Search-Order Independent State Caching[⋆]

Sami Evangelista[1,2] and Lars Michael Kristensen[3]

[1] Computer Science Department, Aarhus University, Denmark
[2] LIPN, Université Paris 13, France
sami.evangelista@lipn.univ-paris13.fr
[3] Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

Abstract. State caching is a memory reduction technique used by model checkers to alleviate the state explosion problem. It has traditionally been coupled with a depth-first search to ensure termination. We propose and experimentally evaluate an extension of the state caching method for general state exploring algorithms that are independent of the search order (i.e., search algorithms that partition the state space into closed (visited) states, open (to visit) states and unmet states).

1 Introduction

Model checking is one of the techniques used to detect defects in system designs. Its principle is to perform an exhaustive exploration of all system states to track erroneous behaviors. Although it provides some advantages compared to other verification methods, its practical use is sometimes prohibited by the well-known state explosion problem: the state space of the system may be far too large to be explored with the available computing resources.

The literature is replete with examples of techniques designed to tackle, or at least postpone, the state explosion problem. While some techniques, like partial order reduction [12], reduce the part of the state space that must be explored while still guaranteeing soundness and completeness, more pragmatic approaches make a better use of available resources to extend the range of systems that can be analyzed. State compression [16], external memory algorithms [1], and distributed algorithms [24] are examples of such techniques. In this paper we focus on the *state caching* method first proposed by Holzmann in [14].

State caching is based on the idea that, in depth-first search (DFS), only the states on the current search path need to be in memory to detect cycles. All states that have been visited but have left the DFS stack can thus be deleted from memory without endangering the termination of the search. This comes at the cost of potentially revisiting states and for many state spaces, time becomes the main limiting factor.

The state caching method has been developed and mostly studied in the context of depth-first search since this search order makes it easy to guarantee

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

termination. In this paper we propose an extension of state space caching to General State Exploring Algorithms (GSEA). Following the definition of [5], we put in this family all algorithms that partition the state space into three sets: the set of *open* states that have been seen but not yet expanded (i.e., some of their successors may not have been generated); the set of *closed* states that have been seen and expanded; and the set of unseen states. DFS, BFS, and directed search algorithms [8] like Best-First Search and A* are examples of such general state exploring algorithms.

The principle of our extension is to detect cycles and guarantee termination by maintaining a tree rooted in the initial state and covering all open states. States that are part of that tree may not be removed from the cache, while others are candidates for replacement. Hence, any state that is not an ancestor in the search tree of an unprocessed state can be removed from memory. This tree is implicitly constructed by the state caching algorithm in DFS, since DFS always maintains a path from the initial state to the current state, while for GSEA it has to be explicitly built. However, our experimental results demonstrate that the overhead both in time and memory of this explicit construction is negligible.

The generalized state caching reduction is implemented in our model checker ASAP [26]. We report on the results of experiments made to assess the benefits of the reduction in combination with different search orders: BFS, DFS, and several variations and combinations of these two; and with the sweep-line method [20] which we show is compatible with our generalized state caching reduction. The general conclusions we draw from these experiments are that (1) the memory reduction is usually better with DFS than with BFS although we never really experienced a time explosion with BFS; (2) BFS is to be preferred for some classes of state spaces; (3) a combination of BFS and DFS often outperforms DFS with respect to both time and memory; (4) state caching can further enhance the memory reduction provided by the sweep-line method.

Structure of the Paper. Section 2 presents the principle of a general state exploring algorithm. Section 3 describes our state caching mechanism for the general algorithm. In Sect. 4 we put our generalized state caching method into context by discussing its compatibility with related reduction techniques. Section 5 reports on the results of experiments made with the implementation of the new algorithm. Finally, Sect. 6 concludes this paper.

Definitions and Notations. From now on we assume to be given a universe of system states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, a set of events $\mathcal{E}$, an enabling function $en : \mathcal{S} \to 2^{\mathcal{E}}$ and a successor function $succ : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$; and that we want to explore the state space implied by these parameters, i.e., visit all its states. A state space is a triple (S, T, s_0) such that $S \subseteq \mathcal{S}$ is the set of reachable states and $T \subseteq S \times S$ is the set of transitions defined by:

$$\begin{array}{rcl} S & = & \{s_0\} \cup \{\, s \in \mathcal{S} \mid \exists s_1, \ldots, s_n \in \mathcal{S} \text{ with } s = s_n \ \wedge\ s_1 = s_0 \ \wedge \\ & & \qquad \forall i \in \{1, \ldots, n-1\} : \exists e_i \in en(s_i) \text{ with } succ(s_i, e_i) = s_{i+1}\} \\ T & = & \{(s, s') \in S \times S \mid \exists e \in en(s) \text{ with } succ(s, e) = s'\} \end{array}$$

Related Work. The principle of state caching dates back to an article of Holzmann [14] in 1985. He noted that, in DFS, cycles always eventually reach a state on the stack and, hence, keeping in memory the states on the current search path ensures termination. Forgetting other states comes at the cost of potentially re-exploring them. In the worst case, if any state leaving the stack is removed from memory, a state will be visited once for each path connecting it to the initial state leading to a potential explosion in run-time. Hence, depending on available memory, a set of states that have left the stack are cached in memory.

The question of the strategy to be used for replacing cached states has been addressed in several papers: [11,14,15,17,18,23]. States can be chosen according to various criteria (e.g., in- and out-degree, visit frequency, stack entry time) or in a purely random way. The experiments reported in [23] stress that no strategy works well on all models and that strategies are, to some extent, complementary.

The *sleep-set* reduction technique [12] is fully compatible with state caching [13]. This reduction eliminates most "useless" interleavings by exploiting the so-called *diamond property* of independent transitions: whatever their execution order they lead to the same state. This has the natural consequence to limit revisits of states removed from the cache. For some protocols (e.g., AT&T's Universal Receiver Protocol, MULOG's mutual exclusion protocol), the result of this combination is impressive: at a reasonable cost in time, the cache size can be reduced to less than 3% of the state space.

All the related work discussed above are coupled with depth-first search. To the best of our knowledge, the only work exploring the combination of state caching with BFS is [21] which is more closely related to our work. Termination in [21] is ensured by taking snapshots of the state space, i.e., memorizing full BFS levels. By increasing the period between two snapshots it is guaranteed that cycles will eventually reach a "pictured" state. This approach is in general incomparable with the present work. The algorithm of [21] has to keep full levels in memory while ours stores some states of each level. Besides this algorithm, [21] also introduces some hierarchical caching strategies and learning mechanisms.

Some other reduction techniques share the philosophy of state caching: only store a subset of the state space while still guaranteeing termination. Examples include the "to-store-or-not" method [3] and the sweep-line method [20]. The compatibility of our algorithm with these two works is discussed in Sect. 4.

