Science of Computer Programming 78 (2013) 778–795

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Dynamic state space partitioning for external memory state space exploration

Sami Evangelista[a,*], Lars Michael Kristensen[b]

[a] LIPN, CNRS UMR 7030, Université Paris XIII, France
[b] Department of Computer Engineering, Bergen University College, Norway

ARTICLE INFO

Article history:
Received 31 January 2011
Received in revised form 15 July 2011
Accepted 12 September 2011
Available online 1 October 2011

Keywords:
Model checking
Explicit state space exploration
External memory verification
Distributed verification
State space partitioning
State explosion problem

ABSTRACT

We describe a dynamic partitioning scheme useable by explicit state space exploration techniques that divide the state space into partitions, such as many external memory and distributed model checking algorithms. The goal of the scheme is to reduce the number of transitions that link states belonging to different partitions, and thereby limit the amount of disk access and network communication. We report on several experiments made with our verification platform ASAP that implements the dynamic partitioning scheme proposed in this paper. The experiments demonstrate the importance of exploiting locality to reduce cross transitions and IO operations, and using informed heuristics when choosing components to be used as a basis for partition refinement. The experiments furthermore show that the proposed approach improves on earlier techniques and significantly outperforms these when provided with access to the same amount of bounded RAM.

1. Introduction

Model checking [1] based on state space exploration is a prominent approach used to prove that finite-state systems match behavioural specifications. In its most basic form, it is based on a systematic exploration of all reachable states in the search for illegal behaviours violating the specification. Despite its simplicity, its practical application is subject to the well-known state explosion problem [2]: the state space may be far too large to be explored in reasonable time or to fit within the available memory. Most techniques devised to alleviate the state explosion problem can be classified as belonging to one of three families. The first family of techniques makes more economical use of available resources by reducing the part of the state space that needs to be explored in such a way that all properties of interest are preserved. Partial order reduction [3] which limits redundant interleavings is an example of such a technique. The second family of techniques aims at providing a compact representation of states. State compression [4] belongs to this family of techniques. The third family of techniques relies on increasing the available computing resources in order to extend the range of problems that can be analysed. Distributed verification [5–7] and external memory (disk-based) verification [8] belong to this second family of techniques.

In the field of external memory and distributed verification, it is common to divide the state space into *partitions* (although some external and distributed algorithms do not rely on such a partitioning, e.g., [8,9]). Partitioning in external memory verification [10] can be used to store the set of currently visited states (and unprocessed states) in a set of files (e.g., one file for each partition). A single partition is then loaded into memory at a time and when no more processing is possible for the currently loaded partition it is moved to disk and another partition is loaded into memory for processing. Another example

[*] Corresponding author.
E-mail addresses: sami.evangelista@lipn.univ-paris13.fr (S. Evangelista), lmkr@hib.no (L.M. Kristensen).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.09.005

of applying partitioning is distributed algorithms [5–7], where each process involved in the verification is responsible for storing and visiting all the states of a partition. Whenever a process generates a state that does not belong to the partition it is responsible for, it sends it to its owner such that the state can be stored and its successor states can be explored. An important component of such algorithms is the *partition function* which is used to map states to partitions. In the ideal case, the partition function should have two properties. First, it should generate as few *cross transitions* as possible. The reason for this is that cross transitions link two states of different partitions and thus systematically generate messages over the network (in a distributed setting) and induce partition swaps and IO operations (in an external memory setting). Second, it should distribute states evenly into partitions to ensure that all processes have the same workload. A hash function based on the bit string of the *state vector* used to represent states may achieve an optimal distribution, but generates many cross transitions due to the insensitivity of hashing to locality.

The contribution of this paper is to address the key issues related to state space partitioning outlined above, by introducing a dynamic partitioning scheme based on the idea of *partition refinement*. Initially, there is a single partition in which the partition function maps all states. Then, whenever a partition has to be split up – for instance because its size exceeds memory capacity – it is divided into sub-partitions and the partition function is refined accordingly. To represent a partition function that can change as the exploration of the state space proceeds, we introduce the concept of *compositional partition functions*. Refinement is done by progressively considering new components of the state vector (descriptor) in the partition function, e.g., variables or communication channels. For instance, after a first refinement step, a state will be mapped to one of the partitions $p_1, \ldots, p_n$ depending only on the value of its ith component in the state vector. Then, if p_1 has to be refined, we consider an additional component of the state vector. As refinement is applied on a single partition at a time, partitions $p_2, \ldots, p_n$ will remain unchanged.

The idea is to take advantage of the fact that events typically only have local effect which means that they modify only a small number of components in the state vector [11]. Thus, if a partition function is based only on a few components of the system and does not consider others, events that do not modify these components will not generate cross transition, and hence disk accesses or network communications will be limited. However, we replace the objective of a uniform distribution of states into partitions by a less ambitious one: partitions may be of different sizes, but we can ensure an upper bound on their size. Even though an uneven distribution does not have direct consequence with an external memory algorithm, it may impact a distributed algorithm in that processes may not receive the same amount of workload.

The refinement algorithm has been implemented in the ASAP model checking platform [12], on top of the external algorithm of [10]. We report the results of several experiments showing that we were able to significantly decrease the number of disk accesses. More importantly, our algorithm improves the algorithm of [10] such that it performs well on classes of models where it previously performed poorly.

Structure of the paper. In the next section, we briefly recall the principle of the two partitioning based algorithms of [5,10] that will be the basis of our work. Section 3 presents related work. Our dynamic scheme based on partition refinement is introduced in Section 4 followed in Section 5 by different heuristics to support the refinement. The experiments conducted with our verification tool are presented in Section 6. Finally, Section 7 concludes this paper and presents some perspectives for future works. We assume the reader is familiar with the basic principles of explicit state space exploration [2]. This paper is a revised version of the conference paper [13] where we significantly expanded and revised the experimental evaluation of the dynamic partitioning scheme.

2. Partitioning the state space

For the presentation of the partitioning scheme, we assume 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 $succ : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$. We want to explore the state space implied by these parameters, i.e., the triple (R, T, s_0) such that $R \subseteq \mathcal{S}$ is the set of *reachable states* and $T \subseteq R \times R$ is the set of *transitions* defined by

$$R = \{s_0\} \cup \{s \in \mathcal{S} \mid \exists s_1, \ldots, s_n \in \mathcal{S} \text{ with } s = s_n \wedge \forall i \in \{0, \ldots, n-1\} : \exists e_i \in en(s_i) \text{ with } succ(s_i, e_i) = s_{i+1}\}$$
$$T = \{(s, s') \in R \times R \mid \exists e \in en(s) \text{ with } succ(s, e) = s'\}$$

Our partitioning scheme assumes that each system state s can be represented as a *state vector* $s = (s_1, s_2, \ldots, s_n)$ where s_i represents the state of the ith system component. Algorithm 1 shows the algorithm of [10] for a state space search using external storage, and Algorithm 2 provides the algorithm of [5] which is the basis of most work in the field of parallel and distributed model checking. Both algorithms use the common $search_i$ procedure in Algorithm 3, and rely on a partitioning function $part : \mathcal{S} \to \{1, \ldots, N\}$ which partitions the set of visited states and the queue of unprocessed states into $\mathcal{V}_1, \ldots, \mathcal{V}_N$ and $\mathcal{Q}_1, \ldots, \mathcal{Q}_N$, respectively.

In the external algorithm, only a single partition i is loaded in memory at a time. The visited states of partition i are stored in memory in $\mathcal{V}_i$, and its unprocessed states reside in the queue $\mathcal{Q}_i$. All other partitions $j \neq i$ are not stored in memory, i.e., $\mathcal{V}_j = \emptyset$, but stored on disk files $\mathcal{F}_j$. Queues are also stored on disk, although, for the sake of simplicity of our presentation, we assume here that they are kept in main memory. Initially, all structures and files are empty. The algorithm inserts the initial state s_0 in the appropriate queue $part(s_0)$ (line 3). Then, as long as one of the queues contains a state, the algorithm selects the longest queue i (lines 4–5), loads the associated partition from disk file $\mathcal{F}_i$ to memory in $\mathcal{V}_i$ (line 6) and starts expanding

Algorithm 1 External-memory algorithm of [10]

```
1: for $i$ in 1 to $N$ do
2:   $\mathcal{Q}_i := \emptyset$ ; $\mathcal{V}_i := \emptyset$ ; $\mathcal{F}_i := \emptyset$
3: $\mathcal{Q}_{part(s_0)}.enqueue(s_0)$
4: while $\exists i : \neg \mathcal{Q}_i = \emptyset$ do
5:   $i := longestQueue()$
6:   $\mathcal{F}_i.load(\mathcal{V}_i)$
7:   $search_i()$
8:   $\mathcal{V}_i.unload(\mathcal{F}_i)$
9:   $\mathcal{V}_i := \emptyset$
```

Algorithm 2 Distributed algorithm of [5]

```
1: execute $proc_1 \parallel \ldots \parallel proc_N$
2:
3: procedure $proc_i$ is
4:   $\mathcal{Q}_i := \emptyset$ ; $\mathcal{V}_i := \emptyset$
5:   if $part(s_0) = i$ then
6:     $\mathcal{Q}_i.enqueue(s_0)$
7:   while $\neg$ $termination()$ do
8:     $search_i()$
```

the states in queue $\mathcal{Q}_i$ using procedure *search*$_i$ (line 7) which will be explained below. When the *search*$_i$ procedure does not have any new state to expand for this partition, it writes back the partition to disk file $\mathcal{F}_i$ and empties $\mathcal{V}_i$ (lines 8–9). The selection of the longest queue (line 5) is mainly a heuristic to perform few partition switches (writing back $\mathcal{V}_i$ in disk file $\mathcal{F}_i$ and selecting a new partition).

In the distributed algorithm, data structures are kept in the memory of the N processes involved in the state space exploration. Each process i owns a partition $\mathcal{V}_i$ and a queue of unprocessed states $\mathcal{Q}_i$ that it has to explore. Both structures are initially empty, and the process that owns the initial state puts it in its queue (lines 5–6). As long as termination is not detected (line 7), the process expands the states in its local queue using procedure *search*$_i$ (line 8). Termination occurs when all queues and communication channels are empty.

