A Database Approach to Distributed State Space Generation ⋆

Stefan Blom,[1,3] Bert Lisser,[2] Jaco van de Pol[2,3] and Michael Weber[2,3]

[1] Institute of Computer Science, University of Innsbruck, Austria

[2] Department of Software Engineering, CWI, The Netherlands

[3] Faculty for Electrical Engineering, Mathematics and Computer Science, University of Twente, The Netherlands

Abstract. We study distributed state space generation on a cluster of workstations. It is explained why state space partitioning by a global hash function is problematic when states contain variables from unbounded domains, such as lists or other recursive datatypes. Our solution is to introduce a database which maintains a global numbering of state values. We also describe tree compression, a technique of recursive state folding, and show that it is superior to manipulating plain state vectors.

This solution is implemented and linked to the μCRL toolset, where state values are implemented as maximally shared terms (ATerms). However, it is applicable to other models as well, e.g., PROMELA or LOTOS models. Our experiments show the trade-offs between keeping the database global, replicated, or local, depending on the available network bandwidth and latency.

Keywords: state space partitioning, state collapsing, tree compression, μCRL

⋆ This work has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.504 (VeriGem) and by the EU under grant number FP6-NEST STREP 043235 (EC-MOAN).

1 Introduction

We study distributed explicit state space generation on a cluster of workstations in the presence of recursive data types, like lists and trees. Recursive data types allow natural modeling of data needed in complicated protocols and distributed systems, e.g., the current knowledge of an intruder in security protocols. Such systems can be analyzed by finite state model checkers, when the scenario is limited to a fixed number of participants. However, an upper bound on the size of the data terms is not known a priori.

Finite state model checking suffers from state space explosion, which can be alleviated by various techniques, such as partial-order reduction, data abstraction and symmetry reduction. In this paper, we focus on distributed model checking, which attacks the state space explosion by using the combined memory and CPU time of a cluster of workstations.

We show that the basic scheme for distributed state space generation based on a shared hash function is limited (Sec. 2.2). It breaks down in the presence of state space generators that produce recursive data types. Implementing them as acyclic pointer structures works well on one computer but sharing pointer structures over a number of workstations is non-trivial.

Our solution (Sec. 3) is to introduce a database (basically an indexed set) that maintains a global numbering of values that occur in state vectors. Instead of exchanging vectors of (serialized) pointer structures, the workers now exchange vectors of indices. In addition, workers must communicate with the database in order to agree on the semantics of these indices.

We improve this basic solution in several steps. In Section 3.2, we replicate the database and introduce piggybacking to reduce synchronisation points, thus decreasing the dependency on network latency.

A further improvement (Sec. 3.3) is to recursively *fold* states using a tree of tables. Each node in this tree represents a set of sub-vectors. The leaf tables store sets of individual state components, while the root tables represents a set of full state vectors by pairs of integers. This so-called *tree compression* reduces the memory needed to store a set of states.

In Sec. 3.4, tree compression is distributed. The leaf database must be maintained globally for consistency reasons, however, the root tables cannot be maintained globally because its size equals the number of reached states. Therefore, each worker keeps a local root database for its own states. The intermediate tables, however, can be kept either local or (replicated) global. In the latter case, workers can exchange shorter *folded vectors*, thus saving on the bandwidth needed to exchange states across the network.

This solution is implemented and linked to the μCRL toolset [6], where state values are implemented as maximally shared terms (ATerms) [7]. However, it is applicable to other models as well. We compare our solution with related work in Sec. 5.

We implemented several versions (Sec. 6), in order to measure the effects of recursive state folding, and the effects of organizing the intermediate tables globally or locally. We report an interesting trade-off for organizing the (intermediate) tables locally or globally, depending on the available bandwidth and latency of the underlying network.

2 Distributed State Space Generation

In the following, we briefly outline the currently prevailing approach to distributed state space generation, which is based on the partitioning of the *closed set* (the set of visited states) across processors with a hash function. We highlight the silently assumed conditions under which this approach is usually im-

plemented, and in Sec. 2.2 we make clear why this simple setup is insufficient not only for μCRL, but more generally for state generators for any language that allows unbounded recursive data types (such as lists, trees), implemented by pointer structures.

2.1 The Traditional Partitioning Approach

The traditional approach to state space generation, as introduced in [10,21], is illustrated by the straightforward algorithm below. Alg. 1 and Alg. 2 are supposed to run concurrently (either in parallel or interleaved). They synchronize on shared data structures.

In these algorithms, the state space is partitioned over the memory of W workers by a hash function. Each worker W_i keeps its own part of the explored state space in $Closed|_{W_i}$. The states that still have to be explored are kept in the set $Open|_{W_i}$. The EXPLORE thread picks an open state, calculates the hash of all its successors in order to put them into the local *Queue* of the right owners. The RECEIVE thread picks states from the local *Queue*, checks if they are new by consulting $Closed|_{W_i}$, and if so, adds the state to both the *Closed* set (to avoid duplicate exploration) and the *Open* set (to be explored by EXPLORE).

Algorithm 1: EXPLORE$_{W_i}$

Data: *Open*, *Closed*, *Queue*

```
1 while not terminated do
2     PICK s from Open|_{W_i};
3     forall transitions s → s' do
4         CALCULATE h = Hash(s');
5         ADD s' to Queue|_{W_h};
```

Result: $Open = Queue = \emptyset$, *Closed* contains all reachable states

We note that this basic scheme relies on a number of assumptions for its correctness and efficiency, which are usually not spelled out explicitly. First, for correctness it must be assumed that states have a globally unique representation, otherwise a worker cannot interpret the states it receives from other workers. Typically, a state consists of a vector of values for locations and state variables. Second, the hash function must be globally known, agreed upon by all workers, and stable over time (unless we take costly rehashing schemes into account), otherwise different workers would add the same state to different owners, leading to exploring states more than once. While this might even be tolerable for simple reachability questions, it is not for other verification algorithms.

For efficiency reasons, it must be assumed that state vectors are small, otherwise local memory and network bandwidth are wasted, and are stored in a contiguous memory area, in order to avoid (de)serialization costs.

These requirements are met in specification languages with "simple" data types, like SPIN [17], NIPS [23], and Petri Nets [2]. Here, data consists of bounded integers, structures, and fixed size arrays. However, for languages that allow unbounded recursive data types, these assumptions are problematic, as we will see in the next section.