2 General State Exploring Algorithm

A general state exploring algorithm is presented in Fig. 1. It operates on two data structures. The set of open states, $\mathcal{O}$, contains all states that have been reached so far, but for which some successor(s) have not yet been computed. Once all these successors have been computed, the state is moved from $\mathcal{O}$ to the set of closed states $\mathcal{C}$. Initially, the closed set is empty, and the open set only contains the initial state. A set of events *evts* is associated with each open state. It consists of its enabled events that have not been executed so far. In each iteration, the algorithm selects an open state s (l. 3), picks one of its executable events e (if any, since the state may be a terminal state) and removes it from the set of events to execute $s.evts$ (ll. 4–5). The successor state s' of s reached via the execution of e is computed, and if it is neither in the closed nor in the open set (ll. 7–9), it is put in $\mathcal{O}$ to be later visited and its enabled events are computed. Once the successor is computed, we check if all the enabled events of s have been executed (ll. 10–11), in which case we move s from $\mathcal{O}$ to the closed set $\mathcal{C}$.

```
1:  $\mathcal{C} := \emptyset$ ; $\mathcal{O} := \{s_0\}$ ; $s_0.evts := en(s_0)$
2:  while $\mathcal{O} \neq \emptyset$ do
3:      $s :=$ choose from $\mathcal{O}$
4:      if there exists $e \in s.evts$ then
5:          $s.evts := s.evts \setminus \{e\}$
6:          $s' := succ(s, e)$
7:          if $s' \notin \mathcal{C} \cup \mathcal{O}$ then
8:              $\mathcal{O} := \mathcal{O} \cup \{s'\}$
9:              $s'.evts := en(s')$
10:     if $s.evts = \emptyset$ then
11:         $\mathcal{C} := \mathcal{C} \cup \{s\}$ ; $\mathcal{O} := \mathcal{O} \setminus \{s\}$
```

Fig. 1. A general state exploring algorithm

In the rest of this paper we shall use the following terminology. *Expanding* a state s consists of executing one of its enabled events and putting its successor in the open set if needed (ll. 6–9). A state s will be characterized as *expanded* if all its successor states have been computed, i.e., $s.evts = \emptyset$, and it has been moved to the closed set; and as *partially expanded* if some of its successors have been computed, but s is still in the open set. A state s *generates* state s' if $succ(s, e) = s'$ for some $e \in en(s)$ and $s' \notin \mathcal{C} \cup \mathcal{O}$ when expanding s. In this case, the transition (s, s') is said to be the *generating transition.*

The algorithm in Fig. 1 differs slightly from explicit state space search algorithms usually found in the literature, e.g., like the GSEA of [5]. In an iteration, the algorithm only executes one event rather than all executable events of a state. This variation is more flexible as it allows us to have open states that are partially expanded. Hence, it naturally caters for search order independence. Depending on the implementation of the open set, the search strategy can be, for instance, depth-first (with a stack), breadth-first (with a queue), or best-first (with a priority queue). Since each search order has its pros and cons, it is of interest to design reduction techniques that work directly on the generic search order independent template, e.g., like the partial order reduction proposed in [5], rather than on a specific instance.

3 State Caching for GSEA

The key principle of state caching for depth-first search is that cycles always eventually reach a state on the DFS stack. Hence, it is only necessary to keep this stack in memory to ensure termination of the algorithm. In breadth-first search, or more generally for a GSEA, we do not have such a structure to rely on in order to detect cycles. Hence, a BFS naively combined with state caching may never terminate.

To overcome this limitation, we propose to equip GSEA with a mechanism that allows it to avoid reentering cycles of states and thereby ensures termination. The principle of this modification is to maintain, as the search progresses, a so-called *termination detection tree* (`TD`-tree). The `TD`-tree is rooted in the initial state s_0 and keeps track of unprocessed states of the open set as explained below. To formulate the requirements of the TD-tree we shall use the term *search tree*. The search tree is the sub-graph of the state space which at any moment during the execution of GSEA covers all open and closed states, and contains only generating transitions. In other words, it consists of the state space explored so far from which we remove transitions of which the exploration led to an already seen state, i.e., in the set $\mathcal{C} \cup \mathcal{O}$.

The three following invariants related to the TD-tree must be maintained during the state space search:

I1 The TD-tree is a sub-tree of the search tree;
I2 All open states are covered by the TD-tree;
I3 All the leaves of the TD-tree are open states.

A sufficient condition for the modified GSEA to terminate is that we always keep in memory the states belonging to the TD-tree. Intuitively, when expanding a state s picked from the open set, we are sure (provided invariants I1 and I2 are valid) that any cycle covering s will at some point contain a state s' belonging to the TD-tree. States that can be deleted from memory are all closed states that are not part of the TD-tree: their presence is not required to detect cycles. Note that only the two first invariants are required for termination. Invariant I3 just specifies that the TD-tree is not unnecessarily large, i.e., it does not contain states we do not need to keep in the closed set to detect cycles. To sum up, all the states that may not be removed from the cache are (besides open states) all closed states that generated (directly or indirectly through a sequence of generating transitions) a state in the open set.

As an example, let us see see how a BFS extended with this mechanism will explore the state space of Fig. 2(top left). Each state is inscribed with a state number that coincides with the (standard) BFS search order. The TD-tree has been drawn in the right box for several steps of the algorithm. The legend for this box is shown in the bottom left box of the figure. Note that these graphical conventions are used throughout the paper. Some reference counters used to maintain the TD-tree appear next to the states. They will not be discussed now. Their use will become clear after the presentation of the algorithm.

After the expansion of the initial state 0, the queue contains its two successor states 1 and 2 and the TD-tree is equivalent to the search tree (see Step 1). At the next level, we expand open states 1 and 2. State 1 first generates states 3 and 4. As state 4 is already in the open set when state 2 is expanded, this one does not generate any new states, which means that at Step 2, state 2 does not have any successors in the search tree. It is deleted from the TD-tree, and we assume that the algorithm also removes it from the closed set. The expansion of open states 3 and 4 at the next level generates the three states 5, 6 and 7. At Step 3 all the states that were expanded at level 2 generated at least one (new) state. Hence, no state is deleted from the TD-tree. Level 3 is then processed. States 5 and 6 do not have any successors and state 7 has a single successor, state 2, which has been visited but deleted from memory. Hence, it is put in the

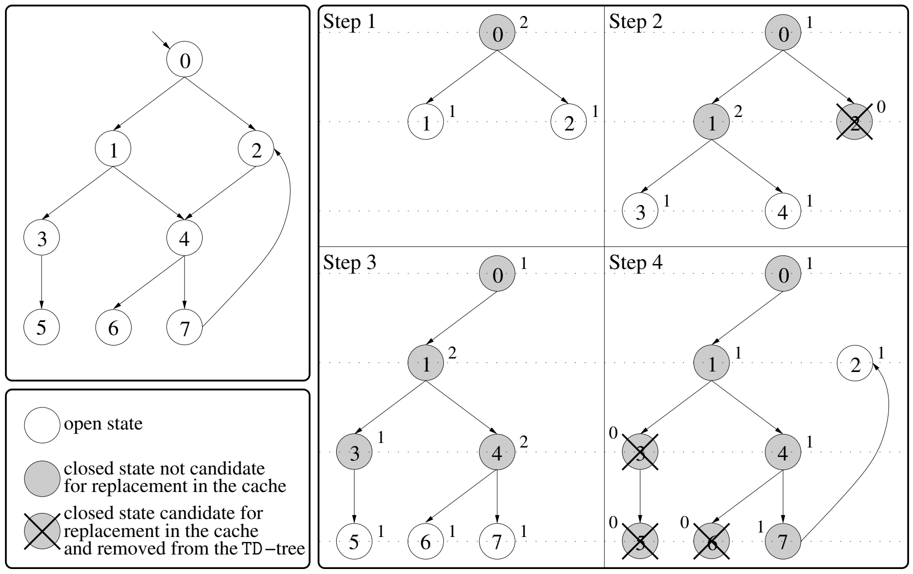
Step 1
Step 2
Step 3
Step 4
open state
closed state not candidate for replacement in the cache
closed state candidate for replacement in the cache and removed from the TD−tree

Fig. 2. A state space (top left), the TD-tree at three different stages of a BFS (right) and the legend for the right box (bottom left). Step 1: after the expansion of state 0. Step 2: after the expansion of states 1 and 2. Step 3: after the expansion of states 3 and 4. Step 4: after the expansion of states 5, 6 and 7.

open set. States 5 and 6 can be deleted from the TD-tree at Step 4 since they are terminal states and do not have any successors in the TD-tree. After the deletion of state 5, state 3 is in the same situation and becomes a leaf of the TD-tree. It is thus also deleted. The queue now only contains state 2 that had already been previously expanded. Its only successor, state 4, belongs to the TD-tree and, hence, is present in memory. The algorithm thus detects the cycle $2 \rightarrow 4 \rightarrow 7 \rightarrow 2$. After this last expansion, the queue is empty and algorithm terminates.

