



Optimal Storage of Combinatorial State Spaces

Alfons Laarman^(✉) 

Leiden University, Leiden, The Netherlands
a.w.laarman@liacs.leidenuniv.nl

Abstract. Efficiently deciding reachability for model checking problems requires storing the entire state space. We provide an information theoretical lower bound for these storage requirements and demonstrate how it can be reached using a binary tree in combination with a compact hash table. Experiments confirm that the lower bound is reached in practice in a majority of cases, confirming the combinatorial nature of state spaces.

1 Introduction

Model checking has proven effective for automatically verifying correctness of protocols, controllers, schedulers and other systems. Because a model checker tool relies on the exhaustive exploration of the system's *state space*, its power depends on efficient storage of states.

To illustrate the structure of typical states in model checking problems, consider Lamport's Bakery algorithm in Fig. 1; a mutual exclusion protocol that mimics a bakery with numbering machine to prioritize customers. Due to limitation of computing hardware, the number is not maintained globally but reconstructed from local counters in $\mathbb{N}[i]$ (for each process i). For two processes, the state vector of this program consists of the two program counters (pc) and all variables, i.e. $\langle \mathbf{E}[0], N[0], pc_0, \mathbf{E}[1], N[1], pc_1 \rangle$.¹ Their respective domains are:

$$\langle \{\top, \perp\}, [0 \dots 2], [0 \dots 7], \{\top, \perp\}, [0 \dots 2], [0 \dots 7] \rangle.$$

There are $2 \times 3 \times 8 \times 2 \times 3 \times 8 = 2304$ possible state vectors. The task of the model checker is determine which of those are reachable from the initial state; here $\iota \triangleq \langle \perp, 0, 0, \perp, 0, 0 \rangle$. It does this using a next-state function, which in this case implements the semantics of the Bakery algorithm to compute the successor states of any state. For example, the successors of the initial state are:

$$\text{NEXT-STATE}(\langle \perp, 0, 0, \perp, 0, 0 \rangle) = \{ \langle \top, 0, 1, \perp, 0, 0 \rangle, \langle \perp, 0, 0, \top, 0, 1 \rangle \}$$

This work is part of the research programme VENI with project number 639.021.649, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).



¹ We opt to order vectors as follows: variables and program counter of Process 0 (pc_0), variables and program counter of Process 1 (pc_1), etc. Section 6 discusses the effect of orderings.

```

bool E[2] = { false, false };
int N[2] = { 0, 0 };
void process(int i) { // with process id i = 0 or 1
0:   E[i] = true;
1:   N[i] = 1 + max(N[0], N[1]);
2:   E[i] = false;
#define j 0
3:   while (E[j]) { } // Wait until thread 0 receives its number
4:   while ((N[j] != 0) && ((N[j],j) < (N[i],i))) { }
#define j 1
5:   while (E[j]) { } // Wait until thread 1 receives its number
6:   while ((N[j] != 0) && ((N[j],j) < (N[i],i))) { }
/* begin critical section .. end critical section */
7:   N[i] = 0;
}

```

Fig. 1. Lamport’s “Bakery” mutual exclusion protocol for two threads. The wait loop is unrolled at Lines 4–7, where the process waits until all threads j , with smaller numbers or with the same number but with higher priority, expressed as $(N[j], j) < (N[i], i)$, passed their critical section. The boolean variable $E[i]$ associated with process i serves to allow other threads to wait until i received a number in $N[i]$. For simplicity, we assume that each line can be executed atomically.

One successor represents the case where the first process executed Line 0; its program counter is set to 1 and $E[0]$ is updated as a consequence. Similarly, the other successor represents the case where the second process executed Line 0.

While exhaustively exploring all reachable states, the model checker searches if it can reach a state from the set *Error*. For the Bakery algorithm with two threads, we have $Error \triangleq \{ \langle b_0, n_0, 7, b_1, n_1, 7 \rangle \mid b_0, b_1 \in \{\top, \perp\}, n_0, n_1 \in [0 \dots 2] \}$, i.e., all states where both processes reside in their critical section (= pc loc. 8). For completeness, Algorithm 1 shows the basic reachability procedure. The more states the reachability procedure can process, the more powerful the model checker, i.e., the larger program instances it can automatically verify. This number depends crucially on the size of the *visited states set* V in memory. Several techniques exist to reduce V : partial order reduction [19, 26], symmetry reduction [10, 30], BDDs [3, 7], etc. Here we focus on explicitly storing the states in V using state compression.

The potency of compression becomes apparent from two related observations:

Locality. Successors computed in the next-state function exhibit *locality*, e.g.,

$$\text{NEXT-STATE}(\langle \perp, 1, 4, \perp, 2, 6 \rangle) = \{ \langle \perp, 1, \mathbf{5}, \perp, 2, 6 \rangle, \langle \perp, 1, 4, \perp, 2, \mathbf{7} \rangle \}$$

Algorithm 1. The reachability procedure in a model checker.

Data: ι , NEXT-STATE
Result: {error, correct}

```

1  $V := \emptyset$ 
2  $Q := \{\iota\}$ 
3 while  $Q \neq \emptyset$  do
4    $Q := Q \setminus \{s\}$  for  $s \in Q$ 
5    $V := V \cup \{s\}$ 
6   for  $s' \in \text{NEXT-STATE}(s)$  do
7     if  $s' \notin V$  then
8       if  $s' \in \text{Error}$  then
9         return error
10       $Q := Q \cup \{s'\}$ 
11 return correct

```

Note that only program counters change value (marked bold in successors).

Combinatorics. Similar to the set of all possible state vectors, the set of reached state vectors is also highly *combinatorial*. Assuming $\langle \perp, 1, 4, \perp, 2, 6 \rangle$ can be reached from the initial state ι , we indeed saw four different vectors sharing large sub-vectors with their predecessors (underlined here):

$$\begin{array}{ll} \langle \perp, 0, 0, \perp, 0, 0 \rangle & \longrightarrow \quad \langle \top, \underline{0}, 1, \underline{\perp}, 0, 0 \rangle, \langle \underline{\perp}, 0, 0, \top, \underline{0}, 1 \rangle \\ \langle \perp, 1, 4, \perp, 2, 6 \rangle & \longrightarrow \quad \langle \underline{\perp}, \underline{1}, 5, \underline{\perp}, \underline{2}, 6 \rangle, \langle \underline{\perp}, \underline{1}, 4, \underline{\perp}, \underline{2}, 7 \rangle \end{array}$$

We hypothesize that the typical locality of the next-state function ensures that the set of reachable states exhibits this combinatorial structure in the limit. Therefore, storing each vector in its entirety in a hash table, would duplicate a lot of data. By folding the reachable state vectors in a tree, however, these shared sub-vectors only have to be stored once (more in Sect. 3).

