In Proc. of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA13), Hano, Vietnam, volume 8172 of Lecture Notes in Computer Science, pp. 208–223, Springer, October 2013.

Multi-threaded Explicit State Space Exploration with State Reconstruction

Sami Evangelista[1] and Lars Michael Kristensen[2] and Laure Petrucci[1]

[1] LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité, 99, av. J.-B. Clément, 93430 Villetaneuse, France
[2] Department of Computing, Mathematics, and Physics, Bergen University College, Nygaardsgaten 112, Postbox 7030, 5020 Bergen, Norway

Abstract. This article introduces a parallel state space exploration algorithm for shared memory multi-core architectures using state compression and state reconstruction to reduce memory consumption. The algorithm proceeds in rounds each consisting of three phases: concurrent expansion of open states, concurrent reduction of potentially new states, and concurrent duplicate detection. An important feature of the algorithm is that it requires little inter-thread synchronisation making it highly scalable. This is confirmed by an experimental evaluation that demonstrates good speed up at a low overhead in workload and with little waiting time caused by synchronisation.

1 Introduction

We consider in this article the problem of explicitly constructing the state space of a system implicitly given through an initial state and a successor function that maps each state to a set of successor states. This is the core operation performed by explicit state model checkers in order to, e.g., verify safety properties and deadlock freedom, and conduct temporal logic model checking. The large number of states combined with the size of each state is a limiting factor for the practical use of standard explicit state space exploration. For complex systems, like software or communication protocols, it is not uncommon that the state vector (the data structure that unambiguously represents a state) consumes up to the order of 100 bytes. One way to overcome this problem is to store only a hash value for each state. This technique is known as *hash compaction* [15] which is an incomplete method in that parts of the state space may not be explored if two states have the same hash value. To guarantee full state space coverage in presence of hash compaction, techniques based on *state reconstruction* [6,14] have been proposed. These techniques reconstruct states from their compressed representation on-demand when comparison of states is required in order to determine whether a newly generated state has been already encountered, i.e. is a duplicate of an already explored state. Clearly, the reconstruction of states implies an increase in exploration time.

One approach [7] to reducing the exploration time in presence of state reconstruction is to delay the duplicate detection and thereby reduce the number of states that needs to be reconstructed. The contribution of this paper is an orthogonal approach in the form of an algorithm that reduces the exploration time by exploiting multiple threads (and multi-core architectures) to perform the reconstruction of states in parallel. In addition, our algorithm processes open states in parallel. The algorithm maintains a *state reconstruction tree* based on which all encountered states can be reconstructed and it proceeds breadth-first in rounds each consisting of three *phases*. In the first phase, the threads traverse the reconstruction tree in order to generate a *frontier set* consisting (in its basic form) of the next breadth-first layer of states. In the second phase, duplicate states in the frontier set are eliminated resulting in a *candidate set* of potentially new states. Finally, in the third phase, threads perform state reconstruction to determine which candidate states are new and such states are then added to the reconstruction tree.

The article is organised as follows. Sect. 2 introduces the basic notations and concepts of transition systems used in order to make our presentation independent of a particular modelling formalism. Sect. 3 gives a high-level overview of the operation of our algorithm by means of a small example, and Sect. 4 provides the formal algorithmic details. An implementation of our algorithm is presented in Sect. 5 together with the findings from an experimental evaluation. Finally, Sect. 6 concludes and discusses further related and future work. The reader is assumed to be familiar with the basic ideas of explicit state space exploration and associated model checking techniques.

2 Background

Let $\mathcal{S}$ be a universe of syntactic states and $\mathcal{E}$ a set of events. The system is given through an initial state $s_0 \in \mathcal{S}$, a mapping $enab : \mathcal{S} \to 2^{\mathcal{E}}$ associating with each state a set of enabled events, and a mapping $succ : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$ used to generate a successor state from a state and one of its enabled events. State space exploration is concerned with computing the set of states reachable from s_0, i.e. states s such that there exist $e_0, \ldots, e_{n-1} \in \mathcal{E}$, $s_1, \ldots, s_n \in \mathcal{S}$ with $s = s_n$ and, for all $i \in \{0, \ldots, n-1\}$: $e_i \in enab(s_i)$ and $succ(s_i, e_i) = s_{i+1}$. For simplicity, we use a function *succ* to obtain a successor state from a given state and an event. This implies that events are assumed to be deterministic in order to reconstruct a unique state from a sequence of events in the state reconstruction. Many modelling formalisms (including Petri nets) have deterministic transitions (events). As shown in [14], state reconstruction can be extended to handle non-deterministic events.

Algorithm 1(left) gives the basic algorithm for explicit state space exploration. It maintains a set $\mathcal{R}$ of reached states and a set O of currently *open states*. The algorithm iterates until there are no open states. In each iteration, an open state s is selected and *state expansion* is performed by exploring all events enabled in s. Successor states that have not been reached earlier are inserted into $\mathcal{R}$ and O.

State reconstruction. Earlier [6,14], we have proposed to implement the reachability set $\mathcal{R}$ as a hash table where full state vectors are not stored but each state is instead represented by an integer (hash value) identifying it. The hash table now represents an inverse spanning tree (a state reconstruction tree) rooted in the initial state, and where nodes have references to one parent, each labelled with an event used to generate the full state vector for the node. Figure 1 illustrates state reconstruction. The top of Fig. 1 shows the state space where the upper part of each node is the state vector, the bottom