The operation of our algorithm is similar to the basic state caching reduction for DFS with the difference that the TD-tree is implicitly maintained by the DFS state caching algorithm: it consists of all the open states located on the DFS stack. Closed states have left the stack so they have not generated the states currently on the stack and do not need anymore to be part of the TD-tree.

We now introduce the algorithm of Fig. 3, a GSEA extended with the state caching mechanism we described above. The main procedure (on the left column) works basically as the algorithm of Fig. 1 except that we inserted the lines preceded by a ▶ to manage the state cache. Apart from the closed set $\mathcal{C}$ and open set $\mathcal{O}$, the algorithm also uses a set $\mathcal{D} \subseteq \mathcal{C}$ that contains all states candidate for deletion if memory becomes scarce, i.e., all the states that left the TD-tree. Hence, the TD-tree is composed of the states in $(\mathcal{C} \cup \mathcal{O}) \setminus \mathcal{D}$.

After each state expansion, the algorithm calls the *garbageCollection* procedure (l. 13) that checks if the number of states kept in memory exceeds some user-defined limit MaxMemory. In that case, one of the candidates for replacement

```
 1: $\mathcal{C} := \emptyset$ ; $\mathcal{O} := \{s_0\}$ ; $s_0.evts := en(s_0)$
 2: ► $\mathcal{D} := \emptyset$ ; $s_0.refs := 1$ ; $s_0.pred :=$ nil
 3: while $\mathcal{O} \neq \emptyset$ do
 4:     $s :=$ choose from $\mathcal{O}$
 5:     if there exists $e \in s.evts$ then
 6:         $s.evts := s.evts \setminus \{e\}$
 7:         $s' := succ(s, e)$
 8:         if $s' \notin \mathcal{C} \cup \mathcal{O}$ then
 9:             $\mathcal{O} := \mathcal{O} \cup \{s'\}$
10:             $s'.evts := en(s')$
11:             ► $s.refs := s.refs + 1$
12:             ► $s'.refs := 1$ ; $s'.pred := s$
13:             ► $garbageCollection()$
14:     if $s.evts = \emptyset$ then
15:         $\mathcal{C} := \mathcal{C} \cup \{s\}$ ; $\mathcal{O} := \mathcal{O} \setminus \{s\}$
16:         ► $unref(s)$
17: procedure $unref(s)$ is
18:     $s.refs := s.refs - 1$
19:     if $s.refs = 0$ then
20:         $\mathcal{D} := \mathcal{D} \cup \{s\}$
21:         if $s.pred \neq$ nil then
22:             $unref(s.pred)$
23:
24: procedure $garbageCollection()$ is
25:     if $|\mathcal{O}| + |\mathcal{C}| >$ MaxMemory then
26:         if $\mathcal{D} = \emptyset$ then
27:             report "out of memory"
28:         else
29:             $s :=$ choose from $\mathcal{D}$
30:             $\mathcal{C} := \mathcal{C} \setminus \{s\}$
31:             $\mathcal{D} := \mathcal{D} \setminus \{s\}$
```

Fig. 3. A general state exploring algorithm combined with state caching

is selected from $\mathcal{D}$ according to a replacement strategy and deleted from both $\mathcal{C}$ and $\mathcal{D}$ (ll. 29–31) to make room for the state newly inserted in $\mathcal{O}$. If there is no candidate (ll. 26–27), the algorithm terminates with a failure: the TD-tree is too large to fit within user-defined available memory.

In order to maintain the TD-tree, two additional attributes are associated with states. The first one, *pred*, identifies the predecessor of the state that previously generated it, i.e., its predecessor in the search tree. It is set at l. 12 when a new state s' is generated from s. The second attribute, *refs*, is a reference counter used by the *garbageCollection* procedure to determine when a closed state leaves the TD-tree and can become a candidate for replacement. The following invariant is maintained by the algorithm for any $s \in \mathcal{C} \cup \mathcal{O}$:

$$\textbf{I4} \quad s.refs = |\{s' \in \mathcal{C} \cup \mathcal{O} \textbf{ with } s'.pred = s \wedge s'.refs > 0\}| + \begin{cases} 0 \textbf{ if } s \notin \mathcal{O} \\ 1 \textbf{ if } s \in \mathcal{O} \end{cases}$$

In other words, $s.refs$ records the number of successors of s in the TD-tree incremented by 1 if s is an open state. It directly follows from invariant I4 that any state s with $s.refs = 0$ must be moved to $\mathcal{D}$ and deleted from the TD-tree in order to satisfy invariant I3. This is the purpose of procedure *unref* called each time a state s leaves the open set (ll. 15–16). Its counter is decremented by 1, and if it reaches 0 (ll. 19–22), s is put in the candidate set and the procedure is recursively called on its predecessor in the TD-tree (if any).

Let us consider again the TD-trees depicted in Fig.2(right). Reference counters are given next to the states. At Step 4, after the expansion of states 5, 6, and 7 of level 3, the reference counter of 5 and 6 reaches 0: they have left the open set and have no successors in the search tree, i.e., they did not generate any new state. They can therefore be put in the candidate set and leave the TD-tree. *unref* is then also recursively called on states 3 and state 4 and their counters are decremented to 0 and 1, respectively. Hence, state 3 is also put in the candidate set. This finally causes *unref* to decrement the reference counter of state 1 to 1.

Lemma 1. *The algorithm of Fig.3 terminates after visiting all states.*

Proof. Let $\mathcal{T}$ be the (only) sub-tree of the search tree that satisfies invariants I1, I2 and I3 and assume that the states belonging to $\mathcal{T}$ always remain in set $\mathcal{C} \cup \mathcal{O}$. Let $s_1, \ldots, s_n \in \mathcal{S}$ be a cycle of states with $\forall i \in \{1, \ldots, n\} : succ(s_i, e_i) = s_{i \bmod n+1}$, and such that s_1 is its first state to enter $\mathcal{O}$. This cycle is necessarily detected, i.e, during the search we reach some $s_i \in \mathcal{C} \cup \mathcal{O}$. Let us suppose the contrary. Then, each state $s_i \in \{s_1, \ldots, s_{n-1}\}$ generates the state $s_j = s_{i+1}$, i.e., $s_j \notin \mathcal{C} \cup \mathcal{O}$ when event e_i is executed from s_i. Hence, after the execution of e_{n-1} by the algorithm it holds from invariants I1 and I2 that $\{s_1, \ldots, s_n\} \subseteq \mathcal{T}$ since $s_n \in \mathcal{O}$ and each $s_j \in \{s_2, \ldots, s_n\}$ was generated by s_{j-1}. Thus, $s_1 \in \mathcal{C} \cup \mathcal{O}$ when event e_n is executed from s_n, which contradicts our initial assumption.

