Dynamic State Space Partitioning for External Memory Model Checking*

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

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

Abstract. We describe a dynamic partitioning scheme usable by model checking techniques that divide the state space into partitions, such as most 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.

1 Introduction

Model checking [3] is a technique used to prove that finite-state systems match behavioral specifications. It is based on a systematic exploration of all reachable states in the search for illegal behaviors violating the specification. Despite its simplicity, its practical application is subject to the well-known state explosion problem [17]: 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 two families. The first family of techniques reduce 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 [9] which limits redundant interleavings is an example of such a technique. More pragmatic approaches do not reduce the state space, but make a more economical use of available resources, or augment them, in order to extend the range of problems that can be analyzed. State compression [11], distributed verification [16], and disk-based verification [4] 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., [4,10]). For example, in the distributed algorithm of [16], 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

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

M. Alpuente, B. Cook, and C. Joubert (Eds.): FMICS 2009, LNCS 5825, pp. 70–85, 2009.

sends it to its owner such that the state can be stored and its successor states can be explored. An important component of this algorithm is the *partition function* (known to all processes) which is used to map states to partitions. In the ideal case, the partition function should have two properties. Firstly, it should generate as few *cross transitions* as possible. Cross transitions link two states of different partitions and thus systematically generate messages over the network. Secondly, it should distribute states evenly into partitions to ensure that all processes have the same workload. A hash function based on the bit string used to represent states may achieve an optimal distribution, but generates many cross transitions due to the insensitivity of hashing to locality.

To address this problem, we introduce 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 over time we introduce the idea 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 i^{th} 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.

Our intuition is to take advantage of the fact that events typically modify a small number of components in the state vector. 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 this does not have any 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 [18] tool, on top of the external algorithm of [1]. 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 [1] 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 [16] and [1] 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. We assume the reader is familiar with the principle of state space exploration.

Definitions and notations. 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:

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

2 Partitioning the State Space

Algorithm 1 (left) shows the algorithm of [1] that mimics a distributed search using external storage, and the distributed algorithm of [16] (right) which is the basis of most work in the field of parallel and distributed model checking. Both algorithms 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 (ll. 1–10), 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 $part(s_0)$ (l. 4). Then, as long as one of the queues contains a state, the algorithm selects the longest queue i (l. 6), loads the associated partition from disk file $\mathcal{F}_i$ to memory in $\mathcal{V}_i$ (l. 7) and starts expanding the states in queue $\mathcal{Q}_i$ using procedure $search_i$ (l. 8) 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 $\mathcal{V}_i$ (ll. 9–10). Selecting the longest queue 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 (ll. 11–19), 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 (ll. 16–17). As long as termination is not detected (l. 18), the process expands the states in its local queue using procedure $search_i$ (l. 19). Termination occurs when all queues and communication channels are empty.

The common part of both algorithms is the $search_i$ procedure that expands all the states queued in $\mathcal{Q}_i$ until it becomes empty. Each state s removed from $\mathcal{Q}_i$ (l. 22) 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 $\mathcal{V}_i$ (l. 24) and then expanded (ll. 25–29). During the expansion, we compute all the successors s' of

Algorithm 1. Two search algorithms based on state space partitioning

```
1:  (* external algorithm of [1] *)
2:  for i in 1 to N do
3:      Q_i := ∅ ; V_i := ∅ ; F_i := ∅
4:  Q_{part(s_0)}.enqueue(s_0)
5:  while ∃i : ¬Q_i = ∅ do
6:      i := longestQueue()
7:      F_i.load(V_i)
8:      search_i()
9:      V_i.unload(F_i)
10:     V_i := ∅

11: (* distributed algorithm of [16] *)
12: execute proc_1 ‖ ... ‖ proc_N
13:
14: procedure proc_i is
15:     Q_i := ∅ ; V_i := ∅
16:     if part(s_0) = i then
17:         Q_i.enqueue(s_0)
18:     while ¬ termination() do
19:         search_i()

20: procedure search_i is   (* search procedure common to both algorithms *)
21:     while Q_i ≠ ∅ do
22:         s := Q_i.dequeue()
23:         if s ∉ V_i then
24:             V_i.insert(s)
25:             for e in en(s), s' = succ(s, e) do
26:                 j := part(s')
27:                 if i = j then    (* local transition *)
28:                     if s' ∉ V_i then Q_i.enqueue(s')
29:                 else  Q_j.enqueue(s')  (* cross transition *)
```