Algorithm 1 A basic state space exploration algorithm (left) and a state space exploration algorithm based on delayed duplicate detection (right)

```
1: $\mathcal{R} := \{s_0\}$
2: $\mathcal{O} := \{s_0\}$
3: while $\mathcal{O} \neq \emptyset$ do
4:     pick $s$ in $\mathcal{O}$ ; $\mathcal{O} := \mathcal{O} \setminus \{s\}$
5:     for $e \in enab(s), s' = succ(s,e)$ do
6:         if $s' \notin \mathcal{R}$ then
7:             $\mathcal{R} := \mathcal{R} \cup \{s'\}$
8:             $\mathcal{O} := \mathcal{O} \cup \{s'\}$
```

```
1: $\mathcal{R} := \{s_0\}$ ; $\mathcal{O} := \{s_0\}$ ; $\mathcal{C} := \emptyset$
2: while $\mathcal{O} \neq \emptyset$ do
3:     pick $s$ in $\mathcal{O}$ ; $\mathcal{O} := \mathcal{O} \setminus \{s\}$
4:     for $e \in enab(s), s' = succ(s,e)$ do
5:         $\mathcal{C} := \mathcal{C} \cup \{s'\}$
6:     if $\mathcal{O} = \emptyset$ or $doDuplicateDetection()$ then
7:         $\mathcal{N} := \mathcal{C} \setminus \mathcal{R}$ ; $\mathcal{C} := \emptyset$
8:         $\mathcal{R} := \mathcal{R} \cup \mathcal{N}$ ; $\mathcal{O} := \mathcal{O} \cup \mathcal{N}$
```

part is its hash value, and the thick edges are edges represented by references in the spanning tree. The lower part of Fig. 1 is a linearised graphical representation of the hash table implementing $\mathcal{R}$ where dashed arcs represent references to parents in the reconstruction tree and are labelled by generating events. Note that state vectors appear in the table for the sake of clarity, but they are not stored in memory.

When required, the state vector for a node can be reconstructed by backtracking up to the root (initial) node for which we have the full state vector, and then forward execute the *reconstructing sequence* of events on the path leading from the initial node (state) to the node in question. This is performed each time the algorithm generates a successor state s' from an open state s. As an example, consider Fig. 1 and assume that the algorithm has explored states s_0 to s_3 and is expanding s_4 corresponding to the exploration of the two dotted edges. The expansion of s_4 generates s_2 and s_5 both hashed to 7. To decide whether s_2 is new, we reconstruct all nodes of the reconstruction tree that are also hashed to 7 as these could potentially have the same state vector. These correspond to the grey cells of the hash table. For the first cell (s_2), we have to follow references labelled b, a to the initial node (state) and finally execute the reconstruction sequence $a.b$ starting from the initial state. Since this execution produces state s_2, we conclude that executing e from s_4 does not generate a new state. For state s_5, we have to reconstruct s_2 using again $a.b$ as reconstructing sequence, and s_4 using the reconstructing sequence $a.c$. Since $succ(succ(s_0,a),b) \neq s_5$ and $succ(succ(s_0,a),c) \neq s_5$, then s_5 is new and is inserted in the hash table with a reference to its parent s_4 labelled with event f.

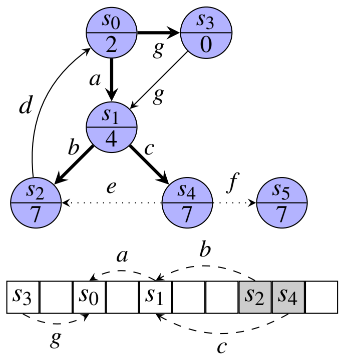


Fig. 1. State reconstruction

Delayed duplicate detection. Duplicate detection refers to the the process of checking if a generated state already belongs to the reachability set $\mathcal{R}$ (l. 6 of Alg.1 (left)) and is often the most expensive operation performed by the algorithm in particular in combination with state reconstruction. The principle of *delayed duplicate detection* is that the number of state reconstructions can be reduced if these checks were grouped and performed once. Duplicate detection is delayed because each generated state is not directly looked for in $\mathcal{R}$ but put in a *candidate set* $\mathcal{C}$, and only occasionally is this set compared to $\mathcal{R}$. Algorithm 1 (right) also performs state space exploration but relies on the principle of delayed duplicate detection. New successor states are put in the candidate set $\mathcal{C}$. When the open set is empty or if, for example, the candidate set reaches a specific threshold (l. 6), the algorithm will identify new candidate states (set $\mathcal{N}$) by comparing sets $\mathcal{C}$ and $\mathcal{R}$. The resulting set will then be inserted in $\mathcal{R}$ and $\mathcal{O}$ as the basic algorithm would have done for individual states. When applied in the context of state reconstruction, delayed duplicate detection allows to group the reconstruction of states and, hence, execute only once the common prefixes of reconstructing sequences [7] and also reduce the number of reconstructions performed. On our example of Fig. 1, the candidate set would contain, after the expansion of s_4, the states s_2 and s_5. Duplicate detection reconstructs s_2 and s_4 together and grouping the execution of reconstructing sequences $a.b$ and $a.c$ allow to execute event a once rather than twice and only reconstruct s_2 once.

3 Algorithm Overview and Example

The primary data structure maintained by our new parallel algorithm is a reconstruction tree as introduced in the previous section representing all encountered states. As a further extension compared to earlier work [6,7,14], we do not use a separate data structure to store open states. Instead, we use the reconstruction tree to also on-the-fly reconstruct open states during exploration which further reduces memory consumption.

Our algorithm proceeds breadth-first in *rounds* where each round explores the next breadth-first search (BFS) level. As depicted in Fig. 2, then each round consists of three *phases*. The first phase uses the current reconstruction tree $\mathcal{T}$ to construct lists of successor states for the currently open states. The second phase merges the lists of successor states to obtain a candidate set $\mathcal{C}$ of potentially new states. The third step performs duplicate detection to find new states, i.e. states in $\mathcal{C}$ that are not represented in $\mathcal{T}$. Within each phase, all threads cooperate and must synchronise before the algorithm proceeds to the next phase (and hence round). Within each phase, additional synchronisation barriers are employed and represent the only form of synchronisation used by our algorithm. The algorithm proceeds into the next round as long as duplicate detection results in states that are not yet represented by the reconstruction tree.

