state table makes it possible, given a compressed state descriptor $H(s)$ for a newly generated state s, to obtain the state numbers for the visited states mapped to the compressed state descriptor $H(s)$. For each such state number $N(s')$ of a state s', the backedge table can be used to obtain the sequence of transitions, $t_1 t_2 \cdots t_n$, on some path (occurrence sequence) in the state space leading from the initial state s_I to s'. As we have initially assumed that transitions are deterministic, executing this occurrence sequence starting in the initial state will reconstruct the full state descriptor for s'. It is therefore possible to compare the full state descriptor of the newly generated state s to the full state descriptor of s' and thereby determine whether s has already been encountered.

Figure 2 (left) shows a snapshot of state space exploration using the ComBack method on the example that was introduced in Fig. 1. The snapshot represents the situation where the successors of the initial state s_1 have been generated, and the states s_2 and s_6 are the states currently in the waiting set. The state number assigned to each state is written inside a box to the upper left of each

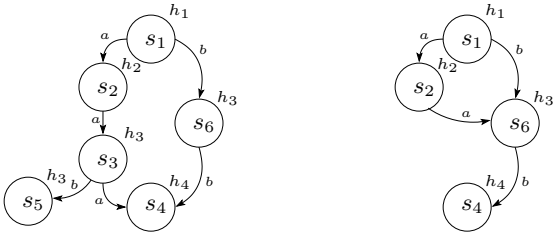

Fig. 1. Full state space (left) and state space explored using hash compaction (right).

state. Figure 2 (middle) shows the contents of the state table, which for each compressed state descriptor h_i lists the state numbers mapped to h_i. Figure 2 (right) shows the contents of the backedge table. The backedge table gives for each state number $N(s)$ a pair $(N(s'), t)$, consisting of the state number $N(s')$ of a predecessor state s' and a transition t such that $s' \xrightarrow{t} s$. As an example, for state number 3 (which is state s_6) the backedge table specifies the pair $(1, b)$ corresponding to the edge in the state space going from state s_1 to state s_6 labelled with the transition b. For the initial state, which by convention always has state number 1, no backedge is specified since backtracking will always be stopped at the initial state.

Assume that s_2 is the next state removed from the waiting set. It has a single successor state s_3 which is mapped to the compressed state descriptor h_3 (see Fig. 1). A lookup in the state table shows that for the compressed state descriptor h_3 we already have a state with state number 3 stored. We therefore need to reconstruct the full state descriptor for state number 3 in order to determine whether s_3 is a newly discovered state. The reconstruction is done in two phases. The first phase uses the backedge table to obtain a sequence of transitions which, when executed from the initial state, leads to the state with number 3. A lookup in the backedge table for the state with state number 3 yields the pair $(1, b)$. Since 1 represents the initial state, the backtracking terminates with the transition sequence consisting of b. In the second phase, we use the transition relation Δ for the system to execute the transition b in the initial state and obtain the full state descriptor for state number 3 (which is s_6). We can now compare the full state descriptors s_3 and s_6. Since these are different, s_3 is a new state and assigned state number 4, which is added to the state table by appending it to the collision list for the compressed state descriptor h_3. In addition s_3 is added to the waiting set, and an entry $(2, a)$ is added to the backedge table for state number 4 in case we will have to reconstruct s_3 later. Figure 3 shows the state space explored, the state table, and the backedge table after processing s_2.

The waiting set now contains s_3 and s_6. Assume that s_3 is selected from the waiting set. The two successor states s_4 and s_5 will be generated. First, we will check whether s_4 has already been generated. As s_4 has the compressed state descriptor h_4, which has no state numbers in its collision list, it is new, and it is assigned state number 5, and an entry $(4, a)$ is added to the backedge table. Then we check if s_5 is new. State s_5 has the compressed state descriptor h_3 and a lookup in the state table yields the collision list consisting of states number

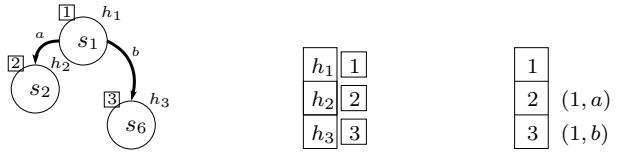

Fig. 2. Before s_2 is processed: state space explored (left), state table (middle), and backedge table (right).

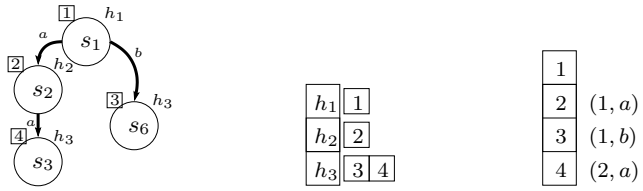

Fig. 3. After processing s_2: state space explored (left), state table (middle), and backedge table (right).

3 and 4. Using the backedge table, we obtain the two corresponding transition sequences: $(1, b)$ and $(2, a)(1, a)$. Executing the occurrence sequences: $s_1 \xrightarrow{b} s_6$ and $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3$ yields the full state descriptors for s_3 and s_6. By comparison with the full state descriptor for s_5 it is concluded that s_5 is new and the state table, the waiting set, and the backedge table are updated accordingly.