Algorithm 2: RECEIVE$_{W_i}$

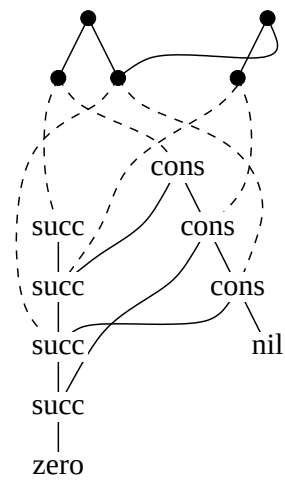
```
Data: Open, Closed, Queue
1 while not terminated do
2     PICK s from Queue|W_i;
3     if s ∉ Closed|W_i then
4         INSERT s into Closed|W_i;
5         INSERT s into Open|W_i;
```

2.2 Special Requirements for μCRL

The μCRL state generator represents state vectors as ATerms [7]. Through ATerms, μCRL allows the use of recursive data types in its specifications, which enables a more natural representation of models in many cases, for example, models of intruder knowledge in security protocols and network routing protocols utilizing dynamic tables [3]. This convenience does not come for free, however.

In a nutshell, ATerms are constructor terms, consisting of a *head* symbol, and a variable number of parameters, which are ATerms themselves. The leaves of the term structure are constant symbols with no arguments, which includes integers. Internally, the collection of all ATerms present is represented as a *maximally shared forest*, i.e., equal sub-terms are only ever stored once, but possibly referenced many times. Unreferenced ATerms are automatically garbage-collected. Maximal sharing allows for a compact representation of ATerm forests, and has other benefits, too.[4]

For example, equality checking of potentially large terms, which would entail a full traversal, now reduces to a (constant-time) pointer comparison, due to maximal sharing. In a sequential setting, this obviates the need for a hash function for fast look-up. The figure on the right shows a particular representation of two 4-variable states $\langle 4, [3, 1, 2], 2, [2]\rangle$ and $\langle 3, [1, 2], 2, [2]\rangle$ as an ATerm forest. Note how vectors $\langle x_1, x_2, x_3, x_4\rangle$ get replaced by trees $\bullet(\bullet(x_1, x_2), \bullet(x_3, x_4))$; how num-



[4] Implementing decision diagrams on top of ATerms is rather trivial: sharing comes for free, only canonicalization rules have to be added.

bers are built from zero and successor and how lists are built from cons and nil. Also note how the resulting sub-terms are shared. In particular, note how the second part of the two states is shared.

One of the biggest drawbacks for distributed computing with ATerms is their representation as a pointer data structure. They are obviously not transportable from one computer to another. Cheap equality checking of ATerms only works locally on the computer they are stored, thus we would need a globally known hash function for fast comparison again. Such a function would require traversing the entire ATerm. This is moderately expensive, but because the same computation is done many times, it is possible to use memoization techniques to overcome the computation time problem, at the expense of memory for the memoization table.

The other problem is that in order to transmit a state across the network it has to be serialized into a flat binary form. Serializing an array of integers is very efficient. Serializing an array of ATerms, however, is a serious problem: the printed version of a *single* ATerm often takes 40 bytes or more, because typically the sharing gets lost. That means that it is factor 10 larger than the pointer we started with. It becomes infeasible, if we consider that a state consists of a vector of such ATerms.

In principle, it would be possible to use buffering to exploit sharing between successively transmitted states, thus reducing the space and time costs of serialization somewhat. But this does not scale up to larger numbers of workers: because the hash function is supposed to be evenly distributed, scaling up can be expected to reduce sharing.

We note that other state generators suffer from the issues described here, for example, DISTRIBUTOR from the CADP toolset [12,14] (version: 2006 "Edinburgh"):

> The current version of Distributor does not handle LOTOS programs containing dynamic data types (such as lists, trees, etc.) implemented using pointers [...][5]

We are aware that a solution which lifts this restriction is being worked on for CADP. Other tools, for example, SPIN and NIPS could benefit as well, as explained in Sec. 5.

3 A Centralized Database

In the following sections, we show how the conditions for hash-based state space partitioning can be recreated by introducing a global database for state parts. We will also see that this setup comes with additional benefits: it allows various schemes of (network-wide) compression of states, thus reducing memory and bandwidth requirements.

3.1 Distributed State Compression

We consider states which are not opaque, but instead have some identifiable structure which we exploit. Thus, a state vector $s_i = \langle p_{i_1}, p_{i_2}, \ldots, p_{i_\ell} \rangle_{\mathrm{SV}}$ is a sequence of *state parts*, here denoted as p_{i_j}. In the case of μCRL these are data terms, representing control locations or data values. Other possibilities for state parts include channels and processes, and their variables. This is the case for states of SPIN and NIPS, for example.

For now, we focus on a static structure that is the same across all states. Furthermore, we assume that the chosen parts exhibit locality, i.e., for the majority of transitions $s \rightarrow s'$ of a state space, most of the parts of state s remain unchanged in s'. This assumption is valid in particular for interleaving semantics of the underlying model. The size of the state space is largely due to the

[5] http://www.inrialpes.fr/vasy/cadp/man/distributor.html

combination of state parts, thus we can assume the number of parts slots ℓ and also number of parts p_{i_j} to be small.

For a basic solution, we first consider a globally accessible, indexed table which maps state parts to indices, and vice versa. For reasons which will become apparent later, we call this the *leaf database*. Through the database, we obtain now a second unique representation of a state vector s_i, in terms of the indices of its parts: $\bar{s}_i = \langle i_1, i_2, \ldots, i_\ell \rangle_{\mathrm{IV}}$. Depending on the size of the state parts, the *index vector* representation $\bar{s}_i$ is in general an order of magnitude or more smaller than s_i, so we may think of this scheme as a simple table compression method.

We note that an index vector by itself is not useful for a state space generator which can only operate on "uncompressed" state vectors. Thus, if we choose index vectors for storing states, we continuously need to map back and forth between two representations.

If we adapt the algorithms from Sec. 2.1 to take the leaf database into account, we can consider three phases.

Exploration. First, for new states the following steps have to be taken:

1. Explore an uncompressed state s by calculating its successors $s_0, \ldots, s_k$.
2. For each $s_i = \langle p_{i_1}, p_{i_2}, \ldots, p_{i_\ell} \rangle_{\mathrm{SV}}$:
   2.1. Resolve all state parts p_{i_j} against the (global) leaf database.
      – Map each state part to its index: $p_{i_j} \mapsto i_j$,
        add p_{i_j} to database, if not already present.
      – This results in index vector $\bar{s}_i = \langle i_1, i_2, \ldots, i_\ell \rangle_{\mathrm{IV}}$ for s_i.
   2.2. Calculate $h = \mathrm{Hash}(\bar{s}_i)$, add $\bar{s}_i$ to $Queue|_{W_h}$

For every state we are now required to look up its state parts in the leaf database which entails additional communication. This also means that still we have to serialize all of the state (although in parts) when adding them to the

database. However, on the plus side, we can now calculate the hash value of a state (which determines its "owner") cheaply over a vector of integers $\bar{s}_i$ instead, no matter what state parts look like. The global uniqueness of state parts can now be locally guaranteed by the leaf database.

