Innovations in Systems and Software Engineering (2019) 15:235–251
https://doi.org/10.1007/s11334-019-00341-7

S.I. : NFM2018

Optimal compression of combinatorial state spaces

Alfons Laarman[1]

Received: 1 October 2018 / Accepted: 2 May 2019 / Published online: 20 May 2019

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 using only the assumption of locality in the model checking input. The theory shows that a set of n vectors with k slots can be compressed to a single slot plus $\mathcal{O}(\log_2(k))$ overhead. Using a binary tree in combination with a compact hash table, we then analytically show that this compression can be attained in practice, without compromising fast query times for state vector lookups. Our implementation of this Compact Tree can compress $n > 2^{32}$ state vectors of arbitrary length $k \ll n$ down to 32 bits per vector. This compression is lossless. Experiments with over 350 inputs in five different model checking formalisms confirm that the lower bound is reached in practice in a majority of cases, confirming the combinatorial nature of state spaces.

Keywords Model checking · Information theory · State compression · Lossless compression · Tree compression · Binary trees · Decision diagrams · BDDs

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 `N[i]` (for each process `i`). For two processes, the state vector of this program consists of the two program counters (pcs) and all variables, i.e., $\langle \texttt{E}[0], N[0], \text{pc}_0, \texttt{E}[1], N[1], \text{pc}_1 \rangle$.[1] Their respective domains are:

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

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 $\iota \triangleq \langle \bot, 0, 0, \bot, 0, 0 \rangle$. 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. The next-state function furthermore accounts for all parallel interleavings, by implementing a worst-case scheduler. For example, the successors of the initial state are:

$$\begin{aligned} &\text{NEXT- STATE}(\langle \bot, 0, 0, \bot, 0, 0 \rangle) \\ &= \{\langle \top, 0, 1, \bot, 0, 0 \rangle, \langle \bot, 0, 0, \top, 0, 1 \rangle\} \end{aligned}$$

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 whether it can reach a state from the set Error. For the Bakery algorithm with two threads, errors are violations of the mutual exclusion principle:

$$\begin{aligned} \text{Error} \triangleq \{\langle b_0, n_0, 7, b_1, n_1, 7 \rangle \mid b_0, b_1 \in \{\top, \bot\}, \\ n_0, n_1 \in [0 \dots 2]\} \end{aligned}$$

[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 different orderings.

✉ Alfons Laarman
a.w.laarman@liacs.leidenuniv.nl

[1] Leiden University, Leiden, The Netherlands

```
    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

So, Error is a collection of all states where both processes reside in their critical section, i.e., where both their program counter locations equal 7.

Algorithm 1: The reachability procedure in a model checker.