When state s_3 has been processed, the waiting set contains the states s_4, s_5, and s_6. The processing of s_4 and s_5 does not result in any new states as these two states do not have successor states. Consider the processing of s_6. We will tentatively denote the full state descriptor for the successor of s_6 corresponding to s_4 by s' as the algorithm has not yet determined that it is equal to s_4. State s' has the compressed state descriptor h_4 and a lookup in the state table shows that we have a single state with number 5 stored for h_4. The backedge table is then used starting from state number 5 to obtain the backedges $(4, a)$, $(2, a)$, and $(1, a)$. Executing the corresponding occurrence sequence $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} s_4$ yields full state descriptor for s_4, and we conclude that this full state descriptor is equal to s', so s' has already been visited and no changes are required to the state table, the waiting set or the backedge table.

Figure 4 shows the situation after state s_6 has been processed. The thick edges correspond to the backedges stored in the backedge table. It can be seen that the backedges stored in the backedge table determine a spanning tree rooted in the initial state in all stages of the construction (Figs. 2–4).

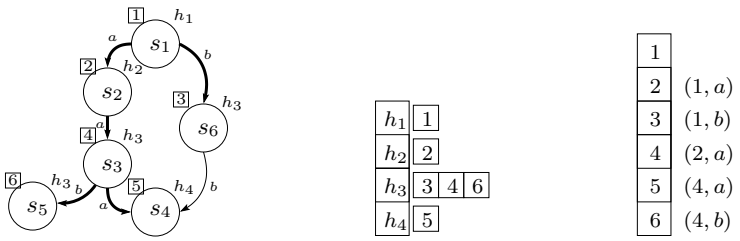

Fig. 4. After processing s_6: state space explored (left), state table (middle), and backedge table (right).

4 The ComBack Algorithm

The ComBack algorithm introduced in the previous section is listed in Algorithm 2. The first part of the algorithm (lines 1–4) initialises the global data structures. The global variable m is used to enumerate the states, i.e., assign state numbers to states, and is initially 1 since the initial state is the first state considered. The state table has an INSERT operation which takes a compressed state descriptor and a state number and appends the state number to the collision list for the compressed state descriptor. The waiting set stores pairs consisting of a full state descriptor and its number. The state number is needed when creating the backedge for a newly discovered state. The backedge table stores pairs consisting of a state number and a transition label. The empty backedge denoted $\perp$ is initially inserted in the backedge table for state number 1 (the initial state).

Algorithm 2 The ComBack Algorithm

```
 1: m ← 1
 2: StateTable.Init(); StateTable.Insert(H(s_I), 1))
 3: WaitingSet.Init(); WaitingSet.Insert(s_I, 1)
 4: BackEdgeTable.Init(); BackEdgeTable.Insert(1, ⊥)
 5:
 6: while ¬ WaitingSet.Empty() do
 7:    (s, n′) ← WaitingSet.Select()
 8:    for all t, s′ such that (s, t, s′) ∈ Δ do
 9:       if ¬ Contains(s′) then
10:          m ← m + 1
11:          StateTable.Insert(H(s′), m)
12:          WaitingSet.Insert(s′, m)
13:          BackEdgeTable.Insert(m, (n′, t))
14:
15: proc Contains(s′) is
16:    for all n ∈ StateTable.Lookup(H(s′)) do
17:       if Matches(n, s′) then
18:          return tt
19:    return ff
20:
21: proc Matches(n, s′) is
22:    return s′ = Reconstruct(n)
23:
24: proc Reconstruct(n) is
25:    if n = 1 then
26:       return s_I
27:    else
28:       (n′, t) ← BackEdgeTable.Lookup(n)
29:       s ← Reconstruct(n′)
30:       return Execute(s, t)
```

The algorithm then executes a while-loop (lines 6–13) until the waiting set is empty. In each iteration of the while-loop, a pair, (s, n'), consisting of a state and its state number is selected from the waiting set (line 7) and each of the successor states, s', of s is examined (lines 8–13). Whether a successor state, s', is a newly discovered state is determined using the CONTAINS procedure, which will be explained below. If s' is a newly discovered state, m is incremented by one to obtain the state number assigned to s', the state number for s' is appended to the collision list associated with the compressed state descriptor $H(s')$, and (n', t) is inserted as a backedge in the backedge table for the state s' which has been given state number m.

The procedure CONTAINS (lines 15–19) is used to determine whether a newly generated state s' has been visited before. The procedure looks up the collision list for the compressed state descriptor $H(s')$ for s', and for each state number, n, in the collision list it checks if s' corresponds to n using the MATCHES procedure. If a reconstructed state descriptor is identical to s', then s' has already been visited and tt (true) is returned. Otherwise ff (false) is returned. The procedure MATCHES (lines 21–22) reconstructs the full state descriptor corresponding to n using RECONSTRUCT procedure and returns whether it is equal to s'.

The procedure RECONSTRUCT recursively backtracks using the backedge table to reconstruct the full state descriptor for state number n. The function recursively finds the state number of a predecessor using the backedge table and calculates the full state descriptor using the EXECUTE procedure. The procedure exploits the convention that the initial state has number 1 to determine when to stop the recursion. The EXECUTE procedure (not shown) uses the transition relation Δ to compute the state resulting from an occurrence of the transition t in the state s, i.e., if $(s, t, s') \in \Delta$ then $\text{EXECUTE}(s, t) = s'$. This is well-defined since we have assumed that transitions are deterministic.