Furthermore, we can communicate the (compressed) index vectors to other workers W. This reduces bandwidth demands between workers, however we must keep in mind their additional communication to the database. We will return to this point in Sec. 3.2.

Queue Management. Next, we consider a state $\bar{s}_i$ arriving in the work queue $\mathit{Queue}|_{W_h}$ of some worker W_h ($h = \mathrm{Hash}(\bar{s}_i)$):

1. Pick $\bar{s}_i$ from $\mathit{Queue}|_{W_h}$
2. Check whether $\bar{s}_i \in \mathit{Closed}|_{W_h}$
3. If yes, $\bar{s}_i$ has been visited before, hence drop it
4. Else, add $\bar{s}_i$ to $\mathit{Closed}|_{W_h}$ and also $\mathit{Open}|_{W_h}$, so that it will be explored eventually.

We note that in this phase, we are dealing with index vectors ($\bar{s}_i$) exclusively. Thus, *Open* set, *Closed* set, and the work queues are storing index vectors.

Decompressing States. In the last phase, before exploration of a new state, we must resolve the index vector representation and rebuild the original state:

1. Pick next state $\bar{s}_i = \langle i_1, i_2, \ldots, i_\ell \rangle_{\mathrm{IV}}$ from $\mathit{Open}|_{W_b}$
2. Resolve all i_j against the leaf database
   - Map indices to state parts $i_j \mapsto p_{i_j}$ (all parts are in the table already, thus look-ups will not fail)
   - We obtain back the original state $s_i = \langle p_{i_1}, p_{i_2}, \ldots, p_{i_\ell} \rangle_{\mathrm{SV}}$
   - Explore new state s_i as detailed in the first phase.

To summarize, with this new scheme, we seemingly have not won much. While resolving indices to state parts, we cause extra communication and costly serialization, for each transition even! What we did achieve, however, is better storage efficiency on each worker, as only compressed states are stored in various data structures. We note that due to the small number of state parts, the leaf database is small and not in any concrete danger of exhausting a worker's memory. In addition, workers among themselves now communicate index vectors, and only with the database they exchange state parts.

We can now leverage existing knowledge how to increase database query performance. We will get to this in the following section.

3.2 Database Replication

Using a central database helped us to overcome the problem of hashing states of the μCRL state generator, but the costly serialization of states remains. We also introduced extra network communication due to round-trips to the leaf database while resolving state parts. In this section, we fix these issues by replicating the tables of the global leaf database on each worker. The additional storage requirements pose no problems to the workers because these tables are small compared to other data structures, like open and closed sets.

During the course of state space generation, the leaf database is updated with new state parts, hence we cannot easily replicate it in one go. Therefore, we describe a protocol which updates the local *replicas* incrementally. A simple approach would cache the query result for each state part when the answer arrives at a worker. This would lead to at most one query per state part per worker. We can improve this by piggybacking each answered query with all the state parts that are not already in the local replica. This only requires replacement of the "Resolve" step in the scheme outlined in Sec. 3.1:

a) Try resolving all state parts p_{i_j} in the *local* leaf table replica. If found, we have $p_{i_j} \mapsto i_j$ *without* communication with the global leaf database.

b) Else, update replica with new parts p_{i_j}:
send current highest index max of local replica, in addition to all unresolved parts $U = \{p_{i_j} \mid p_{i_j} \text{ not locally resolvable}\}$ to the leaf database.

c) The global database replies with the state parts needed to bring the replica up-to-date:

$$(max' - max, p_{max+1}, p_{max+2}, \ldots, p_{max'})$$

In particular, $max\{i_j \mid p_{i_j} \in U\} \leq max'$. We can then resolve all $p_{i_j} \in U$ with the updated local replica.

The above scheme is simple, but effective. We draw from the fact that state parts in the leaf database are only ever added, and never updated or deleted.

We are still requiring the (costly) serialization of all state parts during state space generation, but now in the worst case only once per worker, and once for each worker during a reply to update its local replica of the leaf database. Due to the piggybacking of replica updates, the mentioned worst case is unlikely to occur. When a worker requests a state part, it might well be the case that it is already in its local replica due to an earlier update.

We note that in the specific case of the μCRL toolset, the use of ATerms makes the comparison of state parts p_{i_j} in step (a) very cheap: a pointer comparison suffices, as explained in Sec. 2.2. The hidden cost attached to this efficiency is paid when ATerms are deserialized. However, as we mentioned above, we have limited the number of times this is actually needed, and the remaining deserializations are amortized over the vastly larger number of expected look-ups.

The search-related data structures maintained on each worker remain unchanged from the replication introduced here. As in Sec. 3.1, the open and closed set as well as the queues store states as index vectors. Communication between

Algorithm 3: Tree Compression

```
1  type tree = Node of IndexedSet * tree * tree | Leaf of int;
2  fun newtree(size: int): tree = build(0, size-1);
3  fun build( first : int , last : int ): tree =
4    if first=last then Leaf(first)
5    else let middle = (first+last) div 2
6         in Node(NEWINDEXEDSET, build(first,middle), build(middle+1,last));
7  fun fold( t: tree , vec: int array): int = case t of
8    Leaf(i)                    ⇒ vec[i]
9    Node(table, left , right)  ⇒ PAIRTOINDEX(table, (fold(left,vec),fold( right , vec )));
10 proc unfold(t: tree , index: int , vec: int array): unit = case t of
11   Leaf(i)                    ⇒ vec[i] := index
12   Node(table, left , right)  ⇒ let (i1,i2) = INDEXTOPAIR(table,index)
13                                in unfold( left , i1 , vec); unfold( right , i2 , vec);
```

workers happens in terms of index vectors as well. Network bandwidth requirements are reduced drastically.

3.3 Tree Compression

We have explained how we can transform a vector of variable sized objects into a vector of integers. The length of the vectors (the number of state parts) is typically in the range from 50 to 100 for μCRL.