```
Data: ι, NEXT- STATE
Result: {error, correct}
1  V := ∅
2  Q := {ι}
3  while Q ≠ ∅ do
4      Q := Q \ {s} for s ∈ Q
5      V := V ∪ {s}
6      for s′ ∈ NEXT- STATE(s) do
7          if s′ ∉ V then
8              if s′ ∈ Error then
9                  return error
10             Q := Q ∪ {s′}
11 return correct
```

For completeness, Algorithm 1 shows the basic reachability procedure. The more states the reachability procedure can process, the more powerful the model checker is, 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 [24,38], symmetry reduction [15,43], BDDs [3,9], 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.,

$$\begin{aligned}&\text{NEXT- STATE}(\langle \bot, 1, 4, \bot, 2, 6\rangle)\\ &= \{\langle \bot, 1, \mathbf{5}, \bot, 2, 6\rangle\,, \langle \bot, 1, 4, \bot, 2, \mathbf{7}\rangle\}\end{aligned}$$

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 $\langle \bot, 1, 4, \bot, 2, 6\rangle$ can be reached from the initial state ι, we indeed saw four different vectors sharing large sub-vectors with their predecessors (underlined here):
$\langle \bot, 0, 0, \bot, 0, 0\rangle \longrightarrow \langle \top, \underline{0}, 1, \underline{\bot, 0, 0}\rangle, \langle \underline{\bot, 0, 0}, \top, \underline{0}, 1\rangle$
$\langle \bot, 1, 4, \bot, 2, 6\rangle \longrightarrow \langle \underline{\bot, 1}, 5, \underline{\bot, 2, 6}\rangle, \langle \underline{\bot, 1, 4, \bot, 2}, 7\rangle$

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, e.g., 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 article, we investigate the lower bound on the space requirements of typical state spaces occurring in model checking consisting of n vectors with k u-bit slots each. 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 of approximately $u + \log_2(k) + \epsilon$ bits for the storage requirements of our "state

space stream" (see Theorem 1), far below the uncompressed space requirements of $k \cdot u$ bits.

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 an additional requirement that the query time, the time it takes to look up and insert individual state vectors, is constant with respect to the number of states stored. The storage technique suggested by the information theoretical model, i.e., maintaining differences between successor states, does not satisfy this requirement as it would require the iteration of all inserted vectors in the worst case [16]. Therefore, we utilize a binary tree in combination with a *compact hash table* [12].

In our Compact Tree, the compact hash table has 2^{w+o} buckets storing tuples of w-bit sized pointers from the binary tree. It does so compactly by using the storage location in memory as information [12]. Through this technique, it can chop off $w + o - 3$ bits from the $2w$ bits of each tuple. (The three bits are required for bookkeeping.) Under reasonable assumption on the state length k, number of states n, pointer size w and state slot size u, we analytically show that the Compact Tree can even surpass the information theoretical lower bound when $w - o + 6 \leq u$ (see Theorem 5).[2]