It can be seen that the ComBack algorithm is very similar to the standard algorithm for state space exploration. The main difference is that determining whether a state has already been visited relies on the CONTAINS procedure which uses the backedge table to reconstruct the full state descriptors before the comparison with a newly generated state is done. Since the backedge table at any time during state exploration determines a spanning tree rooted in the initial state for the currently explored part of the state space, we can reconstruct the full state descriptor for any visited state. It follows that the ComBack algorithm terminates after having explored all reachable states exactly once.

Space Usage. The ComBack algorithm explores the full state space at the expense of using more memory per state than hash compaction and by using time on reconstruction of full state descriptors. We will now discuss these two issues in more detail. First we consider memory usage. Let w_N denote the number of bits used to represent a state number, and let w_H denote the number of bits in a compressed state descriptor. Let $|h_i|$ denote the number of reachable states mapped to the compressed state descriptor h_i. The entry corresponding to h_i in the state table can be stored as a pair consisting of the compressed state descriptor and a counter of size w_c specifying the length of an array of state numbers (the collision list). The total amount of memory used to store the

states whose compressed state descriptor is h_i is therefore given by $w_H + w_c + |h_i| \cdot w_N$. Considering all compressed state descriptors, the worst-case memory usage occurs if all collision lists have length 1. This means that the worst-case memory usage for the state table is:

$$|\mathsf{reach}(s_I)| \cdot (w_H + w_c + w_N)$$

We need at least $w_N = \lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits for storing unique numbers for each state and $w_c = \lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits for storing the number of states in each collision list. The worst-case memory usage for the elements in the state table is therefore:

$$|\mathsf{reach}(s_I)| \cdot (w_H + 2 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil)$$

Consider now the backedge table. The entries can be implemented as an array where entry i specifies the backedge associated with state number i. If we enumerate all transitions, each transition in a backedge can be represented using $\lceil \log_2 |T| \rceil$ bits. Each state number in a backedge can be represented using $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits. Observing that each reachable state will have one entry in the backedge table upon termination this implies that the memory used for the elements in the backedge table is given by:

$$|\mathsf{reach}(s_I)| \cdot (\lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil)$$

The above means that the total amount memory used for the elements in the state table and the backedge table is in worst-case given by:

$$|\mathsf{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil)$$

This is $3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil$ bits more per visited state than the hash compaction method. The ComBack method and the hash compaction method both store the full state descriptor for those states that are in the waiting set, but the ComBack method additionally stores the state number of each state in the waiting set which implies that the ComBack method uses $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ more bits per state in the waiting set. In reality, we will not know $|\mathsf{reach}(s_I)|$ in advance, and we will therefore use a machine word (w bits) for storing state numbers. If we furthermore assume that we store each transition using a machine word and use a hash function generating compressed state descriptors of size $w_H = w$, we use a total of $5 \cdot w$ bits or 5 machine words per state, corresponding to 20 bytes on a 32-bit architecture.

Time Analysis. Let us now consider the additional time used by the ComBack algorithm for reconstruction of full state descriptors. Let $\hat{h}_i = \{s_1, s_2, \ldots, s_n\}$ denote the states that are mapped to given compressed state descriptor h_i and assume that they are discovered in this order. The first state s_1 mapped to h_i will not result in a state reconstruction, but when state s_j is discovered the first time it will cause a reconstruction of the states $s_1, s_2, \ldots s_{j-1}$. This means that the number of reconstructions caused by the first discovery of each of the states is given by:

$$\textstyle\sum_{j=1}^{|\hat{h}_i|} (j-1) = \frac{|\hat{h}_i| \cdot (|\hat{h}_i| - 1)}{2}$$

Any additional input edge of an already discovered state mapped to h_i will in worst-case cause all other discovered states to be regenerated. In the worst case, the additional input edges are discovered after all $|\hat{h}_i|$ states have been discovered for the first time. Let $\mathsf{in(s)}$ denote the number of input edges for a state s. The number of reconstructions caused by additional input edges is then given by:

$$|\hat{h_i}| \cdot \textstyle\sum_{s_j \in \hat{h}_i} (\mathsf{in}(s_j) - 1)$$

This means that the total number of state reconstructions for a given compressed state descriptor h_i is given by:

$$\begin{aligned}
\tfrac{|\hat{h}_i| \cdot (|\hat{h}_i|-1)}{2} + |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} (\mathsf{in}(s_j) - 1) &= \tfrac{1}{2}|\hat{h}_i|^2 - \tfrac{|\hat{h}_i|}{2} + |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}(s_j) - |\hat{h}_i|^2 \\
&= -\tfrac{1}{2}|\hat{h}_i|^2 - \tfrac{|\hat{h}_i|}{2} + |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \\
&\leq |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j)
\end{aligned}$$

Let $\hat{H} = \{\, H(s) \mid s \in \mathsf{reach}(s_I) \,\}$ denote the set of compressed state descriptors for the set of reachable states. The number of reconstructions used for the entire state space exploration can be then be approximated by:

$$\begin{aligned}
\textstyle\sum_{h_i \in \hat{H}} |\hat{h}_i| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) &\leq \textstyle\sum_{h_i \in \hat{H}} \left(\max_{h_k \in \hat{H}} |h_k| \cdot \textstyle\sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \right) \\
&= \max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \textstyle\sum_{h_i \in \hat{H}} \sum_{s_j \in \hat{h}_i} \mathsf{in}\,(s_j) \\
&= \max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \textstyle\sum_{s \in \mathsf{reach}(s_I)} \mathsf{in}\,(s_j)
\end{aligned}$$