s and determine the partition j they belong to (l. 26), using 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 the 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 [1] this is implemented by enqueueing 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 [16] it implies to pack 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. Although this objective is more specific to the distributed algorithm, function *part* should also distribute states evenly among partitions so that processes receive a comparable workload.

3 Related Work

Stern and Dill [16], Bao and Jones [1] and Garavel et al. [8] 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 [12]. Assuming that the system to be verified is a set of communicating processes, the partition function proposed in [12] 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 in 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) and, hence, the execution time (up to a factor of 3). The downside is a degraded distribution of states over the nodes of the network.

The dynamic partitioning in [13] 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 [12]. The results of this dynamic partitioning strategy in term of message exchanges and verification time are comparable to the ones of [12]. 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 Sect. 6) a good distribution of states.

An efficient partitioning algorithm based on abstraction and refinement of the state space is introduced in [2]. However, the state space has to be first constructed in order to define the partition function meaning that this approach mainly targets off-line model checking.

In structured duplicate detection [19] 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 the context of model checking due to the difficulty of defining an abstract graph from a complex specification. Close to that idea is the work of Rangarajan et al. [15]. 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 [12] and [13]: only when the selected variable is modified can a cross transition be generated. The experiment made in [15] shows that this method can significantly outperform the local hash partitioning implemented in PSPIN [12].

We propose a way to dynamically (i.e., during state space exploration) modify the partition function by progressively taking into account more components of the underlying system. Our work can be seen as an extension of [15] since some of the ideas we develop were briefly mentioned in [15] – 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 still guarantee an upper bound on the size of any partition loaded in memory which previous approach like [12,13,15] could not. From now on, we focus on the external algorithm of [1] although the proposed method and the heuristics in Sect. 5 can, to a large extent, be applied also to distributed model checking.

Algorithm 2. The partition refinement procedure

```
1: procedure refine_i is
2:     update partition function part: partition i is divided into i_1, ..., i_n
3:     for s ∈ V_i do { s.write(F_part(s)) }
4:     while ¬ Q_i.isEmpty() do { s := Q_i.dequeue() ; Q_part(s).enqueue(s) }
5:     V_i := ∅ ; F_i := ∅ ; go to line 6 of Algorithm 1
```

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. 2 is triggered. It firsts updates the partition function *part* (l. 2) in such a way that each state that was previously mapped to partition i is now mapped to a new partition $j \in \{i_1, \ldots, i_n\}$. Then, it writes the states in $\mathcal{V}_i$ to disk files $\mathcal{F}_{i_1}, \ldots, \mathcal{F}_{i_n}$ (l. 3) and reorganizes the queue $\mathcal{Q}_i$ in the same way (l. 4). Once this reorganization is finished, the table $\mathcal{V}_i$ and the disk file $\mathcal{F}_i$ are emptied (l. 5), and the search can restart by picking a new partition. Note that partition i is the only one to be reorganized; 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 analyzed.

A compositional partition function can be represented as a *partitioning diagram*. Figure 1 is the graphical representation of a diagram D. Rounded boxes represent terminal nodes and branching nodes are drawn using circles. The nodes are labeled 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 codomain can be deduced from the labels of its outgoing arcs: $g : \mathcal{S} \to \{t, f\}$, $h : \mathcal{S} \to \{a, b, c\}$ 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 labeling 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:

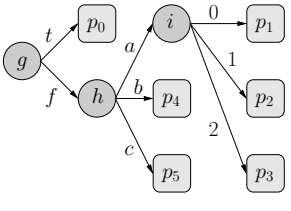

Fig. 1. A compositional partitioning diagram D

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

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 formalizes the notion of partitioning diagrams. Note that the definition of the edge set E implies that partitions are the terminal nodes of the diagram and that functions are branching nodes.