In this paper, we investigate the lower bound on the space requirements of typical state spaces occurring in model checking. We do this by modeling the state spaces as an information stream. The values in this stream probabilistically depend on previously seen values, in effect modeling the locality in the next-state function. A simple application of Shannon’s information theory yields a lower bound for the storage requirements of our “state space stream”.

Subsequently, in Sects. 3 and 4, we investigate whether this lower bound can be reached in practice. To this end, we provide an implementation for the visited set V . A practical compressed data structure has as additional requirement that the query time, the time it takes to lookup and insert individual state vectors, is constant (or at least poly-logarithmic) in the length of the vector. The technique suggested by the information theoretical model, i.e., maintaining differences between successor states, does not satisfy this requirement. Therefore, we utilize a binary tree in combination with a *compact hash table*. By analyzing the best-case compression of this structure, we show that it indeed can reach information theoretical lower bound (at least in theory).

According to the same best-case analysis, our implementation of the ‘Compact Tree’ can compress up to tens of billions of large state descriptors (of tens to hundreds of integers) down to only one 32-bit integer per state. Extensive experimentation in Sect. 5 with diverse input models in four different input languages shows moreover that this compression is also reached in practice, and with little computational overhead.

2 An Information Theoretical Lower Bound

The fact that state spaces have combinatorial values is related to the fact that state generated by a model checker exhibit locality as we discussed in Sect. 1. *We will make no assumptions on the nature of the inputs, besides the locality of state generation.* In the current section, we will derive the *information entropy*—which is equal to the minimum number of bits needed for its storage—of a single state vector using basic notions from information theory.

Information theory abstracts away from the computational nature of a program by considering sender and receiver as black boxes that communicate data (signals) via a channel. The goal for the sender is to encode the data as small as possible, such that the receiver is still able to decode it back to the original. The encoded size depends on the amount of *entropy* in the data. In the most basic case, no statistical information is known about the data: each of X possible messages has an equal probability of taking one of its values and the entropy H is maximal: $H(X) = \log_2(|X|)bit$, i.e., the entropy directly corresponds to using one fixed-sized ($\log_2(|X|)$) bit pattern for each possible message.

If more is known about the statistical nature of the information coming from the sender, the entropy is lower as more elaborate encodings can be used to reduce the number of bits needed per piece of information. A simple example is when we take into account the character frequency of the English language for encoding sentences. Assuming that certain characters are much more frequent, a code of fewer bits can be used for them, while longer codes can be reserved for infrequent characters. To calculate the entropy in this example, we need the probability of occurrence $p(x)$ for each character $x \in X$ in the English language. We can deduce this from analyzing a dictionary, or better a large corpus of texts. The entropy then becomes: $H(X) = \sum_{x \in X} -p(x) \log_2(p(x))$

We apply the same principle now to state vectors. As data source, we use the next-state function to compute new states, as we saw in Sect. 1:

$$\text{NEXT-STATE}(\langle \perp, 1, 4, \perp, 2, 6 \rangle) = \{ \langle \perp, 1, \mathbf{5}, \perp, 2, 6 \rangle, \dots \}$$

As a simplification, let states consist of k variables. By storing full states in the queue Q , the predecessor state is always known in the model checker’s reachability procedure (see s and s' on line 6 in Algorithm 1). Hence, we can abstract away from the one-to-many relation of the next-state function and instead view the arriving states as a k -periodic stream of variable assignments:

$$\langle v_0^0, \dots, v_{k-1}^0 \rangle, \langle v_0^1, \dots, v_{k-1}^1 \rangle, \dots, \langle v_0^{n-1}, \dots, v_{k-1}^{n-1} \rangle$$

It thus makes sense to describe the probability that a variable holds a certain value with respect to the same variable in the predecessor state: For each variable v_j^i with $i \geq 0$ and $0 \leq j < k - 1$, both encoder and decoder can always look at the corresponding variable v_j^{i-1} in the predecessor to retrieve its previous value.

Since we are interested in establishing a *lower bound*, we may safely underestimate the number of variables changing value with respect to a state’s predecessor. It makes sense to assume that only one variable changes value, since with zero changes, the same state is generated (requiring no “new” space in V). Hence, we take the following *relative* probabilities (see example Fig. 2):

$$p(v_j^i \neq v_j^{i-1}) = \frac{1}{k} \qquad p(v_j^i = v_j^{i-1}) = \frac{k-1}{k}$$

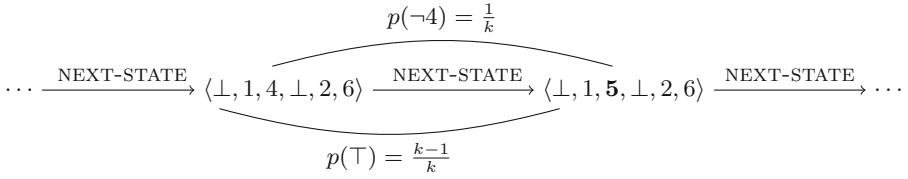


Fig. 2. The states generated with the NEXT-STATE function seen as a stream exhibiting locality. To derive a lower bound, we assume that locality changes only one value in each new vector, i.e., each vector that has to be stored. As there are k variables in the vector, the resulting probability that a variable changes is $1/k$. So the chance that it remains the same with respect to the predecessor is k^{-1}/k .