Below we illustrate the three phases of our algorithm in more detail starting from the partial state space shown in Fig. 3 where the initial state s_0 is assumed to have already been expanded, leading to four new successors s_1, s_2, s_3, and s_4, as shown in Fig. 3. States are stored in the reconstruction tree as a hash value with a reference to

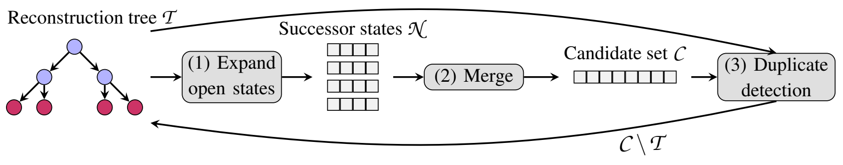

Reconstruction tree $\mathcal{T}$
Successor states $\mathcal{N}$
(1) Expand open states
(2) Merge
Candidate set $\mathcal{C}$
(3) Duplicate detection
$\mathcal{C} \setminus \mathcal{T}$

Fig. 2. A round exploring one BFS level in three phases

their parent node and the event that was used to generated the state from the parent state. We do not explicitly show this in the figures to improve readability. In the example, two states with the same hash value carry the same name, but are distinguished with primes, e.g. s_4 and s'_4 have different state vectors but are mapped to the same hash value. We denote by $\mathcal{W}$ the number of threads taking part in the exploration of the state space, and assume that each worker is identified by an integer in the range from 0 to $\mathcal{W}-1$.

Phase 1: Expand open states. Threads traverse the reconstruction tree using random depth-first search. Randomisation is used in order to break the symmetry between threads and ensure that they can work on different parts of the reconstruction tree. Open states (i.e. leafs at the bottom of the tree) are expanded and their successors put in a *successor lists* structure $\mathcal{N}$ consisting of $\mathcal{W}$ state lists (one per thread). Since each thread w inserts states in the list of the w^{th} slot only, no synchronisation is required. When inserting new states in $\mathcal{N}$, no duplicate detection is performed: each successor state of an open state is inserted at the end of the appropriate list of the structure $\mathcal{N}$ and a state may hence be present in different slots of the array. Considering the example in Fig. 3, then this first step of the algorithm expands the nodes s_1 to s_4 by generating their successors, as depicted by the dotted elements in Fig. 4(left). This phase ends when leaf states have all been expanded (i.e. the current BFS level has been explored) or when the size of structure $\mathcal{N}$ reaches a specific threshold. In the case of four threads, we obtain the successor lists shown in Fig. 4(right). In general, the expansion can be performed concurrently by any number of threads.

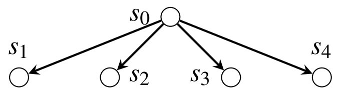


Fig. 3. Initial reconstruction tree

Phase 2: Merge. The second phase consists of merging the successor lists of $\mathcal{N}$ into a single set $\mathcal{C}$, hence removing the duplicate successors present in these lists. Merging the states present in the successor lists in Fig. 4(right) results in the candidate states: $s'_4, s_5, s_4, s_6, s_7, s'_3, s_2$. This second phase can be realised using a parallel sort and merge algorithm [9] or hashing on a matrix structure as in our implementation (see Sect. 5).

Phase 3: Duplicate detection. The last phase removes from $\mathcal{C}$ the states already represented in compressed form in the reconstruction tree. Phase 3 first performs bottom-up tagging (marking) of the nodes to be reconstructed (tagged **R**) and their ancestors (tagged **A**). Considering the example in Fig. 5(a) and the list of candidates from phase

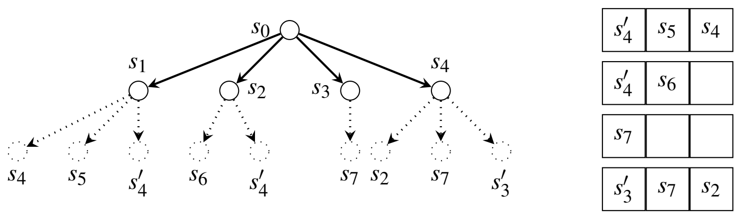
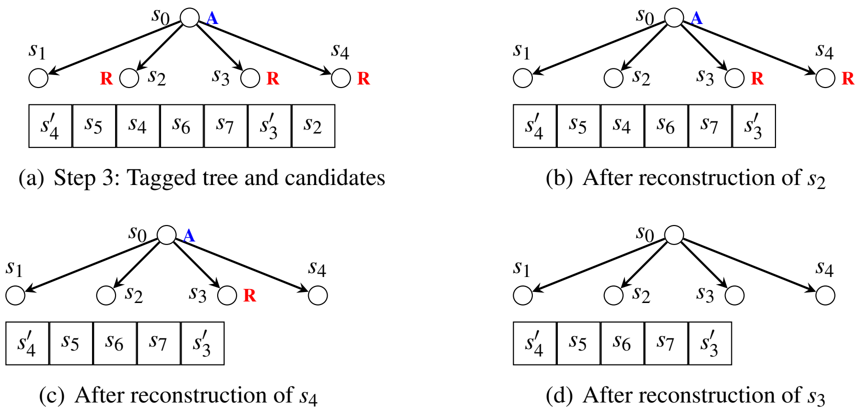


Fig. 4. Phase 1: Expansion of open states (left) and array of successor lists (right)



(a) Step 3: Tagged tree and candidates (b) After reconstruction of s_2 (c) After reconstruction of s_4 (d) After reconstruction of s_3

Fig. 5. Phase 3: Tagging following by state reconstruction and duplicate detection

2: since s_4' is a candidate, and has the same hash value as s_4, s_4 is marked **R** and its only ancestor, s_0 marked **A**. This is achieved using the references to parent nodes. There is no node in the reconstruction tree with the same hash value as state s_5. Then s_4 is a candidate, and it is already marked in the tree. As soon as a state is encountered already having an identical tag, then tagging stops, and the next candidate is processed. The resulting tagged reconstruction tree is depicted in Fig. 5(a).

Each of the nodes marked **R** must be reconstructed, and each node marked **A** is the ancestor of such a node. Starting from the initial state, the threads perform a random forward traversal of the tree via tagged nodes. Let us assume that from s_0, the branch of s_2 is explored. This latter state is reconstructed from s_0 and the event to s_2. The actual value of state s_2 is compared to the candidate states. Since it is found, s_2 already exists and can be removed from the candidates set. Moreover, the **R** tag is removed from s_2. The resulting tree and candidates set are shown in Fig. 5(b). Next, backtracking is performed since s_0 still has tagged successor states, so, e.g. the branch with s_4 may be explored. When s_4 is reconstructed, the reconstructing thread finds that the state is in the candidates set, so it is removed from it, but s_4' is kept since, even though it has the same hash value, the actual state is different. The result in shown in Fig. 5(c). Then the process continues, e.g. with state s_3. Finally, as shown in Fig. 5(d), state s_0 has no more tagged successors, and can be untagged, finishing the reconstruction phase. At the end of the exploration, states remaining in $\mathcal{C}$ are guaranteed to be new and can thus be inserted in the reconstruction tree. The resulting reconstruction tree for the example is shown in Fig. 6. A new round is now initiated on the updated reconstruction tree. To maintain a strict breadth-first search order, the expansion of the new states will occur only when all states of the current BFS level have been expanded. This constraint can be relaxed, but we have chosen to maintain this search order as it guarantees the minimality of reconstruction sequence lengths.

