State Compression Based on One-Sided Communications for Distributed Model Checking

Camille Coti, Sami Evangelista and Laure Petrucci
LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité
99, av. J.-B. Clément, 93430 Villetaneuse, France
{camille.coti,sami.evangelista,laure.petrucci}@lipn.univ-paris13.fr

***Abstract*—We propose a distributed implementation of the collapse compression technique used by explicit state model checkers to reduce memory usage. This adapatation makes use of lock-free distributed hash tables based on one-sided communication primitives provided by libraries such as OpenSHMEM. We implemented this technique in the distributed version of the model checker Helena. We report on experiments performed on the Grid'5000 cluster with an implementation over OpenMPI. These reveal that, for some models, this distributed implementation can altogether preserve the memory reduction provided by collapse compression and reduce execution times by allowing the exchanges of compressed states between processes.**

I. Introduction

Model checking [2] is an automatic verification technique for finite state systems. It consists, in its most basic form, of an exhaustive exploration of a directed graph capturing the system behaviour (*i.e.* its state space) in search for executions violating its expected properties. The nodes of this graph are system states (*e.g.* a memory snapshot in case of a concurrent program) and its edges represent system changes (*e.g.* a statement execution). The main obstacle to the use of this technique on real life systems is the size of this state space graph that can grow exponentially with respect to the number of components in the system (*e.g.* threads). This phenomenon is known as the *state space explosion problem* [17].

Various methods have been proposed to alleviate this problem. Distributing the verification [16], [13] on a cluster is one of them. By benefiting from the aggregate memory and computational power of a machine network it is then possible to analyse larger models and/or reduce exploration times.

The state-of-the-art algorithm [16] upon which most distributed verification algorithm are built distributes the search by partioning the state space among participating processes. A partition function maps state vectors (*i.e.* bit strings encoding states) to processes. Each process is then responsible of any state that is assigned to it: it stores it in a private state table, generates its successors and sends them to their owners that will later process these states in the same way.

Another possibility to tackle the state space explosion problem is to make use of compression techniques such as tree [12] or automaton-based [10] compression in order to compute memory efficient representations of state spaces that could otherwise not fit in the available memory.

Collapse compression described in [16] and initially implemented in the Spin model checker [9] is another alternative. This compression technique is based on the use of hash tables storing local state spaces of components (*e.g.* processes) constituting the analysed system. Global system states can then be represented as integer vectors containing references to component states in their respective hash tables.

Collapse compression is fully compatible with state of the art distributed algorithms since compression operations can remain purely local using private hash tables. However, in such a setting, contents of hash tables may differ among processes, meaning that only full states can be sent through the network as a compressed state is only meaningful for the process that computed it.

In this article we introduce a distributed version of the collapse compression, meaning that the compression operations are made in a distributed and concurrent way and involve all verification processes. We design for this a distributed partitioned hash table structure adapted from [11] and based on RDMA (Remote Direct Memory Access) operations as those provided by the OpenSHMEM specification. They allow for one-sided communications where a process can read and write in another process's memory, thus being the sole actor of the communication.

This method allows for consistent representations of compressed system states among processes and, consequently, compressed states can be exchanged through the network. Hence, it features reduced communication times, and saves memory for communication buffers. We have implemented distributed collapse compression in our model checker Helena [7] and we show through a series of experiments that, for some models, it can significantly reduce exploration times while achieving the same memory reduction as using a non-distributed collapse compression based on private hash tables.

This article is organised as follows. We give in Section II the elements necessary for the understanding of our method. Our distributed compression method is described in Section III. Some variations and optimisations of this method are then given in Section IV. Experimental results are presented in Section V and Section VI concludes our work and gives some perspectives.

II. Background

We review in this section the principle of (distributed) state space exploration, the collapse compression technique and the distributed memory model used in this paper.

```
1: procedure exploreSequential is
2:    Q.init(s0);R.init(s0)
3:    while ¬Q.isEmpty() do
4:       s := Q.remove()
5:       for s' ∈ succ(s) do
6:          if ¬s'.checkInvariant() then
7:             halt and report error
8:          else if ¬R.isIn(s') then
9:             Q.insert(s');R.insert(s')
```

Fig. 1. Sequential state space exploration

A. Sequential and distributed state space exploration

Model checking by state space exploration explores all possible states of the system until it finds a counterexample of the property to be verified, or all states have been explored and thus the property is valid. If it can explore all possible states without finding a counterexample, it concludes that the property is always verified by the system. Therefore, it is of major importance to use an efficient algorithm for this state space exploration.

In this paper we assume a universe of system states S, an initial state $s_0 \in S$ and a mapping $succ : S \to 2^S$, that, from one state s, gives its set of successors. We want to explore the state space induced by these parameters, *i.e.* the smallest set $R \subseteq S$ of reachable states defined inductively as : $s_0 \in R \wedge (s \in R \Rightarrow succ(s) \in R)$.

A sequential state space exploration algorithm usable for checking invariant properties of states is shown in Figure 1. It operates on a queue Q of unexplored states and incrementally builds the reachability set R. Both initially contain the initial state. States are then taken from Q (l. 4), their successors generated and put in R and Q (if not seen before) to be later processed (loop at ll. 5–9). The algorithm terminates when an erroneous state is found (ll. 6–7) or when the queue has been emptied, which is guaranteed to happen for finite-state systems.

The distributed algorithm of [16] that represents the core of many distributed algorithms is given in Figure 2. P exploration processes are used (l. 2). Each process i owns a local portion of the queue and the reachable states. The state space is partitioned among processes using a state hash function. Each exploration process basically acts as the sequential algorithm presented above except that when a state s' is reached, the process checks if it is the owner of this state (condition at l. 8). In that case, it is processed as in the sequential scenario. Otherwise it is sent to its owner and discarded by the current process. Similarly, only the owner of the initial state puts it in its local data structures (ll. 13–14). Processes also have to check for incoming messages (ll. 16–19). A state received is handled as would be any other new state owned by the process (*i.e.* ll. 18–19 and ll. 10–11 match).