Let $\langle d_0^0, \dots, d_{k-1}^0 \rangle, \langle d_0^1, \dots, d_{k-1}^1 \rangle, \dots, \langle d_0^{m-1}, \dots, d_{k-1}^{m-1} \rangle$, be the domains of the state slots. As a simplification, we assume that all domains have u bits, resulting in $y = 2^u$ values. Therefore, there are $y - 1$ possible values for which variable v_j^i can differ from its predecessor v_j^{i-1} . Therefore, the probability for one of these other values $x \in d_j^i$ becomes $p(x) = \frac{1}{k} \times \frac{1}{y-1} = \frac{1}{k(y-1)}$ (this equal probability distribution over the possible values results in higher entropy, but recall that we do not make other assumptions on the nature of the inputs). Of course, there is only one value assignment when the variable v_j^i does not change, i.e., the valuation of the same variable in the predecessor state v_j^{i-1} .² This results in the following definition of entropy per variable in the stream:

$$H_{var}(v_j^i) = -\frac{k-1}{k} \log_2 \left(\frac{k-1}{k} \right) + \sum_{n=1}^{y-1} -\frac{1}{k(y-1)} \log_2 \left(\frac{1}{k(y-1)} \right)$$

After some simplification, we can derive the state vector’s entropy:

$$H_{state}(v_0^i, \dots, v_{k-1}^i) = \sum_{j=0}^{k-1} H_{var}(v_j^i) = \log_2(y-1) + \log_2(k-1) + k \log_2 \left(\frac{k}{k-1} \right) \tag{1}$$

Theorem 1 (Information Entropy of States Exhibiting Locality). For $k > 1$, the information entropy of state vectors in state spaces exhibiting locality, abbreviated with H_{state} , is bound by:

$$\log_2(y-1) + \log_2(k-1) + 1 \leq H_{state} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$$

² The assumption that predecessor is always known of course breaks down for the initial state ι . Our model does not account for the initial storage required for ι . However, as the number of states $|V|$ typically grows very large, this space is negligible.

Proof. We first show that $H_{state} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$. Simplification using $\log_2(k - 1) \leq \log_2(k)$ yields: $(1 + \frac{1}{k-1})^k \leq 4$. As for $k = 2$ (recall that $k > 1$), we have $(1 + \frac{1}{k-1})^k = 4$ and, in the limit, we have $\lim_{k \rightarrow \infty} (1 + \frac{1}{k-1})^k = \lim_{k \rightarrow \infty} (1 + \frac{1}{k})^k = e$, it can be seen that $(1 + \frac{1}{k-1})^k \leq 4$ holds, and hence $H_{state} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$.

Now we show that $\log_2(y - 1) + \log_2(k - 1) + 1 \leq H_{state}$.

Simplification yields: $2 \leq (1 + \frac{1}{k-1})^k$. Again for $k = 2$, we have $(1 + \frac{1}{k-1})^k = 4$ and $\lim_{k \rightarrow \infty} (1 + \frac{1}{k-1})^k = e$, hence $\log_2(y - 1) + \log_2(k - 1) + 1 \leq H_{state}$. \square

Intuitively, this approximation makes sense since a single modification in each new state vector can be encoded with solely the index of the changed variable, in $\log(k)$ bits, plus its new value, in $\log(y) = u$ bits, plus some overhead to accommodate cases where more than one variable changes value. This result indicates that locality could allow us to store sets of arbitrarily long ($k \cdot u$ -bit) state vectors using a small integer of less than $u + \log_2(k) + 2$ bits per vector.

In practice, this could mean that vectors of a thousand (1024) of byte-size variables can be compressed to 20 bits each, which is only slightly more than if these states were numbered consecutively—in which case the states would be 18 bits—but far less than 8192 bits required for storing the full state vectors.

3 An Analysis of Binary Tree Compression

The interpretation of the results in Sect. 2 suggests a trivial data structure to reach the information theoretical lower bound: Simply store incremental differences between state vectors. However, as noted in the introduction, an incremental data structure like that does not provide the required efficiency for lookup up operations (the reachability procedure in Algorithm 1 needs to determine whether states have been visited before on Line 7).

The current section shows how many state vectors can be folded into a single binary tree of hash tables to achieve sharing among subvectors, while also achieving poly-logarithmic lookup times in the worst case. This is the first step towards achieving the optimal compression from Sect. 2 in practice. Section 4 presents the second step. We focus here on the analysis of tree compression. For tree algorithms, refer to [21].

The shape of the binary tree is fixed and depends only on k . Vectors are folded in the tree until only tuples remain. These are stored in the leaves. Using hashing, tuples receive a unique index which is propagated back upwards, forming again new tuple in the tree nodes that can be hashed again. This process continues until a tuple is stored in the root node, representing the entire vector.

Figure 3(a) demonstrates how the state $\langle \perp, 1, 4, \perp, 2, 6 \rangle$ is folded into an empty tree, which consists of $k - 1$ nodes of empty hash tables storing tuples. The process starts at the root of the tree (a), and recursively visits children while splitting the vector (b). When the leaves of the tree (colored gray) are reached, they are filled with the values from the vector (c). The vectors inserted into the

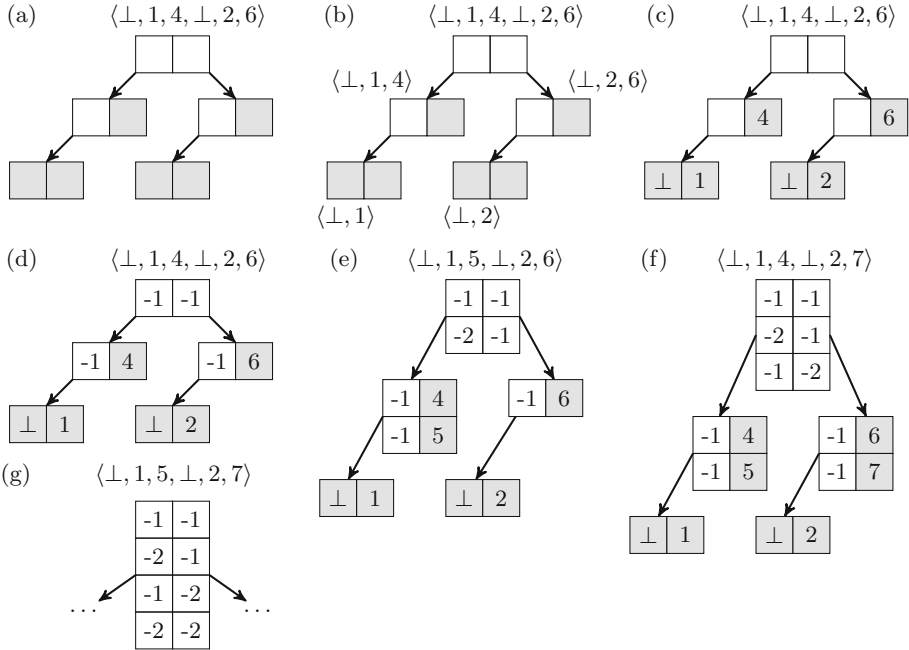


Fig. 3. Tree folding process for $\langle \perp, 1, 4, \perp, 2, 6 \rangle$ (in (a)–(d)), $\langle \perp, 1, 5, \perp, 2, 6 \rangle$ (in (e)), $\langle \perp, 1, 5, \perp, 2, 7 \rangle$ (in (f)) and $\langle \perp, 1, 4, \perp, 2, 7 \rangle$ (in (g)).

hash tables can be indexed (we use negative numbers to distinguish indices). Indices are then propagated back upwards to fill the tree until the root (d).