If we assume that we are using a good hash function for computing the compressed state descriptors, then $|\hat{h}_i|$ will in practice be small (at most 2 or 3). This means that the total number of state reconstructions will be close to the sum of the in-degrees of all reachable states which is equal to number of edges in the full state space. A poor hash function will cause many state reconstructions which in turn will seriously affect the run-time performance of the algorithm. In Sect. 6 we will show how to obtain a good hash function in the context of CP-nets. If the backedge table is implemented as an array, we get a constant look-up time, and a state can be reconstructed in time proportional to the length of the path.

The above is summarised in the following theorem where $\{0,1\}^{w_H}$ denotes the set of bit strings of length w_H.

Theorem 1. *Let $\mathcal{S} = (S, T, \Delta, s_I)$ be a labelled transition system and $H : S \to \{0,1\}^{w_H}$ be a hash function. The ComBack algorithm in Algorithm 2 terminates after having explored all reachable states of $\mathcal{S}$ exactly once. The elements in the state table and the backedge table can be represented using:*

$$|\mathsf{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\mathsf{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil) \ \textit{bits}$$

The total number of state reconstructions during exploration is bounded by:

$$\max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \textstyle\sum_{s \in \mathsf{reach}(s_I)} \mathsf{in}\,(s_j)$$

5 Variants and Extensions

In this section, we sketch several variants of the basic ComBack algorithm. Variants 1 and 2 are aimed at reducing time usage while Variants 3 and 4 are aimed at reducing memory usage. Variant 5 shows how the ComBack method can be used for modelling languages with non-deterministic transitions.

Variant 1: Path Optimisation. The amount of time used on reconstruction of a state s is proportional to the length of the occurrence sequence leading to s stored in the backedge table. If the state space is constructed in a breadth-first order, the backedge table automatically contains the shortest occurrence sequences for reconstruction of states. This is not the case, e.g., when using depth-first exploration. When the state space is not explored breadth-first, it is therefore preferable to keep the occurrence sequences in the backedge table short. As an example consider Fig. 4. The occurrence sequences stored in the backedge table for s_4 (state number 5) is $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{a} s_4$, which is of length 3. A shorter path $s_1 \xrightarrow{b} s_6 \xrightarrow{b} s_4$ has however been found when s_4 was re-discovered from s_6. When re-discovering s_4 from s_6, it is therefore beneficial to replace the backedge $(4, a)$ stored for s_4 to $(3, b)$ such that the shorter occurrence sequence $s_1 \xrightarrow{b} s_6 \xrightarrow{b} s_4$ is stored in the backedge table. It is easy to modify the algorithm to make such simple path optimisations by storing the *depth* of each state in the waiting set along with the full state descriptor and state number. The depth of a state s stored in the waiting set is the length of the occurrence sequence through which s was explored. Whenever a state s is removed from the waiting set in line 7 of Algorithm 2, we obtain the depth d of s. By incrementing d by one, we obtain the depth of each successor state s' of s. If the RECONSTRUCT procedure (see lines 24–30 in Algorithm 2) reconstructs s' based on the backedge table using an occurrence sequences of length greater than $d + 1$, then the backedge stored for s' should be changed to point to the state number of s since going via s results in a shorter occurrence sequence. It is easy to see that the above path optimisation shortens the occurrence sequences stored in the backedge table, but it does not necessary yield the shortest occurrence sequences.

Variant 2: Caching of Full State Descriptors. Another possibility of reducing the time spent on state reconstruction is to maintain a small cache of some full state descriptors for the visited states. As an example, consider Fig. 4 and assume that we have cached state s_3 (with state number 4) during exploration. Then we would not need to do backtracking for state number 4 when we generate state s_5 – we can immediately see that even though states s_3 and s_5 both have the compressed state descriptor h_3, the cached full state descriptor for s_3 is not the same as the full state descriptor for s_5. Caching s_3 also yields an optimisation when we generate state s_4 (with state number 5) when processing s_6. In this case we would not have to backtrack all the way back to the initial state, but as soon as we encounter state number 4 in the backtracking process we can obtain the full state descriptor for s_3 (since it is cached), and it suffices to execute the occurrence sequence $s_3 \xrightarrow{a} s_4$ to reconstruct the full state descriptor for s_4. This shows that caching also optimises state reconstruction for non-cached

states. Another way to further optimise backtracking is to re-order the states in the collision lists according to some heuristics that attempt to predict which state is most likely to be re-visited. A simple heuristic is to move a state number to the front of the collision list every time we re-encounter it.