In order to compress states further, we now introduce a datastructure that can map (long) vectors of indices to single indices, and back. We could use a simple hash table, but then we would have to store a copy of every vector. This would be a waste of memory because large parts of many vectors will probably be identical. Instead of storing long vectors in one table, we could use a main table and two auxiliary tables. The auxiliary tables map the first and second

half of the vector to numbers. The main table maps the pair of numbers to a number. If there are many vectors, but few distinct halves then the auxiliary tables remain small and we save a lot of memory on the main table.

When we recursively apply this splitting method to the auxiliary tables the result is a tree of tables, hence the name *tree compression*. In Alg. 3, we provide pseudo (ML) code for tree compression. To initialize, the user must call newtree with the number of elements which will create a tree with internal nodes containing indexed sets, and leaves containing indices of the array to be folded and unfolded. Function NEWINDEXEDSET creates an empty indexed set. This data structure supports two operations: PAIRTOINDEX(*table*,(*i1*,*i2*)), which inserts pair $(i1, i2)$ in the table if it is not yet present, and always returns its index; and INDEXTOPAIR(*table*,*index*) which returns the pair referred to by *index*. Once a tree is initialized a vector can be folded into a single index, and an index can be unfolded to its original vector. We note that the tables in the tree can be extended by fold and that unfold writes its result into the vector provided as last argument.

In Fig. 1, we have illustrated the result of building a tree for a vector of length 6 and inserting $\langle 1,0,0,0,0,1\rangle_{\mathrm{IV}}$ into it. The first picture shows the state after newtree(6). To insert $\langle 1,0,0,0,0,1\rangle_{\mathrm{IV}}$, we traverse the tree reaching the bottom left binary node. There we visit the leaves and retrieve the contents of the vector (second picture). We insert the pair $\langle 1,0\rangle$ into the table and return 0 because it is that table's first entry[6] (third picture). Backtracking, we fetch the third element of the vector and insert $\langle 0,0\rangle$ yielding 0 (fourth picture). We repeat for the right branch, inserting $\langle 0,0\rangle$ at the top and returning 0 (last picture).

In Fig. 2, we compare the result of inserting 9 vectors into both a normal table and into the tree. On the left, an array of 9 index vectors of length 6 each uses 54 units of memory; on the right, a tree of indexed sets uses 42 units of

[6] subsequent table entries would get the next free index

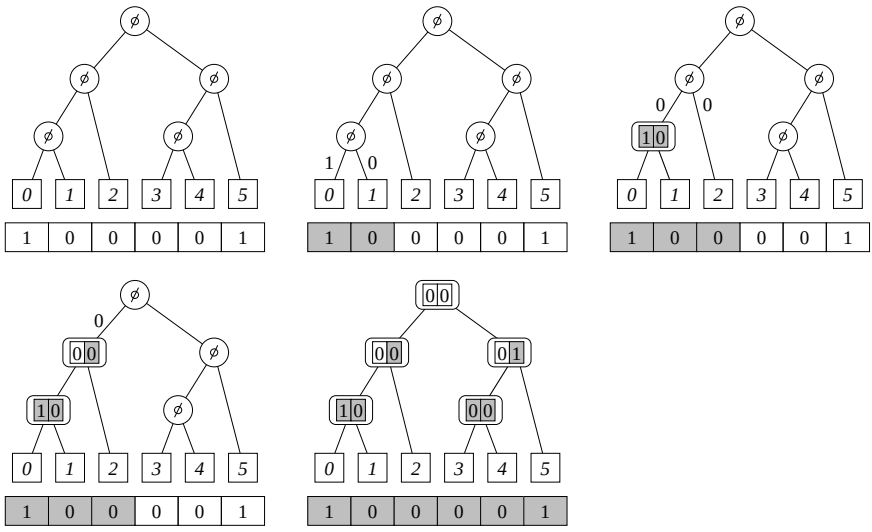


Fig. 1. Tree evolution during the execution of $\mathsf{fold}\,(\mathsf{newtree}\,(6), \langle 1,0,0,0,0,1\rangle_{\mathrm{IV}})$. Grey boxes show progress on the input vector and where vector elements are stored in the tree. Slanted numbers refer to slots in the vector.

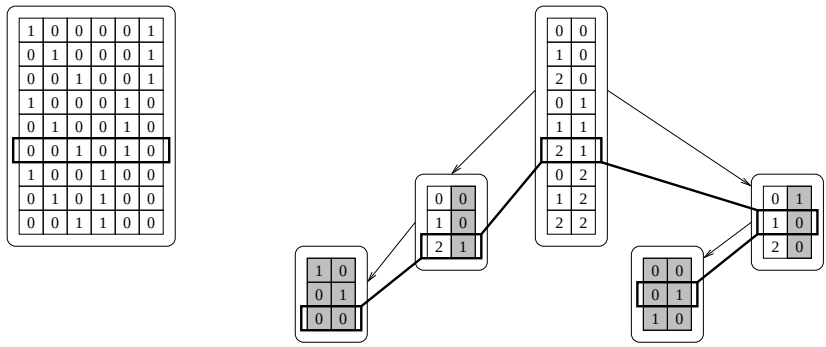


Fig. 2. An example for tree (de)compression. Vector $\langle 0,0,1,0,1,0\rangle_{\mathrm{IV}}$ is represented as $\langle 2,1\rangle_{\mathrm{FV}}$, where index 2 is looked up in the next table to the left, and index 1 to the right, yielding another two pairs of indices. Grey boxes are leaves, and not looked up further. The so selected fringe of grey boxes corresponds to the original index vector if read from left to right.

memory to represent the same data. The elements of the main table at the top of the tree are called *folded vectors* $\langle \ldots \rangle_{\mathrm{FV}}$. The original full length vector is denoted $\langle \ldots \rangle_{\mathrm{IV}}$.

Next, we analyze the best and worst cases of the memory complexity of tree compression. The idea behind tree compression is that when we look at the sub-vectors of a set of vectors, many of these sub-vectors occur more than once. Thus, by storing each sub-vector once and then using a reference we can save memory. E.g., when restricting the table in Fig. 2 to the first three columns, only three distinct sub-vectors occur. If the sets of distinct first and second sub-vectors of a large set of vectors are much smaller than the full set, then the amount of memory used for storing them separately becomes negligible in comparison to the memory needed for the main table.

The worst case for tree compression is that the amount of memory needed increases by a factor of 2. This can happen if for a certain set S, we try to store vectors of ℓ identical elements $\{\langle s, \cdots, s\rangle \mid s \in S\}$. In this case each of the tables will have length $|S|$. Because we have $\ell - 1$ tables of width 2, we need $(\ell - 1) \cdot 2 \cdot |S|$ units of memory compared to the $\ell \cdot |S|$ units needed for storing the vectors directly.