According to the same best-case analysis, our specific implementation of the Compact Tree can compress arbitrarily large state descriptors down to only one 32-bit integer per state (our benchmarks contain inputs with state vectors of a thousand bytes long). Extensive experimentation in Sect. 5 with diverse input models in five different input languages shows moreover that this compression is also reached in practice, and with little computational overhead due to incremental insertion in the tree (see Sect. 3.3).

Surprising is perhaps that the Compact Tree can compress states down to x bits while storing far more than 2^x states. This property of representing a set with less space per element than required for its unique identification is inherited from the compact hash table. As mentioned above, this is realized by using the place in memory as information. Indeed, the case study in Sect. 5.9 demonstrates that the Compact Tree implementation can store $2^{35.6}$ states using only 32.5 bits per state.

This article is an extended version of [33]; the following contributions are added:

- New experiments using the ProB models [2,28], adding more evidence that compact tree compression works regardless of the input language.
- An extensive comparison of compact tree compression with both Binary Decision Diagram and Multi-valued Decision Diagrams in Sect. 5.5.
- A better lower bound in Theorem 5 and additional derivations and proofs for the lower bounds in Theorem 1 and Theorem 5.
- A case study of the GARP specification from [26] in Sect. 5.9 demonstrating that the Compact Tree implementation performs as predicted.

[2] The fact that the lower bound can be surpassed is not entirely surprising as the model to derive the bound in Sect. 2 only takes into account locality as a source of non-entropy. This observation is further discussed in Sect. 4.2.

2 An information theoretical lower bound

The fact that state spaces have combinatorial values is related to the fact that states 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 the 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:

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

As a simplification, let states consist of k variables. By storing full states in the queue Q, the predecessor state is

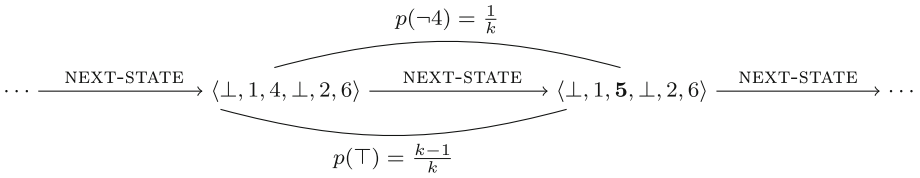

Fig. 2 The states generated with the NEXT- STATE function seen as a stream exhibiting locality. To derive a lower bound, we assume that locality changes only one value in each new vector, i.e., each vector that has to be stored. As there are k variables in the vector, the resulting probability that a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

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:

$$\left\langle v_0^0, \ldots v_{k-1}^0 \right\rangle, \left\langle v_0^1, \ldots v_{k-1}^1 \right\rangle, \ldots, \left\langle v_0^{n-1}, \ldots v_{k-1}^{n-1} \right\rangle$$

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 under-approximate 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):

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

Let $\langle d_0^0, \ldots d_{k-1}^0 \rangle, \langle d_0^1, \ldots d_{k-1}^1 \rangle, \ldots, \left\langle d_0^{n-1}, \ldots d_{k-1}^{n-1} \right\rangle$, 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}.[3] This results in the following definition of entropy per variable in the stream:

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

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

$$\begin{aligned}