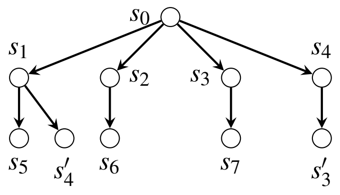


Fig. 6. Updated state tree

4 Algorithmic Details

Reconstruction tree $\mathcal{T}$. The central data structure is the reconstruction tree. For specification of the algorithm, we assume that its nodes are identifiers of some set *ID* (e.g. set of integers) and that from one node identifier its children and its parent in the tree can be obtained using: $parent(id)$ that maps a node identifier *id* to the identifier of its parent (or to $\perp$ if the node is the root); and $children(id)$ that maps a node identifier *id* to the set of pairs (id', e) such that $parent(id') = id$ and the arc from node *id* to id' is labelled with the event e. The only operation that modifies the tree structure is *newNode*. $newNode(id, e)$ inserts a new node and creates a reference from the new node to the node *id* labelled by event e. $newNode(\perp)$ inserts a root in the tree. In both cases, the operation returns the identifier of the new node that is not labelled by any tag.

Node tagging. As discussed earlier, nodes are labelled by a set of tags used to prune the traversal of the reconstruction tree. Three operations are used to manipulate tags: *tagged* (checks whether a node has a specific tag), *tag* (sets a tag on a node), and *untag* (removes a tag from a node). Our example from the previous section introduced two tags (**R** and **A**) used during duplicate detection (phase 3). Two additional tags ($\mathbf{E}_0$ and $\mathbf{E}_1$) are used during the expansion phase (phase 1) to tag nodes to expand. The $\mathbf{E}_0$ and $\mathbf{E}_1$ are used in an alternating manner between rounds such that in odd numbered rounds $\mathbf{E}_0$ marks nodes to be expanded and $\mathbf{E}_1$ marks nodes to be expanded in the next (even numbered) round. In an odd numbered round, a node has the $\mathbf{E}_0$ tag if it has children with this same tag or if it corresponds to an open state (in which case it can not have any children yet). Furthermore, the expansion of open states will then create new nodes in the tree having the $\mathbf{E}_1$ tag and also tag nodes on the way up from these new nodes to the root node with $\mathbf{E}_1$. Traversing nodes with the $\mathbf{E}_0$ (resp. $\mathbf{E}_1$) thus leads to open states of the current (next) expansion phase. When a round is completed $\mathbf{E}_0$ and $\mathbf{E}_1$ swap roles. One **E** tag is not sufficient because nodes can be independently marked for expansion in the current and in the next phase. Note that a node may be simultaneously labelled by several tags.

Successor lists $\mathcal{N}$ and candidate set $\mathcal{C}$. The elements of $\mathcal{N}$ and $\mathcal{C}$ are triples of the form $\mathcal{S} \times ID \times \mathcal{E}$. A triple (s, id, e) represents that state s has been reached from the node *id* by executing event e on the state corresponding to *id*. The first item is used during duplicate detection while the second and third items are used in case the candidate state is actually new and must be inserted in the tree using the *newNode* operation. After merging elements of $\mathcal{N}$ in $\mathcal{C}$ (phase 2 of Fig. 2) there cannot be two elements (s, id, e) in (s', id', e') in $\mathcal{C}$ with $s = s'$.

In addition to the data structures described above, three shared data structures are required by the algorithm. The array *done* contains $\mathcal{W}$ booleans indicating which threads have finished their exploration of the current BFS level. It is used to decide when the current BFS level has been completely expanded and threads can move to the next level. An alternating bit r initialised to 0 identifies which tag among $\mathbf{E}_0$ and $\mathbf{E}_1$ is used to indicate nodes that must be explored during the current expansion phase.

Algorithms 2, 3 and 4 contain the pseudo-code of our algorithm. The w subscript of the procedures identify the working thread executing the procedure. Apart from the

Algorithm 2 *ParReconstruction*, Initialisation and main worker procedure

```
1: algorithm ParReconstruction is
2:     $done$ := [$false,\ldots,false$] ; $r$ := 0
3:     $id_0$ := $newNode(\bot)$ ; $tag(id_0, \mathbf{E}_0)$
4:     spawn $worker_0()$ || ... || $worker_{\mathcal{W}-1}()$
5: procedure $worker_w()$ is
6:     while $tagged(id_0, \mathbf{E}_r)$ do
7:         $expandLevel_w()$
8:         if $w = 0$ then $r := \neg r$
9:         $\underline{barrier}()$
```

Shared Data	
$\mathcal{T}$	the state tree initially empty
$\mathcal{C}$	the candidate set initially empty
$\mathcal{N}$	the successor lists initially empty
done	array of $\mathcal{W}$ booleans
r	bit
id_0	identifier of the initial state in $\mathcal{T}$

shared data specified in Alg. 2, all other variables are thread local. We have underlined the different barriers that must be executed simultaneously by the $\mathcal{W}$ threads so as to highlight synchronisation points. The variable id_0 is used to store the root node. Thread 0 is the only thread that modifies the values of r and id_0.

Initialisation and main worker procedure (Alg. 2). The main procedure *ParReconstruction* inserts the initial state in the reconstruction tree and records its node identifier in the shared variable id_0. This node is then tagged for expansion and the procedure spawns $\mathcal{W}$ instances of the main worker procedure $worker_w$. The loop at l. 6 iterates over all BFS levels expanded using the $expandLevel_w$ procedure introduced below. We use the $\mathbf{E}_r$ tag to decide when a thread can terminate. For any node, the tagging procedure guarantees that if a node is tagged with $\mathbf{E}_r$, then so is its parent in the tree. Thus, an untagged root guarantees that no state is tagged for expansion. The barrier at l. 9 is related to duplicate detection as will be discussed shortly. It is the responsibility of thread 0 to swap the r bit before all threads can start processing the next BFS level.