Termination detection (not shown in the algorithm) is triggered by a unique process (*e.g.* node 0) when this one has been idle (*i.e.* it does not receive any message and its queue is empty) for some amount of time. It then asks its peers if they are in the same situation and if all channels are empty

```
1: procedure exploreDistributed() is
2:    launch explore0 || ... || exploreP−1
3: procedure processQueuei() is
4:    s := Q.remove()
5:    for s' ∈ succ(s) do
6:       if ¬s'.checkInvariant() then
7:          halt and report error
8:       else if s'.hash() mod P ≠ i then
9:          s'.sendTo(s'.hash() mod P)
10:      else if ¬R.isIn(s') then
11:         Q.insert(s');R.insert(s')
12: procedure explorei() is
13:   if s0.hash() mod P = i then
14:      Q.insert(s0);R.insert(s0)
15:   while ¬termination() do
16:      if stateReceived() then
17:         s := receiveState()
18:         if ¬R.isIn(s) then
19:            Q.insert(s);R.insert(s)
20:      if ¬Q.isEmpty() then
21:         processQueuei()
```

Fig. 2. Distributed state space exploration [16]

(check made by counting messages sent and received) before notifying termination to other nodes if both conditions are met.

B. Collapse compression

Collapse compression (or state collapsing) [8] assumes the system to be analysed is composed of a set of N components of which the domains are $C_1, \ldots, C_N$. A system state (or *global state*) is then an item $\langle c_1, \ldots, c_n \rangle \in C_1 \times \ldots \times C_N$, with $c_1, \ldots, c_n$ being the *local states* of components $C_1, \ldots, C_n$ respectively. For instance, in the case of a parallel system, components can be processes (*i.e.* program counters with local variable contents), communication channels and global variables. We define the *local state space* of a component C_i, $i \in \{1, \ldots, N\}$ as: $R_i = \bigcup_{s \in R} \{s[i]\}$ where $s[i]$ denotes the i$^{\text{th}}$ item of tuple s.

The principle of state collapsing is to store each local state space R_i in a separate hash table H_i. A global state inserted in the hash table H storing the system state space then consists of a tuple $\langle idx_1, \ldots, idx_N \rangle$ such that each idx_i points to an item of the local hash table H_i. Local state spaces are not known *a priori* but incrementally built by the exploration algorithm similar to the construction of set R in the algorithms of Figures 1 and 2.

This encoding is illustrated by Figure 3. The model analysed is a two processes system synchronising through global shared variables. We decompose this system into three components: one constituted by global variables (C_1) and one for each process (C_2 and C_3). Local component state spaces are stored in three hash tables H_1, H_2 and H_3. The items of H_2 and H_3 contain program counters (pc) and local process variables. The system state space contains 6 states obtained through various combinations of 3, 2 and 2 local states of components C_1, C_2 and C_3 respectively.

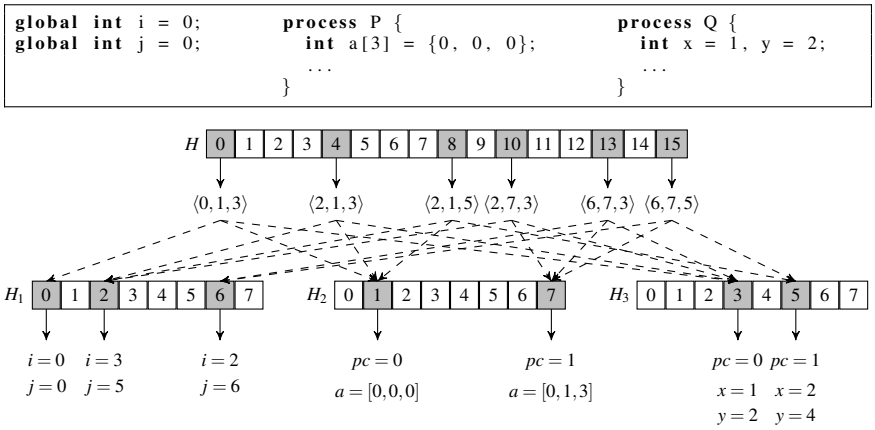


Fig. 3. State space of a two processes system encoded with the collapse compression

Figure 3 illustrates that the efficiency of state collapsing is conditionned by two factors. First local states must be shared (*i.e.* referenced) by many global states. This is usually the case, since state space explosion originates mostly from the combination of small components state spaces. Second, the definition domain of components must be larger than their local state spaces. Otherwise, if $|C_i| = |R_i|$ for some component i, then storing an index to an item of H_i consumes at least as much as memory as the item itself.

In this paper we assume local state spaces are stored in fixed-size linear hash tables. This means that the user has to provide the model checker with an upper bound on the local state space sizes. The recursive indexing method [8] has been designed to alleviate this requirement.

C. RDMA architectures and the OpenSHMEM specification

Our goal in this paper is to propose an implementation of the collapse compression technique using distributed hash tables and usable on a cluster of machines connected via RDMA devices. We therefore give in the remainder of this section a brief presentation of the one-sided communication model we use, and its implementation in OpenSHMEM.

RDMA and one-sided communications: RDMA is a communication mechanism that implements one-sided interprocess communication. It relies on two basic communication primitives: `put()` and `get()`. A process can read (`get()`) and write (`put()`) in another process's memory. In practice, not all the process's memory can be reached from other processes, but only a specific, *public* area.

An attractive feature of one-sided communications is that only the process that initiates the communication needs to take active part in it. The process that owns the memory area it is reading from or writing into is not participating to the communication, nor is it even aware that this communication is happening.

Fast cluster interconnection networks such as InfiniBand implement RDMA communications with zero-copy, meaning that the NIC (Network Interface Card) transfers data directly from one process's memory into the other process's memory without involving its operating system.

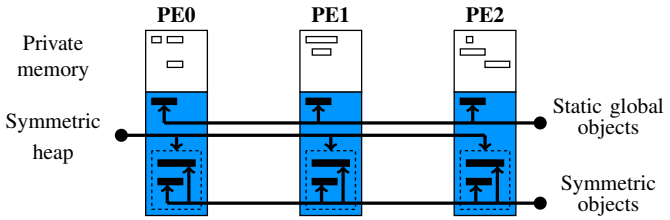


Fig. 4. OpenSHMEM memory model.