It is straightforward to see from invariant I4 that $s.refs > 0 \Leftrightarrow s \in \mathcal{T}$. After an iteration of the algorithm (ll.3–16), invariant I4 is trivially ensured. This implies that any $s \in \mathcal{C}$ with $s.refs = 0$ ($\Leftrightarrow s \in \mathcal{D}$) can be deleted from $\mathcal{C}$.

The modified GSEA of Fig. 3 consumes slightly more memory per state to represent the *pred* and *refs* attributes. In our implementation *pred* is encoded with a 4 byte pointer and *refs* using a single byte. Nevertheless, these 5 bytes are usually negligible compared to the size of the bit vector used to encode states.

4 Compatibility with Other Reduction Techniques

We discuss in this section several aspects of our algorithm and its combination with some selected reduction techniques.

4.1 Single-Successor States Chain Reduction

Closely related to state caching, the idea of [3] is to use a boolean function that, given a state, determines if the state should be kept in the closed set or not. The paper proposes functions that guarantee the termination of the search. One is to only store states having several successors. The motivation is that the revisit of single-successor and deadlock states is cheap. It consists of reexecuting a sequence until reaching a branching state (which has several successors and which is therefore stored). To avoid entering cycles of single-successor states, the k^{th} state of these sequences is systematically stored.

In order to combine this reduction of [3] with our state caching mechanism we have to carefully reduce chains that are part of the TD-tree. First we notice that we do not have to worry about cycles of single-successor states: they will eventually reach a state of the TD-tree and all their states will immediately leave the TD-tree becoming candidates for replacement. To reduce chains of single-successor states we associate with each open state s an attribute *ancestor* that points to the branching state that generated the first state of the chain to which s belongs, or is equal to **nil** if s is a branching state. The following piece of code specifies how this attribute is used to remove such chains from the TD-tree. It must be inserted after the generation of state s' from s at line 12.

```
$s'.ancestor := \begin{cases} \textbf{if } |en(s')| = 1 \textbf{ and } s.ancestor = \textbf{nil then } s \\ \textbf{if } |en(s')| = 1 \textbf{ and } s.ancestor \neq \textbf{nil then } s.ancestor \\ \textbf{else nil} \end{cases}$
if $s'.ancestor =$ nil and $s.ancestor \neq$ nil then
    $a := s.ancestor$ ; $s'.pred := a$ ; $a.refs := a.refs + 1$ ; $unref(s)$
```

ancestor is first set when a single-successor state is generated from a branching state and then propagated later along all the states of the chain. If s' is a branching state and $s.ancestor$ points to some state, this means that we just left a chain. The reduction is done by directly linking s' to $s.ancestor$ and removing all the states of the chain from the TD-tree, by unreferencing s.

Fig. 4 shows an example of this reduction. Dotted arcs graphically represent the *pred* pointer of each state in the TD-tree.At Step 1, $s_1, \ldots, s_n$ form a reducible chain of single-successor states. All their *ancestor* field points to a, the branching state that generated the first state of the chain. State s_n then generates s that has several successors (see Step 2). Hence $s.ancestor = \textbf{nil} \neq s_n.ancestor$ and a reducible chain is detected. The consequence is to break the link from s to s_n and make $s.pred$ directly point to $s_n.ancestor = a$. The chain is then removed from the TD-tree by invoking $unref(s_n)$.

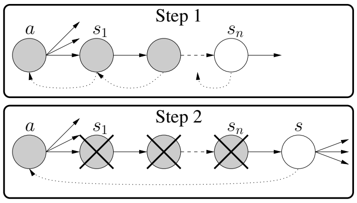

Fig. 4. Reduction of a single-successor state chain

4.2 Distributed Memory Algorithms

Most works in the field of distributed verification follow the seminal work of Stern and Dill [24]. Their algorithm partitions the state space upon several processes using a partition function mapping states to processes. Each process involved in the verification is responsible of storing and exploring the states it is assigned by this function. Whenever a process p generates a state owned by process $q \neq p$ it has to pack it into a message and send it to q that, upon reception, will store it in its open state and expand it later.

Our reduction is compatible with this algorithm. The only issue is raised by the *unref* procedure, used to maintain the TD-tree. This one has to access the ancestors of the unreferenced state — ancestors that may be located on another process — hence generating communications. A possible way to overcome this problem is to only call that procedure when memory becomes scarce. Processes then enter a garbage collection phase where they clean the TD-tree and delete states from memory. Thus, if the aggregated memory is large enough to solve the problem without any reduction then there is no time overhead. This might also help to group states sent to the same owner and thereby reduce communication.

Another solution is to associate with each new state s received from another process a reference number of 2 (rather than 1) to ensure it will never leave the TD-tree (and that no communication will occur). It is then not necessary that the source state of the transition that generated the message keeps a reference to s. An example of a TD-tree (actually a forest) distributed over two processes can be seen in Fig 5. Dashed transitions are not part of the TD-tree. The reference counter of state s is set to 3 whereas it is closed and only has 2 successors. The reason is that it has been first discovered upon its reception by process 1. Hence, it will never leave the TD-tree, guaranteeing that the cycle $t_0.t_1.t_2$ will be detected regardless of the order in which states are visited. Another difference with a "sequential" TD-tree is that the counter of s_0 is set to 1 instead of 2 since it only has one successor in the TD-tree on the same process. Hence, although s will remain in the memory of process 1, s_0 will eventually be allowed to leave the cache.

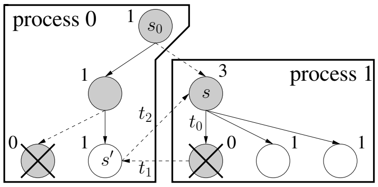

Fig. 5. A distributed TD-tree

Both solutions should benefit from a partitioning exhibiting few cross transitions linking states belonging to different processes. This will limit communications for the first solution and enhance the reduction in the second case.

4.3 Reductions Based on State Reconstruction

Our GSEA is also compatible with the reduction techniques proposed in [10] and [27]. Instead of keeping full state vectors in the closed and open sets, their principle is to represent a state s as a pair $(pred, e)$ where $pred$ is a pointer to the state s' that generated s during the search, and e is the event such that $succ(s', e) = s$. States can be reconstructed from this compressed representation by reexecuting the sequence of events that generated it. At a reasonable cost in time, it allows each state to be encoded by 12–16 bytes whatever the system being analyzed. This reduction fits nicely with the algorithm of this paper. The TD-tree can be compactly stored using this representation of states and, actually, both methods store with each state a pointer to its generating predecessor. The only states that have to be fully stored in memory are those who left the TD-tree since their generating predecessor may not be present anymore in memory.

4.4 The Sweep-Line Method

A sweep-line based algorithm alternates between exploration phases where states are visited and their successor(s) generated; with garbage collection phases where states are removed from memory. A key feature of the method is the progress measure ψ mapping states to (ordered) progress values. It is used to estimate "how far" states are from the initial states and guides the garbage collection procedure: if the minimal progress value found in the set of open states is $\alpha_{min} = \min_{s \in \mathcal{O}} \psi(s)$ then all closed states s with $\psi(s) < \alpha_{min}$ can be deleted from memory. The underlying idea is that if the progress mapping is monotonic, i.e., all transitions (s, s') are such that $\psi(s) \leq \psi(s')$, then a visited state s with a progress $\psi(s) < \alpha_{min}$ will not be visited again. In [20] the method is extended to support progress measures with regress transitions, i.e., transitions (s, s') with $\psi(s) > \psi(s')$, that with the basic method of [6] cause the algorithm to not terminate. The principle of this extension is to mark destination of regress transitions as persistent to prevent the garbage collector from deleting them.