A much better case is a Cartesian product. To store $V \times V$, where $V \subseteq S^\ell$, we need a table with $|V|^2$ entries for the top node plus the tables to store the $|V|$ possibilities for the left and right sub-vectors. So we need $2 \cdot |V|^2$ units for the top node and (using the previously computed worst-case upper bound) less than $(\ell-1)\cdot 2\cdot|V|$ units for each of the sub-vectors for a total of $2\cdot|V|^2+4\cdot(\ell-1)\cdot|V|$ units compared to $2\ell\cdot|V|^2$ for the direct solution. That is, with a perfect balance for the top node the space needed to store N vectors of length ℓ is at most $2N+4(\frac{\ell}{2}-1)\sqrt{N}$, meaning $2+\frac{2(\ell-2)}{\sqrt{N}}$ units on average per vector. Note that we counted just the memory needed for data. However, for the reverse mapping we

Table 1. Average space usage (in units per vector) for N vectors of length ℓ.

	Vector	Tree	
		Perfect Top	Worst Case
Data	ℓ	$2 + \frac{2(\ell-2)}{\sqrt{N}}$	$2(\ell-1)$
Hash Table	≤ 2	$\leq 2 + \frac{2(\ell-2)}{\sqrt{N}}$	$\leq 2(\ell-1)$
Total	$\leq \ell + 2$	$\leq 4 + \frac{4(\ell-2)}{\sqrt{N}}$	$\leq 4(\ell-1)$

also need a hash table. If we also count its usage with a minimum utilization of 50% then we arrive at the numbers in Tab. 1.

In the example and in our implementation, we chose to split the vector in half each time. This is a reasonable assumption if one does not have additional knowledge about the vector. But in some cases, we know in advance that one of the vector positions is going to have a lot of different elements. In that case it would be useful to split the vector in a short and a long part where the element with many different values is in the short part. Permuting the vector can also have large effects. We leave research in this direction for future work.

From the analysis, one might draw the conclusion that just splitting the vector into two parts once and then using a hash table for the components has practically the same performance. We identify two reasons why this is not true in practice. First, it might happen that the top node does not split perfectly, but the second node does. So using the same trick recursively improves our chances of getting good performance. Second, in the distributed setting, tree compression can be used to influence bandwidth requirements as well as memory requirements.

3.4 Distributed Tree Compression

Tree compression can be used to further reduce the communication bandwidth needs of a distributed state space generator. Instead of sending and receiving index vectors as a whole, we fix a part of the tree as local and the remainder as global. Note that the root table has as many entries as the total number of states. So the top node must be a local node on each worker, storing only those states that it owns. Furthermore, the parent of a local node must be local. That is, the local nodes are a non-empty prefix of the whole tree. Local nodes are stored in hash tables which are unique to a worker. Global nodes use tables which are kept synchronized across all workers just like the leaf database in Sec. 3.2. This allows us to compress in two steps. In the first step we apply all global tables to get an intermediate folded vector. In the second step we apply the local tables. Because the intermediate vector is computed using globally known tables, we can transmit intermediate folded vectors rather than index vectors to other workers. The length of the intermediate vectors is one more than the number of local tables. The fully folded vectors are used for storage on the workers.

We have implemented two strategies: *local* in which both the root and intermediate tables are local and *global* in which the root table is the only local table. To transmit N vectors of length ℓ in local mode, we need to send $\ell \cdot N$ integers. In global mode, we need $2 \cdot N$ integers for the real messages plus $3 \cdot W \cdot T$ integers for replicating the global tables (assuming W workers, sending a query and getting response of together 3 integers for each of the T entries of the global tables). If $T < \frac{(\ell-2)N}{3W}$ then the global method has a bandwidth advantage over the local method. With perfect balancing, T can be as small as $2\sqrt{N}$. In practice, we have seen pathological cases with $T \approx \frac{1}{2}N$, which with $\ell > 50$ and $W = 16$ should still give a gain. However, it also comes with a latency penalty: each of the T look-ups might require a round-trip to the database. Again, we can use the

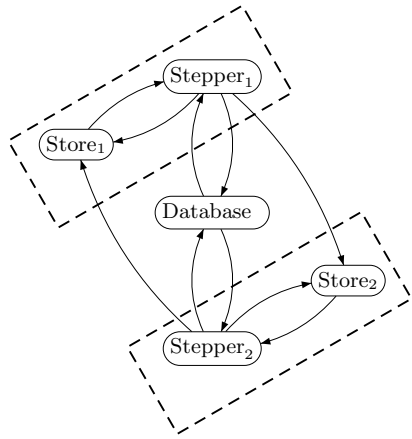


Fig. 3. Data flow between tool components (configuration with two workers).

piggybacking principle explained in Sec. 3.2 to alleviate the influence of latency somewhat.

4 Implementation

For implementation purposes, the tool is divided into several components. Fig. 3 depicts a configuration with only two workers.

A worker is divided into a *stepper* and a *store*. The store contains all closed states assigned to this worker. The stepper component processes this worker's part of the open set, and adds states to the queue (next level). States in the queue are sent to the store of the worker that owns them. Thus, steppers implement the *Exploration* and *Decompression* phases as described in Sec. 3.1.

The store component receives queued states from all steppers, matches them against its part of the closed set, and adds them to the open set if they are new. This corresponds to the *Queue Management* phase from Sec. 3.1.

We have chosen this setup for its flexibility. Usually, the store and stepper of one worker are on one machine. But it is for instance possible to run all the stores

on a (well-connected) computer with large amount of RAM, and the steppers on fast machines with less memory.

4.1 "Global" Mode

Each stepper communicates with the database to map state parts to indices and vice versa, as well as indices to pairs for tree compression. The central database component stores the master copy of all mappings (*leaf tables* and *intermediate tables*), and each stepper replicates them fully.

Each store's data structures work in terms of indices only, hence there is no need to access or replicate the central database.

4.2 "Local" Mode

In "local" mode, the components remain the same as in the "global" mode, however, they have slightly different responsibilities. Each stepper replicates the leaf database to be able to compress states to index vectors, but no intermediate tables for global tree compression. Instead, index vectors of the states in the open set and the queue are compressed locally per stepper. The resulting intermediate tables are stored in the *open set trees*. They accumulate information for states belonging to a worker, but also information of their successor states (which are ultimately sent to another worker). In order to avoid this pollution and the resulting extra storage overhead, it is important to flush the open set trees regularly, for example, after every BFS level.