The OpenSHMEM communication and memory model: OpenSHMEM is a parallel programming interface that defines a set of one-sided, RDMA communication routines, aiming specifically at clusters featuring low-latency networks [1]. The processes are called *Processing Elements*. Each PE has its own (private) memory, and exhibits a public heap. One particularity of OpenSHMEM is that this heap is *symmetric*: every PE has a shared heap of the same size and that contains the same allocated objects and static global objects (Figure 4).

Symmetry is maintained between shared heaps through the use of dedicated memory management routines (*e.g.* `shmem_malloc()` and `shmem_free()`). The OpenSHMEM specification states that these routines are *collective* routines and must end by something semantically equivalent to a barrier. Hence, every object is allocated at the same offset from the beginning of the buffer on all the PEs [5]. Besides, global static variables are also located in the shared heaps and therefore remotely accessible.

The OpenSHMEM specification also defines some other interfaces for collective operations, locks and atomic remote accesses (such as fetch-and-add or compare-and-swap), which are of particular interest for our distributed state compression technique.

III. Distributed state collapsing based on one-sided communications

A. Motivations

Collapse compression does not depend on a specific exploration algorithm and is therefore fully compatible with the distributed algorithm of Figure 2. This requires each PE to

possess a local hash table for each component, but in practice the memory used by these is low compared to the global hash table H. Hence, a direct use of state collapsing provides the same memory reduction as in the sequential case.

Nevertheless, a proper combination of state collapsing in distributed state space exploration should also impact network usage which this simple combination fails to provide. Indeed, since each PE owns a portion of the global state space, the contents of local hash tables (tables H_i) may differ among PEs. Moreover, due to hash collisions, the same local state could be found at different indexes among PEs. These reasons prevent the exchanges of collapsed states between PEs: only uncompressed global states can be sent through the network because a collapsed state is only meaningful for the PE that computed it. This naturally impacts transmission times but also increases the memory allocated to reception buffers.

B. Design of partitioned hash tables

We now propose a distributed state collapsing method that addresses the issue of exchanging collapsed states. This method is based on the use of distributed partitioned hash tables that can be efficiently implemented with the communication primitives provided by the OpenSHMEM specification.

The principle of our distributed collapse compression is to maintain consistency between the different local hash tables of PEs storing local state spaces. Let c be a component of the system analysed and PE be the set of PEs. A simple solution would be to assign c to a single PE p, in other words, this PE would then be responsible for storing the most up-to-date version of the local state space in a hash table H_c^p made available to other PEs via the shared heap. All PEs would then concurrently access H_c^p to insert or retrieve local states. This solution would however break the symmetry between processes which in turn would penalise the performance of our method as it is likely that some PEs would then have their shared heap accessed more frequently (*e.g.* those that are responsible of larger local state spaces), resulting in a network bottleneck.

To maintain, as much as possible, this symmetry between processes, the solution we propose is to partition the local state space table of a component c into $|PE|$ blocks of size $B = \frac{L_c}{|PE|}$, L_c being the number of slots in the hash table associated with component c (assumed to be a multiple of $|PE|$ for the sake of simplicity). A PE $p \in \{0,\ldots,|PE|-1\}$ is then responsible of the B consecutive slots within range $[p \cdot B, (p+1) \cdot B - 1]$. Only these B slots are shared with other PEs through the symmetric heap. These slots are said to be *owned* by PE p. Other slots remain in the private area.

Figure 5 illustrates the memory layout of this solution for a system composed of two components and with $|PE| = 4$ and $L_1 = L_2 = 8$. Each PE p shares exactly two slots of every table H_c^p (H_c^p being the local state space hash table of component c on PE p). Other slots remain in the PE private memory. This layout implies that if a PE hashes a component value and finds out that the corresponding slot is owned by another PE, it will first look in its (private) copy of the slot to check whether the value is present. Then, only if this slot is empty, will it try to insert the item in the owner's slot (using remote put and atomic compare-and-swap operations as described below).

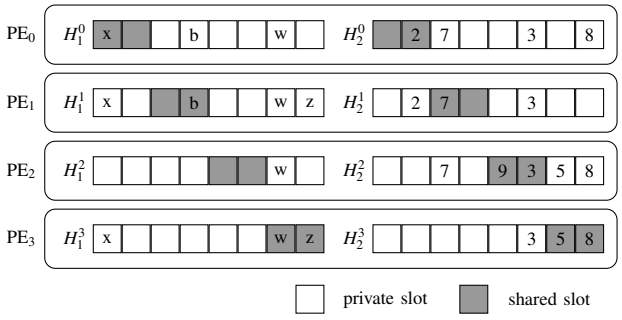


Fig. 5. Memory layout of the distributed partitioned hash tables used by distributed state collapsing method for a two components system example

The consistency of this layout is expressed through the two following invariant properties verified for any PE p:

(I1) If a slot owned by p is empty, then this slot is also empty on all other PEs.

(I2) If a slot owned by p contains an item, then this slot may not contain a different item on any other PE.

Note that the symmetry of this layout makes it particularly suited to the OpenSHMEM memory model.

C. Implementation of the partitioned tables

We detail now the implementation of the *compress* and *uncompress* functions used by our algorithm. Our hash tables must basically only support: a find-or-put operation that returns the index of an item in the table after its insertion, if required; and a lookup operation that retrieves the item at a specific index. For this, we use a straightforward adaptation of the hash table structure proposed in [11] also adapted in [14] for RDMA networks (but for non-partitioned tables and with asynchronous communications). We assume below that the following procedures are provided by the communication layer:

- $getMem(p,o)$ returns the shared object o stored on PE p
- $putMem(p,o,data)$ stores $data$ in the shared object o of PE p
- $cswap(p,b,old,new)$ atomically checks if the shared byte b stored on PE p has value old and, if it is the case, updates it with value new. Function $cswap$ returns the value of the shared byte before its execution.

Figure 6 gives the pseudo-code of the find-or-put and lookup operations required for our distributed collapse compression method. Our hash tables are arrays of slots (type *hashSlot*) that comprise two values, a status and the item stored in that slot. The status is used to control the content of slots as well as concurrent accesses performed by the find-or-put operation detailed below. Initially, all slots have the EMPTY status.