Variant 3: Backwards State Reconstruction. Some modelling languages, including PT-nets and CP-nets, allow transitions to be executed backwards, i.e. we can obtain a function Δ^{-1} such that $\Delta^{-1}(s', t) = s \iff (s, t, s') \in \Delta$. This can be used to execute occurrence sequences from the backedge table backwards, starting from the full state descriptor of a newly generated state s', in order to determine whether s' has already been visited. This has two benefits. Firstly, we do not need to store the occurrence sequence obtained from the backedge table in memory, but can just iteratively look up a backedge in the backedge table and transform the current state using Δ^{-1}. Secondly, the backtracking process may stop early if we encounter an *invalid state*. What qualifies as an invalid state depends on the modelling formalism. A simple implementation for PT-nets and CP-nets is to consider states to be invalid if there is a negative amount of tokens on a place (which may happen when transitions are executed backwards).

Variant 4: Reconstruction of Waiting Set States. In the basic ComBack algorithm we store the full state descriptors for the states in the waiting set. This may take up a considerable amount of memory. It can be observed that we do not actually need to store the full state descriptor for states in the waiting set. It suffices to store the state number as the full state descriptor can be reconstructed from the state number and the backedge table when the state number is selected from the waiting set. This reduces memory usage at the expense of having to make up to $|\mathsf{reach}(s_I)|$ extra reconstructions of states. We can alleviate this, however, if we do depth-first exploration and cache at least the last state that was processed.

Variant 5: Non-deterministic Transitions For modelling languages with non-deterministic transitions we may have $(s, t, s') \in \Delta \wedge (s, t, s'') \in \Delta$ such that $s' \neq s''$. This means that we may not have a single unique state when executing occurrence sequences obtained from the backedge table, and a state reconstruction procedure is required that operates on sets of states. Consider the reconstruction of a visited state with number n. From the backedge table we obtain (as before) a sequence of backedges $(n_m, t_m) \cdots (n_i, t_i) \cdots (n_2, t_2)(n_1, t_1)$ where $n_1 = 1$ (the initial state). In the i'th step of the reconstruction process when considering the backedge (n_i, t_i), now have a set of states S_1 containing the states that can be reached by executing the transition sequence $t_1 t_2 t_{i-1}$ starting in the initial state. From this set we obtain a new set of states S_2 which is the set of states obtained by executing t_i in those states of S_1 where t_i is enabled. To reduce the size of the set S_2 we observe that S_2 should only contain those states that has the same compressed state descriptor as state number n_{i+1}. The compressed state descriptor for state number n_{i+1} can be obtained from the state table. With a good hash function H, this is expected to keep the size of the sets of states considered during state reconstruction small.

Algorithm 3 MATCHES and RECONSTRUCT procedures for Variant 5

```
 1: proc MATCHES(n, s) is
 2:   return s ∈ RECONSTRUCT(n)
 3:
 4: proc RECONSTRUCT(n) is
 5:   if n = 1 then
 6:     return {s_I}
 7:   else
 8:     (n', t) ← BACKEDGETABLE.LOOKUP(n)
 9:     S_1 ← RECONSTRUCT(n')
10:     S_2 ← {s_2 ∈ S | ∃s_1 ∈ S_1 : (s_1, t, s_2) ∈ Δ}
11:     S_3 ← {s_2 ∈ S_2 | n ∈ STATETABLE.LOOKUP(H(s_2))}
12:     return S_3
```

Revised MATCHES and RECONSTRUCT procedures for Variant 5 are shown in Algorithm 3. The RECONSTRUCT procedure is changed to return a set of possible states matching the state number n, so MATCHES is changed to check if s is among those (line 2). The only state corresponding to state number 1 is the initial state (line 6). In line 8 we look up the number of a predecessor state in the backedge table and recursively reconstruct all states that can match that state (line 9). Then we calculate all possible successors of those states (line 10). After that we check that the state number we are looking for, n, is actually in the collision list of the compressed state descriptor of all calculated successors (line 11), and finally return the result. The algorithm will work without the weeding of states in line 11, but at the expense of considering larger state sets.

6 Experimental Results

A prototype of the basic algorithm as described in Sects. 3 and 4 has been implemented in CPN Tools [18] which supports construction and analysis of CPN models [11]. The algorithm is implemented in Standard ML of New Jersey (SML/NJ) [23] version 110.60.

The STATETABLE is implemented as a hash mapping (using lists for handling collisions) and the BACKEDGETABLE is implemented as a dynamic extensible array. This ensures that we can make lookups and insertions in (at least amortized) constant time. The collision list is implemented using SML/NJ's built-in list data type, which is a linked list (rather than an array with a length). A more efficient implementation of the STATETABLE could be obtained using very tight hashing [5]. This would allow us to remove some redundant bits from the compressed state descriptor. We have implemented both depth-first exploration (DFS) and breadth-first exploration (BFS).

The compressed state descriptors calculated by the hash function as well as the state numbers are 31-bit unsigned integers as SML/NJ uses the 32^{nd} bit for garbage collection. The hash function used is defined inductively on the state of the CPN model. In CP-nets, a state of the system 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 also implemented caching of full state descriptors as explained in Sect. 5. The caching strategy used is simple: we use a hash mapping from state numbers to full state descriptors, which does not account for collisions of hash values. That way, if we allocate a hash mapping of, say, size 1000, we can store at most 1000 full state descriptors in the cache. We have not implemented re-ordering of states in the collision lists, as the collision lists have length at most 2 (with two exceptions) for all our examples.