Phase 1: State expansion (Alg. 3). The expansion procedure $expandLevel_w$ expands open states of a BFS level. The working thread w first launches a DFS exploration from the root of the tree using procedure $dfsExpand_w$ that is parameterised by the identifier of the visited node in the tree (id) and the corresponding full state vector (s). In case the visited node is a leaf (ll. 12–15), the worker puts all its successors in list $\mathcal{N}[w]$ and enters the duplicate detection phase if this structure reaches a specified threshold. Otherwise (ll. 16–19), the thread picks all children of the node one by one in the random order obtained using the *random* function, and explores those that have the $\mathbf{E}_r$ tag set as these nodes may lead to open states. To maintain the correspondence between nodes and states, procedure $dfsExpand_w$ is recursively called (l. 19) with the identifier of the child id' and the state obtained by executing on s the event e labelling the arc from id to id', i.e. the state s' such that $succ(s,e) = s'$. When node id has been processed, the $\mathbf{E}_r$ tag can be removed from it to signal to other threads that this node has been processed.

After the threads have finished exploring the tree, a last duplicate detection is required since successor lists might not be empty (l. 6). Moreover, after exploration, a thread also has to be ready to perform duplicate detection as long as some workers have not yet finished their exploration. These threads may still be feeding successor lists and hence call the duplicate detection procedure (l. 15) that must be executed by all threads. This is where the shared array *done* is used. A thread will keep performing duplicate

Algorithm 3 Procedures used during the state expansion phase

```
1: procedure $expandLevel_w()$ is
2:     $done[w]$ := false
3:     $dfsExpand_w(id_0, s_0)$
4:     $done[w]$ := true
5:     do
6:         $allDone := duplicateDetection_w()$
7:     while $\neg allDone$
8: procedure $expandState_w(id, s)$ is
9:     for $e \in enab(s)$ do
10:        $append(\mathcal{N}[w], (id, succ(s,e), e))$
11: procedure $dfsExpand_w(id, s)$ is
12:    if $children(id) = \emptyset$ then
13:        $expandState_w(id, s)$
14:        if $|\sum_x \mathcal{N}[x]| >$ MemoryLimit then
15:            $duplicateDetection_w()$
16:    else
17:        for $(id', e) \in random(children(id))$ do
18:            if $tagged(id', \mathbf{E}_r)$ then
19:                $dfsExpand_w(id', succ(s,e))$
20:    $untag(id, \mathbf{E}_r)$
```

detection until all threads have finished their exploration and executed the assignment at l. 4. Procedure *duplicateDetection*$_w$ introduced below returns a boolean value specifying if all threads have finished expanding the current level. This check is performed between two barriers (l. 3 of Alg. 4) in a block of statements that does not modify the content of array *done* to ensure that its outcome will be the same for all threads.

Phases 2 and 3: Merge and duplicate detection (Alg. 4) Procedure *duplicateDetection*$_w$ corresponds to phases 2 and 3 and can be decomposed into four sub-steps. The entry in each sub-step is protected by a barrier. A thread first awaits all its peers to have called the procedure and be waiting at the barrier at l. 2. The parallel merge of successor lists $\mathcal{N}$ in the candidate set $\mathcal{C}$ (Phase 2 of Fig. 2) can then take place (l. 2).

Before exploring the reconstruction tree to remove duplicate states from $\mathcal{C}$, all threads first start to mark (with **R** and **A** tags) the appropriate branches using procedure *tagNodesforDD*$_w$ (l. 4) in order to avoid reconstructing all states. For efficiency reasons, we assume that the data structure implementing the candidate set can be partitioned in $\mathcal{W}$ classes and that a thread can recover the class it is responsible for using the function *ownedCandidates* (l. 17). For a candidate state c, the nodes that must be reconstructed are all nodes that have the same hash value as c (ll. 18–21) because these might, after reconstruction, match with c. It is the purpose of the *conflict* function used at l. 19 to return the identifiers of these nodes. The *tagPath*$_w$ procedure is used to put a specific tag on a node and all its ancestors. It stops as soon as it reaches the root or a node that already has this tag (in which case all its ancestors also have it).

The reconstruction begins when all threads have finished tagging the branches that need to be explored. The exploration procedure *dfsDD*$_w$ used to reconstruct states follows the same pattern as procedure *dfsExpand*$_w$ of Alg. 3. Reconstructed states removed from the candidate set $\mathcal{C}$ are those with the **R** tag (ll. 23–25) and nodes explored are those with the **A** tag (ll. 26–31). In both cases, a processed node is untagged.

All threads must have finished their exploration in order to decide which candidate states are actually new. Thread 0 is then responsible (l. 6) for inserting the new states (those that are still present in $\mathcal{C}$) in the tree using the procedure *insertNodes*$_w$. Only the last two components of the candidates are required for insertion: the identifier (*parentId*) of the node of which the expansion generated the candidate and the event used to generate it. A new node is then inserted in the tree and the tag $\mathbf{E}_{\neg r}$ is put on all nodes on the path from the initial node to this new node to signify that this node must be