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

- *$G = (V, E)$ 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;*
- *$\mathcal{F} = \{f_i : \mathcal{S} \to \mathcal{L}_i\}$ is a set of branching functions;*
- *$\mathcal{P} \subseteq 2^{\mathcal{S}}$ is a set of state partitions;*
- *$E \subseteq \mathcal{F} \times \mathcal{L} \times V$ (with $\mathcal{L} = \cup_i \mathcal{L}_i$), 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 formalized 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 labeling the new branching node.

Our refinement algorithm assumes the global system can be viewed as a set of distinct *components* $C_1 \in \mathcal{D}_1, \ldots, C_n \in \mathcal{D}_n$ and that a state of the system is obtained from the state of these components, i.e., $\mathcal{S} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$. This naturally capture systems with a statically defined state vector (e.g., DVE systems [5], Petri nets). 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 from a given state s returns the value of component C_i. During the refinement of partition p, the partition diagram is modified as follows. The algorithm first inspects the diagram to determine the functions 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. At last, p_i 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}.

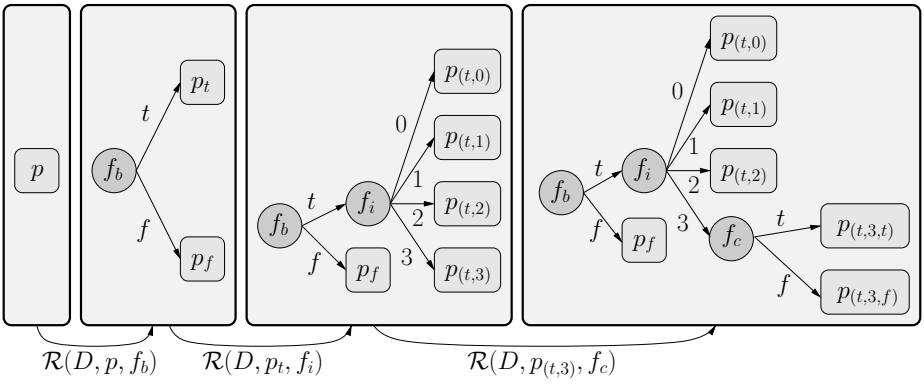

Fig. 2. A dynamic compositional diagram D

Figure 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: $b \in \{t, f\}$, $c \in \{t, f\}$ 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. Hence, 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 formalizes 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 $f : \mathcal{S} \to \mathcal{L}_f$ and $p \in \mathcal{P}$ is the CPD $D' = (V', E', r'_0, \mathcal{F}', \mathcal{P}')$ with:*

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

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 minimize 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 efficiently select components to be used as a basis for the refinement. We classify these in two categories. *Static 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 *dynamic heuristics*, the component selected is chosen during the refinement step on the basis of data collected on-the-fly during the search.

Static 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 component, the number of events that modify it and order components accordingly in increasing order. Some weights may also be associated with events as we did in our implementation for the DVE language. For example, events nested in loops can be assigned a high weight as we can reasonably assume that their execution will occur frequently.

Heuristic **SS***: Static 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., initialization events, whereas some will generate most of the state transitions. With heuristic SS, we attempt 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 randomized search in order to explore 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).

Dynamic Heuristics

Heuristic **DR***: Dynamic Randomized* 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 dynamic strategies below.

Heuristic **DE***: Dynamic Event execution* Heuristic DE is the dynamic equivalent of the heuristic SA: the 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 **DD***: Dynamic 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 DD, the refinement procedure simulates all possible refinements by computing for each state s of the partition to be refined the values $f_{C_{i_1}}(s)$, ..., $f_{C_{i_k}}(s)$, 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, 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 DE and DDE (that extends DE with this "simulation" process, see below) exhibited comparable performances in term of disk accesses, the execution times were roughly the same.