When considering an *item*, *findOrPut* looks at slots starting from $hash(item) \mod H.length$. If it meets an empty slot in its private copy of the table (l. 25), it performs a remote atomic swap on the slot status of the owner of the slot (l. 26) to change this status from EMPTY to WRITING. By this update, the PE notifies its peers that it is currently updating the cell.

```
 1: type pe is {0,...,|PE|−1}
 2: type slotStatus is {EMPTY, WRITING, READY}
 3: type hashSlot is status : slotStatus × item : data
 4: type hashTable is array of hashSlot
 5:
 6: procedure compress(s : state, p : pe) is
 7:    return ⟨findOrPut(H_1^p, s[1]), ..., findOrPut(H_N^p, s[N])⟩
 8:
 9: procedure uncompress(s : compressedState, p : pe) is
10:    return ⟨lookup(H_1^p, s[1]), ..., lookup(H_N^p, s[N])⟩
11:
12: procedure lookup(H : hashTable, idx : int) is
13:    if H[idx].status ≠ READY then
14:       H[idx] := getMem(idx/|PE|, H[idx])
15:    return H[idx].item
16:
17: procedure getItem(H : hashTable, owner : pe, idx : int) is
18:    repeat
19:       H[idx] := getMem(owner, H[idx])
20:    until H[idx].status = READY
21: procedure findOrPut(H : hashTable, item : data) is
22:    idx := hash(item) mod H.length
23:    loop
24:       owner := idx/|PE|
25:       if H[idx].status = EMPTY then
26:          st := cswap(owner, H[idx].status, EMPTY, WRITING)
27:          if st = EMPTY then
28:             putMem(owner, H[idx].item, item)
29:             putMem(owner, H[idx].status, READY)
30:             H[idx] := (READY, item)
31:          else if st = WRITING then
32:             getItem(H, owner, idx)
33:          else /* ⇒ st = READY */
34:             H[idx] := (READY, getMem(owner, H[idx].item))
35:       else if H[idx].status = WRITING then
36:          getItem(H, owner, idx)
37:       if item = H[idx].item then
38:          return idx
39:       else
40:          idx := (idx+1) mod H.length
```

Fig. 6. Compression and decompression procedures used by our distributed state collapsing method

If this swap succeeds, the PE writes the item in the slot of its owner, changes again its status to READY (meaning the slot is now occupied and available to other PEs) and updates its private copy of the slot (ll. 27–30). Note that access with a WRITING status only occurs once, for a single PE, to set the value; subsequent accesses are only reading the slot and can be concurrent. If the swap fails, two cases have to be considered. Either another PE is currently updating the slot (ll. 31–32), in which case, the PE has to wait for it to become ready by periodally interrogating its owner (procedure *getItem* at ll. 17–20). Or, the slot has already been assigned an item (ll. 33–34), in which case the PE just has to recover it from its owner. Note that the *getMem* operation is used for either retrieving the status and the item (*e.g.* l. 19), or the item only (when the status is known, *e.g.* l. 34).

When checking for the status of the slot in its private copy, the PE may also find out that this slot is currently being updated (ll. 35–36). This may happen if and only if the PE is the owner of the slot. In that case, it also waits for the slot to become ready.

At l. 37, it is guaranteed that the status of the slot is READY and that it contains the item stored on the owner of the slot. Then, the PE just has to compare the item to the content of the slot, and return the index if both are equal, or move to the next slot.

Using *findOrPut*, the *compress* procedure (ll. 6–7) simply returns the N-tuples of indexes of the state component values in their respective hash tables. Similarly the *uncompress* procedure (ll. 9–10) can rebuild the full state vector by recovering the N state component values using the *lookup* procedure (ll. 12–15). In this latter procedure, the PE first has to recover the slot from its owner if its local copy of the slot does not contain an item (test at l. 13).

Now that the algorithm has been detailed, we show that the invariants (I1) and (I2) hold.

Sketch of proof of invariants: When a value is processed by a PE, if the slot is empty in its owner's hash table, the item is inserted in the hash table of the owner and of the current PE, if it is not the owner. Hence the first invariant (I1). Moreover, this operation sets together the value in the owner and the PE hash tables. If the slot is not empty in the owner's hash table, it is copied from it and put in the PE hash table. Thus, the value in the owner's table cannot be changed, and is copied by all PEs that encounter the corresponding item. Hence they all carry the same value, as stated by invariant (I2).

D. Use of partitioned tables in distributed state collapsing

With the distributed hash table structure described above, it is straightforward to implement a distributed state collapsing method. When a PE p reaches a state belonging to another PE q, it first computes a compressed representation of this state (*i.e.* an integer tuple) which is sent to q. Upon reception, q stores the compressed state in its private hash table H storing global (compressed) states. Later, this state will be uncompressed in order to be processed by q (*i.e.* its successors generated).

IV. Variations and optimisations

We discuss in this section some variations and optimisations of our distributed state collapsing method.

A. Compatibility with variable length state vectors

The method presented in the previous section assumes that components have constant sizes. This disallows its use for the verification of some models specified in a language such as high-level Petri nets. A simple way to address this issue is to store local component states in a separate buffer of the shared heap that would grow as the exploration progresses. An item

of a partitioned hash table would then consist of a pointer to the location of the local state in this buffer, coupled with the length of the local state. Two remote operations are thus necessary to get/put a state from/to its owner (one to access the table and one to access the buffer).

B. Recovering multiple items simultaneously

In the algorithm of Figure 6, each remote access allows us to get a single item from a partitioned table. The second variation we propose is, when requesting an item, to get several consecutive items owned by the interrogated PE, starting from the requested one. This is indeed possible since PEs own blocks of consecutive items in a partitioned table. For instance, in the example of Figure 5, if PE_1 requests local state 5 of component 2 from PE_3 it could also get local state 8 using the same *getMem* operation. This will possibly save a future request for local state 8. This modification does not affect the algorithm in any other way.

This optimisation is parametrised by the size of the blocks recovered this way. It will be named BR(B) (Block Retrieval) thereafter with B being the block size (a size of 0 indicating that the optimisation is turned off).