The sweep-line method can also be used in conjunction with our reduction. This stems from the fact that the algorithm of [20] is also an instance of the GSEA of Fig. 1 that keeps open states in a priority queue (priority being given to states having the lowest progression). However, one has to proceed carefully when combining both methods: the *unref* procedure of our algorithm may not put in the set $\mathcal{D}$ of candidates for replacement an unreferenced state (i.e., with $s.refs = 0$) that has been marked as persistent by the sweep-line reduction. Note that the predecessor of a persistent state may however be unreferenced.

Running the sweep-line algorithm in combination with our state caching algorithm causes the deletion of non-persistent states stored in the `TD`-tree. This means that we only store the parts of the `TD`-tree corresponding to the states determined to be in memory by the sweep-line method. The role of the `TD`-tree (which now becomes a forest) is to ensure termination of each of the phases of the sweep-line method, while the overall termination of the combined search is guaranteed by the persistent states stored by the sweep-line method.

5 Experiments

The technique proposed in this paper has been implemented in the ASAP verification tool [26]. ASAP can load models written in DVE [7], the input language of the DiVinE verification tool [2]. This allowed us to perform numerous experiments with models from the BEEM (BEnchmarks for Explicit Model checkers) database [22] although “Puzzles” and “Planning and scheduling” problems were not considered. These are mostly toy examples having few characteristics in common with real-life models. We performed two experiments, studying the performance of our reduction in combination with basic search algorithms for the first one; and with the sweep-line method for the second one. Due to lack of space, some data has been left out in this section but may be found in [9].

5.1 Experiment 1: State Caching with Basic Search Strategies

State caching is a rather unpredictable technique in the sense that its performance depends on a large range of parameters, e.g., the size and replacement strategy for the cache, the characteristics of the state space. It is usually hard to guess which configuration should be used before running the model checker. State caching must therefore be experimented with in a wide range of settings and with many different state spaces in order to get a good insight into its behavior. In this experiment we performed more than 1,000,000 runs using different search algorithms, caching strategies, cache sizes, and state space reduction techniques.

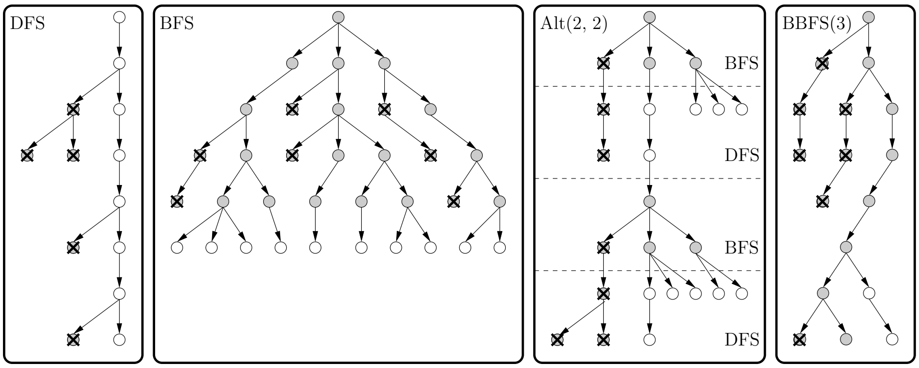

Fig. 6. Snapshot of the search trees with different search strategies

Experimentation Context

Search strategies. Several preliminary experiments revealed important variations of our state caching reduction when combined with DFS or BFS. Therefore, it seemed interesting to combine both algorithms to observe if such a combination could improve on pure breadth- or depth-first searches. Thus, in addition to DFS and BFS, we also experimented with the two following variations of these search strategies devised for the sake of our experimentation.

– *Alternation of breadth- and depth-first search.* This search strategy is parametrized by two integers b and d. It starts breadth-first on the b first levels. Then for the states at level b the search switches to a DFS until the depth $b + d$ is reached. At that point, the algorithm mutates back to a BFS and so on. This search will be denoted by Alt(b, d) in the following.
– *Bounded-width breadth-first search.* This search strategy, denoted BBFS(w), proceeds as a BFS except that the queue at each level may not contain more than w states (the width of the search). Open states of previous levels are kept in a stack to be expanded later when all next levels have been processed.

Note that DFS and BFS are special instances of these search strategies since it holds that DFS $\equiv$ BBFS(1) $\equiv$ Alt(0, ∞) and BFS $\equiv$ BBFS(∞) $\equiv$ Alt(∞, 0).

A snapshot of the search trees of different state spaces induced by these different strategies can be seen in Fig. 6. With DFS (left), all states outside the stack are candidates for replacement in the cache. With BFS (second left), any ancestor in the search tree of an open state must remain in the cache while others may be replaced. Open states, with algorithm Alt(2, 2) (second right) are those still present in the stacks and queues used to perform “local” DFSs and BFSs. At last, the tree of BBFS(3) (right) is a BFS tree where each level can contain at most 3 states. Unlike BFS, some previous levels may contain open states as is the case here with the penultimate level.

All these algorithms are implemented using the generic template of Fig. 3 and parametrized by the type of the $\mathcal{O}$ data structure. BFS is implemented with a queue, DFS with a stack, BBFS(w) with a stack of arrays of size w, and Alt(b, d) with a stack containing single states for DFS levels and queues for BFS levels.

Cache replacement strategies. We implemented various strategies from the literature. The garbage collector can select states according to their in- and out-degree, their distance from the initial state (i.e., the depth at which the state has been generated), or in a purely random fashion. Stratified caching [11] has also been implemented. Due to a lack of space, we will not compare these strategies here. For DFS, the reader may consult the large body of work on that subject, e.g., [11,23]. With BFS, distance seems to be the criterion with most impact.

State space reduction. Sleep-set reduction has been shown in [13] to drastically reduce state revisits when using state caching. Rather than sleep-sets, we implemented the reduction of [4] which proposes a sleep-set like technique for both DFS and BFS that has two advantages over it: it does not require any memory overhead whereas the algorithm of [13] associates a set of transitions (the sleep set) with each closed and open set; and it is easier to implement.

We also implemented and experimented with the single-successor state chain reduction of [3] which is compatible with our algorithm as explained in Sect. 4.1. This reduction will be denoted by CR in the following.

Table 1 summarizes the different parameters and instances we experimented with. Unlike the reduction of [4] which was always turned on, the reduction of [3] was a parameter of each run. A run ended in one of three situations:

success. The search could finish within allocated resources.
out of memory. The cache was two small to contain the `TD`-tree.
out of time. The algorithm visited more states than the specified threshold.

Experimental Results

Experimental data is reported in Table 2. Due to space constraints, the table only contains data for 25 selected instances although the average on all instances experimented with is reported on the last line. Under each instance name, we give its number of states |S| and its average degree d as the ratio of transitions over states. All tests performed for each instance were divided into 8 groups according to the search algorithm they used (columns DFS, BFS, BBFS, and Alt) and according to whether or not they used the chain compression reduction (column CR). We then ordered, within each group, all successful runs first by ascending cache size (i.e., memory) and then by ascending number of state visits (i.e., time). Each cell of the table contains data for the best run according to that order: the number of stored states, i.e., the cache size, (column S) and visited states (column V) both expressed as a percentage of the state space. Additionally, for algorithms Alt and BBFS, columns (b, d) and w specify the parameters of the search that the best run used.