Heuristic **DDE***: Dynamic Distribution and Event execution.* This last heuristic combines the idea of heuristics DD and DE: 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:

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

where $updates[i]$ is the number of modifications of component i recorded so far and $std(C_i)$ is the standard deviation in the sizes of sub-partitions obtained if component C_i is chosen for refinement. The algorithm picks the candidate having the lowest value.

6 Experiments

The PART algorithm of [1] as well as our dynamic partitioning technique have been implemented in the ASAP model checking platform [18]. We report in this section on experimental results obtained with this implementation. Additional data from the experiments can be found in [7].

Application of refinement to DVE systems. All models we have used are written in the DVE language and comes from the BEEM database [14]. 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. In the DVE language, the system is described as a set of automata synchronizing through communication channels and global variables. Communications 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; a variable (global or local); and the content of a communication channel. Arrays were considered as components although this obviously was not a good solution in some cases. We plan to refine that in a future implementation. 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).

Experimental context. Apart from our refinement technique, we also implemented the static and dynamic partitioning schemes of [12] and [13] both 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 [13], a partition is split in two sub-partitions when it exceeds memory capacity: half of the classes that comprises the partition are put in a new partition. The process used for hashing was selected after an initial sampling of the state space. We selected the process which achieved both a uniform state distribution and a low number of cross transitions using heuristic h in Equation 1 from the previous section. The initial sampling was stopped after 100,000 states had been visited. This represents from 10% of the state space to less than 0.2% for the largest instances.

We experimented with PART using different partitioning schemes on 35 instances of the BEEM database having from $1{\cdot}10^6$ to $60{\cdot}10^6$ states. Since instances of the same model often have similar state spaces, we only kept for each model the instance with the largest state space. During each run we gave the PART algorithm the possibility to keep in memory at most 2% of the state space[1]. Half of this amount was given to the memory buffer of the state queue (remember that PART stores the queue on disk) and half was given to the partition loaded in memory. Hence, each partition could contain at most 1% of the total state space size. With static partitioning, it is impossible to put an upper bound on

[1] Other sizes were experimented: 10%, 5% and 1%. Due to lack of space, these experiments have been left out in this section, but can be found in [7].

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 1% 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 [13]. As noted earlier, the algorithm of [13] cannot guarantee an upper bound on a partition size: when a partition contains a single class it cannot be further reorganized.

Experimental results. Table 1 shows the result of our experiments. Due to lack of space, we only report the data for 14 representative instances, but still provide the average over the 35 instances experimented[2]. We performed 10 runs per instance, each with a different partitioning strategy. Each column provides data for a single run. For static and dynamic settings, GHC stands for "Global Hash Code": the partition function is the global hash function modulo the number of partitions; and LHC stands for "Local Hash Code": only the part of the state vector corresponding to a specific process is hashed, that is, the algorithms of [12] (in the static setting) and [13] (in the dynamic setting). Dynamic + Compositional is our refinement algorithm with the different heuristics proposed in Section 5: SA (Static + Analysis), SS (Static + Sample), DR (Dynamic + Randomized), DE (Dynamic + Event execution), DD (Dynamic + Distribution) and DDE (Dynamic + Distribution and Event execution). For heuristic SS, we performed exactly the same preliminary search as the one performed for strategies with LHC: we stopped the search after the visit of 100,000 states. For each instance, rows CT and IO provide the number of cross transitions and disk accesses performed (both for the queue and for disk partitions). Absolute values are given for column Static - GHC. Other values are relative to this one. The ϵ symbol is used to denote values less than 0.001. Best values have been highlighted in bold.

We first observe that, for most instances, heuristic DR performs worse than other heuristics. We found no instances where heuristic DR generated fewer cross than heuristics SS and DE. This confirms our initial intuition on the impact of the candidate's choice made during refinement.