Using a similar process, we can insert vector $\langle \perp, 1, 5, \perp, 2, 6 \rangle$ (e). The hash tables in the tree nodes extended with index -2 storing $[-1, 5]$ in the left child of the root, while the root is extended with the tuple $[-2, -1]$. Notice how sub-vector sharing already occurs since the tuple $[-1, 5]$ in the left child of the root points again to $[\perp, 1]$. In (f), the vector $\langle \perp, 1, 4, \perp, 2, 7 \rangle$ is also added. In this case, only the right child of the root needs to be extended, while the tuple $[-1, -2]$ is added to the root.

With these three vectors in the tree (f), we can now easily add a new vector $\langle \perp, 1, 5, \perp, 2, 7 \rangle$ by merely adding the tuple $[-2, -2]$ to the root of the tree (g). We observe that an entire state vector (of length k in general) can be compressed to a single tuple of integers in the root of the tree, provided that the sub-vectors are already present in the left and the right sub-tree of the root.

The tree containing four vectors in Fig. 3 (g) uses 20 “places” (= 10 tuples in tree nodes) to store four vectors with a total of 24 variables. The more vectors are added, the more sharing can occur and the better the compression. We now recall the worst-case and the best-case compression ratio for this tree database. We make the following reasonable assumptions about their dimensions:

- The respective database stores $n = |V|$ state vectors of k u -bit variables.
- The size of tree tuples is $2w$ bits, and w bits is enough to store both a variable valuation (in a leaf) or a tree reference (in a tree node), hence $u \leq w$.
- Keys can be stored without overhead in tables.³
- k is a power of 2.⁴

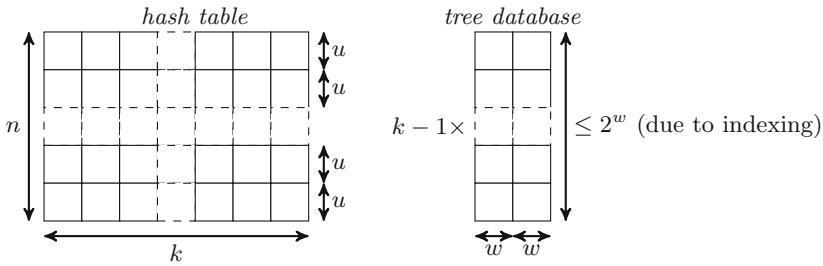


Fig. 4. From left to right: a hash table and a tree table with their dimensions.

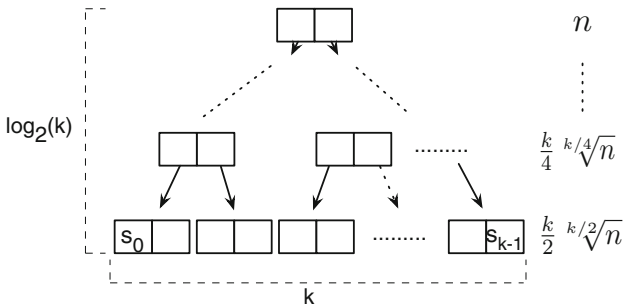


Fig. 5. Optimal entries per tree node level.

Figure 4 provides an overview of the different data structures and the stated assumptions about their dimensions.

To arrive at the worst-case compression scenario (Theorem 2), consider the case where all states $s \in V$ have k identical data values: $V = \{v^k \mid v \in \{1, \dots, n\}\}$, where v^k is a vector of length k : $\langle v, \dots, v \rangle$. No sharing can occur between state vectors in the database, so for each state we store $k - 1$ tuples at the tree nodes.

Theorem 2 ([4]). *In the worst case, the tree database requires at most $k - 1$ tuple entries of $2w$ bits per state vector.*

³ [21] explains in detail how this can be achieved.

⁴ Solely assumed to simplify the formulae below.

Table 1. Theoretical bounds for the compressed state sizes in the tree database and in plain hash table storage. Note that while $u \leq w$, often u, w are in the same ballpark.

Structure	Worst case	Best case
Hash table	ku	ku
Tree database	$2kw - 2w$	$2w + \epsilon w$

The best-case scenario (Theorem 3) is easy to comprehend from the effects of a good combinatorial structure on the size of the parent tables in the tree. If a certain tree table contains d tuple entries, and its sibling contains e entries, than the parent can have up to $d \times e$ entries (all combinations, i.e. the Cartesian product). In a tree that is perfectly balanced ($d = e$ for all sibling tables), the root node has n entries (1 per state), its children have \sqrt{n} entries, its children’s children $\sqrt[4]{n}$, etc. Figure 5 depicts this scenario.

Hence there are a total of $n + 2\sqrt{n} + 4\sqrt[4]{n} + \dots (\log_2(k)\text{times}) \dots + k/2 \sqrt[k/2]{n}$ tuple entries. Dividing this series by n gives a series for the expected number of tuple entries per state: $\sum_{i=0}^{\log_2(k)-1} 2^i \frac{\sqrt[2^i]{n}}{n}$. It is hard to see where this series exactly converges, but Theorem 3 provides an upper bound. The theorem is a refinement of the upper bound established in [4]. Note that the example above of a tree with the four Bakery algorithm states already represents an optimal scenario, i.e., the root table is the cross product of its children.

Theorem 3. *In the best case and with $k \geq 8$, the tree database requires less than $n + 2\sqrt{n} + 2\sqrt[4]{n}(k - 4)$ tuple entries of $2w$ bits to store n vectors.*

Proof. In the best case, the root tree table contains n entries and its children both contain \sqrt{n} entries. The entries in the 4 children’s children of the root represent vectors of size $k/4$. These 4 tree nodes contain each $\sqrt[4]{n}$ entries that each require $k/4 - 1$ tuples taking the worst case according to Theorem 2 (hence also $k \geq 8$). □

Corollary 1 ([21]). *In the best case, the total number of tuple entries l in all descendants of root table is negligible ($l \ll n$), assuming a relatively large number of vectors is stored: $n \gg k^2 \gg 1$.*

Corollary 2 ([21]). *In the best case, the compressed state size approaches $2w$.*

Table 1 lists the achieved compressed sizes for states, as stored in a normal hash table and a tree database. As a simplifying assumption, we take u to be equal w , which can be the case if the tree is specifically adapted to accommodate u bit references.