Table 1. Instances and parameters used during Experiment 1

Selected instances	135 instances with { 1,000, . . . , 1,000,000 } states
Maximal state visits	$5 \cdot \lvert S \rvert$ (where $\lvert S \rvert$ is the state space size)
Cache size	{ 5, 10, 15, 20, . . . } (as a % of the state space size) until a successful run could be found
Cache replacement strategy	60 caching strategies selected after experimentation with a small sample of 10 instances
Search strategy	DFS, BFS, BBFS(w) for $w \in \{4, 16, 256\}$, Alt(b, d) for $b, d \in \{1, 4, 8\}$
Reductions used	Reduction of [4] for all runs and CR for some runs

State caching apparently provides a better memory reduction when coupled with DFS than with BFS. The size of the `TD`-tree, with BFS, is lower bounded by the width of the state space, i.e., the size of the largest level, which can be high for some models. The DFS stack can also contain a large proportion of the state space but the reduction of [4] not only reduces interleavings but also the stack size, whereas it is not helpful in BFS.

We still found some models for which BFS outperformed DFS with respect to both time and memory. This is the case for instances `extinction.3`, `firewire_tree.4` and `leader_election.4`. We will see later why BFS is to be preferred for these models.

Some instances like `cambridge.5` have typical characteristics that make state caching inadequate: many cycles and a high degree. This inevitably leads to a time explosion with DFS even using partial order reduction. BFS seems to be more resilient with respect to these instances. We found a couple of similar instances during our experiments.

Although state caching is generally more memory efficient when coupled with DFS, BFS still provides a notable advantage: it is less subject to a time explosion. Even in cases where the cache size was close to its lower bound, i.e., the maximal size of the `TD`-tree, the time brutally increased in very few cases. With BFS, the distribution of run failures is the following: 98% are "out of memory", and 2% are "out of time". With DFS, these percentages become respectively 61% and 39%. Even in cases where BFS "timed out", increasing the maximal number of state visits from $5 \cdot |S|$ to $20 \cdot |S|$ could turn all these runs into successes. This is, from the user point of view, an appreciable property. With DFS, when the user selects a small cache size, and the search takes a long time, he/she can not know if it is due to a high rate of state revisits or if it is because the state space is very large and state caching is efficient. This situation never occurred with BFS.

In general, we found out that BFS is far less sensitive to the caching strategy than DFS. For a specific cache size, it was not unusual, with DFS, that only one or two replacement strategies could make the run successful. Whereas, with BFS, once a successful run could be found with some strategy, it usually meant that many other runs using different strategies (and with the same cache size) could also terminate successfully. On an average made on all models we calculated

Table 2. Summary of data for Experiment 1

Model	CR	DFS		BFS		BBFS			Alt		
		S	V	S	V	S	V	w	S	V	(b,d)
`at.2`	no	35	228	40	130	30	369	4	30	219	(8,4)
|S|=49,443 d=2.9	yes	30	428	40	130	30	373	4	30	222	(8,4)
`bakery.4`	no	20	271	25	216	25	173	4	20	185	(8,4)
|S|=157,003 d=2.6	yes	20	271	25	149	25	168	4	20	185	(8,4)
`bopdp.2`	no	15	215	30	149	15	246	4	15	213	(1,8)
|S|=25,685 d=2.8	yes	10	437	25	162	10	398	4	10	406	(1,8)
`brp.3`	no	5	235	15	112	5	163	256	5	160	(8,8)
|S|=996,627 d=2.0	yes	5	152	10	116	5	168	256	5	153	(8,8)
`brp2.5`	no	5	141	30	112	5	130	4	5	135	(8,1)
|S|=298,111 d=1.4	yes	5	140	20	103	5	131	4	5	137	(8,1)
`cambridge.5`	no	45	285	35	121	45	203	4	45	232	(8,4)
|S|=698,912 d=4.5	yes	45	287	35	120	45	200	16	45	233	(8,8)
`collision.3`	no	10	484	30	119	10	487	16	10	239	(8,1)
|S|=434,530 d=2.3	yes	10	338	25	169	10	412	16	10	286	(8,1)
`extinction.3`	no	10	185	10	100	10	186	4	10	176	(4,4)
|S|=751,930 d=3.5	yes	10	184	10	100	10	187	4	10	176	(4,4)
`firewire_link.7`	no	5	327	25	101	5	278	256	5	214	(8,1)
|S|=399,598 d=2.7	yes	5	321	20	101	5	348	256	5	230	(8,1)
`firewire_tree.4`	no	10	337	10	100	15	190	4	10	313	(4,4)
|S|=169,992 d=3.7	yes	10	345	10	100	15	191	4	10	311	(4,4)
`gear.2`	no	15	109	10	100	10	100	256	10	102	(4,4)
|S|=16,689 d=1.3	yes	15	106	5	102	5	101	256	10	101	(4,4)
`iprotocol.2`	no	5	359	20	132	5	250	4	5	296	(8,1)
|S|=29,994 d=3.3	yes	5	364	20	132	5	245	16	5	294	(1,1)
`lamport_nonatomic.3`	no	45	398	50	120	45	257	4	45	237	(8,8)
|S|=36,983 d=3.3	yes	45	366	50	119	45	260	4	45	232	(8,1)
`leader_election.4`	no	15	450	10	100	15	456	16	15	273	(4,1)
|S|=746,240 d=5.0	yes	15	316	10	100	15	453	16	15	275	(4,1)
`lifts.6`	no	5	123	15	123	5	125	4	5	124	(1,1)
|S|=333,649 d=2.1	yes	5	124	15	112	5	132	4	5	124	(1,1)
`lup.2`	no	30	478	30	142	40	369	4	15	429	(8,8)
|S|=495,720 d=1.8	yes	30	294	30	142	35	324	4	15	428	(8,8)
`needham.3`	no	5	412	30	100	5	409	256	5	435	(1,1)
|S|=206,925 d=2.7	yes	5	415	30	100	5	417	256	5	435	(1,1)
`peterson.3`	no	25	345	35	142	25	374	4	25	329	(1,8)
|S|=170,156 d=3.1	yes	25	331	35	136	25	356	4	25	320	(8,8)
`pgm_protocol.7`	no	10	152	10	106	5	360	16	5	493	(8,1)
|S|=322,585 d=2.5	yes	10	152	5	112	5	392	4	10	145	(4,1)
`plc.2`	no	5	100	30	100	10	101	4	5	100	(1,8)
|S|=130,777 d=1.6	yes	5	100	10	100	5	113	4	5	101	(8,8)
`production_cell.4`	no	10	176	10	100	10	162	4	10	157	(8,1)
|S|=340,685 d=2.8	yes	10	175	10	100	10	162	4	10	158	(8,1)
`rether.3`	no	30	118	40	104	35	132	256	25	113	(8,1)
|S|=305,334 d=1.0	yes	25	152	15	111	15	117	256	15	128	(8,1)
`synapse.6`	no	5	148	30	111	5	133	16	5	129	(8,8)
|S|=625,175 d=1.9	yes	5	118	25	296	5	136	16	5	129	(8,8)
`telephony.3`	no	30	394	50	124	25	482	256	25	368	(8,1)
|S|=765,379 d=4.1	yes	25	491	50	123	25	470	16	25	388	(8,1)
`train-gate.5`	no	10	114	20	100	10	110	4	10	102	(4,1)
|S|=803,458 d=2.1	yes	10	114	20	100	10	105	256	10	102	(4,1)
Average on	no	18.5	259	30.1	131	19.4	236		16.4	239	
135 models	yes	17.7	251	26.3	143	17.4	241		15.5	249	

that, with DFS and for the smallest cache size for which at least a run turned out to be successful, only 25% of all runs were successful. With BFS, this same percentage goes up to 66%.

These observations are in line with a remark made in [21], p. 226: "*Compared to BFSWS* [BFS With Snapshots]*, the success of DFS setups differs a lot from one case to another.*" One of our conclusions is indeed that BFS has the advantage of exhibiting more predictable performance.