Algorithm 4 Procedures used during merge and duplicate detection phases

```
1:  procedure $duplicateDetection_w()$ is
2:      $\underline{barrier}()$; $parallelMerge(\mathcal{N}, \mathcal{C})$
3:      $allDone := \wedge_{x \in \{0,\dots,\mathcal{W}-1\}} done[x]$
4:      $\underline{barrier}()$; $tagNodesForDD_w()$
5:      $\underline{barrier}()$; $dfsDD_w(s_0, id_0)$
6:      $\underline{barrier}()$; if $w = 0$ then $insertNodes_w()$
7:      return $allDone$
8:  procedure $tagPath_w(id, tag)$ is
9:      while $id \neq \perp \wedge \neg tagged(id, tag)$ do
10:         $tag(id, tag)$
11:         $id := parent(id)$
12: procedure $insertNodes_w()$ is
13:     for $(\_, parentId, e) \in \mathcal{C}$ do
14:         $id := newNode(parentId, e)$
15:         $tagPath_w(id, \mathbf{E}_{\neg r})$
16: procedure $tagNodesForDD_w()$ is
17:     for $(s, \_, \_) \in ownedCandidates(w)$ do
18:         $h := hash(s)$
19:         for $id \in conflict(h)$ do
20:             $tag(id, \mathbf{R})$
21:             $tagPath_w(parent(id), \mathbf{A})$
22: procedure $dfsDD_w(id, s)$ is
23:     if $tagged(id, \mathbf{R})$ then
24:         $removeCandidate(s)$
25:         $untag(id, \mathbf{R})$
26:     if $tagged(id, \mathbf{A})$ then
27:         for $(id', e) \in random(children(id))$ do
28:             if $tagged(id', \mathbf{A})$
29:               or $tagged(id', \mathbf{R})$ then
30:                 $dfsDD_w(id', succ(s, e))$
31:         $untag(id, \mathbf{A})$
```

expanded. This insertion step is performed only by thread 0 because the data structure we have chosen for the reconstruction tree does not easily support concurrent insertions although it allows for multiple concurrent read accesses (or node tagging/untagging). Therefore, other threads may proceed to the next expansion step as thread 0 inserts new nodes in the tree. As an extension, this insertion step could also be parallelised. The only situation where other threads have to wait for thread 0 to finish this insertion is when a BFS level has been completely processed, i.e. duplicate detection and insertion was not triggered by a threshold in l. 13 of procedure *dfsExpand*. They will then be waiting at the barrier of procedure $expandLevel_w$ (l. 7 of Alg. 3).

5 Implementation and Experimental Evaluation

We have integrated our algorithm in the Helena tool [4]. We discuss below the most important implementation aspects and present the results from an experimental evaluation of our algorithm based on the Helena implementation.

Implementation. The implementation uses the pthread library that provides synchronisation barriers. The reconstruction tree $\mathcal{T}$ is implemented as a fixed size hash table using open addressing with linear probing. This allows to support multiple read accesses with a single insertion as performed when thread 0 inserts new nodes in the reconstruction tree while other threads continue their expansion of open states. A main requirement is the possibility to get the parent and children of a node. To reduce the number of pointers and save memory, we represent kinships as linked lists where the parent node stores the identifier (i.e. the slot of the hash table) of its first child in field *fstChild* and each child stores the identifier of the next child in field *next*. Only the last child of the list (identified using a *last* bit set to 1 while the previous children have it set to 0) points with *next* to the parent node. Fig. 7 provides an illustration of this for an example tree. To enumerate the children of a node, we first follow its *fstChild* pointer and then the *next* pointers of its children until the last child with $last = 1$ is met. Recovering the parent of a node is done by following the *next* pointers until the last child is met and then by following the *next* pointer to reach the parent. This means that we cannot get the parent of a node in constant time but this is not a practical problem as the number of children is usually low.

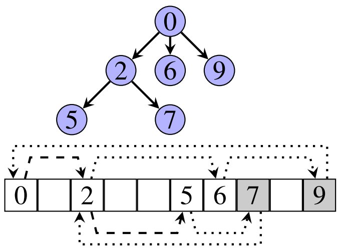


Fig. 7. Implementation of the reconstruction tree. Dashed lines represent the *fstChild* pointers. Dotted lines represent the *next* pointers. Gray cells identify nodes with $last = 1$.

This representation requires $2 \cdot \log_2(|ID|) + 1$ bits per node for the *fstChild* and *next* pointers and the *last* bit. Four more bits are required for tags (**R**, **A**, $\mathbf{E}_0$ and $\mathbf{E}_1$) and $\log_2(|\mathcal{E}|)$ bits for the event generating the node from its parent. Hence, a node can be encoded in $2 \cdot \log_2(|ID|) + 5 + \log_2(|\mathcal{E}|)$ bits. To have a representation that is model independent, we encode in each node an event number (the number in the list of enabled events of the parent) rather than the event itself. This requires to recompute enabled events when exploring the tree but leads to significant savings for models such as high-level Petri nets, where events are often complex data structures. We also implemented the fresh successor heuristic [10] that, in our context, forces threads to engage in part of the state tree that no thread is currently visiting. Implementing this heuristic requires two bits per node (one for the expansion step and one for the duplicate detection step) to tag branches where threads are currently engaged.

To avoid concurrent accesses in the successor lists of $\mathcal{N}$, we implemented this data structure as a matrix of size $|\mathcal{W} \times \mathcal{W}|$ where each cell stores a list and using a hash function *hash* on states. During the expansion step, a worker w inserts any new successor state s in the list of cell $(w, hash(s) \mod \mathcal{W})$. During the merging step performed to merge states of successor lists $\mathcal{N}$ into table $\mathcal{C}$, a worker thread $w \in \{0, \ldots, \mathcal{W} - 1\}$ is responsible for moving from $\mathcal{N}$ to $\mathcal{C}$ all states s with $hash(s) \mod \mathcal{W} = w$. Thus, during the merging step, it only needs to merge states contained in the list of cells (x, w) for $x \in \{0, \ldots, \mathcal{W} - 1\}$ and because of the use of the hash function no such states can be in a hash conflict with states processed by other threads. Since the expansion and merging steps cannot overlap due to the use of barriers, there cannot be any concurrent insertions or deletions on the lists in $\mathcal{N}$. The candidate set $\mathcal{C}$ is also implemented as a fixed size hash table, and a state can only be inserted (if not already present) in a slot l of $\mathcal{C}$ such that $l \mod \mathcal{W} = w$. Hence, there cannot be concurrent accesses on a same slot of $\mathcal{C}$ during the merging step. During the duplicate detection step occurring right after, the deletion of a state is simply made by swapping a bit in the slot of the deleted state. These choices imply that $\mathcal{C}$ can be implemented without locks.