C. Broadcasting new local states

It is clear from the layout of Figure 5 that maintaining the partitioned hash table storing the local state space of a component c requires a number of remote operations in $O(|R_c| \cdot |PE|)$ (assuming an ideal situation with no hash conflict) since each local state can be retrieved from its owner by any other PE. This unfortunately means that the number of remote operations used to maintain local state spaces increases linearly with the number of PEs involved which may in turn impact the acceleration of the exploration algorithm. This becomes problematic for large local state spaces.

To alleviate this issue, we implemented an optimisation consisting of broadcasting new local states discovered. Basically, each time a PE inserts a new local state in a partitioned table (ll. 28–29 in the algorithm of Figure 6), it also puts it (*i.e.* a triple containing the component number, the index of the new local state in its partitioned table, and the value of this local state) in the output buffers storing compressed global states to be sent to other PEs. Upon reception of this local state, a PE inserts it in its local state space.

The principle of this optimisation is that when a PE reaches a state that causes the insertion of a new local state in a partitioned table, it is likely that other PEs will later have to retrieve this local state. Hence, to avoid having these PEs retrieving it using a costly remote get carrying a single local state, it is put in output buffers to be sent together with other data. Proceeding this way can thus save subsequent individual *getMem* operations that would otherwise be performed by these PEs to request the new local state.

This optimisation will be named BS (Broadcast of new local States) thereafter.

V. Experiments

We have implemented our method in the Helena verification tool [7] on top of the distributed algorithm of [6]. We have conducted experiments with input models written in the DVE language, the input language of the DiVinE model checker [3]. A future implementation of the variation proposed in Section IV-A to deal with state vectors of variable length will allow us for using also high-level Petri nets as a formalism for input models.

A. An implementation of state collapsing for DVE models

DVE allows to model concurrent systems made of processes having private variables and synchronising through global shared variables or communication channels.

We have seen that collapse compression relies on a decomposition of the system into components. A straightforward decomposition scheme associates each process to a component and all global data (variables or channels) to another component. This scheme will be named DP hereafter (Decomposition by Process).

Scheme DP is however not always adapted. This may, for instance, be the case if the system is composed of a large number of small components (*e.g.* 1 or 2 bytes), in which case the memory reduction ratio will be close to 1 or even larger. Similarly, if some component represents a large fraction of the state vector, it is likely that its local state space will also suffer from combinatorial explosion, thereby cancelling the benefit of state collapsing.

Another decomposition algorithm we used tries to tackle these issues by proceeding as follows. A first decomposition is done using scheme DP. Components are then ordered by increasing size and then merged or split according to the following two rules:

- Consecutive components are merged together in a new one if their overall size does not exceed M bytes, M being a parameter of this scheme. The decomposition algorithm always tries to merge as many components as possible.
- Components exceeding M bytes are split into sub-components of size M bytes. The last sub-component contains the remaining bytes of the component if its size is not a multiple of M.

This scheme will be named DMS(M) hereafter (Decomposition by Merge and Split).

B. Input models

We used models from the BEEM database for benchmarkings [15]. We experimented with models of Table I selected according to their size (state space and state vector). The table gives model names, numbers of states and transitions in the state space, the decomposition strategy used (see Section V-A) and, in the last five columns, compression information for this specific strategy. Column *Partitioned table size* gives the size (*i.e.* number of slots in the table) of a single partitioned table storing local states. All partitioned tables are given the same size. Hence, the number of bits required to encode a compressed state vector equals

$$\text{Components} \cdot log_2(\text{Partitioned table size})$$

TABLE I
DVE MODELS USED FOR BENCHMARKING

Name	States	Transitions	Decomposition strategy	Components	Partitioned table size	Local states	State vector (B)	Compressed state vector (B)
firewire_tree.6	22 690 105	126 238 660	DMS(20)	21	2^{12}	5 255	483	32
leader_election.6	35 777 100	233 195 212	DMS(20)	11	2^{16}	24 823	235	22
firewire_tree.7	121 230 111	778 073 817	DMS(20)	28	2^{12}	7 780	647	42
leader_election.7	235 183 948	1 712 371 948	DMS(20)	12	2^{16}	29 242	281	24
lifts.9	266 445 936	846 144 885	DMS(16)	3	2^{20}	918 975	43	8
collision.5	431 965 993	1 644 101 878	DMS(20)	3	2^{12}	6 289	52	5
pgm_protocol.11	499 396 802	1 207 512 586	DMS(8)	19	2^{20}	503 957	129	48
brp.8	1 526 547 707	3 207 513 490	DP	7	2^{22}	1 358 728	18	20
synapse.9	1 675 298 471	3 291 122 975	DMS(12)	4	2^{18}	48 959	58	9

Column *Local states* gives the number of unique local states over all components, (*i.e.* this equals $\sum_{c\in\{1,\ldots,N\}} |R_c|$). This number corresponds to the number of items stored in the partitioned hash tables.

To determine the decomposition strategy we picked DMS(20) as the default strategy. We then started with a partitioned table size of 2^8 and progressively increased (quadrupled) its size until the search could terminate successfully. Indeed, since partitioned hash tables cannot be resized nor extended, the search has to be aborted as soon as one of these tables fills up as it is done with the non-distributed collapse compression [8]. If the search could not terminate with a size of 2^{22}, we then tried with a lower value of M (DMS(16), DMS(12), ...) and repeated this process until a run could finish successfully. For model brp.8 we had to fall back to strategy DP, since no run could successfully terminate with strategy DMS. Note that state collapsing is useless for this model as the compressed state is larger than the original one.

C. Experimental environment

Experiments presented in this paper were carried out using the Grid'5000 [4] testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organisations (see https://www.grid5000.fr).

We used the Graphene cluster, which is composed of 144 nodes, each of which features a quad-core Intel Xeon X3440 running at 2.53 GHz, 16 GiB of RAM and a 20G InfiniBand network interconnection. The nodes were running a 64-bit Linux 4.9 kernel. All the code was compiled using the GNU gcc 6.3.0 with -03 optimisation flag. We used the OpenSHMEM implementation provided by OpenMPI 2.0.1 and the InfiniBand communication libraries libverbs 1.2.1 and librdmacm 1.1.0.