Algorithm 3 Search procedure common to Algorithms 1 and 2

```
1: procedure $search_i$ is
2:   while $\mathcal{Q}_i \neq \emptyset$ do
3:     $s := \mathcal{Q}_i.dequeue()$
4:     if $s \notin \mathcal{V}_i$ then
5:       $\mathcal{V}_i.insert(s)$
6:       for $e$ in $en(s), s' = succ(s, e)$ do
7:         $j := part(s')$
8:         if $i = j$ then  (* local transition *)
9:           if $s' \notin \mathcal{V}_i$ then
10:            $\mathcal{Q}_i.enqueue(s')$
11:          else  (* cross transition *)
12:            $\mathcal{Q}_j.enqueue(s')$
```

The common part of both algorithms is the *search*$_i$ procedure in Algorithm 3 that expands all the states queued in $\mathcal{Q}_i$ until it becomes empty. Each state s removed from $\mathcal{Q}_i$ (line 3) is checked to be in partition $\mathcal{V}_i$. This check is performed since a state in $\mathcal{Q}_i$ may have been inserted in $\mathcal{Q}_i$ because it was a destination state of a cross transition. If s has not been met before it is inserted into $\mathcal{V}_i$ (line 5) and then expanded (lines 6–12). During the expansion, we compute all the successors s' of s and determine the partition j they belong to (line 7), using the function *part*. If $i = j$ the transition from s to s' is a *local transition*. We can simply check if s' is in memory in table $\mathcal{V}_i$ and put it in queue $\mathcal{Q}_i$ if needed. Otherwise, this is a cross transition, and the partition of state s' is not available in memory (it is stored on disk or belongs to another process). We thus unconditionally put it in $\mathcal{Q}_j$. For the external algorithm of [10] this is implemented by enqueuing the state in the memory queue $\mathcal{Q}_j$ (and possibly writing s' in the disk file associated with $\mathcal{Q}_j$), whereas for the distributed algorithm of [5] it implies to serialise the state in a message and send it to the owner of the appropriate partition, i.e., process j. Upon reception, the state is enqueued by the receiving process in $\mathcal{Q}_j$.

The performance of these algorithms depends to a large extent on the partition function *part*. In the distributed algorithm, cross transitions highly impact the number of messages exchanged and thereby indirectly the execution time. For the external algorithm, partition swaps, and hence disk accesses, are generated by cross transitions. In particular for the distributed algorithm, it is desirable that function *part* distribute states evenly among partitions so that processes receive a comparable workload.

3. Related work on state space partitioning

Stern and Dill [5], Bao and Jones [10] and Garavel et al. [14] left open the problem of the partition function. They used in their experiments a standard hash function taking as input the entire state vector. The importance of the partition function was stressed in [11]. Assuming that the system to be verified is a set of communicating *processes*, the partition function proposed in [11] only hashes the part of the state vector describing a selected process p. Thus, only when that part changes, i.e., the search algorithm explores events of process p, is a cross transition generated. Compared to a global hash function, this scheme efficiently reduces the number of messages exchanged (up to a factor of 5 according to the experiments in [11]) and, hence, the execution time (up to a factor of 3 according to the experiments in [11]). The downside is a degraded distribution of states over the nodes of the network.

The dynamic partitioning in [15] groups states into *classes* and partitions consist of a set of classes. When memory becomes scarce, the partition function is modified by reassigning some classes of the overflowing partition to other partitions. The function mapping states to classes can be a local hash function as in [11]. The results of this dynamic partitioning strategy in terms of message exchanges and verification time are comparable to the ones of [11]. The main advantage is that no knowledge of the system is necessary: run-time information is used to keep the partitioning balanced and, indeed, we generally observed in our experiments (to be discussed in Section 6) a good distribution of states using this dynamic partitioning. An efficient partitioning algorithm based on abstraction and refinement of the state space is introduced in [16]. However, the state space has to be first constructed in order to define the partition function meaning that this approach mainly targets off-line analysis.

In structured duplicate detection [17] as used in external graph search, an abstraction of the state space is used to determine when to load/unload partitions from/to disk. However, this approach seems hard to apply in our context due to the difficulty of defining an abstract graph from a complex specification. Close to that idea is the work of Rangarajan et al. [18]. The algorithm they propose first explores a sample of the state space. This sample is abstracted into a higher level graph using a single variable v of the system. An abstracted state aggregates all states having the same value of v. A partition function can then be constructed from this abstracted graph. The algorithm can reiterate this process on all variables to improve the quality of the function. The underlying principle is the same as in [11,15]: only when the selected variable is modified can a cross transition be generated. The experiment made in [18] shows that this method can significantly outperform the local hash partitioning implemented in PSPIN [11] (a distributed extension of the SPIN tool [19]). The partition function proposed in [6] also relies on a preliminary random walk of the state space. A balanced binary tree is built during this sampling. Its nodes are states and leaves are partitions. As the real search progresses, the tree is used to determine the partition the states reached belong to.

The contribution of this paper is to propose a way to dynamically (i.e., during state space exploration) modify the partition function. Our approach is based on progressively taking into account more components of the underlying system. Part of our work can be seen as an extension of [18] since some of the ideas that we develop were briefly mentioned in [18] — like the one of considering several variables of the system to define the partition function. Another contribution of our dynamic approach is that it can guarantee an upper bound on the size of any partition loaded in memory which previous approach like [11,15,6,18] could not. In the following sections, we develop our dynamic partitioning scheme in the context of the external algorithm of [10] (Algorithm 1). In the conclusion section we discuss the application of the proposed method and the heuristics developed in Section 5 in the context of distributed exploration.

4. Dynamic partitioning based on refinement

Our dynamic partitioning scheme is based on the principle of *partition refinement*. The algorithm starts with a single partition to which all states are initially mapped. If the state space is small enough to be kept in main memory the algorithm acts as a standard RAM algorithm. Otherwise, whenever the partition $\mathcal{V}_i$ currently loaded in memory exceeds the memory capacity, procedure *refine*$_i$ of Algorithm 4 is triggered. It first updates the partition function *part* (line 2) in such a way that each state that was previously mapped to partition i is now mapped to a new partition $j \in \{i_1, \ldots, i_n\}$ (we explain how the partition function is updated shortly). Then, it writes the states in $\mathcal{V}_i$ to disk files $\mathcal{F}_{i_1}, \ldots, \mathcal{F}_{i_n}$ (line 3) and reorganises the queue $\mathcal{Q}_i$ in the same way (lines 4–6). Once this reorganisation is finished, the table $\mathcal{V}_i$ and the disk file $\mathcal{F}_i$ are emptied (lines 7–8), and the search can restart by picking a new partition. Note that partition i is the only one to be reorganised; all other partitions remain unchanged. Our focus is now on the implementation of line 2 of procedure *refine*$_i$. We describe in the rest of this section how our algorithm uses a *compositional partition function* that can change during the state space exploration. We propose a way to dynamically refine the partition function by gradually considering more components of the state vector of the system being analysed.

A compositional partition function can be represented as a *partitioning diagram*. Fig. 1 is the graphical representation of a diagram D. Rounded boxes represent *terminal nodes* and *branching nodes* are drawn using circles. The nodes are labelled either with a partition, e.g., p_0, p_1, or with a *branching function* of which the domain is the universe of states $\mathcal{S}$ and the co-domain can be deduced from the labels of its outgoing arcs. For the diagram in Fig. 1 we have $g : \mathcal{S} \to \{t, f\}, h : \mathcal{S} \to \{a, b, c\}$

Algorithm 4 The partition refinement procedure

```
1: procedure $refine_i$ is
2:   update partition function $part$: partition $i$ is divided into $i_1, \dots, i_n$
3:   for $s \in \mathcal{V}_i$ do $s.write(\mathcal{F}_{part(s)})$
4:   while $\neg\, \mathcal{Q}_i.isEmpty()$ do
5:     $s := \mathcal{Q}_i.dequeue()$
6:     $\mathcal{Q}_{part(s)}.enqueue(s)$
7:   $\mathcal{V}_i := \emptyset$
8:   $\mathcal{F}_i := \emptyset$
9:   go to line 6 of Algorithm 1
```

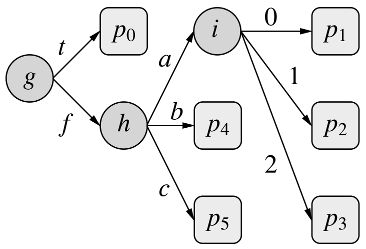


Fig. 1. A compositional partitioning diagram D.

and $i : \mathcal{S} \to \{0, 1, 2\}$. This diagram induces a partition function part_D mapping states to partitions. Starting from the root g of this diagram, we successively apply to the state the different functions labelling the branching nodes of the diagram until reaching a terminal node, i.e., a partition. The branches to follow are given by the labels of the outgoing edges from branching nodes. Below is a few examples of application of part_D:

$$\begin{array}{lcl}\mathsf{part}_D(s) = p_0 & \Leftrightarrow & g(s) = t \\ \mathsf{part}_D(s) = p_5 & \Leftrightarrow & g(s) = f \wedge h(s) = c \\ \mathsf{part}_D(s) = p_3 & \Leftrightarrow & g(s) = f \wedge h(s) = a \wedge i(s) = 2\end{array}$$

Three functions (g, h and i) are used to decide if a state belongs to partition p_3. Hence we say that partition p_3 is *dependent* on functions g, h and i.

The following definition formalises the notion of partitioning diagrams. Note that the definition of the edge set in item 1 implies that partitions are the terminal nodes of the diagram and that functions are branching nodes. Furthermore, the deterministic requirement in item 4 ensures the uniqueness of the partition to which a state is mapped.

Definition 1 (*Compositional Partitioning Diagram (CPD)*). A CPD is a tuple $D = (V, E, r_0, \mathcal{F}, \mathcal{P})$ such that:

1. $G = (V, E)$ is a directed acyclic graph with vertices $V = \mathcal{F} \cup \mathcal{P}$ and edges E, and $r_0 \in V$ is the only root node of G;
2. $\mathcal{F} = \{f_i : \mathcal{S} \to \mathcal{L}_i\}$ is a set of branching functions mapping where f_i maps from the set of states S into a set of labels L_i;
3. $\mathcal{P} \subseteq 2^{\mathcal{S}}$ is a set of state partitions;
4. $E \subseteq \mathcal{F} \times \mathcal{L} \times V$ with $\mathcal{L} = \cup_i \mathcal{L}_i$ is such that for all $f_i \in \mathcal{F}$, $l \in \mathcal{L}_i$ there exists exactly one $v' \in V$ such that $(f_i, l, v') \in E$.

A CPD determines a partition function as formalised in the following definition.