Heuristic SS and heuristic DE (the dynamic equivalent of SS) exhibit comparable performances. This indicates that the preliminary randomized search often provides a very good sample of the state space. We only observed a notable difference for a few instances, especially the largest ones for which the sample was too small to be representative enough (e.g., `train-gate.7` and `collision.4`).

In [6], 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 [16]. To illustrate that, let us consider the extreme case where the graph is a long sequence of states. Using a good hash function we can assume the probability of a transition to be a cross transition is close to $\frac{N-1}{N}$ (where N is the number of partitions). Hence, with this state space structure, most transitions will immediately be followed by a partition swap. With a distributed algorithm, the search will consist of

[2] The complete table can be found in [7].

Table 1. Performance of PART with different partitioning schemes

	Static		Dynamic		Dynamic + Compositional					
	GHC	LHC	GHC	LHC	SS	SA	DR	DE	DD	DDE
`bopdp.3`					*1,040,953 states*				*2,747,408 transitions*	
CT	2.7 M	0.091	0.965	**0.078**	0.223	0.300	0.311	0.183	0.256	0.306
IO	39 M	**0.148**	1.008	0.189	0.311	0.243	0.324	0.370	0.323	0.304
`brp.6`					*42,728,113 states*				*89,187,437 transitions*	
CT	88 M	0.281	0.899	0.277	**0.040**	0.083	0.286	0.042	0.170	0.049
IO	5.9 G	0.346	1.057	0.292	0.132	0.130	0.979	0.123	**0.046**	0.082
`collision.4`					*41,465,543 states*				*113,148,818 transitions*	
CT	112 M	0.088	0.969	0.087	0.078	0.030	0.255	**0.011**	0.131	0.056
IO	1.5 G	0.183	1.135	0.235	0.178	0.220	0.395	**0.176**	0.211	0.294
`firewire_link.5`					*18,553,032 states*				*59,782,059 transitions*	
CT	59 M	0.262	0.981	0.254	0.054	0.050	0.173	**0.010**	0.346	0.017
IO	788 M	2.282	0.971	0.869	0.224	**0.190**	0.488	0.220	0.715	0.206
`firewire_tree.5`					*3,807,023 states*				*18,225,703 transitions*	
CT	18 M	0.111	0.983	0.109	0.114	0.190	0.153	**0.065**	0.195	0.138
IO	141 M	**0.177**	0.969	0.461	2.148	0.323	0.665	0.757	0.248	0.287
`fischer.6`					*8,321,728 states*				*33,454,191 transitions*	
CT	33 M	0.109	0.966	**0.107**	0.474	0.470	0.629	0.474	0.683	0.468
IO	130 M	**0.478**	1.221	0.547	0.896	0.915	0.980	0.874	0.855	0.840
`iprotocol.7`					*59,794,192 states*				*200,828,479 transitions*	
CT	196 M	0.276	0.958	0.152	**0.003**	0.114	0.319	0.004	0.170	0.021
IO	2.6 G	1.390	1.090	0.634	**0.190**	0.220	1.383	0.209	0.784	0.211
`msmie.4`					*7,125,441 states*				*11,056,210 transitions*	
CT	10 M	0.048	0.852	**0.047**	0.204	0.794	0.528	0.211	0.448	0.253
IO	97 M	**0.315**	0.925	0.419	0.603	1.013	0.739	0.545	0.797	0.556
`pgm_protocol.8`					*3,069,390 states*				*7,125,121 transitions*	
CT	6.3 M	0.273	0.932	0.268	**0.024**	0.110	0.208	**0.024**	0.286	0.025
IO	255 M	0.447	0.800	0.303	**0.100**	0.145	0.373	**0.100**	0.185	0.102
`plc.4`					*3,763,999 states*				*6,100,165 transitions*	
CT	6.0 M	0.018	0.985	0.017	ϵ	ϵ	0.073	ϵ	0.104	0.001
IO	1.3 G	0.085	1.251	0.107	0.030	**0.018**	0.393	0.030	0.110	0.020
`rether.7`					*55,338,617 states*				*61,198,113 transitions*	
CT	60 M	0.040	0.980	**0.039**	0.042	0.049	0.183	0.051	0.106	0.093
IO	3.7 G	0.170	1.150	0.244	0.151	**0.128**	0.383	0.164	0.198	0.217
`synapse.7`					*10,198,141 states*				*19,893,297 transitions*	
CT	19 M	0.301	0.970	0.297	0.014	0.012	0.238	**0.010**	0.451	0.015
IO	161 M	0.792	1.096	0.778	0.372	0.396	0.659	**0.325**	0.768	0.369
`telephony.7`					*21,960,308 states*				*114,070,470 transitions*	
CT	111 M	0.245	0.976	**0.239**	0.450	0.450	0.495	0.447	0.708	0.505
IO	619 M	0.838	1.110	0.854	0.958	0.972	1.272	0.977	**0.715**	1.263
`train-gate.7`					*50,199,556 states*				*106,056,460 transitions*	
CT	105 M	0.028	0.976	**0.027**	0.359	0.270	0.566	0.212	0.783	0.224
IO	1.7 G	**0.105**	1.332	0.142	0.392	0.498	1.771	0.318	0.751	0.321
Average on 35 models										
CT	1.000	0.255	0.962	0.236	0.163	0.206	0.327	**0.152**	0.419	0.179
IO	1.000	0.504	1.050	0.496	0.458	0.423	0.661	0.411	0.531	**0.393**