Since index vectors are exchanged between all components, the store needs to maintain its own compression tables for the closed set, stored in *closed set trees*. The separation also facilitates the mentioned flushing.

The global database is responsible only for the mapping between indices and state parts, hence also does not store intermediate tables anymore.

5 Related Work

The classical approach of state space partitioning in the setting of Petri Nets dates back to at least the work of Ciardo et al. [10]. For in-depth explanations and variations we refer to Ciardo [9].

The database and state compression approach presented here is based on earlier work of Blom, Langevelde and Lisser [5, Sec. 4] on file formats for distributed state space generation. As a follow-up, we focus here on the changes needed to integrate μCRL with classical state space partitioning: we introduce a global database and several query and update protocols. We also provide measurements to show the trade-offs between several of these protocols depending on the hardware used.

We utilize loss-less state compression schemes for efficient storage and network transmission, and regard lossy compression as out of scope. The simple index table compression which is crucial for μCRL works essentially in the same way as SPIN's initial COLLAPSE method [22,17], and was probably pioneered in XESAR [15]. Holzmann describes *recursive indexing* for the Revised COLLAPSE method, however, despite the name this is actually only a two-level approach (variables and processes). More importantly, decompression is never needed in the case of SPIN, and there are no provisions to keep the indices unique in a distributed setting. In contrast, our *state folding* method indeed aggregates state parts recursively, and is designed for a distributed setting, which also requires decompression.

In typical PROMELA models, a state is represented as vector of 50 to 500 bytes. It consists of around 10 parts: processes, channels and a block of global variables. Depending on how this byte vector is compressed into an index vector by Revised COLLAPSE, we can obtain an index vector of length 10. Storing these in an indexed set still costs a lot of memory. Hence, we believe that

our recursive tree folding could be profitable for model checkers with flat data, such as SPIN and NIPS as well.

Ciardo et al. consider multi-valued decision diagrams (MDDs) for efficient storage of state sets [11]. A distributed version is described in [8]. These solutions use a single MDD that branches on the value of state variables to store a set of states. In contrast, we use a tree for every state. These trees branch on the *position* of variables in the state vector. One might say that our tree structures are a specialized implementation of multi-terminal binary decision diagrams (MTBDDs).

To the best of our knowledge, currently no other distributed state space generator can handle recursive datatypes.

6 Measurements

The μCRL toolset has been used in a number of case studies [3], yet the benefits and trade-offs of the different state representations we have presented here, have not been assessed before. To fill this gap, we experimented with five models (size information can be found in Tab. 3):

Lift5 describes an elevator system with five legs for lifting large vehicles [16].

SWP is a version of the sliding window protocol [1].

CCP33 describes an instance of the cache coherence protocol *Jackal* for Java programs with 3 processes and 3 threads [19].

1394fin describes the physical layer service of the 1394 or firewire protocol and also the link layer protocol entities [18,20]. We use an instance with 3 links and 1 data element.

Franklin53 describes a leader election protocol for anonymous processes along a bidirectional ring of asynchronous channels, which terminates with probability one [13]. We chose an instance with 5 nodes and 3 identities.

We considered three implementations. All three implementations utilize a global (but replicated) leaf database which is used to map states to index vectors, but they differ in the following characteristics:

vector Workers store and exchange full index vectors.

local Workers exchange full index vectors, but store them compressed. This requires local intermediate tables on each worker.

global Workers exchange and store only compressed vectors of indices. This requires a global (but replicated) database with intermediate tables.

We performed the experiments on two clusters:

CWI The *Spin* cluster at the CWI, using 16 nodes with AMD Athlon$^{\text{TM}}$ 64 3500+ 2.2 GHz processors and 1 GB RAM each, all interconnected with Gigabit switched Ethernet.

TUE The *Sandpit* cluster at the TU Eindhoven, again with 16 nodes, each equipped with a 32-bit Intel Pentium 3.06 GHz processor and 2 GB RAM, also interconnected with Gigabit switched Ethernet.

Next, we will describe the three sets of tests we ran.[7]

The first set of experiments was to compare the first two case studies for two data structures on both clusters using version 2.17.13 of the μCRL toolset. In Tab. 2, we present run times and memory use[8] of these experiments.

The second set of experiments was a speedup test for the CCP33 problem on both clusters. The results can be found in Fig. 4. For these measurements we scaled the number of processors from 2 to the maximum available to us (16 on the TUE cluster, 32 on the CWI cluster.) Run times are averaged over three runs, and vary very little on the TUE cluster, as evident from the small error

[7] Additional details can be found at http://www.cwi.nl/~mcrl/pdmc-2007/

[8] The total memory usage is higher, as we omitted open set and buffers here.

Table 2. Measurements for the "vector" and the "local" implementations.

		Run Time [sec.]		Memory
		TUE	CWI	
Lift5	vector	81	160	561.1M
	local	96	160	301.9M
SWP	vector	294	387	2.0G
	local	248	372	417.1M

Proc.	Average Run Time [sec.]	
	TUE	CWI
2	51,593.33	(OOM)
4	26,422.67	56,056.00
8	13,450.00	31,792.00
16	6,905.33	14,752.67
28	N/A	7,502.00
32	N/A	6,458.50

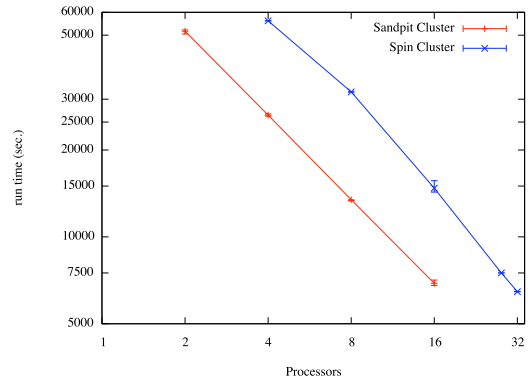


Fig. 4. Speedup measurements performed with CCP33 and the "global" implementation. With only two processors assigned, the CWI cluster runs out of memory (TUE cluster machines have more RAM installed). The log-log plot of the data reveals a close to linear speedup for up to 32 processors.

bars in the plot. On the CWI cluster, variation is slightly more visible due to interference from other users of the cluster.