Performance. We conclude the current section with a note on the performance of the tree database compared to a plain hash table. The tree trades ku bit vector lookups for $k - 1$ of $2u$ -bit tuple lookups in its nodes. Apart from the additional

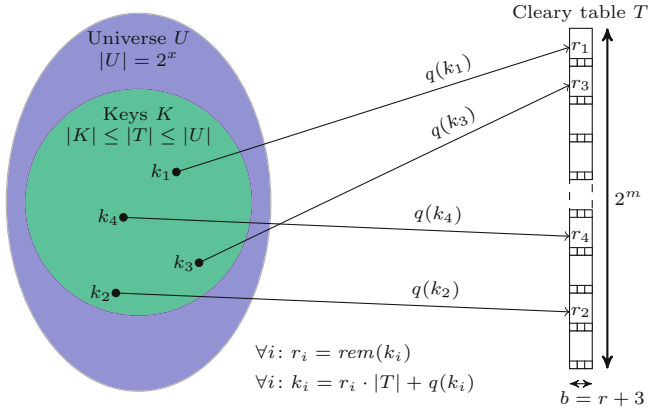


Fig. 7. Cleary table T storing keys K from universe U with three admin. bits/bucket. (We omit that keys should be hashed, with invertible function, for good distribution.)

If, due to collisions, the key is not stored at its home location, additional information is needed. Cleary [8] solved this problem with little overhead by imposing an order on the keys in T and introducing three administration bits per bucket. For details, see [8, 12, 28]. Because of the administration bits, the bucket size b of compact hash tables is $b = r + 3$ bits. The ratio b/x can approach zero arbitrarily close, yielding good compression. For instance, a compact table only needs 5 bits per bucket to store 2^{30} 32-bit keys (Fig. 7).

Compact Tree Database. To create a compact tree database, we replace the hash tables in the tree nodes with compact hash tables.

Let the tree references again be w bits; Tuples in a tree node table are $2w$ bits. The tree node table’s universe therefore contains 2^{2w} tuples. However, tree node tables cannot contain more than 2^w entries, otherwise the entries cannot be referenced (with w -bit indices) by parent tree node tables. As the tree’s root table has no parent, it can contain up to 2^{2w} entries. Let o be the overcommit of the tree root table T_{root} , i.e., $\log_2(|T_{root}|) = 2^{w+o}$ for $0 \leq o \leq w$.

Overcommitting the root table in the tree can yield better reductions as we will see. However, it also limits the subsets of the state universe that the tree can store. Close-to-worst-case subsets might be rejected as the left or right child (2^w tuples max) of the root can grow full before the root does (2^{w+o} tuples max).

We will only focus on replacing the root table with a compact hash table as it dominates the tree’s memory usage in the optimal case according to Corollary 1. The following parameters follow for using a compact hash tables for T_{root} :

- $x = 2w$, (universe bits)
- $m = w + o$, (quotient bits)
- $r = 2w - w - o = w - o$, and (remainder bits)
- $b = 2w - w - o + 3 = w - o + 3$. (bucket bits)

Let the Compact Tree Database be a Tree Database with the root table replaced by a compact hash table with the dimensions provided above, ergo: $n = |V| = |T_{root}| = 2^{w+o} = 2^m$. Theorem 4 gives its best-case memory usage.

Theorem 4 (Compact Tree Best-Case). *In the best case and with $k \geq 8$, the compact tree database requires less than $CT_{opt} \triangleq (w - o + 3)n + 4w\sqrt{n} + 4w\sqrt[4]{n}(k - 4)$ bits to store n vectors.*

Proof. According to Theorem 3, there are at most $n + 2\sqrt{n} + 2\sqrt[4]{n}(k - 4)$ tuples in a tree with optimal storage. The root table contains n of these tuples, its descendants use at most $2\sqrt{n} + 2\sqrt[4]{n}(k - 4)$ bits. The n tuples in the root table can now be stored using $w - o + 3$ bits in the compact hash table buckets instead of $2w$ bits, hence the root table uses $n(w - o + 3)$ bits. \square

Finally, Theorem 5 relates the compact tree compression results to our information theoretical model in Sect. 2, under the reasonable assumptions that $8 \leq k \leq \sqrt[4]{n} + 4$. As a consequence, when the overcommit ($o - 7$ bits) fills the gap of $w - u$ bits between the sizes of references in the tree (w bits) and the sizes of variables (u bits), the optimal compression of the compact tree is approached. If $o - 7 > w - u$, the compact tree can even surpass the compression predicted by our information theoretical model. This is not surprising as the tree with $k = 2$ reduces to a compact hash table, for which a different information theoretical model holds [12, 27].

Theorem 5. *Let CT_{opt} be the best-case compact-tree compressed vector sizes. We have $CT_{opt} \leq \frac{w-o+7}{u}H_{state}$ provided $8 \leq k \leq \sqrt[4]{n} + 4$.*

Proof. According to Theorem 1, $nH_{state} \leq un + \log_2(k)n + 2n$ bits. According to Theorem 4, the compact tree database uses at most $CT_{opt} \triangleq (w - o + 3)n + 4w\sqrt{n} + 4w\sqrt[4]{n}(k - 4)$ bits in the best case and with $k \geq 8$.

We show that $CT_{opt} \leq cH_{state}$ using lower bound Theorem 1 and derive c .

After simplification using $(u - 1) \leq \log_2(y - 1)$ and $0 \leq \log_2(k - 1)$, we obtain: $4w/\sqrt{n} + 4w(k - 4)/n^{3/4} \leq cu - w + o - 3$. As the premise ensures that $n \geq (k - 4)^4$, this can be further simplified to $4w/\sqrt{n} + 4w\sqrt[4]{n}/n^{3/4} \leq cu - w + o - 3$ and then to $8w/\sqrt{n} \leq cu - w + o - 3$.

In an intermediate step, we show that $w/\sqrt{n} \leq 1/2$ under the premises $n \geq (k - 4)^4$ and $k \geq 8$. We have $w \leq \log_2(n)$ in order to accommodate the worst-case compression (see Theorem 2 in Sect. 4). Therefore, we can also prove $\log_2(n)/\sqrt{n} \leq 1/2$. Implied by the two earlier assumptions from the premise, we have $n \geq 256$ for which the inequality indeed holds.