We use a test-suite consisting of three kinds of models: small examples, medium-sized examples and real-life applications. In the first category, we have three models: a model of the dining philosophers system (DP), a model of replicating database managers (DB), and a model of a stop-and-wait network protocol (SW). In the second category, we have a model of a telephone system (TS). In the last category, we have a model of a protocol (ERDP) for distributing network prefixes to gateways in a network consisting of standard wired networks and wireless mobile ad-hoc networks [14]. All of the models are parametrised: DP by the number of philosophers, DB by the number of database managers, SW by the number of packets transmitted and the capacity of the network, TS by the number of telephones, and ERDP by the number of available prefixes and the capacity of the network. We will denote each model by its name and its parameter(s), e.g. DP22 denotes DP with 22 philosophers and ERDP6,2 denotes the ERDP protocol with six prefixes and a network capacity of two.

We have evaluated the performance of the ComBack method without cache, denoted by ComBack, and with cache of size n, denoted ComBack n. We have compared the ComBack method with implementations of basic hash compaction [27], bit-state hashing [8] by means of double hashing [4] which uses a linear combination of two hash functions to compute, in this case, 15 compressed state descriptors. Instead of storing the compressed state descriptors, like hash compaction, bit-state hashing uses the values to set bits in a bit-array. Finally, we compare the ComBack method to standard state space exploration of the full state space using a hash table for storing the full state descriptors. For each model, we have measured how much memory and how much CPU time was used to conduct the state space exploration. Memory is measured by performing a full garbage collection and measuring the size of the heap. This is done every 0.5 second or 40 states, whichever comes last. As garbage collection takes time, the CPU time used is measured independently. We have measured the time three times and used the average as the result.

Table 1 shows the results of the experiments. For each model (column 1) and each exploration method (column 2), we show the number of nodes (states) and arcs explored (columns 3 and 4). We also show the CPU time spent (in seconds) and the amount of space (memory) used (in mega-bytes) for a depth-first traver-

sal (DFS) and a breadth-first traversal (BFS) of the state space (columns 5, 7, 10, and 12). In addition we show how much time and memory is used relative to traversal using a standard exploration using DFS (columns 6, 8, 11, and 13) and how much memory (in bytes) is used per state (columns 9 and 14).

We note that for each model, independent of the reduction technique, either DFS performs better memory-wise than BFS or vice versa. For the more realistic examples, TS and ERDP, DFS is slower than BFS. This is due to the fact that the models resemble real systems and have more complex behaviour, which leads to very long occurrence sequences in the backedge table, and thus impacts the performance of the ComBack method. If we instrument the ComBack method with even a small cache when using DFS, or if we use BFS, processing is much faster for realistic examples. We see that the ComBack method uses quite a bit more memory than the 5 machine words predicted in the previous section. One cause for this is that the calculation in Sect. 4 did not take the WaitingSet into account and only considered the *elements* of the state table and the backedge table, not the tables themselves. Furthermore, SML is not very

Table 1. Experimental results

model	method	nodes	arcs	DFS					BFS				
				time		space			time		space		
				sec	%	Mb	%	/state	sec	%	Mb	%	/state
DP22	ComBack	39604	481625	2791	10337	23.0	97	608	59	219	9.8	42	260
	ComBack 100	39604	481625	800	2963	23.0	98	610	56	207	9.9	42	261
	ComBack 1000	39604	481625	98	363	23.6	100	625	57	211	10.6	45	281
	Hash compaction	39603	481609	25	93	20.8	88	550	26	96	8.4	35	222
	Bit-state	39604	481609	28	104	32.0	135	846	29	107	20.0	85	531
	Standard	39604	481625	27	100	23.6	100	625	27	100	14.3	61	380
DB9	ComBack	59051	314947	60	214	4.5	10	80	63	225	11.9	28	212
	ComBack 100	59051	314947	50	178	4.7	11	83	51	182	12.1	28	214
	ComBack 1000	59051	314947	48	171	5.5	13	98	44	157	12.9	30	229
	Hash compaction	59049	314937	25	89	1.4	3	25	27	96	10.1	23	179
	Bit-state	59051	314947	29	104	12.3	28	218	33	118	21.3	49	379
	Standard	59051	314947	28	100	43.3	100	769	28	100	43.4	100	770
DB10	ComBack	196832	1181001	286	44	15.4	9	82	307	48	43.1	25	230
	ComBack 100	196832	1181001	247	38	15.6	9	83	264	41	43.3	25	231
	ComBack 1000	196832	1181001	240	37	16.6	10	89	250	39	44.4	26	236
	Hash compaction	196798	1180790	118	18	4.9	3	26	133	21	36.8	21	196
	Bit-state	196832	1181001	138	21	12.3	7	66	152	24	46.3	27	247
	Standard	196832	1181001	643	100	174.0	100	927	693	106	174.0	100	927
SW7,4	ComBack	215196	1242386	115	319	17.5	41	85	115	319	20.1	47	98
	ComBack 100	215196	1242386	68	189	17.6	41	86	100	278	20.2	47	98
	ComBack 1000	215196	1242386	64	178	17.9	42	87	93	258	20.6	48	100
	Hash compaction	214569	1238803	33	92	5.2	12	25	37	103	9.8	23	48
	Bit-state	215196	1242386	41	114	12.3	28	60	46	128	18.3	43	89
	Standard	215196	1242386	36	100	43.0	100	210	40	100	43.1	100	210
TS5	ComBack	107648	1017490	3302	6115	51.4	84	500	103	191	17.6	29	172
	ComBack 100	107648	1017490	933	1728	51.4	83	501	102	189	17.7	29	172
	ComBack 1000	107648	1017490	207	383	51.9	85	506	107	198	18.4	30	180
	Hash compaction	107647	1017474	50	93	45.7	75	445	52	96	14.7	24	143
	Bit-state	107648	1017490	58	107	55.4	90	540	62	115	25.8	42	251
	Standard	107648	1017490	54	100	61.2	100	596	57	106	45.0	73	438
ERDP6,2	ComBack	207003	1199703	986	865	29.1	33	147	867	761	35.7	41	181
	ComBack 100	207003	1199703	259	227	29.0	33	147	481	422	35.8	41	181
	ComBack 1000	207003	1199703	205	180	29.6	34	150	402	353	36.4	42	184
	Hash compaction	206921	1199200	106	93	5.1	6	26	114	100	18.6	21	94
	Bit-state	207003	1199703	123	108	12.3	14	62	135	118	27.3	31	138
	Standard	207003	1199703	114	100	87.4	100	443	131	115	88.5	101	449
ERDP6,3	ComBack	4277126	31021101	42711	-	572.3	-	140	65354	-	708.1	-	174
	ComBack 100	4277126	31021101	18043	-	571.2	-	140	-	-	-	-	-
	ComBack 1000	4277126	31021101	23084	-	571.7	-	140	-	-	-	-	-
	Hash compaction	4270926	30975030	3341	-	113.5	-	28	20512	-	403.6	-	99
	Bit-state	4277125	31021091	3732	-	12.1	-	3	17481	-	347.9	-	85
	Standard	-	-	-	-	-	-	-	-	-	-	-	-