The third set of experiments is a comparison of local and global for all five models on both clusters. The results can be found in Table 3. Note that these results were produced with the improved version 2.18.1 of the toolset. The "wall time" column contains the time in seconds elapsed until job completion. Under

Table 3. Measurements for the comparison of local (L) and global (G) modes.

cluster	mode	Wall Time [sec.]	Messages to DB Count	Messages to DB Latency [msec.]	Set Memory Open	Set Memory Queue	Set Memory Closed	Tree Memory Open/ Queue	Tree Memory Closed	Transfer Size	Transfer Time [sec.]
Lift5:			**103 levels**		**2,165,446 states**				**8,723,465 transitions**		
TUE	L	102	1,252	2.60	1.4M	11.1M	32.5M	6.8M	26.1M	2.3G	219
TUE	G	91	251,347	0.25				285.0M		267.0M	133
CWI	L	172	1,247	11.28				6.8M	26.1M	2.3G	256
CWI	G	257	254,319	5.16				285.0M		267.0M	109
SWP:			**61 levels**		**19,466,100 states**				**93,478,264 transitions**		
TUE	L	255	895	1.52	12.6M	121.2M	276.5M	20.2M	55.9M	11.1G	608
TUE	G	225	420,932	0.23				172.9M		2.7G	151
CWI	L	364	860	6.22				20.1M	55.3M	11.1G	640
CWI	G	469	421,736	5.23				172.9M		2.7G	168
1394fin:			**170 levels**		**88,221,818 states**				**152,948,696 transitions**		
TUE	L	6,339	1,955	1.07	64.1M	245.2M	1.1G	31.6M	211.5M	39.3G	1,913
TUE	G	5,517	1,313,124	0.22				252.3M		4.5G	356
CWI	L	9,010	1,977	5.87				31.4M	211.8M	39.3G	1,919
CWI	G	9,111	1,311,443	5.89				252.3M		4.5G	353
Franklin53:			**82 levels**		**84,381,157 states**				**401,681,445 transitions**		
TUE	L	1,782	1,079	1.44	39.8M	385.1M	1.1G	278.7M	1.6G	91.2G	5,382
TUE	G	2,066	38,647,751	0.33				7.8G		11.9G	456
CWI	L	2,003	1,109	2.33				287.3M	1.6G	91.2G	4,336
CWI	G	11,506	38,644,914	3.89				7.8G		11.9G	374
CCP33:			**298 levels**		**97,451,014 states**				**1,061,619,779 transitions**		
TUE	L	8,569	2,928	0.46	17.8M	391.3M	1.2G	121.1M	850.1M	407.3G	21,421
TUE	G	7,067	14,780,196	0.23				2.7G		31.6G	1,141
CWI	L	12,562	2,945	5.36				126.6M	1,019.8M	407.3G	21,746
CWI	G	15,800	14,815,199	5.12				2.7G		31.6G	1,048

column “Messages to DB”, one finds a count of queries to the global database and the average round-trip time per query (including not only network transmission but also processing). The memory use is split over 5 (4) columns for local (global). In each column, we account for the sum of the memory used in all 16 workers. The first three columns show the memory used for the root tables of the open set, the queue and the closed set. In local mode we have separate intermediate tables in the stepper (storing the open set and the queue) and in the store (storing the closed set). In global mode all workers share one database which is then replicated. The memory used by these tables is listed in the “Tree Memory” columns. The final two columns list how much data is transferred between workers and how much time it takes. These columns are also sums over all workers.

Tab. 2 is the relevant part of the first data set in our earlier PDMC paper [4]. The data in Fig. 4 is new. The measurements in Tab. 3 are redone with a new implementation, and extended by two models.

6.1 Evaluation

First, we observe in Tab. 2 that the “vector” implementation uses much more memory than the “local” implementation (e.g., 2.0 GB versus 417.1 MB for SWP in Tab. 2). This is explained by the compression due to sharing: “vector” stores the open and closed sets as arrays of vectors of integers, while “local” stores them as short vectors, plus local tree compression tables. Larger models, like CCP33, could not even be generated in the vector implementation. Moreover, contrary to what one would expect, the “vector” method is not significantly faster than the “local” method.

Next, we compare keeping the intermediate tables “local” or “global” (but replicated). As expected, the local tables reduce the communication of workers with the global database drastically (Tab. 3, column “Messages to DB”). It is

only needed for leaves, and mainly during the initial phase of a run. However, network traffic between workers is much higher, for the models presented here around factor 5–13. For example, running the "local" implementation on CCP33 caused a data exchange of 407.3 GB in total between workers, whereas in the "global" version only 31.6 GB were exchanged. This is due to the fact that with "local" tables, workers exchange long index vectors, while with "global" tables they can exchange small folded vectors.

Surprisingly, the winner in overall time (Tab. 3, first column) depends on the actual cluster: the "local" implementation is faster than the "global" implementation on the CWI cluster, but slower on the TUE cluster (except franklin53). We attribute this to the difference in database latency between the clusters (Tab. 3, column 5), for the models used here by a factor of up to 23. Note that the traffic between workers is asynchronous (latency hiding through buffering), while the traffic with the database is synchronous. High network latency mainly influences database traffic, while low available bandwidth affects the communication between workers. The fact that local is faster on both clusters for the franklin model is due to the fact that the amount of messages sent is larger than for any other model.

Considering the approximately similar networking hardware of both clusters, the latency difference is unexpected. Indeed, on both clusters the fastest queries are almost instantaneous, and despite some fluctuation there is no alarming difference between the slowest queries. However, looking at the distribution of query latencies we found that for SWP and CCP33 models, consistently 95% of the time spent on database communication is due to the slowest 2% of all queries, on both clusters. The rest of the messages are negligibly (and equally) fast. That is, on the TUE cluster, the slowest 2% of all messages account for around 3,171 sec. cumulated time over all workers, while on the CWI cluster, the slowest 2% of all messages need 74,257 sec. Eventually, the slow queries

could be traced to the CWI cluster's suboptimal handling of buffers within the network stack when dealing with dropped packets. The same situation happens on the TUE cluster, but it is handled much faster.

Another unexpected result is that the tree compression tables for the closed set (column "tree memory") require more memory in the "global" version than both sets of compression tables in the "local" version combined. The difference is that the "global" version contains a full replica of the global database, while "local" contains only entries for state parts which have been encountered locally (when storing states permanently due to ownership). Apparently, the assumption that all workers need nearly all entries of the intermediate tables of the global database is wrong. We may have been too optimistic for tables higher up in the folding trees, that represent longer sub-vectors.

7 Conclusion and Future Work

We enhanced the basic scheme of distributed state space generation with a global database, in order to provide a globally unique representation of values from recursive data types. The round-trip costs are lowered by using database replication and a piggybacking scheme. Furthermore, we introduced tree compression to reduce the storage size of state spaces by recursive state folding. Local and global (but replicated) implementations of index databases have been implemented and their effect on latency and throughput was measured.