With $w/\sqrt{n} \leq 1/2$, the above gives $4 \leq cu - w + o - 3$ and $\frac{w-o+7}{u} \leq c$.

Therefore, we obtain $CT_{opt} \leq \frac{w-o+7}{u}H_{state}$, provided that $n \geq (k - 4)^4$. \square

5 Experiments

We implemented the Compact Tree in the model checker LTSMIN [22]. This implementation is based on two concurrent data structures: a tree database [21]

and a compact hash table [28], based on Cleary’s approach [8]. The parameters of the Compact Tree Table in this implementation are (for details see [23]):

- $w = 30$ bits (The internal tree references are 30 bit)
- $u = 30$ bits (The state variables can be 30-bit integers, often less is used)
- $o = 2$ bits (The root table fits a maximum of 2^{32} elements)

LTSMIN is a language-independent model checker based on a *partitioned next-state interface* [18]. We exploit this property to investigate the compression ratios of the Compact Tree for four different input types: DVE models written for the DIVINE model checker [1], Promela models written for the SPIN model checker [15], process algebra models written for the MCRL2 model checker [9], and Petri net models from the MCC contest [20]. Table 2 provides an overview of the models in each of these input formats and a justification for the selection criterion used. In total, over 400 models were used in these benchmarks.

Table 2. Input languages and model selection criteria

DVE	All 267 benchmarks from the BEEM database [24] that completed within one hour in (sequential) LTSMIN are selected. (This selection criterium is more stringent than for the other languages, because the set of models is large and the presence of differently sized versions of the same type of model still ensures that the selection is varied.)
Promela	All models currently supported by LTSMIN [2] with the same state count as in SPIN are selected. This includes case studies of the GARP, the i-, x509 and BRP protocols
Petri net	All models from the MCC 2016 competition [20] that are also considered by Jensen et al. [16] and complete within 10 h in (sequential) LTSMIN. (Again this ensures a varied selection, since Jensen et al. [16] only feature instances that resulted in best-case, worst-case and average-case compression using a Trie data structure.)
MCRL2	We selected all industrial case studies from the MCRL2 toolset that completed within 10 h in (sequential) LTSMIN

All experiments ran on a machine with 128 GB memory and 48 cores: four AMD Opteron™ 6168 processors with 12 cores each.

Compression Ratio. Compressed state sizes of our implementation can roughly approach $w - 2 + 3 = 31$ bits or ± 4 Bytes by Corollary 1 and Theorem 4. We first investigate whether this compression is actually reached in practice. Figure 8 plots the compressed sizes of the state vectors against the length of the uncompressed vector. We see that for some models the optimal compression is indeed reached. The average compression is 7.88 Bytes per state. The fact that there is little correlation with the vector length confirms that the compressed size indeed tends to be constant and vectors of up to 1000 Bytes are compressed

to just above 4 Bytes. Figure 9 furthermore reveals that good compression correlates positively with the state space size, which can be expected as the tree can exhibit more sharing.

Only for Petri nets and for DVE models, we find models that exhibit worse compression (between 10 and 15 Bytes per state) even when the state space is large. However, we observed that in these cases, the vector length k is also large, e.g., the two Petri net instances with a compressed size of around 12 have $k > 400$. Based on some earlier informal experiments, we believe that with some variable reordering, these compression might very well be improved to reach the optimum. Thus far, however, we were unable to derive a reordering heuristic that consistently improves the compression.

Runtime Performance and Parallel Scalability. In the introduction, we mentioned the requirement that a database visited set ideally features constant lookup times, like in a normal hash table. To this end, we compare the runtime of the DVE models with the SPIN model checker; a model checker known for its fast state generator.⁵ Figure 10 confirms that the runtimes of LTSMIN with Compact Tree are sequentially on par with those of SPIN, and often even better. We attribute this performance mainly to the incremental vector insertion discussed in Sect. 3 (see Fig. 6). Based on the MCC 2016 [20] results, we believe that LTSMIN’s performance is on par with other Petri net tools as well.

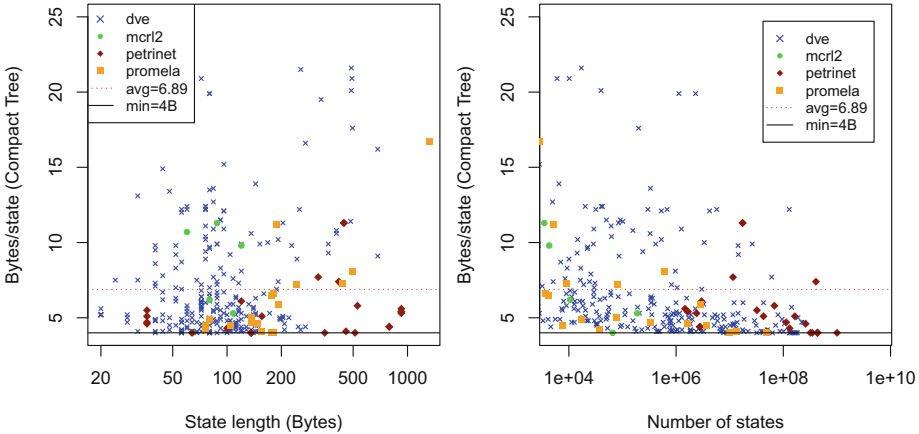


Fig. 8. Compressed sizes in Compact Tree for all benchmarks against the length k of the uncompressed state vector. **Fig. 9.** Compressed sizes in Compact Tree for all benchmarks against the size n of state space.

⁵ The DVE models are translated to Promela and we only selected those (76/267) which preserved state count. This comparison can be examined interactively at <http://fmt.ewi.utwente.nl/tools/ltsmin/performance/> (select LTSmin-clearly-dfs).

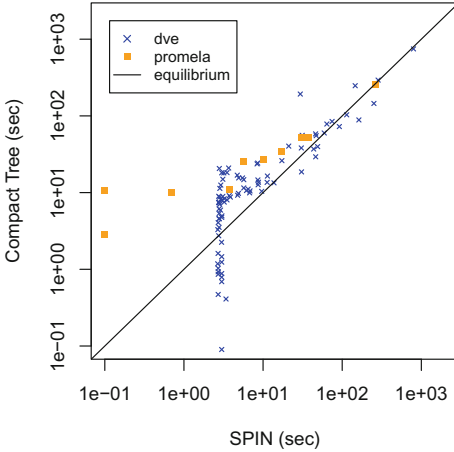


Fig. 10. Sequential runtimes of LTSMIN with Compact Tree and SPIN with optimal settings (as reported in [2]) on (translated) DVE models and Promela models.