Definition 2 (*Compositional Partition Function*). Let $D = (V, E, r_0, \mathcal{F}, \mathcal{P})$ be a CPD. The function $\mathsf{part} : V \times \mathcal{S} \to \mathcal{P}$ is defined by

$$\mathsf{part}(v, s) = \begin{cases} v & \text{if } v \in \mathcal{P} \\ \mathsf{part}(v', s) & \text{if } v \in \mathcal{F}, \text{ where } (v, v(s), v') \in E \end{cases}$$

The compositional partition function $\mathsf{part}_D : \mathcal{S} \to \mathcal{P}$ is defined by

$$\mathsf{part}_D(s) = \mathsf{part}(r_0, s)$$

Refinement of a CPD consists of replacing a terminal node representing a partition by a new branching node. Thus, a state s that was previously mapped to the refined partition is now redirected to a new sub-partition according to the value of $g(s)$ where g is the function labelling the new branching node.

Our formulation of the refinement algorithm assumes that the global system can be viewed as a set of distinct *components* $C_1 \in \mathcal{D}_1, \dots, C_n \in \mathcal{D}_n$ and that a state of the system is obtained from the state of these components, i.e., $\mathcal{S} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$. This naturally capture systems with a statically defined state vector such as DVE systems [20] and Petri nets [21]. However, as the partition function dynamically evolves as the search progresses, this constraint could easily be relaxed. We denote by f_{C_i} the function that for a given state s returns the value of the component C_i in the state vector. During the refinement of partition p, the partition diagram is modified as follows. The algorithm first inspects the diagram to determine the functions

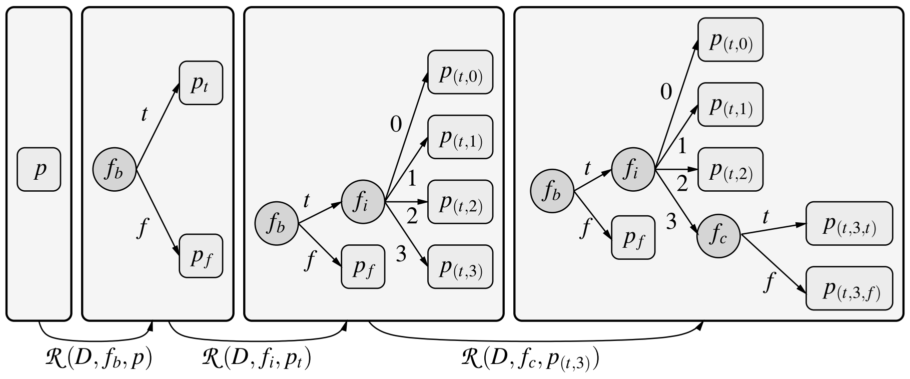


Fig. 2. Refinement of a dynamic compositional diagram D.

F on which partition p is dependent. These functions label the branching nodes on the path from the root to the terminal node associated with p in the diagram. Then it picks a function $f_{C_i} \notin F$. Each of its outgoing branches leads to a new partition. Finally, p is replaced in the diagram by the branching node f_{C_i}. We shall use the term *candidate component* (or simply candidate), to denote a component that can be used to refine a partition, i.e., any component C_i such that the refined partition is not already dependent on f_{C_i}.

Fig. 2 shows the graphical representation of a compositional partition diagram D that dynamically evolves as described above. We assume the following components are part of the underlying system, i.e., that the state vector has the form (b, c, i) such that $b \in \{t, f\}$, $c \in \{t, f\}$ and $i \in \{0, 1, 2, 3\}$. Initially, there is a single partition p and all states are mapped to that partition. As p exceeds the allowed size, it is refined into p_t and p_f after the selection of the boolean component b to be used for the refinement. States already visited with $b = t$ are put in partition p_t and states with $b = f$ are put in p_f. Later, partition p_t becomes too large. Since this partition is already dependent on function f_b it would not make sense to refine it using component b: all states of p_t would be redirected to the same partition. In this case, the algorithm selects to refine it using component i. Partition p_t is thus split in $p_{(t,0)}$, $p_{(t,1)}$, $p_{(t,2)}$ and $p_{(t,3)}$ and states that were previously in p_t, i.e.,with $b = t$, are redirected to one of these according to the value of component i. Note that p_f is unchanged. Thus, states that satisfy $b = f$ will still be mapped to this partition whatever the value of their other components.

The definition below formalises this idea of partition refinement.

Definition 3 (*Partition Refinement*). Let $D = (V, E, r_0, \mathcal{F}, \mathcal{P})$ be a CPD. The refinement $\mathcal{R}(D, f, p)$ of D with respect to a function $f : \mathcal{S} \to \mathcal{L}_f$ and $p \in \mathcal{P}$ is the CPD $D' = (V', E', r'_0, \mathcal{F}', \mathcal{P}')$ with

1. $\mathcal{F}' = \mathcal{F} \cup \{f\}$;
2. $\mathcal{P}' = \mathcal{P} \setminus \{p\} \cup \mathcal{P}_{\mathcal{L}_f}$, where $\mathcal{P}_{\mathcal{L}_f} = \{p_l \mid l \in \mathcal{L}_f\}$ and $p_l = \{s \in p \mid f(s) = l\}$;
3. $E' = E \setminus \{(v, l, q) \in E \mid q = p\} \cup \{(f, l, p_l) \mid l \in \mathcal{L}_f\} \cup \{(v, l, f) \mid (v, l, p) \in E\}$;
4. $r'_0 = f$ (if $r_0 = p$), and $r'_0 = r_0$ (otherwise).

Item 1 adds the new branching function f to the set of branching function and item 2 removes the partition p being refined and replaces it with the new sub-partitions based on the application of the new branching function f to the states in p. Item 3 removes all edges leading to the partition p being refined, replaces them with edges leading to the new branching node for f, and adds edges leading from f to the new sub-partitions. Finally, item 4 handles the case where it is the initial partition being refined which means that f becomes the root of the refined diagram.

The motivation behind this dynamic partitioning scheme is to benefit as much as possible from system properties. Usually, realistic systems are composed of many components, and events only modify a small fraction of them leaving others unchanged. Our refinement algorithm tries to minimise cross transitions by only selecting for each partition a few components to depend on. Let us consider, for instance, the last step in the evolution of the compositional diagram of Fig. 2. All the states of partition $p_{(t,2)}$ have in common that $b = t \wedge i = 2$. Hence, from any state of this partition, an event that does not change the value of b or i will not generate a cross transition. Clearly, the way the component is selected during a refinement step largely impacts the number of cross transitions it will cause. For instance, the worst choice would be to select a global variable updated by all events. In that case, any transition from a state of the resulting partitions will be a cross transition.

5. Selection of candidate components

We propose in this section several heuristics to select components to be used as a basis for the refinement. We classify these in two categories. *Off-line heuristics* perform an analysis of the model or sample the state space to order components. Then, during the search, the next component is always chosen according to that predetermined order. Hence along two different paths (of same length) of the partitioning diagram, we always find the same components in the same order. With *online heuristics*, the component selected is chosen *during* the refinement step on the basis of data collected on-the-fly during the search.

5.1. Off-line heuristics

Heuristic **SA***: Static Analysis*. With this first heuristic, the algorithm tries to predict from a static analysis of the model the modification frequency of components. The analysis performed is simple. We count for each state vector component, the number of events in the model that modify it and order components accordingly in increasing order. This means that we prioritise the components that are the least frequently modified in an attempt to reduce the number of cross transitions. Some weights may also be associated with events as we did in our implementation for the DVE language. For example, events inside loop-constructs were assigned a high weight as it is reasonably assume that their execution will occur frequently (each time the loop is executed).

Heuristic **SS***: State Space Sample*. Heuristic SA from above is based on a static analysis of the model and as such assumes a uniform distribution of event executions. However, in practice, this assumption is not always valid. Some events are typically executed only a few times, e.g., initialisation events, whereas some will generate most of the state transitions. With heuristic SS, we attempt to tackle this problem by first exploring a sample of the state space. An array of integers indexed by state components is maintained and each time an event is executed, the counters of all modified components are incremented. State components are then ordered according to the values of their counters, lowest values first. It is very important to perform a randomised search in order to seek a reasonably representative sample of the state space. A breadth-first search, for instance, would only explore the states of the first levels of the state space, and these usually share very few characteristics with the states we can find at deeper levels (and hence different executable events).

5.2. On-line heuristics

Heuristic **RD***: RanDomised*. This strategy picks out a component randomly from a set of candidates. The purpose of this strategy is only to serve as a baseline to assess the other on-line strategies below.

Heuristic **EE***: Event Execution*. Heuristic EE is the dynamic equivalent of the heuristic SA: an array of integers specifying, for each component, the number of modifications of that component, is maintained as the state space exploration progresses. During a refinement step, the algorithm selects, among candidates, the one which has, until now, been the least frequently modified.

Heuristic **PD***: Partition Distribution*. The previous heuristics do not consider how well states are distributed among sub-partitions during a refinement step. This may, however, have important consequences in subsequent steps. Suppose that a partition p is refined in two sub-partitions, the first one, p_1, receiving 95% of the states of p, and the second one, p_2, receiving 5% of these states. Then, it is likely that during the next expansion step of partition p_1, new states will be added to p_1 which will cause it to exceed the maximal allowed size and hence to be refined. We can thus reasonably consider the first refinement to be useless. As refinement steps are costly – it entails writing back to disk each state in the partition currently loaded in memory – these refinements should be avoided as much as possible.

With heuristic PD, the refinement procedure simulates all possible refinements by computing for each state s of the partition to be refined the values $f_{C_{i_1}}(s), \ldots, f_{C_{i_k}}(s)$, where $f_{C_{i_1}}, \ldots, f_{C_{i_k}}$ are the partition functions for the candidate components. This indicates how good the state distributions induced by the different candidates are. Then, the algorithm picks the component that achieves the lowest standard deviation of sub-partition sizes, that is, the most even distribution of states among partitions. Applying $f_{C_{i_1}}, \ldots, f_{C_{i_k}}$ on all states does not incur a major time penalty. In the worst case (if all components are candidates), this is equivalent to compute a hash value on the entire state vector, which is usually negligible compared to the later writing of the state in the sub-partition file. In our experiments, we observed that, when heuristics EE and DE (that extends EE with this "simulation" process, see below) exhibited comparable performances in terms of disk accesses, the execution times were roughly the same.