H_{\text{var}}(d_j^i) &= -\frac{k-1}{k} \log_2 \left(\frac{k-1}{k} \right) + \sum_{n=1}^{y-1} -\frac{1}{k(y-1)} \log_2 \left(\frac{1}{k(y-1)} \right) \\
&s = -\frac{k-1}{k} \log_2 \left(\frac{k-1}{k} \right) - \frac{1}{k} \log_2 \left(\frac{1}{k(y-1)} \right) \\
H_{\text{state}} &\triangleq H_{\text{state}}(d_0^i, \ldots, d_{k-1}^i) = \sum_{j=0}^{k-1} H_{\text{var}}(d_j^i) \\
&= -(k-1) \log_2 \left(\frac{k-1}{k} \right) - \log_2 \left(\frac{1}{k(y-1)} \right) \\
&= -(k-1) \log_2 \left(\frac{k-1}{k} \right) - \log_2 \left(\frac{1}{y-1} \right) + \log_2(k) \\
&= -(k-1)(\log_2(k-1) - \log_2(k)) - \log_2 \left(\frac{1}{y-1} \right) + \log_2(k) \\
&= -k \log_2(k-1) + k \log_2(k) + \log_2(k-1) \\
&\quad - \log_2(k) - \log_2 \left(\frac{1}{y-1} \right) + \log_2(k) \\
&= -k \log_2(k-1) + k \log_2(k) + \log_2(k-1) + \log_2(y-1) \\
&= \log_2(y-1) + \log_2(k-1) + k \log_2 \left(\frac{k}{k-1} \right) \qquad (1)
\end{aligned}$$

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 bounded by:*

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

[3] 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 $H_{\text{state}} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$.

$$\log_2(y-1) + \log_2(k-1) + k\log_2(\frac{k}{k-1})$$
$$\overset{?}{\leq} \log_2(y) + \log_2(k) + 2$$
$$\log_2(y) + \log_2(k) + k\log_2(\frac{k}{k-1})$$
$$\overset{?}{\leq} \log_2(y) + \log_2(k) + 2 \quad \text{[increase left]}$$
$$k\log_2(\frac{k}{k-1}) \overset{?}{\leq} 2$$
$$\log_2(\frac{k}{k-1}) \overset{?}{\leq} 2/k$$
$$\frac{k}{k-1} \overset{?}{\leq} \sqrt[k]{4}$$
$$1 + \frac{1}{k-1} \overset{?}{\leq} \sqrt[k]{4}$$
$$(1 + \frac{1}{k-1})^k \overset{?}{\leq} 4$$

For $k = 2$ (recall that $k > 1$), we have $(1 + \frac{1}{k-1})^k = 4$. In the range $[2, \infty)$, this function monotonically decreases toward the limit $\lim_{k\to\infty}(1+\frac{1}{k-1})^k = \lim_{k\to\infty}(1+\frac{1}{k})^k = e$. Hence, it holds that $(1 + \frac{1}{k-1})^k \leq 4$ and $H_{\text{state}} \leq \log_2(y) + \log_2(k) + 2 = u + \log_2(k) + 2$ for $k > 1$.

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

$$\log_2(y-1) + \log_2(k-1) + 1 \overset{?}{\leq} \log_2(y-1)$$
$$+ \log_2(k-1) + k\log_2(\frac{k}{k-1})$$
$$1 \overset{?}{\leq} k\log_2(\frac{k}{k-1})$$
$$1/k \overset{?}{\leq} \log_2(\frac{k}{k-1})$$
$$\sqrt[k]{2} \overset{?}{\leq} \frac{k}{k-1}$$
$$\sqrt[k]{2} \overset{?}{\leq} 1 + \frac{1}{k-1}$$
$$2 \overset{?}{\leq} (1 + \frac{1}{k-1})^k$$

Again for $k = 2$, we have $(1+\frac{1}{k-1})^k = 4$ and $\lim_{k\to\infty}(1+\frac{1}{k-1})^k = e$ (monotonically decreasing), hence $\log_2(y-1) + \log_2(k-1) + 1 \leq H_{\text{state}}$. □

The entropy of states H_{state} provides a lower bound on the number of bits required to encode states. Consequently, the upper bound on the entropy in Theorem 1 gives a conservative estimation of the lower bound. Intuitively, the upper bound from Theorem 1 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) 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 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 sub-vectors while also achieving poly-logarithmic lookup times in the worst case. This is the first step toward 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 [29].

3.1 Tree compression