D. Impact of the optimisations

In tables II, III and IV, we compare the execution times (in seconds) on 256, 128 and 32 cores, with various optimisation options on a set of models. The fastest time is coloured in green; those slower by 10% or less are coloured in lighter green; those slower by 20% or less are coloured in light green.

The lift.9 and brp.8 models stand apart from the other models, since they are the only ones on which the local compression version is faster, for any number of processes. We believe that this can be explained by the relatively small state vector (40 bytes with lifts.9, 18 bytes for brp.8 and the state spaces: lifts.9 has only 266 millions of states, whereas although brp.8 has a large state space (1.5 billions), it has a lot of local states compared to the state space (1.4 millions).

Hence, the state vectors are too small to benefit from the states compression before they are sent (on brp.8 the state is so small that the compressed vector is bigger than the original one). In their case, maintaining the shared hash tables induces a lot of communications, since they contain a lot of local states. Therefore, the performance gain from the state compression is not significant enough to compensate the overhead due to these additional communications.

On the other hand, the model with a very small local state collision.5 gets better performance with no broadcast on a small number of processes, and scales better with a larger number of processes.

We can see that, at small scale, the relative difference is small (almost all the optimisations give a performance within 20% of the fastest time) and the difference increases with the number of processes used. On most models, broadcasting the new local states (see Sec. IV-C) gives better performance or, when it is not the case, very close to the best performance. In particular, on a small number of processes, when a broadcast operation is not very expensive, broadcasting the new states is always the winning strategy (most MPI implementations implement their broadcast in log(P) steps, except for large messages, which is not our case).

On models that are too small to scale well such as firewire_tree.6, the additional communications of the distributed collapse algorithm harm the scalability. However, as presented in section V-F, it does not really make sense to run this model on 256 processes.

On 256 processes, small block sizes (see Sec. IV-B) gives the best performance, except on the two leader_election models. They have relatively large numbers of local states (w.r.t. the state space size) and medium-size state vectors. As explained about brp.8 and lifts.9, the performance gain obtained by state vector compression does not overcome the overhead caused by maintaining the hash table at this scale.

Model	Local collapse	Distributed collapse													
		BS off							BS on						
		BR(0K)	BR(4K)	BR(8K)	BR(16K)	BR(32K)	BR(64K)	BR(128K)	BR(0K)	BR(4K)	BR(8K)	BR(16K)	BR(32K)	BR(64K)	BR(128K)
firewire_tree.6	9.37	13.04	12.35	12.49	12.23	13.18	18.89	12.42	17.43	11.84	11.93	12.01	12.15	12.08	12.18
leader_election.6	9.0	8.52	7.92	7.89	7.85	8.04	7.97	7.79	7.32	7.2	7.26	7.17	7.37	7.29	7.45
firewire_tree.7	37.49	27.72	26.73	28.06	29.18	26.72	32.76	26.53	26.01	26.51	27.14	27.175	27.16	26.71	26.29
lifts.9	9.89	381.65	84.2	61.94	30.97	25.66	48.48	35.28	12.6	12.63	12.75	13.04	13.7	14.6	14.87
leader_election.7	35.53	22.14	21.27	21.53	21.29	27.26	21.32	21.3	20.84	20.68	20.68	20.91	20.65	20.89	20.6
collision.5	14.4	12.84	11.39	11.38	11.47	11.42	11.42	11.4	11.1	10.98	11.11	11.15	11.13	11.36	11.18
pgm_protocol.11	31.12	57.96	50.46	54.98	42.09	44.66	49.32	47.81	31.3	44.66	31.55	31.57	32.57	32.2	32.47
synapse.9	34.74	36.62	32.65	30.44	30.25	30.28	30.7	30.41	28.77	29.85	29.81	28.95	29.61	29.0	29.04
brp.8	39.09	140.56	60.23	43.94	56.52	48.91	53.3	58.71	43.5	36.83	37.07	37.18	38.16	48.06	41.65

TABLE II
EXECUTION TIME WITH VARIOUS OPTIMISATION OPTIONS ON 256 PROCESSES

Model	Local collapse	Distributed collapse													
		BS off							BS on						
		BR(0K)	BR(4K)	BR(8K)	BR(16K)	BR(32K)	BR(64K)	BR(128K)	BR(0K)	BR(4K)	BR(8K)	BR(16K)	BR(32K)	BR(64K)	BR(128K)
firewire_tree.6	10.19	7.81	7.64	7.67	11.8	7.71	7.57	7.82	7.53	7.57	8.38	7.43	7.63	7.62	7.49
leader_election.6	9.49	7.41	8.15	7.05	9.5	7.01	7.04	7.02	6.42	7.68	6.49	6.54	6.49	6.46	6.5
firewire_tree.7	73.08	45.22	48.17	47.37	45.79	47.09	46.7	45.57	44.74	45.98	46.61	45.55	46.06	44.29	45.23
lifts.9	11.1	61.97	25.44	23.7	23.36	26.21	29.42	33.68	12.51	12.96	12.8	13.12	14.1	15.85	17.92
leader_election.7	63.73	34.83	36.51	34.85	35.91	34.15	34.41	33.97	34.11	35.76	34.14	33.63	33.52	34.04	34.08
collision.5	19.22	14.05	13.81	14.4	13.96	13.79	13.83	14.29	13.93	14.79	14.92	13.72	13.55	13.62	13.52
pgm_protocol.11	40.88	51.52	45.09	44.22	44.45	47.11	52.87	57.91	33.56	33.61	33.68	33.72	33.86	34.38	34.57
synapse.9	60.01	53.6	50.83	50.72	50.52	52.58	52.47	51.3	55.1	49.77	49.67	49.79	49.58	49.67	49.89
brp.8	61.95	112.8	66.58	64.6	65.19	68.4	74.7	82.33	55.97	55.69	55.93	56.28	57.11	58.38	60.94