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 instances `brp.6`, `iprotocol.7`, `pgm_protocol.8`, `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 Static - GHC and Dynamic - GHC with respect to both cross transitions and disk accesses. Also, except for `rether.7`, compositional partitioning performs significantly better than partitioning based on a local hash code.

For `firewire_tree.5`, heuristic DE generates, after refinements, unfair state distributions which leads to most time being spent on reorganizing partitions. Thus, although it is the one that generates the fewest cross transitions, this has little consequences on the overall number of disk accesses. Heuristics DD and DDE that try to distribute states equally among partitions largely outperforms DE on that instance. This observation can be generalized to most instances experimented: heuristic DE is the one that minimizes cross transitions, but not necessarily disk accesses. It would be interesting to experiment with the two heuristics in a distributed environment. In [13] it is advised to delete states after a reorganization 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 DE performs more refinements it should cause the deletion and revisit of more states than DDE and, hence, generate more cross transitions and message exchanges. It is therefore not immediately clear which one should be preferred in a distributed setting.

Although we see some correlation between the number of cross transitions and disk accesses, this is not always the case. Firstly, for the reason that explains the bad performances of heuristic DE for instance `firewire_tree.5`: disk accesses are also triggered by partition refinements. Secondly, 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 guarantee that partition j will eventually be loaded in memory. All subsequent cross transitions do not affect the algorithm. Hence, a large number of cross transitions linking two partitions is not necessarily a bad thing.

Synchronizations in models `telephony` and `fischer` are realized through global arrays modified by most events. As the refinement procedure currently implemented considers arrays as single components, our algorithm is not really efficient in these cases. A better management of arrays should improve this. This remark applies to most mutual exclusion algorithms we have experimented with.

Table 1 indicates that our refinement algorithm outperforms the partitioning algorithms of [12] and [13] although only slightly. However, the experiment reported here is quite unfair to our algorithm as no memory limit was (and could be) given to strategies Static - LHC and Dynamic - LHC whereas our refinement algorithm works within a bounded amount of RAM. Table 2 gives for all instances of Table 1 and for these two partitioning schemes, the proportion

Table 2. Ratio of overflowing states (given by Eq. 2) with static (S-LHC) and dynamic (D-LHC) partition functions of [12] and [13] based on local hash code

	S-LHC	D-LHC
bopdp.3	0.677	0.677
brp.6	0.735	0.735
collision.4	0.722	0.722
firewire_link.5	0	0
firewire_tree.5	0.785	0.785
fischer.6	0.969	0.969
iprotocol.7	0	0