The effect of reduction CR is more evident in the case of BFS. The cache size could be further reduced by an average of 4–5% for a marginal cost in time, whereas with DFS the reduction achieved is negligible. In some instances BFS could, with the help of this reduction, significantly outperform DFS with respect to memory consumption. This is for instance the case for `rether.3` which has a majority of single-successor states.

Lastly, we notice that Alt and BBFS sometimes cumulate the advantages of both BFS (w.r.t. time) and DFS (w.r.t. memory) and perform better than these, e.g., `at.2`, `bakery.4`, `firewire_link.7` and `lup.2`. Search Alt also seems to be more successful than BBFS. It could on average reduce the cache size from 17 to 15% of the state space, when compared to DFS.

Influence of the State Space Structure in BFS. We previously noticed that the width of the graph is a lower bound of the `TD`-tree in BFS. More generally, there is a clear link between the shape of the BFS level graph and the memory reduction in BFS. Figure 7 depicts this graph for several instances that are of particular interest to illustrate our purpose. For each BFS level, the value plotted specifies the number of states (as a percentage of the full state space) belonging to a specific level. For instances `telephony.3` and `synapse.6`, a large proportion of states is gathered on a few neighbor levels. The algorithm will thus have to store most states of these levels and it is not surprising to observe in Table 2 that state caching is not efficient in these cases. Instances `pgm_protocol.7` and `gear.2` have the opposite characteristic: the distribution of states upon BFS levels is rather homogeneous and there is no sequence of neighbor levels containing many states. This explains the good memory reduction observed with BFS on these examples.

With DFS, the time increase is closely related to the average degree of the state space. This factor has a lesser impact with BFS: the proportion of backward transitions[1] plays a more important role. Indeed, the search order of BFS implies the destination state of a forward transition to necessarily be in the open set. Hence, only backward transitions may be followed by state revisits. Instances `extinction.3` and `firewire_tree.4` have rather high (3–5) degrees but few or no backward transitions, which led to very few state revisits with BFS; whereas using DFS we often experienced a time explosion with these instances. The fact that state spaces of real-world problems often have few backward transitions [25] may explain the small state revisit factors usually observed with BFS.

[1] (s, s') is a backward transition if the BFS levels d and d' of s and s' are such that $d \geq d'$. The length of (s, s') is the difference $d - d'$.

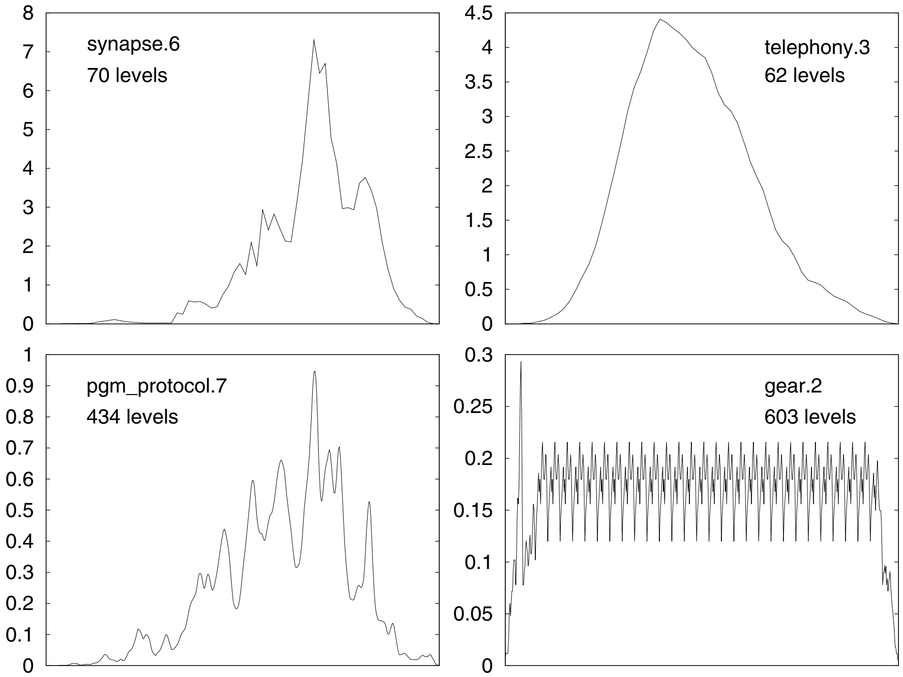

Fig. 7. BFS level graph of some instances

5.2 Experiment 2: State Caching with the Sweep-Line Method

In a second experiment we studied how our state caching algorithm combines in practice with the sweep-line method, as described in Section 4.4. We analyzed the same instances used in the first experiment. We automatically derived from their full state spaces several progress measures identified by a level ranging from 0 to 5. The higher the level, the more precise the progress mapping and the more aggressive the reduction. Our progress measures are abstractions in that they project states to some components of the underlying system. At level 0, the progress mapping is guaranteed to not generate any regress transition since we only consider components having a monotonic progression, e.g., an increasing sequence number; and as the level progresses, the mapping is refined by including in it more and more components in order to multiply the different progress values possible and increase the potential of the reduction (while also introducing more regress transitions). Progress values are then fixed-size vectors of (selected) components that can be compared via a lexical ordering. Each level corresponds to an estimation of the upper bound of the proportion of regress transitions: level 0 $\rightarrow$ no regress transition, level 1 $\rightarrow$ at most 1% of regress transitions, level 2 $\rightarrow$ 2%, level 3 $\rightarrow$ 5%, level 4 $\rightarrow$ 10% and level 5 $\rightarrow$20%.

Table 3 summarizes the data collected during this experiment for some selected instances and progress measures. For each instance (column Model) and progress measure (of which the level appears in column PM) we performed a first

Table 3. Summary of data for Experiment 2

Model	|S|	PM	SL				SL + SC	
			S	V	P	L	S	V
bopdp.2	25,685	3	52.4	231	0.5	51.9	20.4	498
		4	26.8	242	0.8	26.0	14.9	480
		5	13.7	259	1.9	8.3	12.7	482
brp.3	996,627	2	13.3	145	0.1	8.3	4.9	203
		3	10.6	126	0.5	2.9	9.7	154
		4	10.0	122	2.4	0.3	9.9	120
		5	8.1	124	3.8	0.1	8.2	119
collision.3	434,530	4	24.1	100	0.0	24.1	13.5	482
		5	21.1	232	9.0	12.2	16.0	337
iprotocol.1	6,814	5	50.4	100	0.0	47.3	21.0	416
lifts.4	112,792	5	33.3	100	0.0	33.3	8.5	489
needham.3	206,925	5	20.9	100	0.0	0.1	21.0	101
pgm_protocol.3	195,015	3	2.4	144	1.1	0.0	2.1	129
		4	3.2	110	3.1	0.0	3.1	110
		5	9.0	110	8.9	0.0	9.0	108
plc.2	130,777	3	1.8	100	0.0	1.7	0.6	102
		4	1.2	105	1.2	0.0	1.3	105
		5	1.2	105	1.2	0.0	1.3	105
production_cell.4	340,685	4	27.7	100	0.0	22.7	14.1	485
		5	11.1	100	2.0	6.9	5.5	438
rether.3	305,334	4	25.3	161	3.9	21.8	5.0	200
		5	6.3	152	6.0	0.6	6.1	144
synapse.6	625,175	3	26.6	100	0.0	26.6	7.4	477
		4	27.4	316	2.7	16.4	17.2	477
		5	18.1	150	10.8	2.9	17.4	136

run with the sweep-line algorithm of [20] (column SL) and then several runs with the same algorithm extended with our state caching reduction (column SL + SC) using different cache sizes. We used a variation of the stratified caching strategy [11] that revealed to be the most efficient one during the first experiment. As in the first experiment we only kept, for algorithm SL + SC, the "best" run, that is, the one that used the smallest amount of memory with a state visit factor less than 5. For these two runs columns S and V provide the number of stored and visited states. Additionally, for algorithm SL the table gives in column P the number of persistent states at the end of the search ; and in column L the size of the largest class of states sharing the same progress value and present in memory at the same time (before being garbage collected). All these values are expressed as a percentage of the state space size given in column |S|. The data of the run that provided the best memory reduction for a model has been highlighted using a gray background.