We see three lines of future research regarding tree compression. So far, we only experimented with exchanging long index vectors (no tree compression) or index vectors of length 2 (full tree compression). Intermediate solutions are possible too. It would be interesting to experimentally establish an optimal cut-off point for state vector compression, or even build an adaptive tool that dynamically finds the optimum w.r.t. a given model and cluster.

Our experiments so far were restricted to relatively small cluster sizes. We could imagine that hundreds of workers could bring down the central database. Once we confirm this as a bottleneck with actual experiments, we would like to try out existing database technology to deal with the problem, for example, striping the global tables across several servers, etc., instead of a home-brewn solution.

Finally, another interesting possibility is to adapt our scheme to heterogeneous systems, where several clusters of workstations are connected by a high-latency, high-bandwidth network to form a *grid*. In such settings, databases could be local to a cluster, providing indices that are unique within a cluster. This would allow to exchange compressed vectors within a cluster, while across clusters uncompressed vectors have to be exchanged in order to contain the effects of latency. The worker-worker traffic is not affected because it is a-synchronous.

Acknowledgement. We thank Aad van der Klaauw for tracing the reported latency issues at the network layer. We thank TU Eindhoven for access to their Sandpit cluster. We thank Rena Bakhshi for the μCRL model of the Franklin protocol.

References

1. Badban, B., W. Fokkink, J. F. Groote, J. Pang and J. van de Pol, *Verification of a sliding window protocol in μCRL and PVS*, Formal Aspects of Computing **17** (2005), pp. 342–388.
2. Bell, A. and B. R. Haverkort, *Sequential and distributed model checking of Petri net specifications*, in: *Proc. 1st Workshop on Parallel and Distributed Methods for Verification*, ENTCS **68**, 2002.
3. Blom, S., J. R. Calamé, B. Lisser, S. Orzan, J. Pang, J. v. d. Pol, M. Torabi Dashti and A. J. Wijs, *Distributed analysis with μCRL: A compendium of case studies*,

in: O. Grumberg and M. Huth, editors, *TACAS 2007*, LNCS **4424** (2007), pp. 683–689, iSBN 978-3-540-71208-4.

4. Blom, S., B. Lisser, J. van de Pol and M. Weber, *A database approach to distributed state space generation*, ENTCS **198** (2007), pp. 17–32.
5. Blom, S., I. van Langevelde and B. Lisser, *Compressed and distributed file formats for labeled transition systems.*, ENTCS **89** (2003), pp. 68–83.
6. Blom, S. C. C., W. J. Fokkink, J. F. Groote, I. A. v. Langevelde, B. Lisser and J. C. v. d. Pol, *μCRL: A toolset for analysing algebraic specifications*, in: G. Berry, H. Comon and A. Finkel, editors, *Computer Aided Verification (CAV 2001)*, LNCS **2102**, 2001, pp. 250–254.
7. Brand, M. G. J. v. d., H. A. d. Jong, P. Klint and P. A. Olivier, *Efficient annotated terms*, Software – Practice & Experience **30** (2000), pp. 259–291.
8. Chung, M.-Y. and G. Ciardo, *Saturation NOW*, in: *QEST* (2004), pp. 272–281.
9. Ciardo, G., *Distributed and structured analysis approaches to study large and complex systems.*, in: E. Brinksma, H. Hermanns and J.-P. Katoen, editors, *European Educational Forum: School on Formal Methods and Performance Analysis*, LNCS **2090** (2000), pp. 344–374.
10. Ciardo, G., J. Gluckman and D. Nicol, *Distributed state-space generation of discrete-state stochastic models*, INFORMS Journal on Comp. **10** (1998), pp. 82–93.
11. Ciardo, G., R. M. Marmorstein and R. Siminiceanu, *Saturation unbound*, in: H. Garavel and J. Hatcliff, editors, *TACAS*, LNCS **2619** (2003), pp. 379–393.
12. Fernandez, J.-C., H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier and M. Sighireanu, CADP *(Cæsar/Aldébaran development package): A protocol validation and verification toolbox*, in: R. Alur and T. A. Henzinger, editors, *Proc. 8th CAV*, LNCS **1102** (1996), pp. 437–440.
13. Franklin, W. R., *On an improved algorithm for decentralized extrema finding in circular configurations of processors*, Commun. ACM **25** (1982), pp. 336–337.
14. Garavel, H., R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm and G. Stragier, *DISTRIBUTOR and BCG_MERGE: Tools*

for distributed explicit state space generation., in: H. Hermanns and J. Palsberg, editors, *TACAS*, LNCS **3920** (2006), pp. 445–449.

15. Graf, S., J.-L. Richier, C. Rodriguez and J. Voiron, *What are the limits of model checking methods for the verification of real life protocols?*, in: J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS **407** (1989), pp. 275–285.
16. Groote, J. F., J. Pang and A. G. Wouters, *A Balancing Act: Analyzing a Distributed Lift System*, in: S. Gnesi and U. Ultes-Nitsche, editors, *Proc. 6th Workshop on Formal Methods for Industrial Critical Systems*, 2001, pp. 1–12.
17. Holzmann, G. J., *State compression in SPIN: Recursive indexing and compression training runs*, in: *Proc. 3th International SPIN Workshop*, 1997.
18. Luttik, S., *Description and formal specification of the link layer of P1394*, Technical Report SEN-R9706, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands (1997).
19. Pang, J., W. J. Fokkink, R. F. Hofman and R. Veldema, *Model checking a cache coherence protocol of a Java DSM implementation*, JLAP **71** (2007), pp. 1–43.
20. Sighireanu, M. and R. Mateescu, *Verification of the link layer protocol of the IEEE-1394 serial bus (FireWire): An experiment with E-LOTOS*, STTT **2** (1998), pp. 68–88.
21. Stern, U. and D. L. Dill, *Parallelizing the Murφ verifier*, in: O. Grumberg, editor, *Computer-Aided Verification, 9th International Conference*, LNCS **1254** (1997), pp. 256–267.
22. Visser, W. and H. Barringer, *Memory efficient state storage in SPIN*, in: J.-C. Grégoire, G. J. Holzmann and D. Peled, editors, *The Spin Verification System*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science **32** (1997).
23. Weber, M., *An embeddable virtual machine for state space generation*, in: D. Bošnački and S. Edelkamp, editors, *Proc. 14th SPIN Workshop*, LNCS **4595** (2007), pp. 168–185.