	S-LHC	D-LHC
msmie.4	0.939	0.939
pgm_protocol.8	0	0
plc.4	0	0
rether.7	0.550	0.192
synapse.7	0.090	0.035
telephony.7	0.827	0.827
train-gate.7	0.950	0.950

$$\sum_{p \in \text{partitions}} \frac{\max(\text{size of partition } p - \text{ memory limit per partition}, 0)}{\text{state space size}} \tag{2}$$

of overflowing states (see Eq. 2 of Table 2) where, again, the memory limit per partition that was given to our algorithm is 1% of the total number of states.

When the algorithm could stay within allowed memory, LHC based partitioning is clearly outperformed by a refinement based partition function. This is evidenced by Table 1 showing that for models `firewire_link.5`, `iprotocol.7`, `pgm_protocol.8`, `plc.4` and `synapse.7`, refinement based partitioning generates — sometimes considerably — fewer cross transitions and disk accesses. In contrast, 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 `fischer.6`, `msmie.4` and `train-gate.7`.

7 Conclusions and Future Work

We have proposed in this paper a dynamic partitioning algorithm for external and distributed model checking, and extensively experimented with the disk-based algorithm of [1]. 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 analyzed system. Different heuristics have been proposed to appropriately select 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 [12,13,15] could not. In addition to this, we have presented a common framework for external and distributed algorithms based on partitioning.

Our framework and results are also valid in the context of distributed memory verification. However, 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. As part of a future work, we therefore plan to explore heuristics specifically designed for a distributed context.

References

1. Bao, T., Jones, M.: Time-Efficient Model Checking with Magnetic Disk. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 526–540. Springer, Heidelberg (2005)
2. Bourahla, M., Benmohamed, M.: Efficient Partition of State Space for Parallel Reachability Analysis. In: AICCSA 2005, p. 21. IEEE Computer Society, Los Alamitos (2005)
3. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
4. Dill, D.L., Stern, U.: Using Magnetic Disk Instead of Main Memory in the Murϕ Verifier. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 172–183. Springer, Heidelberg (1998)
5. DVE Language, `http://divine.fi.muni.cz/page.php?page=language`
6. Evangelista, S.: Dynamic Delayed Duplicate Detection for External Memory Model Checking. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 77–94. Springer, Heidelberg (2008)
7. Evangelista, S., Kristensen, L.M.: Dynamic State Space Partitioning for External and Distributed Model Checking. Technical report, DAIMI – Aarhus University (2009), `http://www.cs.au.dk/~evangeli/doc/ss-partitioning.pdf`
8. Garavel, H., Mateescu, R., Smarandache, I.: Parallel State Space Construction for Model-Checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 217–234. Springer, Heidelberg (2001)
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
10. Holub, V., Tuma, P.: Streaming State Space: A Method of Distributed Model Verification. In: TASE 2007, pp. 356–368. IEEE Computer, Los Alamitos (2007)
11. Holzmann, G.J.: State Compression in Spin: Recursive Indexing and Compression Training Runs. In: SPIN 1997 (1997)
12. Lerda, F., Sisto, R.: Distributed-Memory Model Checking with SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 22–39. Springer, Heidelberg (1999)
13. Lerda, F., Visser, W.: Addressing Dynamic Issues of Program Model Checking. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 80–102. Springer, Heidelberg (2001)
14. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
15. Rangarajan, M., Dajani-Brown, S., Schloegel, K., Cofer, D.D.: Analysis of Distributed Spin Applied to Industrial-Scale Models. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 267–285. Springer, Heidelberg (2004)
16. Stern, U., Dill, D.L.: Parallelizing the Murphi Verifier. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 256–278. Springer, Heidelberg (1997)
17. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
18. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: ATPN 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009)
19. Zhou, R., Hansen, E.A.: Structured Duplicate Detection in External-Memory Graph Search. In: AAAI 2004, pp. 683–689. AAAI Press/The MIT Press (2004)