Heuristic **DE***: Distribution and Event execution*. This last heuristic combines the idea of heuristics PD and EE: we prefer candidates that achieve a good state distribution and which is not frequently modified. During a refinement step, the following metric is computed for each candidate C_i:

$$h(C_i) = updates[i] \cdot \max(std(C_i), 1) \tag{1}$$

where $updates[i]$ is the number of modifications of component i recorded so far (cf. heuristic EE) and $std(C_i)$ is the standard deviation in the sizes of sub-partitions obtained if component C_i is chosen for refinement (cf. heuristic PD). The algorithm picks the candidate having the lowest value. We take the maximum of $std(C_i)$ and 1 so that updates are taken into account in case sub-partitions are of equal size, i.e., $std(C_i) = 0$.

6. Experiments

The PART algorithm of [10] as well as our dynamic partitioning technique have been implemented in the ASAP model checking platform [12]. We report in this section on experimental results obtained with this implementation.

Table 1
Input models used for experimentation.

Model	Comp.	Update factor	Vector	States	Transitions	BFS levels
Mutual exclusion algorithms						
anderson.6	14	2.00	19	18,206,917	86,996,322	182
at.5	10	2.34	14	31,999,440	125,231,180	52
Controllers						
bopdp.7	26	2.07	26	15,236,725	42,498,986	171
elevator.5	30	2.82	54	185,008,051	185,008,051	122
lifts.9	39	2.84	43	266,445,936	846,144,885	372
plc.4	52	2.49	113	3,763,999	6,100,165	7852
production_cell.6	25	1.98	25	10,689,251	33,005,890	69
train-gate.7	24	3.98	32	50,199,556	106,056,460	232
Communication protocols						
brp.6	18	2.06	18	42,728,113	89,187,437	659
cambridge.7	14	11.24	64	11,465,015	54,850,496	340
collision.5	22	1.77	52	431,965,993	1,644,101,878	182
firewire_link.3	32	2.45	41	425,333,983	1,621,543,475	416
iprotocol.8	19	2.00	45	447,570,146	1,501,247,756	353
pgm_protocol.10	46	3.87	133	29,679,589	61,980,099	919
rether.7	44	2.18	55	55,338,617	61,198,113	419
Leader election algorithms						
extinction.7	45	2.53	61	20,769,427	76,184,301	150
firewire_tree.6	351	2.06	483	22,690,105	126,238,660	268
leader_election.6	82	3.21	235	35,777,100	233,195,212	222
leader_filters.8	17	1.5	32	431,401,020	1,725,604,080	83
Other protocols						
msmie.4	24	1.8	28	7,125,441	11,056,210	63
needham.4	28	1.87	76	6,525,019	22,203,081	34
public_subscribe.5	34	1.78	36	1,153,014,089	5,447,695,171	170
synapse.9	28	4.85	58	1,675,298,471	3,291,122,975	89
telephony.7	15	2.2	24	21,960,308	114,070,470	76

6.1. Application of refinement to DVE systems

We have evaluated our algorithms on models written in the DVE language [20]. In this language, the system is described as a set of automata synchronising through communication channels and global variables. Communication can either be synchronous or asynchronous. An automaton is described as a set of states, local variables, and guarded events. To use our refinement algorithm, we considered as components each of the following items: the state of an automaton, i.e., its program counter; variables (global or local); and the content of communication channels. Arrays were considered as a single component for simplicity. In a future implementation, this could be further refined by considering the elements in arrays as components. Since the domain of variables can be very large and cannot be defined a priori, we used for each component C_i the component function $f_i = h_i(C_i) \; mod \; p$ where h_i is a hash function from $\mathcal{D}_i$ (the domain of component C_i), to $\mathbb{N}$ and p is the maximum number of sub-partitions we want a partition to be refined in (p was set to 20 in our implementation).

6.2. Input models

All our input models come from the BEEM database [22]. We have pursued a representative set of models according to several criteria: type of system modelled (e.g., communication protocol, mutual exclusion algorithm), size and shape of the state space graph (e.g., short or long graphs), type of synchronisation used (through global variables or communication channels), and the size of the state descriptor. We did not experiment with models belonging to the categories "Planning and scheduling" and "Puzzles" that are mostly toy examples having few common characteristics with real-life models. This selection resulted in 24 DVE instances having from 4×10^6 up to 1.6×10^9 states. The experiments we report on in this section have all been performed on this input set. The characteristics of the chosen instances are listed in Table 1. Column Comp. gives, for each model, the number of components that can be used by our algorithm during a refinement step. Column Update factor gives the average number of components modified by events. This data is obtained through a static analysis of the model (and not from its state space). Column Vector gives, in bytes, the size of the state vector used to encode states. Columns States and Transitions give, respectively, the number of reachable states and the number of executable transitions in the system. Finally, column BFS levels provides the number of BFS levels in the state space, where level l consists of states for which the shortest transition sequence from the initial state is of length l.

6.3. Experiment 1: evaluation of partitioning strategies

The goal of this first experiment is to evaluate the heuristics from Section 5, and compare our method to existing partitioning functions found in the literature. All partition functions have been evaluated through an implementation on top of the PART algorithm [10].

6.3.1. Context

In addition to our refinement technique, we also implemented the static and dynamic partitioning schemes of [11] and [15] using a local hash function that only refers to the part of the state vector corresponding to a specific process of the system. In our implementation of [15], a partition is split in two sub-partitions when it exceeds memory capacity: half of the classes that comprises the partition are moved to new partitions. The process used for hashing was selected after an initial sampling of the state space. We selected the process that achieved both a uniform state distribution and a low number of cross transitions using heuristic h in Eq. (1) from the previous section. The initial sampling was stopped after 100,000 states had been visited. This represents, for most models, a very small fraction of the state space.

During each run we gave the PART external-memory algorithm the possibility to keep in memory at most 1% of the state space. Half of this amount was given to the memory buffer of the state queue (recall that PART stores the queue on disk) and half was given to the partition loaded in memory. Hence, each partition could contain at most 0.5% of the total state space size. Note that this setting is valid throughout this section. With static partitioning, it is impossible to put an upper bound on a partition size. Therefore, assuming the distribution of states upon partitions might be unfair, we configured the static schemes with 256 partitions to guarantee (to the extend possible) that a partition will not contain more than 0.5% of the state space. For dynamic partitioning strategies, when a partition exceeded this capacity, it was automatically split using refinement with our algorithm or by reassigning classes of states to partitions with the algorithm of [15]. As noted earlier, the algorithm of [15] cannot guarantee an upper bound on a partition size: when a partition contains a single class it cannot be further reorganised.

Naming conventions. We denote by S-GHC and D-GHC the Static and Dynamic partitioning strategies based on the Global Hash Code: the partition function is the global hash function modulo the number of partitions. Similarly S-LHC and D-LHC denote the static and dynamic partition functions based on the Local Hash Code: only the part of the state vector corresponding to the selected process is hashed, that is, the algorithms of [11] (in the static setting) and [15] (in the dynamic setting). We refer to our partition function using the name of the used heuristic: SA (Static Analysis), SS (State space Sample), RD (RanDomized), EE (Event Execution), PD (Partition Distribution), and DE (Distribution and Event execution).

6.3.2. Results

The histograms of Fig. 3 summarise our results. Each value plotted corresponds to the number of cross transitions, IO operations or execution time of a single run using a specific partitioning strategy. Strategy S-GHC is used as an index (reference) value, i.e., it corresponds to a value of 1. For the IOs and Time histograms, the bottom part in white filled and dashed boxes correspond, respectively, to the IOs that were due to partition reorganisation and the time spent in reorganisation. For the sake of clarity, some data are not plotted on these histograms which are those of strategies D-GHC, S-LHC, and RD. For the two first ones, the results were very similar to their static and dynamic analogues, respectively. Heuristic RD performed much worse than the other heuristics which confirmed our initial intuition on the impact of the component chosen during the refinement step.

Quality of the sample. Heuristic SS and heuristic EE (its on-line equivalent) exhibit comparable performances. This indicates that the preliminary randomised search of SS often provides a good sample of the state space even when this sample represented a very small fraction of the state space (e.g., for models public_subscribe.5 and synapse.9). However, we found two models (firewire_tree.6 and, to a lesser extent, production_cell.6) for which this observation is not valid demonstrating the relevance of heuristic EE.

Influence of the graph structure. In [23], we observed that PART (using a global hash function to partition the state space) is not designed for long state spaces, i.e., state spaces with many levels, which should also hold for the distributed algorithm of [5]. To illustrate that, consider the extreme case where the graph is a long sequence of states. Using a good hash function we may assume that the probability of a transition being a cross transition is close to $\frac{N-1}{N}$ (where N is the number of partitions). Hence, with this state space structure, most cross transitions will immediately be followed by a partition swap. With a distributed algorithm, the search will consist of a long series of message exchanges with processes constantly waiting for new states. A partition function that exploits model structure can fill that gap. For models brp.6, iprotocol.8, pgm_protocol.10, plc.4, and rether.7 that have long state spaces (up to almost 8000 levels for plc.4), all partitioning strategies based on the model structure significantly outperform strategies S-GHC and D-GHC. Also, except for rether.7, compositional partitioning performs significantly better than LHC based partitioning. Model cambridge.7 should normally be put in that category as it has a long state space with 340 BFS levels. However, it does not respect the assumption our refinement algorithm is based on: events of this model modify many components (an average of 11.24 out of 14 components). This explains the relatively poor performances of structure based partitioning strategies. In contrast, for short graphs (e.g., telephony.7 with 76 BFS levels or needham.4 with 34 BFS levels), structure based partitioning has a lesser impact.