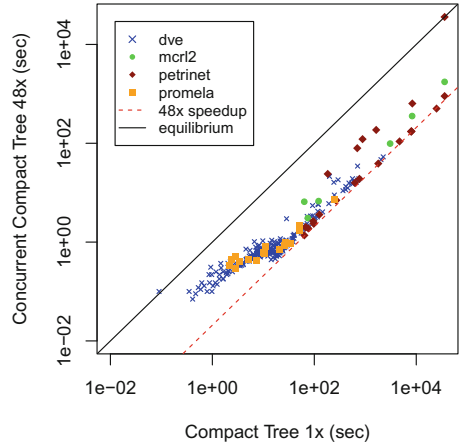


Fig. 11. Runtimes, sequentially and with 48 threads, of LTSMIN with compact tree on all models: DVE, Promela, process algebra and Petri nets.

The measured performance first of all confirms that the Compact Tree satisfies its requirements. Secondly, it provides a good basis for the analysis of parallel scalability (if we had chosen to implement the Compact Tree in a slow scripting language, the slowdown would yield “free” speedup). Figure 11 compares the sequential runtimes to the runtimes with 48 threads. The measured speedup often surpasses 40x, especially when the runtimes are longer and there is more work to parallelize. Speedups are good regardless of input language.

Comparison with Other Data Structures. SPIN’s collapse compression uses the structure in the model to fold vectors, similar as in tree compression. The lower bounds reported in the current paper cannot be reached with collapse due to the n -ary tree and the two levels. Figures 12 and 13 show additional experiments that show an order of magnitude difference in practice.

Jensen et al. [16] propose a Trie for storing state vectors. Tries compress vectors by ensuring sharing between prefixes. BDDs [6] also store state vectors efficiently, however, Jensen et al. [17] figure them too slow for state space exploration. We compared both Tries and BDDs with the Compact Tree and found that (1) the Trie’s compression is less than the Compact Tree though sometimes faster (Figs. 14 and 15), and (2) that BDD’s are not *always* prohibitively expensive with LTSMIN (because it learns the transition relation [18]), but nonetheless hard to compare to Tree Compression (Figs. 17 and 16).

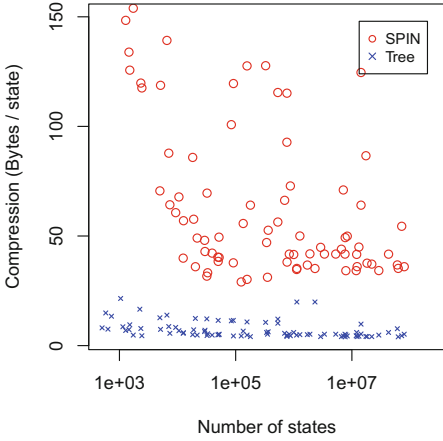


Fig. 12. Compressed sizes per state of LTSMIN with Compact Tree and SPIN with collapse compression [14] on DVE models.

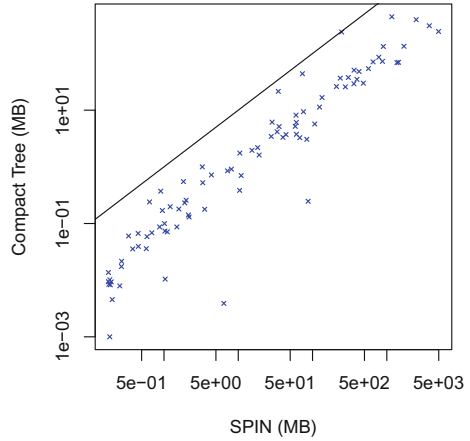


Fig. 13. Absolute memory use of LTSMIN with Compact Tree and SPIN with collapse compression [14] on DVE models.

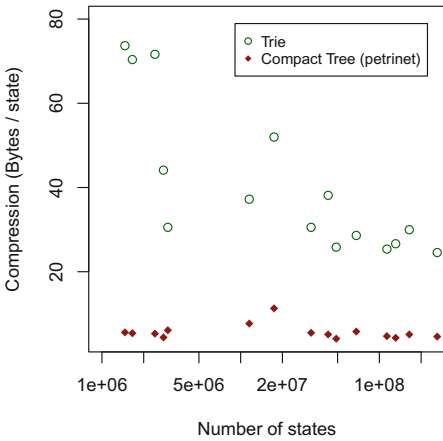


Fig. 14. Memory use per state of LTSMIN with Compact Tree and Trie from [16] on Petri net models.

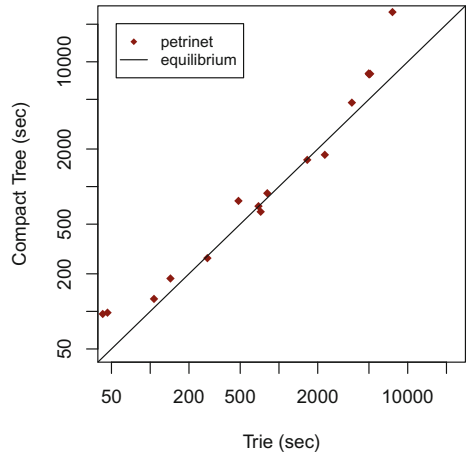


Fig. 15. Runtime (sequential) of LTSMIN with Compact Tree and Trie from [16] on Petri net models.

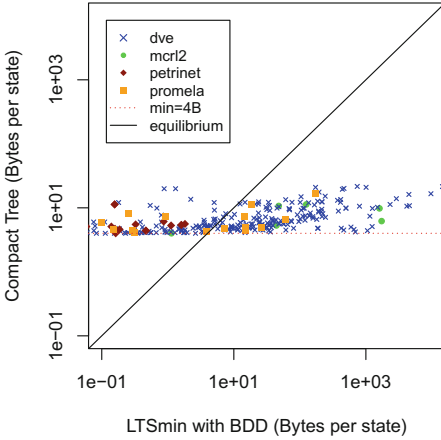


Fig. 16. Memory use per state of LTSMIN with Compact Tree and BDD [3] on MCRL2, Promela and Petri net models.