TABLE III
EXECUTION TIME WITH VARIOUS OPTIMISATION OPTIONS ON 128 PROCESSES

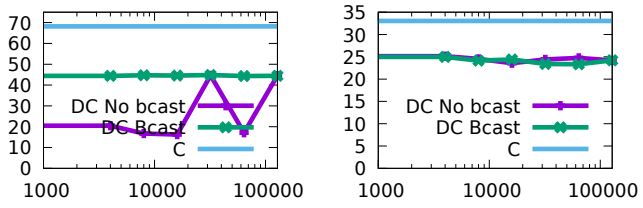
Model	Local collapse	Distributed collapse													
		BS off							BS on						
		BR(0K)	BR(4K)	BR(8K)	BR(16K)	BR(32K)	BR(64K)	BR(128K)	BR(0K)	BR(4K)	BR(8K)	BR(16K)	BR(32K)	BR(64K)	BR(128K)
firewire_tree.6	34.78	47.21	32.79	24.63	23.03	22.68	23.4	17.7	18.16	17.83	18.03	17.99	18.28	18.19	17.81
leader_election.6	34.03	18.43	18.14	18.15	17.83	18.01	17.71	17.96	17.39	17.7	17.85	17.32	17.39	17.14	17.22
firewire_tree.7	328.21	188.75	177.3	177.2	177.6	177.36	174.17	178.11	178.34	176.45	177.84	22.45	22.89	23.14	22.51
lifts.9	33.79	64.26	43.7	41.76	41.48	42.89	47.78	52.18	29.54	29.73	30.06	30.65	31.23	34.28	37.48
leader_election.7	302.82	149.37	151.33	150.85	148.98	150.09	149.53	150.09	147.84	151.14	150.04	147.35	147.4	149.78	148.88
collision.5	68.17	100.27	20.47	16.71	16.23	44.62	17.12	44.56	46.25	44.39	44.72	44.56	44.83	44.33	44.43
pgm_protocol.11	139.58	120.26	115.94	115.8	114.61	117.47	122.94	129.21	105.17	104.79	105.58	104.82	105.86	105.55	106.92
synapse.9	125.25	96.05	94.44	94.85	92.31	95.28	93.93	94.38	92.92	93.52	94.0	93.11	94.52	94.01	92.16
brp.8	134.89	175.87	145.65	142.6	145.32	148.64	156.24	163.72	132.11	128.9	135.37	134.34	130.77	136.63	136.78

TABLE IV
EXECUTION TIME WITH VARIOUS OPTIMISATION OPTIONS ON 32 PROCESSES

E. Buffer size

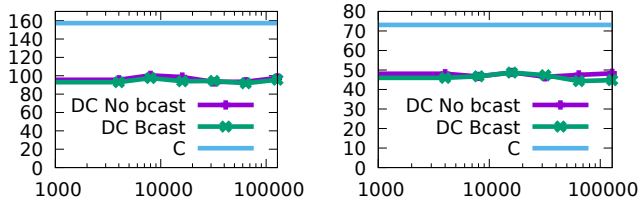
In order to observe the impact of the buffer size on the performance, we measured the execution time on a set of models on a given number of processes with respect to the buffer size with and without broadcast and with local collapse. Some selected results are given Fig. 7, 8, 9, 10 and 11. As seen in section V-D, on some models the distributed collapse is slightly slower than the local collapse.

We notice that the buffer size has little impact on the performance.



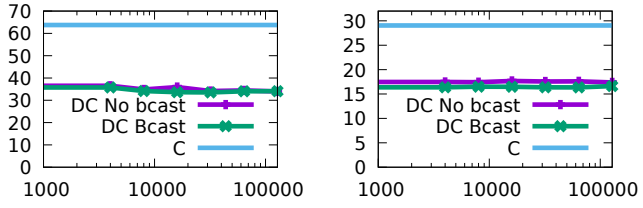
(a) 32 procs (b) 64 procs

Fig. 7. Impact of the buffer size: `collision 5`. The x axis gives the buffer size while the y axis displays the execution time.



(a) 64 procs (b) 128 procs

Fig. 8. Impact of the buffer size: `firewire tree 7`. The x axis gives the buffer size while the y axis displays the execution time.

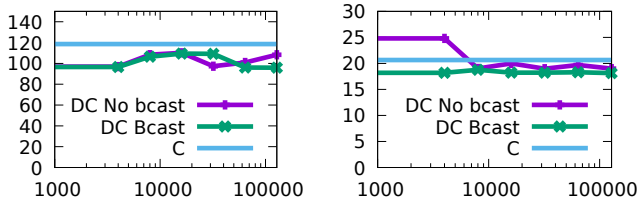


(a) 128 procs (b) 384 procs

Fig. 9. Impact of the buffer size: `leader election 7`. The x axis gives the buffer size while the y axis displays the execution time.

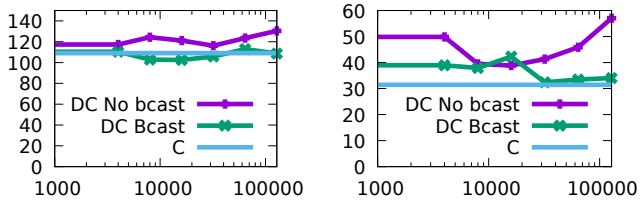
F. Scalability

On Fig. 12 and 13 we compare the scalability of the various options. The local compression is slower than distributed compression, but it scales better. The difference between the other options used for distributed compression is not significant, neither in terms of raw performance nor on the scalability.



(a) 64 procs (b) 384 procs

Fig. 10. Impact of the buffer size: `synapse 9`. The x axis gives the buffer size while the y axis displays the execution time.



(a) 64 procs (b) 384 procs

Fig. 11. Impact of the buffer size: `brp 8`. The x axis gives the buffer size while the y axis displays the execution time.

G. Lessons learned from the experiments

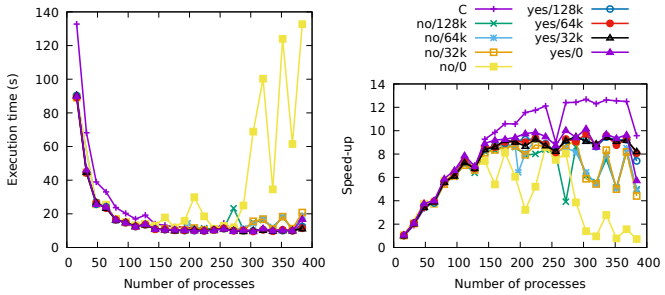
To summarise the conclusions from the experimental results:

- Except on a small number of processes and with a small local state, broadcasting the new local states always gives better performance;
- However, since broadcasting the new states induces additional communications, it does not scale as well as without this broadcast;
- The number of states that are retrieved at the same time (called the buffer size) has little impact on the performance when broadcast is enabled, but a middle-size buffer is efficient when there is no broadcast;
- On models with a small state vector and large local state (compared to the state space), the distributed collapse algorithm is slightly slower than the local collapse one;
- In all the other cases, the distributed collapse algorithm with a small block size for multiple simultaneous retrieval is faster.