Cost of partition reorganisation. For models firewire_tree.6, leader_election.6 and needham.4, heuristics SS and EE generate, after refinements, unfair state distributions which leads to a large amount of disk accesses due to partition reorganisation. Thus, even if they are, among refinement based strategies, the ones generating the fewest cross transitions, this has little consequences on the overall number of disk accesses. Heuristics PD and DE that aim to distribute states equally among partitions largely outperforms both on these three problems when considering disk accesses. This observation can be generalised to most models we experimented with: heuristics EE and SS minimise cross transitions, but not necessarily

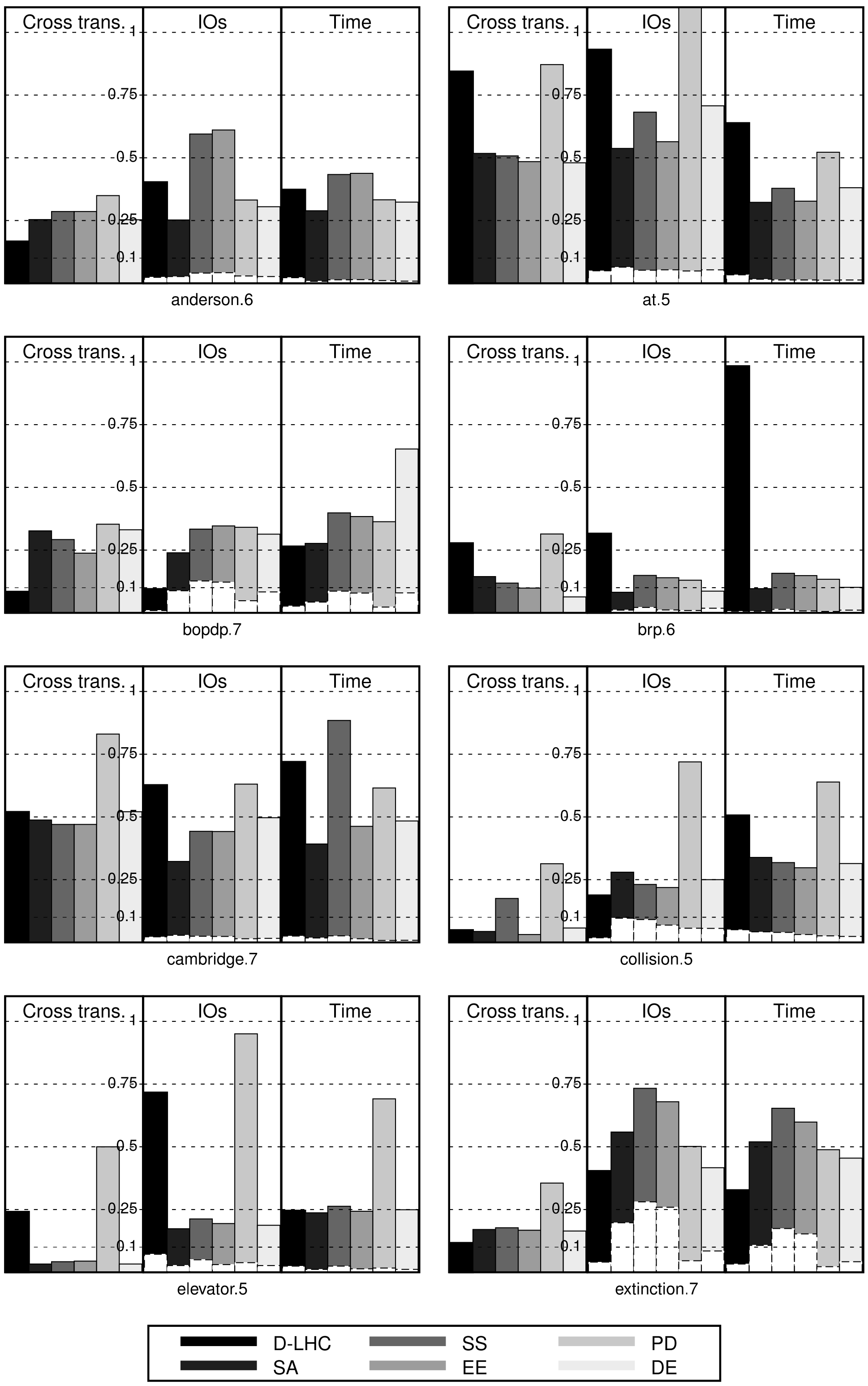


Fig. 3. Comparison of different partitioning strategies (spans 3 pages).

disk accesses and execution times. Fig. 3 also shows that heuristics PD and DE can efficiently limit the cost of reorganisation. On all models the bottom part of the boxes representing reorganisation IOs and times are negligible or at least lower than with other strategies.

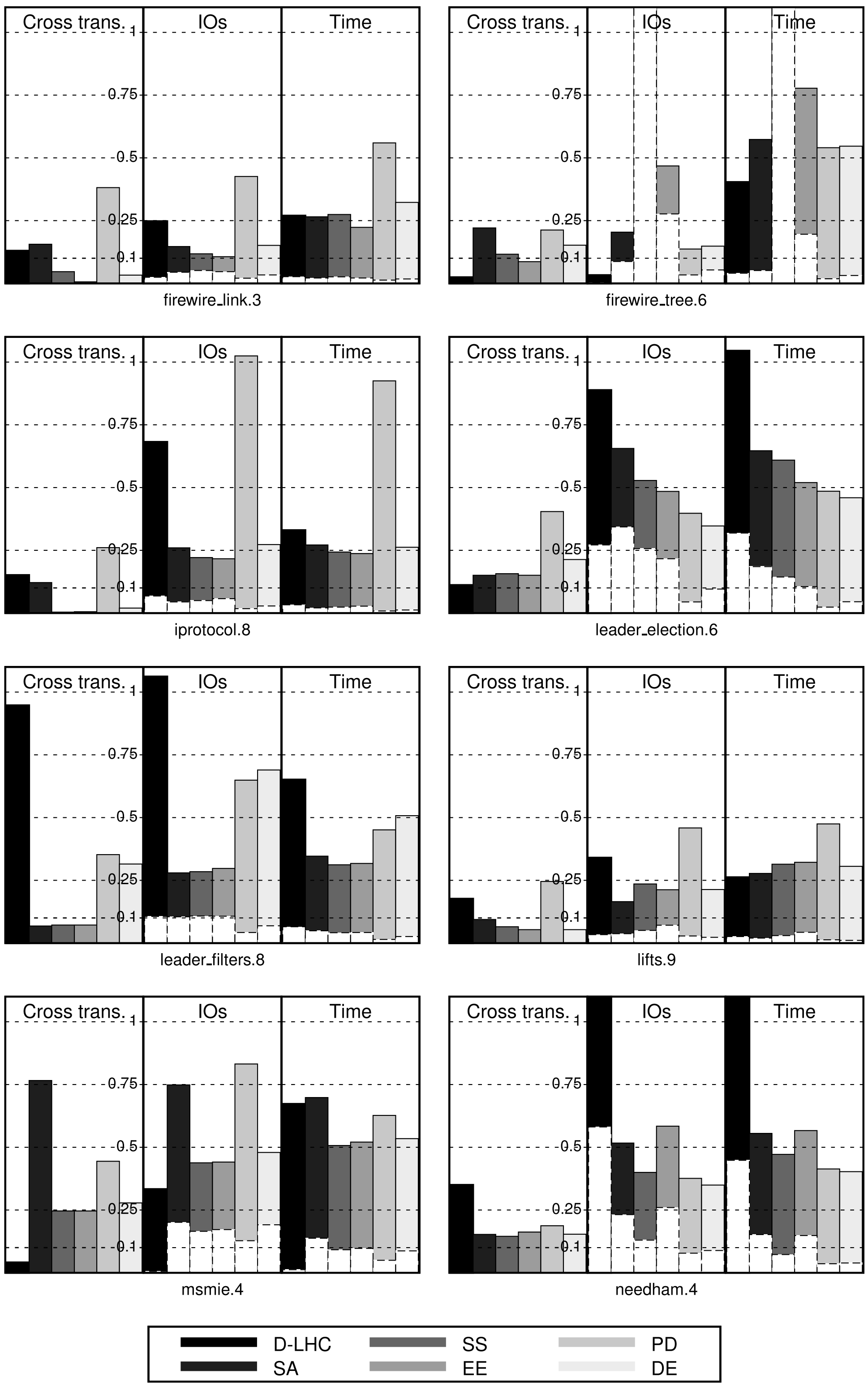


Fig. 3. (Continued).

It should be noted that even when reorganisation generates a large amount of IOs, this does not necessarily have such a negative impact on the execution time. An important factor seems to be the state vector size. For models with large vectors (e.g., 483 bytes for firewire_tree.6, and 235 bytes for leader_election.6), reorganisation IOs obviously contribute to

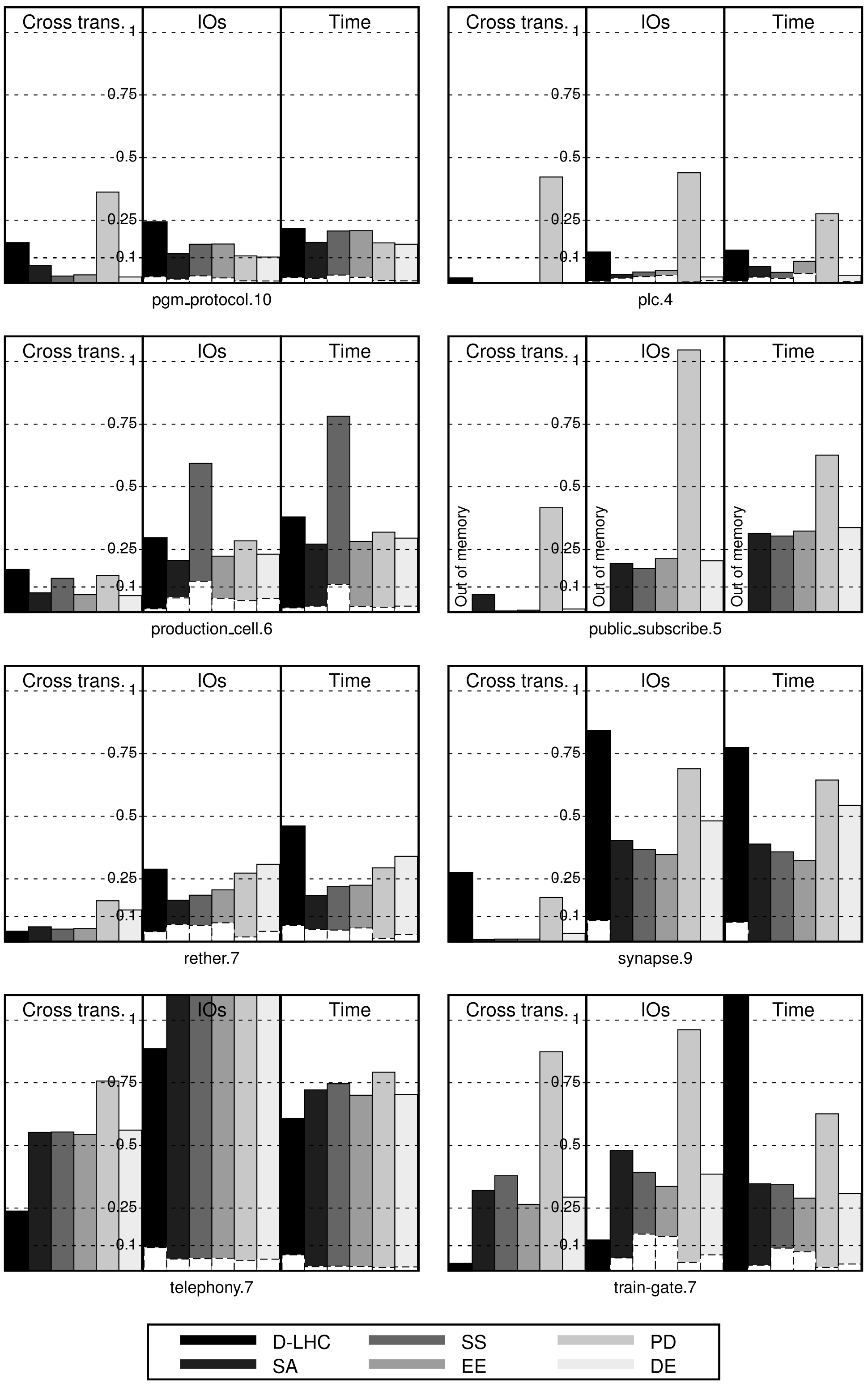


Fig. 3. (Continued).

a significant increase of the run time. In contrast, for models with small state vectors (e.g., 41 bytes for firewire_link.3) the proportion of reorganisation IOs is not reflected in the execution time.