memory-efficient, doubling usage. We also note that the standard exploration as well as the ComBack method using BFS were not able to complete due to lack of memory for the ERDP6,3 model. The hash compaction bit-state hashing were also not able to explore all states for this example (as can be seen in the nodes column of Table 1). This means we are comparing methods guaranteeing full coverage with methods that do not, so while the hash compaction and bit-state hashing methods seem to perform well, they do so at a cost.

Figure 5 shows charts depicting memory and time usage relative to standard DFS exploration (i.e. one chart for columns 5 and 7 and another chart for columns 10 and 12). These charts allow us to better understand how the different exploration methods perform compared to each other, independent of the example. We see that the values fall into 7 rectangles corresponding to 6 different exploration methods and an abnormal experiment. Rectangle 1: standard exploration; all results are near 100% on both axes, showing that when we store full state descriptors in a hash table, it does not matter whether we use DFS or BFS. Rectangle 2: hash compaction; all are near 100% on the time axis and between 2% and 100% (DFS) or 20% and 40% (BFS) on the memory axis, showing that hash compaction uses as much time as storing the full state descriptors, but significantly less space. Rectangle 3: bit-state hashing; all are near 100% time-wise, but slightly higher than 1 and 2 (this is probably because we have to calculate two hash values instead of just one). All range between 15% and 150% memory-wise. The bit-state hashing method consistently uses 12.5 mega-bytes plus the size of the waiting set, so it performs well memory-wise on models with large state spaces, but performs poorly on models with small state spaces. This means that memory optimisations are possible, but customisation is required by the user. All the show models are reasonable large, leading to reasonable performance of bit-state hashing. Rectangle 4: ComBack without cache; all are above 150% time-wise and between 10% and 100% (DFS) or 25% and 40% (BFS) memory-wise. Time is also better bounded in the BFS results. This indicates that ComBack without cache yields a reduction (it is never above 100%), and when using BFS we have better control of the time and memory used. DFS makes it possible to save more memory, but can be very costly time-wise, and sometimes we do not save any memory at all (e.g. in the Dining philosophers example, where we can end up with most of the state space in the waiting set). Rectangles 5+6: ComBack with cache; these use slightly more memory but less time than 4, in particular in the DFS case. More cache yields more memory and less time used, but the differences are not that large, and even a small cache yields great optimizations in time compared to the ComBack method with no cache at all. Rectangle 7: DB10; these points fall outside of all the other boxes. Inspection of the data in Table 1 shows that the DB10 example has irregular behaviour, as exploration using the standard exploration is slow as a full state descriptor for this model is large, and thus the SML/NJ garbage collector is invoked often. This yields a performance penalty and causes all other experiments, as they are relative to the standard exploration, to fall outside the other boxes. ERDP6,3 is not shown as the standard exploration was unable to terminate.