Experimental setup and results. We have conducted our experiments on the models of Table 1. The *Time* reported (in seconds) are those obtained with a sequential algorithm, i.e., with a single worker. Helena has its own modelling language for high-level Petri nets and can also analyse automata written in the DVE language, the input lan-

Table 1. Models used for experimental evaluation

Helena models			
Model	States	Arcs	Time
eratosthene	195.3 M	1.252 G	9,689
leader	188.9 M	2.530 G	24,086
neo-election	406.1 M	3.796 G	40,943
peterson	172.1 M	860.7 M	5,105
slotted	189.1 M	1.742 G	12,018

DVE models			
Model	States	Arcs	Time
collision.5	431.9 M	1.644 G	7,570
firewire-link.3	425.3 M	1.621 G	8,782
iprotocol.8	447.5 M	1.501 G	7,505
pub-sub.5	1.153 G	5.447 G	49,395
synapse.9	1.675 G	3.291 G	64,842

guage of the DiVinE model checker [1]. We have selected a set of 10 models having a set of reachable states ranging from 172 millions (M) to 1.675 billions (G) of states. The memory limit, measured as the maximal number of state vectors the algorithm can keep in memory (in the successor lists and, hence, in the candidate set) was set, for all runs, to one thousandth of the reachable states of the model. We performed our experiments on a 12-core computer with 64 GB of RAM and evaluated our algorithm on each model with 1 to 12 worker threads. Note that, when using a single thread, our algorithm becomes identical to the sequential algorithm of [7] that uses state compression, state reconstruction, and delayed duplicate detection, except for some minor differences on the data structures used, and the implicit representation of the open set.

Our experimental results have been plotted in Fig 8. On the horizontal axis of the three plots is the number of working threads used ranging from 1 to 12. The top plot, entitled *Speed-up*, gives execution times as the ratio of the execution time for 1 thread over the execution time for *n* threads. The middle plot, entitled *Event execution*, gives the total number of events executed (at any step of the algorithm and by any thread) for *n* threads relatively to the same number for 1 thread. This measure provides a means to evaluate how good the work is balanced among threads. The bottom plot, *Barrier time*, gives, as a percentage of the total execution time, the time spent by threads at barriers waiting for other threads to join them. It is computed as the average over all threads of the individual barrier times and is reported in percentage of the overall execution time.

Interpretation of results. The *Speed-up* plot shows a stable speed-up as the number of threads involved increases. As a general trend, Helena models (see Table 1(left)) have better speed-ups than DVE models (see Table 1(right)) considering that Helena models are penalised by a larger redundant work factor (see the *Event execution* plot). We conjecture that this is due to the cost of model operations (computing enabled events, executing events) that are much more costly for high-level Petri nets than for DVE models. Since these operations are purely local and do not need to access shared data structures (the reconstruction tree or the candidate set) they can be more efficiently parallelised. Also note that the algorithm is resilient with respect to the random function we used. We do not provide these results here due to lack of space, but we observed that performance never significantly differed between two runs on the same model.

As expected, the workloads shown by the *Event execution* plot increase with the number of threads but usually following a logarithmic progression. For the four models standing out with a larger amount of redundant work (eratosthene, leader, neo-election and slotted), we see a correlation with the high proportion of arcs of the state space with an average number of arcs per state ranging from 6 up to 13 (see Table 1); and with their state spaces that have diamond like structures. Due to the way nodes are inserted in the reconstruction tree, this automatically increases the proportion of nodes that have no (or few) children nodes in the tree which in turn decreases the potential parallelism. For instance, if two states at the same BFS level have the same successors states, the first state visited among the two will have some successors in the tree whereas the second one will not have any. This situation often occurred for the models mentioned. For these four models we observed that once the exploration completed, the proportion of leaves in the tree reached 65%–70%. For other models, this proportion is around 40%–50%.

The *Barrier time* plot shows that the waiting times remain low with an average (over all runs involving more than 1 thread) of less than 2% of the total exploration time. In the worst case, this time represented around 2.5% of the exploration time (for model synapse.9). Moreover, unlike for event executions, there is no real correlation between the number of working threads and waiting times observed and increasing the number of threads does not seem to have any negative effects in that respect.