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 3a 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 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 $\langle \perp, 1, \mathbf{5}, \perp, 2, 6\rangle$ (e). The hash tables in the tree nodes

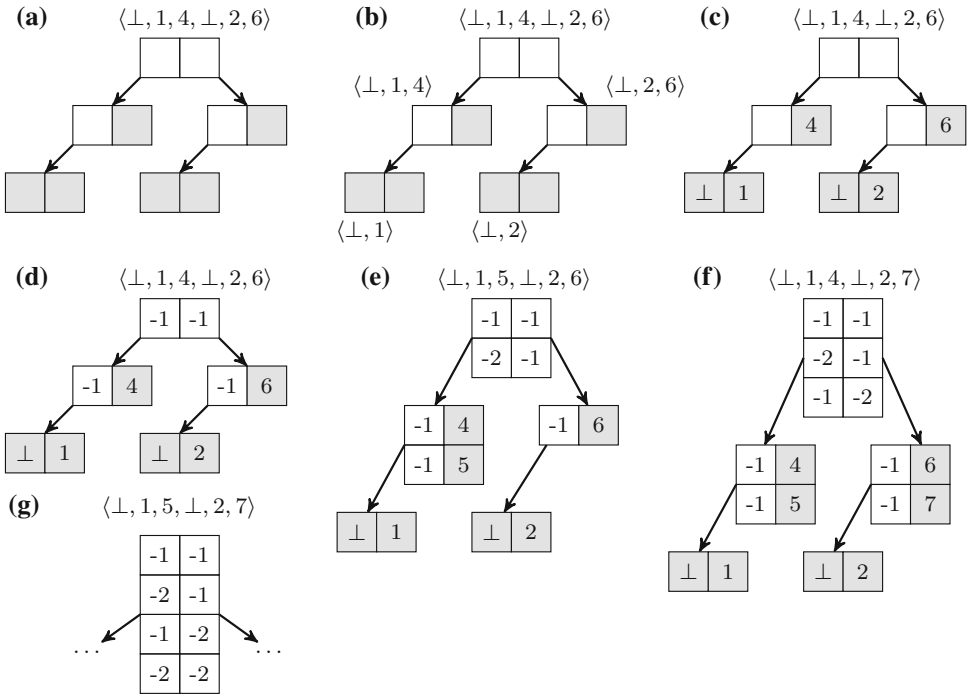

Fig. 3 Tree folding process for $\langle \bot, 1, 4, \bot, 2, 6\rangle$ (in **a**–**d**), $\langle \bot, 1, 5, \bot, 2, 6\rangle$ (in **e**), $\langle \bot, 1, 5, \bot, 2, 7\rangle$ (in **f**) and $\langle \bot, 1, 4, \bot, 2, 7\rangle$ (in **g**)

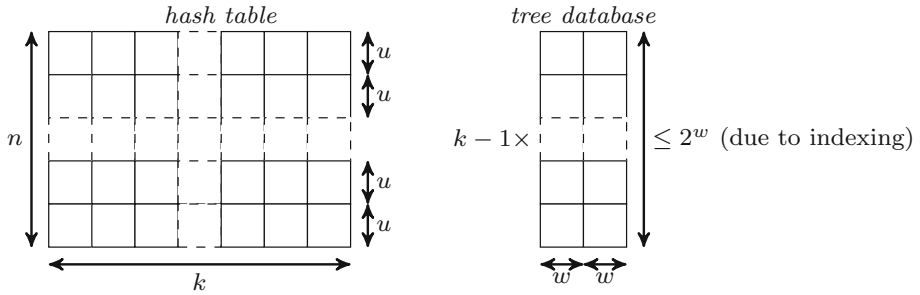

Fig. 4 From left to right: a hash table and a tree table with their dimensions

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 [⊥|1]. In (f), the vector $\langle \bot, 1, 4, \bot, 2, 7\rangle$ 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 $\langle \bot, 1, 5, \bot, 2, 7\rangle$ 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.

3.2 Analysis of compression ratios

The tree containing four vectors in Fig. 3g 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 ratios 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 are 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.[4]
- k is a power of 2.[5]

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

[4] [29] explains in detail how this can be achieved.

[5] Solely assumed to simplify the formulae below.

identical data values: $V = \{v^k \mid v \in \{1, \ldots, n\}\}$, where v^k is a vector of length k: $\langle v, \ldots, v \rangle$. 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.*

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, then 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 $n + 2\sqrt{n} + 4\sqrt[4]{n} + \cdots$ ($\log_2(k)$times) $\cdots + k/2\ \sqrt[k/2]{n}$ 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 $n + 2\sqrt{n} + \sqrt[4]{n}(k - 4)$ 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 four children's children of the root represent vectors of size $k/4$. These four tree nodes contain each of the $\sqrt[4]{n}$ entries that each require $k/4 - 1$ tuples taking the worst case according to Theorem 2 (hence also $k \geq 8$). □

Corollary 1 ([29]) *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 ([29]) *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.

3.3 Poly-logarithmic-time tree updates by incremental insertion

The Compact Tree trades ku bit vector lookups (in a plain hash table) for $k - 1$ of $2u$-bit tuple lookups in its nodes, assuming $w \approx u$. This makes the data structure already constant time with respect to the number of vectors n, as required in Sect. 1. Moreover, the tree requires only few additional data accesses compared to a hash table, i.e., $\mathcal{O}(ku - 2u)$ bits (assuming hash table accesses are indeed amortized constant time).

However, far worse for modern computers with steep memory hierarchies is that the Compact Tree makes $k - 1$ times more random memory accesses compared to a plain table (which, after all, can store key data consecutively in memory). Luckily, we can exploit locality again to speed up tree lookups by keeping the tree pointers of the predecessor state in the search stack (Q), as explained in [29]. Figure 6 illustrates this. The resulting *incremental insertion* yields a poly-logarithmic query time of $\mathcal{O}(\log(k) \times u)$, which is even faster than $\mathcal{O}(ku)$ time required by a plain hash table, but still requires $\log(k)$ times more random memory accesses (note that both are still constant in n). Nonetheless, in the case of good compression, the lower tables in the tree typically contain fewer entries which can more easily be cached.