To have a better understanding of these results it seems necessary to briefly recall the principle of algorithm SL. At each step the algorithm explores a class of states sharing a common progress value ψ. All their successors are put in the priority queue implementing the open set, and once this expansion step is finished, i.e., no state with progress value ψ is in the queue, the algorithm deletes from memory all expanded states. It then reiterates this process with the next progress value found in the queue until it is empty. The size of the largest class of states with the same progress value given in column L is thus a lower bound on the memory consumption of the algorithm. By implementing state caching on top of this algorithm we can hope to reduce only the class of states the algorithm is currently working on. Indeed if we denote by ψ the progress value of this class then all the states with a progress value $\psi' < \psi$ have been garbage collected by the sweep-line reduction and states with a progress value $\psi' > \psi$ present in memory are necessarily in the open set and hence can not be removed from the cache. The potential of the state caching reduction is thus given by the ratio $\frac{\mathsf{L}}{\mathsf{S}}$ (in column SL). It is therefore not surprising that the gain of using state caching depends on the size L of this largest class. Instances `brp.3`, `plc.2` and `pgm_protocol.3` are the typical examples of models for which the sweep-line method is well suited: they have long state spaces with a clear progression which enables progress mappings to be defined that divide the state space into many small classes. Hence, the sweep-line method used solely can provide a very good reduction that can not be significantly enhanced by state caching.

Increasing the PM level has three noteworthy consequences. First, for most models, the peak number of states stored by SL decreases although this is not always the case: by refining the progress mapping we usually increase the proportion of regress transitions generating persistent states that will never be garbage collected. Instance `pgm_protocol.3` is a good illustration. Second, by multiplying progress values we naturally decrease the effect of state caching since, as we previously saw, state caching is helpful to reduce classes of states sharing the same progress value. We indeed observe that the values in column L decrease as we increase the PM level, with the consequence that numbers converge to the same values with both algorithms. A last observation is that, regarding the number of states stored, SL and SL + SC often follow opposite behaviors. For instances `brp.3`, `collision.3`, `plc.2`, `rether.3` and `synapse.6`, SL consumes less memory when we refine the progress measure whereas SL + SC needs a larger cache. More generally, we observed that the average ratio of the number of states stored by SL over the number of states stored by SL + SC is maximal at level 3 (2.93) and then decreases to 1.97 at level 5. This is an interesting property from a user perspective. This indeed means that he/she does not necessarily have to provide a very fine tuned progress mapping to the model checker. In many cases, a basic mapping that extracts some monotonic component(s) from the model will be sufficient: the state caching reduction will then fully complement the sweep-line reduction flaws and provide an even better reduction compared to a very precise progress mapping that will cancel the benefits of state caching.

6 Conclusion

In this paper we have proposed an extension of state caching to general state exploring algorithms. Termination is guaranteed by maintaining, as the search progresses, a tree rooted in the initial state that covers all open states. This ensures that any cycle will eventually reach a state of the tree. Closed states that have left the tree can therefore be deleted from memory without endangering the termination of the algorithm. Extensive experimentation with models from the BEEM database has revealed that state caching can reduce the memory requirements of a BFS by a factor of approximately 4. Although this is usually not as good as the reduction observed with DFS, BFS offers an advantage in that the reduction comes almost for free: the average increase in run-time that we observed with BFS was usually around 30–40%, and we observed very few cases of time explosion, whereas this is quite common with DFS even when using partial order reduction. Combining both search strategies can also bring advantages: in some cases, we found that state caching coupled with a combination of BFS and DFS could bring the same (or an even better) reduction as with DFS while limiting the run-time explosion that could occur with this one. Last but not least, our experiments revealed that our reduction can also enhance the sweep-line method as the algorithm it relies on is also an instance of the GSEA.

The algorithm we proposed is fully language-independent in that it only relies on a successor function to explore the state space. Nevertheless, it should be worth experimenting it with other formalisms than DVE and especially Colored Petri Nets (CPN) [19]. It is possible that the high level constructs provided by CPN to express the successor function may impact the structural characteristics of state spaces that, as previous works on state caching and our own experiments revealed, are tightly linked to the performance of state caching. Such an experimentation is therefore the next research direction we will focus on.

References

1. Bao, T., Jones, M.: Time-Efficient Model Checking with Magnetic Disk. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 526–540. Springer, Heidelberg (2005)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, 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)
3. Behrmann, G., Larsen, K.G., Pelánek, R.: To Store or Not to Store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
4. Bošnački, D., Elkind, E., Genest, B., Peled, D.: On Commutativity Based Edge Lean Search. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 158–170. Springer, Heidelberg (2007)
5. Bošnački, D., Leue, S., Lafuente, A.L.: Partial-Order Reduction for General State Exploring Algorithms. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 271–287. Springer, Heidelberg (2006)
6. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
7. DVE Language, `http://divine.fi.muni.cz/page.php?page=language`

8. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed Explicit-State Model Checking in the Validation of Communication Protocols. STTT 5, 247–267 (2004)
9. Evangelista, S., Kristensen, L.M.: Search-Order Independent State Caching. Technical report (2009), `http://daimi.au.dk/~evangeli/doc/caching.pdf`
10. 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)
11. Geldenhuys, J.: State Caching Reconsidered. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 23–38. Springer, Heidelberg (2004)
12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
13. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-Space Caching Revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
14. Holzmann, G.J.: Tracing Protocols. AT&T Technical J. 64(10), 2413–2434 (1985)
15. Holzmann, G.J.: Automated Protocol Validation in Argos: Assertion Proving and Scatter Searching. IEEE Trans. Software Eng. 13(6), 683–696 (1987)
16. Holzmann, G.J.: State Compression in Spin: Recursive Indexing and Compression Training Runs. In: SPIN 1997 (1997)
17. Jard, C., Jéron, T.: On-Line Model Checking for Finite Linear Temporal Logic Specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 189–196. Springer, Heidelberg (1990)
18. Jard, C., Jéron, T.: Bounded-memory Algorithms for Verification On-the-fly. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 192–202. Springer, Heidelberg (1992)
19. Jensen, K., Kristensen, L.M.: Coloured Petri Nets — Modeling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
20. Kristensen, L.M., Mailund, T.: A Generalised Sweep-Line Method for Safety Properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
21. Mateescu, R., Wijs, A.: Hierarchical Adaptive State Space Caching Based on Level Sampling. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 215–229. Springer, Heidelberg (2009)
22. 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)
23. Pelánek, R., Rosecký, V., Sedenka, J.: Evaluation of State Caching and State Compression Techniques. Technical report, Masaryk University, Brno (2008)
24. Stern, U., Dill, D.L.: Parallelizing the Murphi Verifier. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 256–278. Springer, Heidelberg (1997)
25. 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)
26. 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)
27. 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)