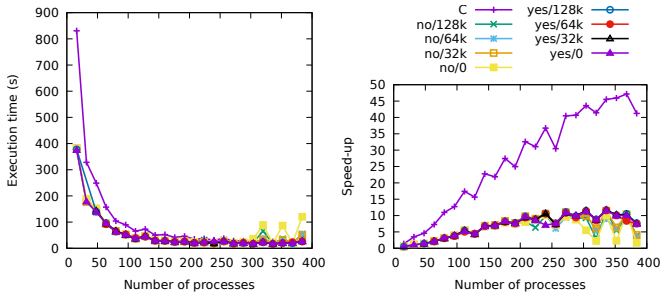
VI. Conclusion

This paper addressed the optimisation of distributed model-checking with one-sided communications by using state compression. This new approach has been implemented in the distributed version of the Helena model-checker, and experiments on a range of different benchmarks has shown its advantages.

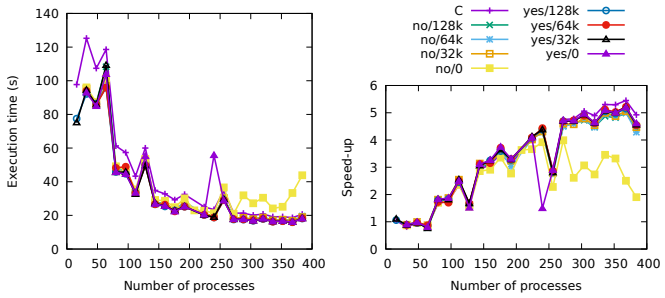
One limitation of our distributed compression technique is the necessity for the model checker user to provide an upper bound on the partitioned hash table sizes. Indeed, if the table fills up, a PE will not be able to insert new item to it, thus aborting the search. The user then has to rerun the search with a larger size. This adds to the necessity of providing a value for parameter M if decomposition strategy DMS is used. Some



(a) Collision 5

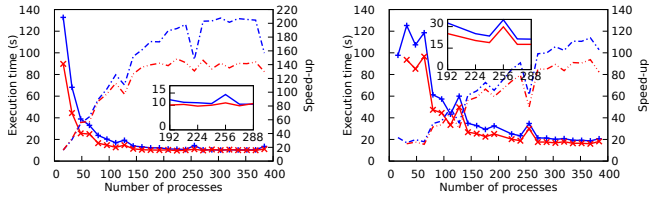


(b) Firewire Tree 7



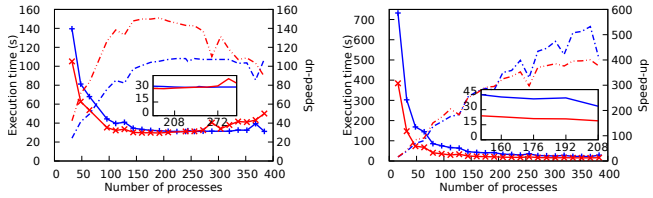
(c) Synapse 9

Fig. 12. Execution time and speed-up comparison on three models. The left plot gives the execution time using various buffer sizes, broadcast and no broadcast, and local compression; the right plot gives the speed-up normalized using the smallest number of processes we could execute on.



(a) Collision 5

(b) Synapse 9



(c) PGM protocol 11

(d) Leader election 7

Fig. 13. Compared scalability with local (in blue) and distributed compression (in red). The plain lines represent the execution time, the dashed lines represent the speed-up, normalized with the smallest number of processes we could execute on.

heuristics for computing a suitable M will be designed in the future.

Moreover, for sequential model checkers, the recursive indexing compression [8] enhances the collapse compression to address this issue. One perspective is to extend our technique in a similar way.

References

[1] OpenSHMEM Application Programming Interface version 1.4. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf, Dec 2017.

[2] C. Baier and J-P Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] Z. Baranovà, J. Barnat, K. Kejstovà, T. Kučera, H. Lauko, J. Mrázek, and P. Ročkai a nd V. Štill. Model Checking of C and C++ with DIVINE 4. In *ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017.

[4] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Vicat-Blanc Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*, pages 99–106, Seattle, Washington, USA, November 2005. IEEE/ACM.

[5] C. Coti. POSH: Paris OpenSHMEM: A High-Performance OpenSHMEM Implementation for Shared Memory Systems. In *Procedia Computer Science, special issue on the 2014 International Conference on Computational Science (ICCS 2014)*, volume 29, pages 2422–2431, 2014.

[6] C. Coti, S. Evangelista, and L. Petrucci. One-Sided Communications for more Efficient Parallel State Space Exploration over RDMA Clusters. In *Proceedings of the 24th European Conference on Parallel and Distributed Computing (EuroPar'18)*, Torino, Italy, August 2018. To appear.

[7] S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'2005*, volume 3536 of *LNCS*, pages 455–464. Springer, 2005.

[8] G. J. Holzmann. State Compression in Spin: Recursive Indexing and Compression Training Runs. In *SPIN'1997*, 1997.

[9] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley,, 2004.

[10] G. J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. *STTT*, 2(3):270–278, 1999.

[11] A. Laarman, J. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *FMCAD'10*, pages 247–255. IEEE, 2010.

[12] A. Laarman, J. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN'2011*, volume 6823 of *LNCS*, pages 38–56. Springer, 2011.

[13] F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *SPIN'1999*, volume 1680 of *LNCS*, pages 22–39. Springer, 2000.

[14] W. Oortwijn, T. van Dijk, and J. van de Pol. Distributed Binary Decision Diagrams for Symbolic Reachability. In *SPIN'2017*, pages 21–30. ACM, 2017.

[15] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'2007*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[16] U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *CAV'1997*, volume 1254 of *LNCS*, pages 256–278. Springer, 1997.

[17] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.