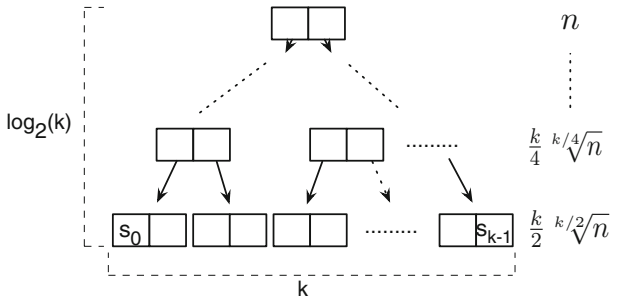

Fig. 5 Optimal entries per tree node level

Table 1 Theoretical bounds for the compressed state sizes in the tree database and in plain hash table storage

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

Note that while $u \leq w$, often u, w are in the same ballpark

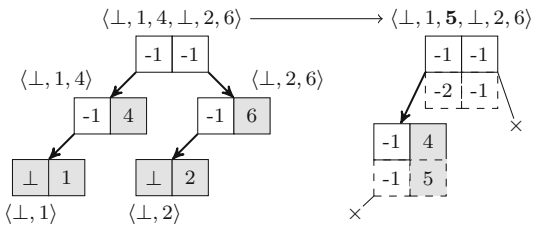

Fig. 6 Incremental insertion of state $\langle \bot, 1, \mathbf{5}, \bot, 2, 6 \rangle$. Only a small part of the tree needs to be updated (dashed boxes), because the predecessor state $\langle \bot, 1, 4, \bot, 2, 6 \rangle$ is used to look up unchanged parts (the crosses in the tree of the successor state)

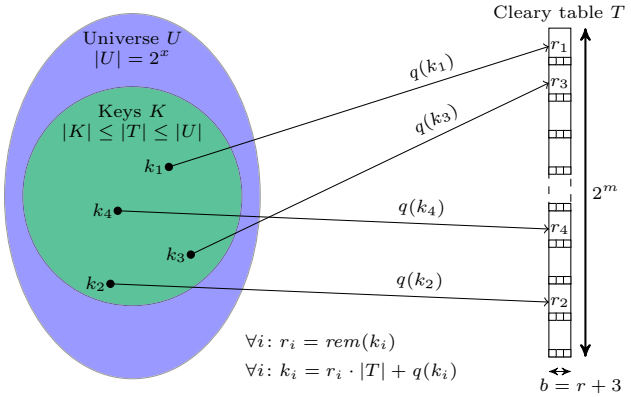

Fig. 7 Cleary table T storing keys K from universe U with three admin. bits/bucket. (We omit that keys should be hashed, with invertible function, for good distribution.)

4 A novel compact tree

The current section shows how a normal tree database can be extended to reach the information theoretical optimum using a compact hash table.

4.1 Hash tables and compact hash tables

A hash table stores a subset of a large *universe* U of keys and provides the means to look up individual keys in constant time. It uses a *hash function* to calculate an address h from the unique *key*. The entire key is then stored at its hash or home location in a table T (an array of *buckets*): $T[h] \leftarrow$ key. Because typically $|U| \gg |T|$, multiple keys may have the same hash location. These so-called *collisions* are handled by calculating alternate hash locations and inserting the key there if empty. This process is known as *probing*. For this reason, the entire key needs to be stored in it, to distinguish which key is currently mapped to a bucket of T (Fig. 7).

Observe, however, that in the case that $|U| \leq |T|$, the table can be replaced with a *perfect hash function* and a bit array. *Compact hashing* [12] generalizes this idea for the case $|U| > |T|$. (The table size is relatively close to the size of the universe.) The compact table first splits a key k into a quotient $q(k)$ and a remainder $\text{rem}(k)$, using a reversible operation, e.g., $q(k) = k\,\%\,|T|$ and $\text{rem}(k) = k/|T|$. When the key is $x = \lceil log_2(|U|)\rceil$ bits, the quotient $m = \lceil log_2(|T|)\rceil$ bits and the remainder $r = x - m$ bits. The quotient is used for addressing in T (like in a normal hash table). Now, only the remainder is stored in the bucket. The complete key can now be reconstructed from the value in T and the home location of the key. If, due to collisions, the key is not stored at its home location, additional information is needed. Cleary [12] 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 [12,17,42]. 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 five bits per bucket to store 2^{30} 32-bit keys.

4.2 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., $\log_2(|T_{\text{root}}|) = 2^{w+o}$ 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 grows full before the root (2^{w+o} tuples max).