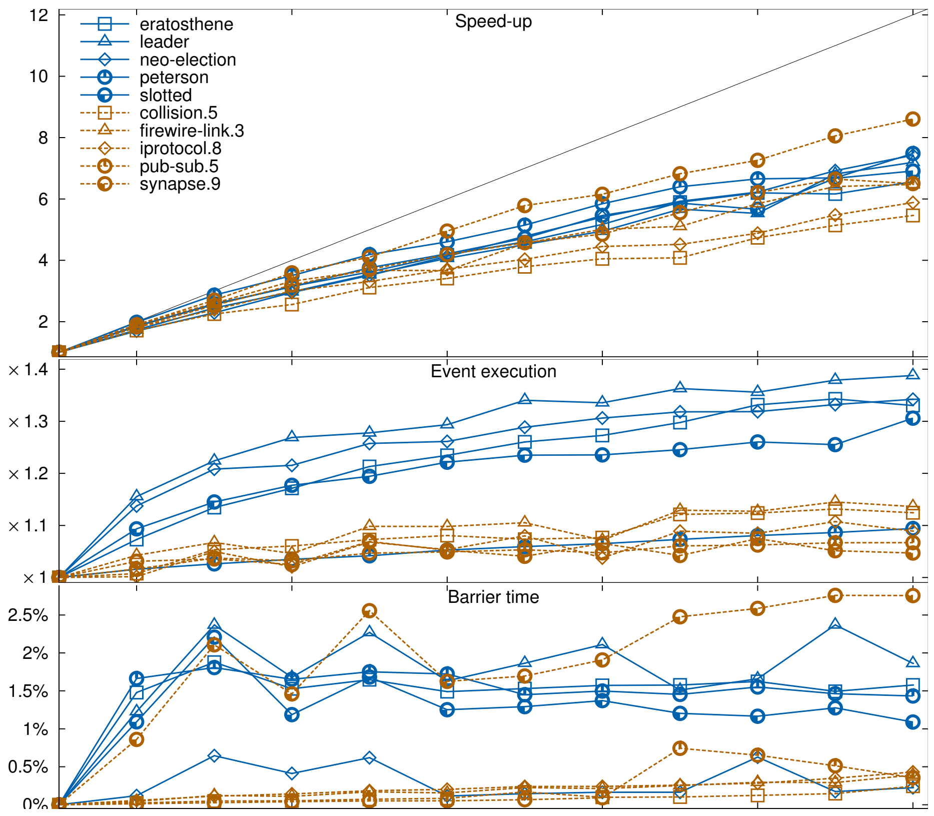


Fig. 8. Experimental results

6 Conclusion and Perspectives

This paper is the logical continuation of previous work based on the principle of state reconstruction that can significantly reduce the memory usage in state space exploration by avoiding the storage of full state vectors of states while maintaining a full coverage of the state space. The foundations of this method have been established in [6,14] and then extended in [7] with the principle of delayed duplicate detection that allows grouping state reconstructions and, hence, saves the redundant executions of shared reconstructing sequences. We also conjectured in [7] that parallelisation could further reduce the cost of duplicate detection. Following this intuition, we developed in this paper an algorithm designed for multi-core processors that preserves previous characteristics in terms of state reconstruction [6,14] and delayed duplicate detection [7].

A main feature of our algorithm is that locks can be avoided by the use of synchronisation barriers separating the different steps of the algorithm. A key property of the algorithm is also that the size of the frontier set can be bounded by a predefined threshold making the memory consumption predictable. A series of experiments done with Helena on a 12-core computer has shown good speed-up with negligible waiting times and a low amount of redundant work (threads that simultaneously engage in the same branch of the tree). The low barrier times show that synchronisation represents a very small overhead which leaves little room for further improvement in that respect. Our observations rather lead us to pursue two directions to further improve the speed-up of our algorithm. First, despite its low memory usage, the data structure we have chosen for the reconstruction tree does not show enough locality. Since the children of a node can be anywhere in the state table due to the hashing mechanism, traversing the tree requires accessing multiple memory areas which in turn means frequent cache misses. It is then relevant to design and experiment with a different data structure that takes better advantage of caching. Second, redundant explorations are still problematic for some models even if the fresh successor heuristic turned out to be quite efficient in that it allowed to reduce the overall workload by a factor of 10–20%. Besides the use of appropriate heuristics, it is also important to address the unbalanced distribution of child nodes (i.e. situations where two states have similar successor states but only one has children in the state tree) that we observed for a few models.

Related Works Several data structures have been designed for multi-core model checking or reachability analysis: [11,12,13]. All have in common to avoid the use of locks. The approach that seems the closest to ours is the tree database proposed in [12] as it is designed for high scalability while making use of state compression. Speed-ups reported in [12] are clearly better with this tree database structure: the average speed-up on all models of the BEEM database is almost optimal. Nevertheless our algorithm still has some advantages over [12]. First, its memory usage is model independent which is not true for the tree database although, on the average, compression ratios are excellent for BEEM models (around 8 bytes per state [12]). Second, the algorithm of [12] assumes fixed width state vectors, an assumption that does not hold for specification languages such as high-level Petri nets. Last, the support of a delete operation does not seem straightforward in the tree database and hence it is not obvious how to combine this with reduction techniques based on on-the-fly state deletions.

Perspectives To further assess the scalability of our algorithm a direct practical perspective is to experiment with it on larger models and massively on parallel architectures (e.g. 256-core machines).

The reconstruction tree can be used to check basic properties such as system invariants or to perform offline LTL or CTL model checking. It is relevant to study how our algorithm could serve as a basis for the implementation of on-the-fly LTL algorithms that are compliant with the breadth-first search order, e.g. [3,8]. A last perspective is to study how our algorithm combines with other reduction techniques. The state caching reduction we proposed in [5] maintains a termination detection tree (TD-tree) to keep track of open states. Termination is guaranteed if all states of the TD-tree are kept in memory and these consists of all nodes tagged with $\mathbf{E}_0$ or $\mathbf{E}_1$ in our parallel algorithm. Hence, all other nodes can be safely discarded. For partial order reduction, a breadth-first search compatible solution has been proposed in [2]. For both reductions [2,5] it remains to be investigated how they can be efficiently combined with our algorithm.

References

1. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker. In *HiBi/PDMC'10*, pages 4–7. IEEE, 2010.
2. D. Bosnacki and G.J. Holzmann. Improving Spin's Partial-Order Reduction for Breadth-First Search. In *SPIN'05*, vol. 3639 of *LNCS*, pages 91–105. Springer, 2005.
3. L. Brim, I. Cerná, P. Moravec, and J. Simsa. Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In *FMCAD'04*, vol. 3312 of *LNCS*, pages 352–366. Springer, 2004.
4. S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'05*, vol. 3536 of *LNCS*, pages 455–464. Springer, 2005.
5. S. Evangelista and L.M. Kristensen. Search-Order Independent State Caching. *ToPNoC Journal*, 6550(4):21–41, 2010.
6. S. Evangelista and J.-F. Pradat-Peyre. Memory Efficient State Space Storage in Explicit Software Model Checking. In *SPIN'05*, vol. 3639 of *LNCS*, pages 43–57. Springer, 2005.
7. S. Evangelista, M. Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *ToPNoC Journal*, 5800(3):189–215, 2009.
8. G. J. Holzmann. Parallelizing the Spin Model Checker. In *SPIN'2012*, vol. 7385 of *LNCS*, pages 155–171. Springer, 2012.
9. J. Jaja. *Parallel Algorithms*. Addisson-Wesley, 2002.
10. A. Laarman and J. van de Pol. Variations on Multi-Core Nested Depth-First Search. In *PDMC'11*, pages 13–28, 2011. `http://arxiv.org/abs/1111.0064v1`.
11. A. Laarman, J. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *FMCAD'10*, pages 247–255. IEEE, 2010.
12. A. Laarman, J. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN'11*, vol. 6823 of *LNCS*, pages 38–56. Springer, 2011.
13. R. T. Saad, S. Dal-Zilio, and B. Berthomieu. Mixed Shared-Distributed Hash Tables Approaches for Parallel State Space Construction. In *ISPDC*, pages 9–16. IEEE, 2011.
14. M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. The Comback Method - Extending Hash Compaction with Backtracking. In *ATPN'07*, vol. 4546 of *LNCS*, pages 445–464. Springer, 2007.
15. P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *CAV'93*, vol. 697 of *LNCS*, pages 59–70. Springer, 1993.