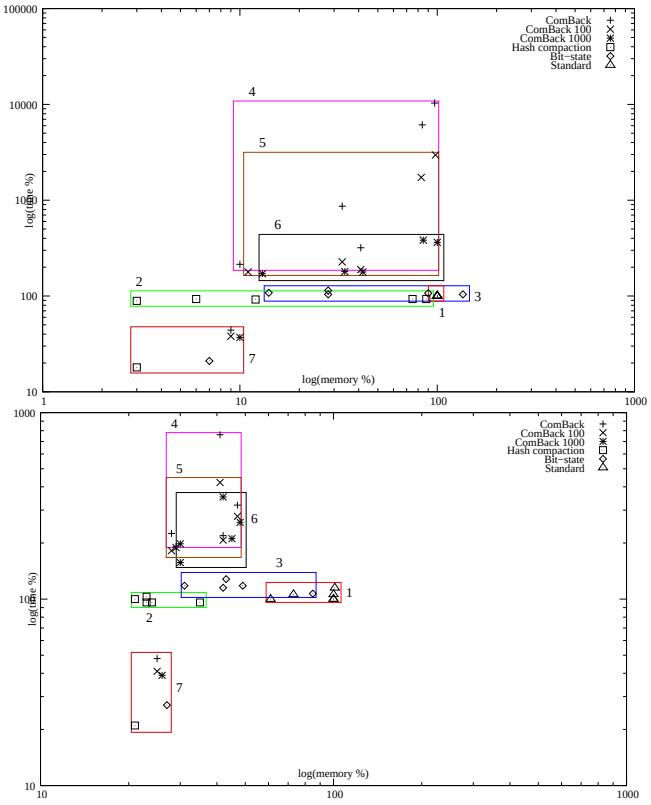

Fig. 5. Time and memory usage for the various reduction techniques using DFS exploration (top) and BFS exploration (bottom). Values are relative to corresponding values for standard depth-first exploration.

All of the shown experiments have been performed using a hash-function generating 31-bit compressed state descriptors. We have also tested the method using a hash function generating 62-bit compressed state descriptors, but have not shown those results, as the time usage is the same but more memory is consumed, as the 31-bit hash function causes few collisions. We have verified the quality of the hash-function by calculating the lengths of the collision lists for all examples. The worst case is example SW7,4, where there are 214009 collision lists of length 1, 592 lists of length 2 and 1 list of length 3, so 99.7 % of the collision lists have the minimum length. It also means that hash compaction misses at least $1 \cdot 592 + 2 \cdot 1 = 594$ states due to hash collisions.

7 Conclusions and Future Work

In this paper we have presented the ComBack method for alleviating the state explosion problem. The basic idea of the method is to augment the hash com-

paction method with a backedge table that makes it possible to reconstruct full state descriptors and ensure full coverage of the state space. We have made a prototype implementation of the method in CPN Tools and our experimental results demonstrate that the method (as expected) uses more time and memory than hash compaction, but less memory than ordinary state space exploration.

The advantage of the ComBack method is that it guarantees full coverage of the state space, unlike related methods such as hash compaction and bit-state hashing. From a practical viewpoint one could therefore use methods such as hash compaction in early phases of a verification process to discover errors, and when no further errors can be detected, the ComBack method could be used for formal verification of properties.

In this paper we have not discussed verification of properties using the ComBack method. It can be observed that the method explores the full state space without mandating a particular exploration order. Furthermore, the state reconstruction that occurs when checking whether a state has already been visited can be made fully transparent to the verification algorithm being applied in conjunction with the state space exploration. This makes the method compatible with most on-the-fly verification algorithms (e.g., verification of safety properties and on-the-fly LTL model checking [26]). The ComBack method is also compatible with off-line verification algorithms such as CTL model checking [15] since the backedge table allows the reconstruction of any of the full state descriptors which in turn allows the forward edges between states to be reconstructed. Alternatively, we can simple store the forward edges in an additional table during state space exploration.

The ComBack method opens up several areas for future work. One topic is the integration of verification algorithms as sketched in the previous paragraph. Future work also includes implementation of the additional variants presented in Sect. 5, and the development and evaluation of caching strategies and organisation of collision lists to reduce the time spent on state reconstruction. It would also be interesting to compare the ComBack method to other complete techniques such as state caching [6]. Another important topic is to explore the combination of the ComBack method with other reduction methods. For this purpose, partial-order methods [17, 24] appear particularly promising as they reduce the in-degree of states which in turn will lead to a reduction in the number of state reconstructions.

References

1. G. Behrmann, K.G. Larsen, and R. Pelnek. To Store or Not to Store. In *Proc. of CAV 2003*, volume 2725 of *LNCS*, pages 433–445. Springer, 2003.
2. R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
3. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.

4. P.C. Dillinger and P. Manolios. Fast and accurate Bitstate Verification for SPIN. In *Proc. of SPIN 2004*, volume 2989 of *LNCS*. Springer-Verlag, 2004.
5. J. Geldenhuys and A. Valmari. A Nearly Memory-Optimal Data Structure for Sets and Mappings. In *Proc. of SPIN 2003*, volume 2648 of *LNCS*, pages 136–150. Springer, 2003.
6. P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.
7. C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
8. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.
9. G.J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
10. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9, 1996.
11. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.
12. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996.
13. T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, University of California at Berkeley, 1995.
14. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Proc. of INT'04*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.
15. O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, 2000.
16. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
17. D. Peled. All for One, One for All: On Model Checking Using Representatives. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.
18. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
19. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
20. K. Schmidt. LoLA - A Low Level Analyser . In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 465–474. Springer-Verlag, 2000.
21. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 206–224. Springer-Verlag, 1995.
22. U. Stern and D.L. Dill. Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 172–183. Springer-Verlag, 1998.
23. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
24. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets '90*, volume 483 of *LNCS*, pages 491–515. Springer-Verlag, 1990.
25. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.
26. M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. of IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.
27. P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.