In [15] it is advised to delete states after a reorganisation rather than sending them to its new owner which is claimed to be too expensive. This comes at the cost of possibly revisiting states that have been deleted. Intuitively, since heuristic EE performs more refinements it should cause the deletion and revisit of more states than DE and, hence, generate more cross transitions and message exchanges. It is therefore not immediately clear which one should be preferred in a distributed setting which in turn demonstrates the relevance of evaluating our heuristics separately in a distributed environment.

Granularity of components. Synchronisations in model at.5 and telephony.7 are realised through global arrays modified by most events. As the refinement procedure currently implemented considers arrays as single components, our algorithm is not as efficient in these cases as it could potentially be. A better management of arrays should improve this. This remark applies to most mutual exclusion algorithms we have experimented with.

Effects of cross transitions. Although Fig. 3 exhibits some correlation between the number of cross transitions and disk accesses, this is not always the case. First, for the reason that explains the bad performances of heuristic EE for models firewire_tree.6 and needham.4: disk accesses are also triggered by partition refinements. Second, because the consequences of cross transitions largely depend on the stage of the search they occur at: as the search progresses, partitions contain more and more states which increases the cost of swapping. Finally, it suffices that one cross transition leads to a state of partition j when queue $\mathcal{Q}_j$ is empty to cause partition j to be eventually loaded in memory again. All subsequent cross transitions do not affect the algorithm. Hence, a large number of cross transitions linking two partitions is not necessarily problematic in terms of disk accesses.

Impact of disk accesses on execution times. Similarly to cross transitions, we observe a relationship between disk accesses and execution times. This is not surprising since IO operations are usually the most time expensive operations performed by external-memory algorithms. However, in some contexts, the execution time with strategy D-LHC is not correlated with the number of disk accesses. This remark holds especially when PART with strategy D-LHC consumed more memory than using our refinement algorithm (see Table 2 introduced in the next paragraph). In these situations, data structures handled by the algorithm are much larger and the program spends more time in garbage collection. Model train-gate.7 is a pathological example. Strategy D-LHC outperforms our refinement algorithm with respect to cross transitions and disk operations, but this does not have any consequence on the execution time. This stems from the fact that PART with strategy D-LHC consumed much more memory than what was given to our refinement algorithm (73.9% vs. 0.5%).

Memory use of LHC based strategies. Our refinement algorithm outperforms the partitioning algorithms of [11] and [15] although only slightly. However, the experiment reported here is somewhat unfair to our algorithm as no memory limit was (and could be) given to strategies S-LHC and D-LHC whereas our refinement algorithm works within a bounded amount of RAM. Table 2 gives for all our input models and for these two partitioning schemes, the size (as a proportion of the state space size) of the largest partition. We recall that the memory limit per partition that was given to our algorithm is 0.5% of the total number of states. As Table 2 shows, only in a few cases are the algorithms of [11] and [15] able to stay within the allowed memory. When the algorithm could do so (i.e., for values less than or equal to 0.5% in the table), LHC based partitioning is clearly outperformed by a refinement based partition function. This is also evident in Fig. 3 which shows that for iprotocol.8, pgm_protocol.10, plc.4 and synapse.9, refinement based partitioning outperforms strategy D-LHC regardless of the performance criterion considered. When LHC based partitioning performed better it usually meant that it used more memory than what was given to our refinement algorithm. This is especially the case for models bopdp.7, firewire_tree.6, msmie.4 and train-gate.7. Still, for some models like firewire_link.3, LHC based strategies and refinement based strategies performed comparably with respect to IOs, execution time and RAM usage.

Conclusion on compositional strategies. A main conclusion which can be drawn from the experiment is that no heuristic is clearly superior. Even heuristic SA based on a very simple static analysis of the model is competitive for certain models. Moreover, there can be important variations on the same model between two heuristics. This is quite inconvenient from a user perspective. The design of an adaptive heuristic could therefore be an interesting possibility to tackle this issue. We have chosen heuristic DE to be used in the next experiments since, on the average, it is the one that generates fewest disk accesses.

6.4. *Experiment 2: partitioning issues and scalability of heuristic DE*

In this second experiment, we provide a closer analysis of our refinement algorithm from two perspectives: the quality of the partitioning realised and the scalability as the available RAM decreases and the number of partitions increases.

State space partitioning. The sizes of the 200 largest partitions are plotted in Fig. 4 for 15 out of our 24 input models. This size is expressed as a percentage of the state space size. The data are extracted from the runs of experiment 1. Therefore, a partition could not contain more than 0.5% of the state space. In order to ease the readability of the figure, we have removed some curves that did not bring additional information. The conclusion here is somewhat negative in the sense that heuristic DE – as all other heuristics we proposed – often generates partitions of unequal sizes. We still observe that for some models, e.g., brp.6, cambridge.7, production_cell.6, elevator.5, the curve goes through some thresholds. It is likely that these models exhibit some sort of symmetry, at least if we consider some of its components. The refinement of a partition can then generate sub-partitions of uniform sizes.

This observation has little consequence in the case of an external memory algorithm, but it is likely to have a negative impact on a distributed algorithm in that working processes may receive different workloads. It will obviously happen if the

Table 2
Size of the largest partition (as a percentage of the state space size) with static and dynamic partition functions of [11,15] based on local hash code.

	S-LHC	D-LHC
anderson.6	6.8%	6.8%
at.5	0.3%	0.4%
bopdp.7	24.6%	21.6%
brp.6	20.5%	20.5%
cambridge.7	0.2%	0.4%
collision.5	4.7%	3.9%
elevator.5	1.8%	1.3%
extinction.7	4.9%	4.7%
firewire_link.3	2.3%	2.0%
firewire_tree.6	61.0%	56.1%
iprotocol.8	0.1%	0.4%
leader_election.6	23.5%	23.5%
leader_filters.8	0.1%	0.5%
lifts.9	1.0%	0.7%
msmie.4	47.7%	47.7%
needham.4	6.7%	6.7%
pgm_protocol.10	0.2%	0.4%
plc.4	0.1%	0.4%
production_cell.6	6.4%	6.4%
public_subscribe.5	Out of memory	
rether.7	4.8%	4.8%
synapse.9	0.5%	0.4%
telephony.7	17.5%	17.5%
train-gate.7	73.9%	73.9%

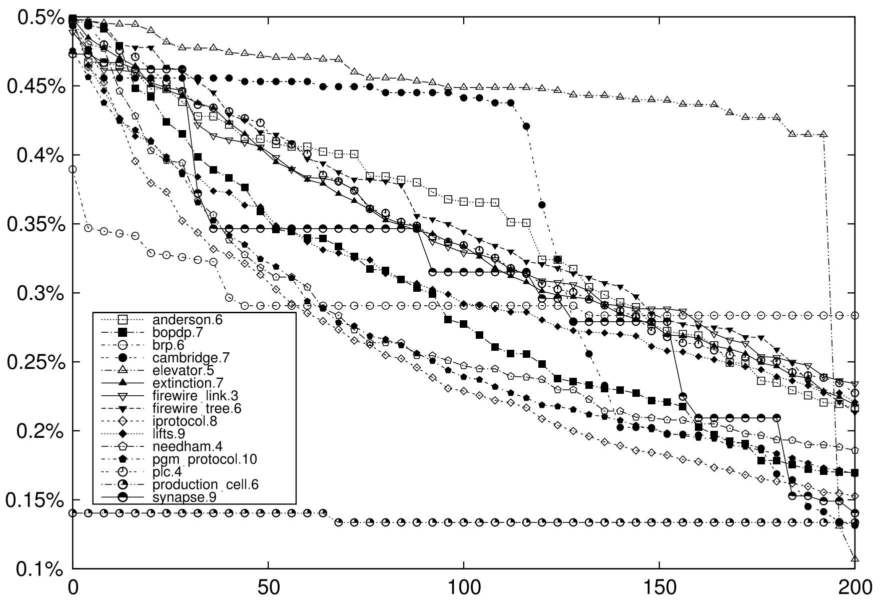
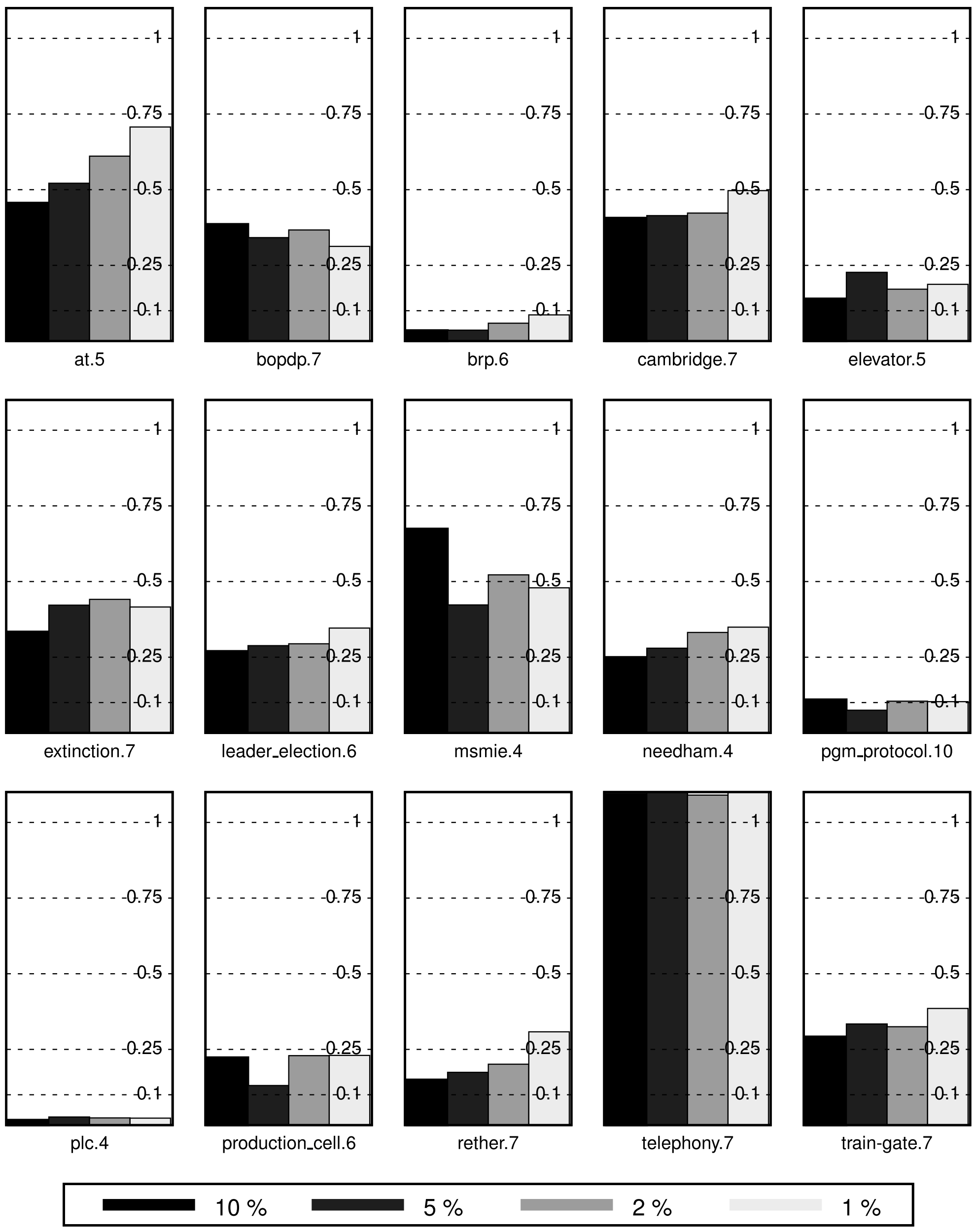