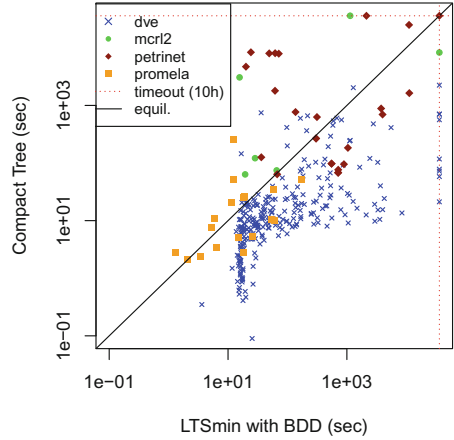


Fig. 17. Runtime (sequential) of LTSMIN with Compact Tree and BDD [3] on MCRL2, Promela and Petri net models.

6 Discussion and Conclusion

The tree compression method discussed here is a more general variant of recursive indexing [14], which only breaks down processes into separate tables. Hash compaction [25] compresses states to an integer-sized hash, but this lossy technique becomes redundant with the compact tree database. Bloom filters [13] still present a worthwhile lossy alternative using only a few bits per state, but of course abandon soundness when applied in model checking.

Evangelista et al. [11] report on a hash table storing incremental differences of successor states (similar to the incremental data structure discussed in Sect. 3). Their partial vectors take $2u + \log(E)$ bits, where E is the set of (deterministic) actions in the model. Defying our requirement of poly-logarithmic for lookups, Evangelista et al. reconstruct full states by reconstructing all ancestors.

A Binary Decision Diagram (BDD) [6] can store an astronomically sized state set using only constant memory (the true leaf). Our information theoretical model suggests however that compressed sizes are merely linear in the number of states (and constant in the length of the state vector). We can explain this with the fact that we only assume locality about inputs. Compression in BDDs, on the other hand, depends on the entire state space. Therefore, we would have to assume structural, global properties to describe the non-linear compression of BDDs (e.g. the input's decomposition into processes, symmetries, etc.).

Much like in BDDs [5], the variable ordering influences the number of nodes in a tree table and thus the compression, as mentioned in Sect. 1. Consider the vector set $\{i, i, j, j \mid i, j \in [1 \dots N]\}$: Only the root node in a compact tree will contain N^2 entries, while the leaf nodes contain N entries. On the other hand, we have no such luck for the set $\{i, j, i, j \mid i, j \in [1 \dots N]\}$. Preliminary research [29]

revealed that the tree's optimum can be reached in most cases for DVE models, but we were unable to find a heuristic to consistently realize this.

Acknowledgements. The author thanks Yakir Vizel for promptly pointing out the natural number as a limit and Tim van Erven for a fruitful discussion.

References

1. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkal, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
2. van der Berg, F., Laarman, A.: SpinS: extending LTSmin with Promela through SpinJa. ENTCS **296**, 95–105 (2013)
3. Blom, S., van de Pol, J., Weber, M.: LTSMIN: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_31
4. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A database approach to distributed state space generation. ENTCS **198**(1), 17–32 (2008)
5. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**, 993–1002 (1996)
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS, pp. 428–439 (1990)
8. Cleary, J.G.: Compact hash tables using bidirectional linear probing. IEEE Trans. Comput. **C-33**(9), 828–834 (1984)
9. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Weselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_15
10. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_25
11. Evangelista, S., Kristensen, L.M., Petrucci, L.: Multi-threaded explicit state space exploration with state reconstruction. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 208–223. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_16
12. Geldenhuys, J., Valmari, A.: A nearly memory-optimal data structure for sets and mappings. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 136–150. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_9
13. Holzmann, G.J.: An analysis of bitstate hashing. In: Dembiński, P., Średniawa, M. (eds.) PSTV 1995. IFIPAICT, pp. 301–314. Springer, Boston (1996). https://doi.org/10.1007/978-0-387-34892-6_19
14. Holzmann, G.J.: State compression in SPIN: recursive indexing and compression training runs. In: Proceedings of 3rd International SPIN Workshop (1997)
15. Holzmann, G.J.: The model checker SPIN. IEEE TSE **23**, 279–295 (1997)

16. Jensen, P.G., Larsen, K.G., Srba, J.: PTrie: data structure for compressing and storing sets via prefix sharing. In: Hung, D., Kapur, D. (eds.) ICTAC 2017. LNCS, vol. 10580, pp. 248–265. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67729-3_15
17. Jensen, P.G., Larsen, K.G., Srba, J., Sørensen, M.G., Taankvist, J.H.: Memory efficient data structures for explicit verification of timed systems. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 307–312. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_26
18. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
19. Katz, S., Peled, D.: An efficient verification method for parallel and distributed programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1988. LNCS, vol. 354, pp. 489–507. Springer, Heidelberg (1989). <https://doi.org/10.1007/BFb0013032>
20. Kordon, F., et al.: Complete results for the 2016 edition of the model checking contest, June 2016. <http://mcc.lip6.fr/2016/results.php>
21. Laarman, A., van de Pol, J., Weber, M.: Parallel recursive state compression for free. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22306-8_4
22. Laarman, A., van de Pol, J., Weber, M.: Multi-core LTSMIN: marrying modularity and scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_40
23. Laarman, A.: Scalable multi-core model checking. Ph.D. thesis, UTwente (2014)
24. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_17
25. Stern, U., Dill, D.L.: Improved probabilistic verification by hash compaction. In: Camurati, P.E., Evesking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60385-9_13
26. Valmari, A.: Error detection by reduced reachability graph generation. In: APN, pp. 95–112 (1988)
27. Valmari, A.: What the small Rubik’s cube taught me about data structures, information theory, and randomisation. STTT **8**(3), 180–194 (2006)
28. van der Vegt, S., Laarman, A.: A parallel compact hash table. In: Kotásek, Z., Bouda, J., Černá, I., Sekanina, L., Vojnar, T., Antoš, D. (eds.) MEMICS 2011. LNCS, vol. 7119, pp. 191–204. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25929-6_18
29. de Vries, S.H.S.: Optimizing state vector compression for program verification by reordering program variables. In: 21st Twente SConIT, vol. 21, 23 June 2014
30. Wahl, T., Donaldson, A.: Replication and abstraction: symmetry in automated formal verification. Symmetry **2**(2), 799–847 (2010)