Fig. 4. Sizes of the 200 largest partitions using heuristic DE for 15 representative models.

mapping assigning partitions to processes is bijective. A possible way to limit this problem would be to consider partition sizes in the attribution of partitions. A direct research perspective will be to use PART with our heuristics, but allow it to keep several partitions in memory at the same time (provided it still does not exceed some limit). This may limit further disk accesses and better simulate a distributed algorithm based on state space partitioning. Consequently cross transitions between these small partitions loaded in memory will not be treated as such.

Scalability. We also experimented with our refinement algorithm and heuristic DE using different amounts of available RAM to store the queue of unprocessed states and the partition loaded in memory: 10%, 5%, 2% and 1% (as in Experiment 1) of the whole state space. Due to memory constraints, we could not experiment with all our input models and had to select 15 models out of the 24 models listed in Table 1. Fig. 5 shows for these 15 models the evolution of disk accesses as the available memory decreases. Partition strategy S-GHC is taken again as a reference. Hence a value of 1 means that heuristic DE generated exactly the same amount of disk accesses as strategy S-GHC for a specific setting. The table below the histograms gives the average values over these 15 models for the different memory sizes we experimented with. As a general trend we observe that the number of IOs with both partitioning methods slowly converge. This was to be expected. Indeed, the proportion of cross transition naturally grows as partitions shrink. Since, using strategy S-GHC almost all transitions are



	10%	5%	2%	1%
Average over 15 models	0.322	0.334	0.346	0.375

Fig. 5. Number of disk operations for 15 input models with heuristic DE and different amounts of available RAM (from 10 to 1% of the entire state space) relatively to strategy S-GHC.

already cross transitions, any partitioning function will necessarily degenerate in function S-GHC. In the extreme situation, every partition contains a single state, and every transition is a cross transition, whatever the chosen partition function.

Nevertheless, the table shows that our algorithm still has a rather good resistance in that respect. Even using only 1% of the required RAM (which gives at least 200 partitions) we generated on the average 37.5% of the number of disk accesses generated by function S-GHC. This is to relate to the fact that using 10 times more memory this ratio goes down from 37.5 to 32.2%.

6.5. *Experiment 3: comparison of* PART *with other state space search algorithms*

In this last experiment we compare PART equipped with our refinement technique with two search algorithms that do not rely on graph partitioning. The first one (BFS-MEM) is a pure internal memory breadth-first search algorithm. The second

Table 3
Comparison of PART with compositional refinement and heuristic DE, BFS-HD3 and BFS-MEM.

Model	States (E6)	Time (s)				IOs per state	
		BFS-MEM		BFS-HD3	PART	BFS-HD3	PART
anderson.6	18	52		× 23.82	× 5.80	134.8	÷ 14.6
at.5	31	132		× 6.87	× 3.41	20.1	÷ 1.2
bopdp.7	15	43		× 20.16	× 3.86	99.8	÷ 9.8
brp.6	42	88		× 90.28	× 5.40	389.6	÷ 34.8
cambridge.7	11	211		× 9.50	× 3.23	217.7	÷ 6.5
collision.5	431	3,135	⋆	× 27.63	× 3.34	92.0	÷ 11.1
elevator.5	185	301	⋆	× 34.12	× 12.09	50.4	÷ 9.3
extinction.7	20	150		× 10.50	× 2.57	63.2	÷ 6.1
firewire_link.3	425	4,922	⋆	× 26.81	× 2.97	186.2	÷ 20.3
firewire_tree.6	22	5,253	⋆	× 13.28	× 1.02	156.0	÷ 21.2
iprotocol.8	447	2,024	⋆	× 59.68	× 5.41	161.9	÷ 12.0
leader_election.6	35	783	⋆	× 16.81	× 3.81	173.9	÷ 17.8
leader_filters.8	431	2,064	⋆	× 14.34	× 6.98	41.7	÷ 2.2
lifts.9	266	2,940	⋆	× 12.81	× 2.74	166.0	÷ 13.8
msmie.4	7	30		× 5.96	× 3.43	28.6	÷ 3.8
needham.4	6	30		× 10.26	× 3.06	23.0	÷ 4.2
pgm_protocol.10	29	144	⋆	× 51.86	× 6.22	272.1	÷ 21.9
plc.4	3	22		× 385.45	× 4.22	2156.0	÷ 334.1
production_cell.6	10	29		× 9.44	× 2.55	17.7	÷ 3.2
public_subscribe.5	1,153	10,362	⋆	× 11.65	× 6.02	69.2	÷ 10.2
rether.7	55	120	⋆	× 59.55	× 9.18	208.0	÷ 10.9
synapse.9	1,675	7,448	⋆	× 36.13	× 10.45	31.9	÷ 4.6
telephony.7	21	156		× 6.86	× 6.68	39.0	÷ 1.1
train-gate.7	50	125		× 30.21	× 5.13	143.7	÷ 10.7
Average on 24 models				× 40.58	× 4.98	–	÷ 17.08

one [23] (BFS-HD3) is an external memory breadth-first search that extends the algorithm of [8] with the principle of hash-based delayed duplicate detection [24]. This extension allows to perform a single file scan per BFS level in order to detect duplicate states whereas the algorithm of [8] additionally performs a duplicate detection when internal memory is full. The goal of this comparison is to evaluate how close PART (equipped with our partitioning function) is to an algorithm that does not use external memory and hence does not have the overhead of IO operations and how it performs compared to an existing external memory algorithm.

We compared the three algorithms on our 24 input models and measured their execution times. For multiple models, the whole state space could not be kept in RAM. In these situations we interrupted BFS-MEM after the exhaustion of 80% of the available memory (i.e., 3.2 GB). If the search was interrupted this way, we then extrapolated the final data from the sample explored so far. The same amount of RAM was given to BFS-HD3 and PART, i.e., the possibility to keep at most 1% of the state space in internal memory. Since they perform exactly the same operations in internal memory (e.g., computation of admissible transitions, generation of successors, state serialisation), we also chose the number of IOs performed as a comparison criterion.

Table 3 lists the data collected during this last experiment. Column Time gives the execution times of the three algorithms for each input model. It is expressed in seconds for algorithm BFS-MEM. A star next to this time value indicates that BFS-MEM had to be interrupted and that the time reported is thus an extrapolation. For algorithms BFS-HD3 and PART the time value is expressed relatively to the execution time of BFS-MEM. Column IOs gives the average number of disk operations per state performed by BFS-HD3 (in absolute value) and PART (relatively to the number of IOs of BFS-HD3).

The comparison of BFS-HD3 and PART is clearly in favour of PART. We did not find any models on which BFS-HD3 was faster than PART or generated more disk accesses. For a few cases, both algorithms generated a comparable number of IOs. This occurred mainly for short graphs, like telephony.7. This stems from the facts that the amount of disk accesses performed by BFS-HD3 is a direct consequence of the height (number of levels) of the graph and that, as demonstrated by our first experiment, our partitioning function does not bring significant improvements for this kind of graph structure. This second observation also holds for models composed of a few components, like at.5. In contrast, for graphs having the opposite characteristics, e.g., plc.4, PART is considerably faster. An important advantage in favour of the PART algorithm is also that it does not impose a particular state space search order.

The relative performances of algorithms BFS-MEM and PART seem to largely rely on the complexity of the model in terms of the cost of computing executable events, executing events, and hashing states. As this complexity grows, internal RAM operations tend to dominate IO operations and PART can then almost compete with BFS-MEM. Models firewire_tree.6 and firewire_link.3 are good illustrations. These models have a large number of (syntactical) events: approximately 1700 and 400 respectively. Hence, the computation of enabled events at each state is the most time consuming operation. Since heuristic DE performs very well on these two problems, it is competitive with the BFS-MEM algorithm in this case.

Using our partitioning function, PART is on the average 5 times slower than what could be achieved in the ideal case. This must however be compared to the fact that PART only used 1% of the RAM given to BFS-MEM.

7. Conclusions and future work

We have proposed in this paper a dynamic partitioning algorithm applicable in external-memory and distributed state space exploration. As part of this, we have presented a common framework for external and distributed algorithms based on partitioning. Our algorithm is based on the key idea of partition refinement. The search starts with a single partition and as memory becomes scarce, partitions are refined using new components of the analysed system. Different heuristics have been proposed for selection of components during refinement steps. This scheme allows us to efficiently limit cross transitions at the cost of possibly generating unequal state distributions upon partitions compared to a partition function hashing the global state vector. However, our algorithm can still guarantee an upper bound on the size of any partition loaded in memory which previous approach like [11,15,18] could not.

We have extensively experimented with our approach in the context of the PART external-memory algorithm of [10]. The experiments have demonstrated the importance of exploiting model structure and the importance of making an informed selection of the state vector component used in a refinement step. We have experimentally demonstrated that a refinement based partitioning combined with the heuristics developed is able to outperform earlier approaches. Provided with access to the same amount of bounded RAM our approach clearly outperformed earlier approaches in that it could complete state space exploration without causing memory overflow. Our experimental evaluation also highlighted that the best choice of heuristic is model-dependent and hence it is desirable to further investigate the possibility of developing adaptive heuristics. Finally, we have demonstrated that a refinement based external memory algorithm does not have an intolerable overhead compared to a pure RAM algorithm.

Application to distributed state space exploration. Our framework and results are also relevant in the context of distributed memory algorithms relying on state space partitioning. However, our experiments raised several issues specific to that context and that shall be addressed in future works. First, the choice of the heuristic is still an open question. Heuristic DE was apparently the best regarding cross transitions, but may not be the most appropriate as it can generate unfair state distributions and consequently more refinements that imply the deletion (and revisit) of more states. Second, it remains to be investigated to which extent the non-uniform partitions obtained with our heuristic is an issue. It could be that assigning several small partitions to the same working process could solve that problem. As part of a future work, we therefore plan to experiment with our algorithm in a distributed setting and possibly explore heuristics specifically designed for that context. Furthermore, with our dynamic partitioning scheme cycles may span multiple partitions which our technique incompatible with techniques such as [25,26].

Application to safety property model checking. In this paper we have focused on the use of our refinement algorithm for state space exploration algorithms. A relevant question is then its compatibility with more general model checking algorithms. *Safety properties* forms a simple class of analysable properties which can be locally checked on each reachable state and is hence independent from the specific exploration algorithm used. Our refinement algorithm is therefore immediately applicable in safety model checkers. A difficulty is, however, related to *counter-example* production. If an erroneous state is discovered, the model checker should provide the user with an event sequence leading to the faulty state. In general, this can be done by associating with each state s a unique identifier and storing with s the identifier of one of its predecessor s' coupled with the *generating event* e for which $succ(s', e) = s$. The counter-example can then be constructed by recursively following these pointers up to the initial state. However, if there is no way to map an identifier to a partition (which is the case with our algorithm since the partitioning is constantly evolving) finding a state from its identifier could require to read all partition files which would make the reconstruction excessively slow. The solution we propose is instead to store in a single file a spanning tree rooted in the initial state and covering all states visited so far. This tree is stored as a sequence of pairs (idx^{pred}, e) where idx^{pred} is the index in the tree file of the predecessor state and e the identifier of the generating event. Finding a path to a specific state can then be done by going up in the spanning tree using indexes. The events on the path obtained are used to reconstruct states of the counter-example. Maintaining the tree file comes at the cost of an additional IO operation per state to store the predecessor reference and the event. Although we have not implemented this solution it is reasonable to assume that it would only have a marginal impact on performance.

Application to LTL model checking. A more difficult problem is the verification of properties expressed in *Linear Time temporal Logic* (LTL). Explicit state model checkers often reduces the LTL model checking problem to a graph problem: find a cycle in the synchronised graph (obtained through the composition of the state space and the buchi automaton of the LTL formula to check) containing at least one *accepting* state. State of the art algorithms, such as nested depth-first search [27] or algorithms based on SCC decomposition [28,29], rely on a depth-first search of the state space for discovering acceptance cycles. Our refinement technique can clearly not be used in conjunction with these algorithms as we assume some form of *delayed duplicate detection* (i.e., a state reached through event execution is not directly checked to be already visited) incompatible with depth-first search algorithms.

Our technique is however compatible with some IO efficient LTL model checking algorithms found in the literature and that often are adapted from distributed algorithms. They, as such, make use of delayed duplicate detection and do not rely on depth-first search traversal of the state space. As part of future work, we will focus on the integration of our technique with the OWCTY [30,31] and MAP [32,33] algorithms. For instance, the authors of [34] build on the OWCTY algorithm to define a distributed and external-memory algorithm for LTL model checking. It uses a hash function to assign states to working processes that should be replaceable by our dynamic partitioning function.

Acknowledgements

We are grateful for the constructive and detailed comments and references provided by the reviewers that have helped us to improve the paper.

References

[1] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.
[2] A. Valmari, The state explosion problem, in: Lectures on Petri Nets I: Basic Models, in: LNCS, vol. 1491, Springer, 1998, pp. 429–528.
[3] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems, in: LNCS, vol. 1032, Springer, 1996.
[4] G.J. Holzmann, State compression in SPIN: recursive indexing and compression training runs, in: SPIN'1997. Available via: http://spinroot.com/spin/Workshops/ws97/gerard.pdf, 1997.
[5] U. Stern, D.L. Dill, Parallelizing the Murphi verifier, in: CAV'1997, in: LNCS, vol. 1254, Springer, 1997, pp. 256–278.
[6] D. Nicol, G. Ciardo, Automated parallelization of discrete state-space generation, Journal of Parallel and Distributed Computing 47 (1997) 153–167.
[7] G. Ciardo, J. Gluckman, D. Nicol, Distributed state-space generation of discrete-state stochastic models, INFORMS Journal of Computing 10 (1996) 82–93.
[8] D.L. Dill, U. Stern, Using magnetic disk instead of main memory in the Murøverifier, in: CAV'1998, in: LNCS, vol. 1427, Springer, 1998, pp. 172–183.
[9] V. Holub, P. Tuma, Streaming state space: a method of distributed model verification, in: TASE'2007, IEEE Computer, 2007, pp. 356–368.
[10] T. Bao, M. Jones, Time-efficient model checking with magnetic disk, in: TACAS'2005, in: LNCS, vol. 3440, Springer, 2005, pp. 526–540.
[11] F. Lerda, R. Sisto, Distributed-memory model checking with SPIN, in: SPIN'1999, in: LNCS, vol. 1680, Springer, 1999, pp. 22–39.
[12] M. Westergaard, S. Evangelista, L. Kristensen, ASAP: an extensible platform for state space analysis, in: ATPN'2009, in: LNCS, vol. 5606, Springer, 2009, pp. 303–312.
[13] S. Evangelista, L.M. Kristensen, Dynamic state space partitioning for external memory model checking, in: International Workshop on Formal Methods for Industrial Critical Systems, FMICS, in: LNCS, vol. 5825, Springer, 2009, pp. 70–85.
[14] H. Garavel, R. Mateescu, I. Smarandache, Parallel state space construction for model-checking, in: SPIN'2001, in: LNCS, vol. 2057, Springer, 2001, pp. 217–234.
[15] F. Lerda, W. Visser, Addressing dynamic issues of program model checking, in: SPIN'2001, in: LNCS, vol. 2057, Springer, 2001, pp. 80–102.
[16] M. Bourahla, M. Benmohamed, Efficient partition of state space for parallel reachability analysis, in: AICCSA'2005, IEEE Computer Society, 2005, 21.
[17] R. Zhou, E.A. Hansen, Structured duplicate detection in external-memory graph search, in: AAAI'2004, AAAI Press/The MIT Press, 2004, pp. 683–689.
[18] M. Rangarajan, S. Dajani-Brown, K. Schloegel, D. Cofer, Analysis of distributed spin applied to industrial-scale models, in: SPIN'2004, in: LNCS, vol. 2989, Springer, 2004, pp. 267–285.
[19] G.J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[20] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, P. Šimeček, DiVinE — a tool for distributed verification (Tool Paper), in: Computer Aided Verification, in: LNCS, vol. 4144, Springer, 2006, pp. 278–281.
[21] T. Murata, Petri nets: properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.
[22] R. Pelánek, BEEM: benchmarks for explicit model checkers, in: SPIN'2007, in: LNCS, vol. 4595, Springer, 2007, pp. 263–267.
[23] S. Evangelista, Dynamic delayed duplicate detection for external memory model checking, in: SPIN'2008, in: LNCS, vol. 5156, Springer, 2008, pp. 77–94.
[24] R.E. Korf, Best-first frontier search with delayed duplicate detection, in: AAAI'2004, AAAI Press/The MIT Press, 2004, pp. 650–657.
[25] A.L. Lafuente, Simplified distributed LTL model checking by localizing cycles, Tech. Rep., Institute of Computer Science, Albert-Ludwings Universitt Freiburg, 2002.
[26] J. Barnat, L. Brim, I. Cerna, Property driven distribution of nested DFS, in: VCL'02, 1–10, 2002.
[27] C. Courcoubetis, M. Y. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithms for the verification of temporal properties, in: CAV'1990, in: LNCS, Springer, 1990, pp. 233–242.
[28] J.-M. Couvreur, On-the-fly verification of linear temporal logic, in: FM'1999, in: LNCS, vol. 1708, Springer, 1999, pp. 253–271.
[29] J. Geldenhuys, A. Valmari, Tarjan's algorithm makes on-the-fly LTL verification more efficient, in: TACAS'04, in: LNCS, vol. 2988, Springer, 2004, pp. 205–219.
[30] I. Cerná, R. Pelánek, Distributed explicit fair cycle detection (set based approach), in: SPIN'2003, in: LNCS, vol. 2648, Springer, 2003, pp. 49–73.
[31] J. Barnat, L. Brim, P. Simecek, I/O efficient accepting cycle detection, in: CAV'2007, in: LNCS, vol. 4590, Springer, 2007, pp. 281–293.
[32] L. Brim, I. Cerná, P. Moravec, J. Simsa, Accepting predecessors are better than back edges in distributed LTL model-checking, in: FMCAD'2004, in: LNCS, vol. 3312, Springer, 2004, pp. 352–366.
[33] J. Barnat, L. Brim, P. Simecek, M. Weber, Revisiting resistance speeds Up I/O efficient LTL model checking, in: TACAS'2008, in: LNCS, vol. 4963, Springer, 2008, pp. 48–62.
[34] J. Barnat, L. Brim, P. Simecek, Cluster-based I/O-efficient LTL model checking, in: ASE'2009, IEEE, 2009, pp. 635–639.