We will only focus on replacing the root table T_{root} 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 immediately:

- $x = 2w$, (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: $n = |V| = |T_{\text{root}}| = 2^{w+o} = 2^m$. 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 $CT_{opt} \triangleq (w-o+3)n + 4w\sqrt{n} + 2w\sqrt[4]{n}(k-4)$ bits to store n vectors.*

Proof According to Theorem 3, there are at most $n + 2\sqrt{n} + \sqrt[4]{n}(k-4)$ tuples in a tree with optimal storage. The root table contains n of these tuples, its descendants use at most $2\sqrt{n} + \sqrt[4]{n}(k-4)$ 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. □

Finally, Theorem 5 relates the compact tree compression results to our information theoretical model in Sect. 2, under the reasonable assumption that $8 \leq k \leq \sqrt[4]{n} + 4$. It shows that CT_{opt} can approach H_{state} up to a factor $\frac{w-o+6}{u}$. As a

consequence, when the overcommit ($o-6$ 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 is approached with the compact tree. If $o-6 > w-u$, 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 [17,39].

Theorem 5 *Let CT_{opt} be the best-case compact tree compressed vector sizes. We have $CT_{opt} \leq H_{state}$ provided that $w-o+6 \leq u$ and $8 \leq k \leq \sqrt[4]{n}+4$.*

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

We now derive c in $\text{CT}_{\text{opt}} \leq cH_{\text{state}}$ using the lower bound from Theorem 1.

$$
\begin{aligned}
&(w-o+3)n+4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} c\log_2(y-1)n+c\log_2(k-1)n+cn\\
&wn-on+3n+4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} c\log_2(y-1)n+c\log_2(k-1)n+cn\\
&(w-o-c+3)n+4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} c\log_2(y-1)n+c\log_2(k-1)n \quad [-cn]\\
&(w-o-c+3)n+4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} c(u-1)n+c\log_2(k-1)n \quad [r.r.^{6}]\\
&4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} (c(u-1)-w+o+c-3)n+c\log_2(k-1)n \quad [-..]\\
&4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} (cu-w+o-3)n+c\log_2(k-1)n\\
&4w\sqrt{n}+2w\sqrt[4]{n}(k-4)\\
&\quad\overset{?}{\leq} (cu-w+o-3)n \quad [\text{reduce right by } c\log_2(k-1)n)]\\
&4w/\sqrt{n}+2w(k-4)/n^{3/4}\\
&\quad\overset{?}{\leq} cu-w+o-3 \quad [\text{divide by } n]\\
&4w/\sqrt{n}+2w\sqrt[4]{n}/n^{3/4}\\
&\quad\overset{?}{\leq} cu-w+o-3 \quad [\text{increase left by } n \geq (k-4)^4]\\
&4w/\sqrt{n}+2w/\sqrt{n} \overset{?}{\leq} cu-w+o-3\\
&6w/\sqrt{n} \overset{?}{\leq} cu-w+o-3\\
&3 \overset{?}{\leq} cu-w+o-3 \quad [\text{increase left by } w/\sqrt{n} \leq 1/2^{7}]\\
&w-o+6 \overset{?}{\leq} cu \quad [+w-o+3]\\
&\frac{w-o+6}{u} \overset{?}{\leq} c
\end{aligned}
$$

Taking $c=1$, we obtain that $\text{CT}_{\text{opt}} \leq H_{\text{state}}$ provided that $w-o+6 \leq u$ and $n \geq (k-4)^4$. □

5 Experiments

We implemented the Compact Tree in the model checker LTSMIN [30]. This implementation is based on two concurrent data structures: a tree database [29] and a compact hash table [42], based on Cleary's approach [12]. The parameters of the Compact Tree Table in this implementation are (for details see [32]):

- $w = 30$ 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* [23]. 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 [20], process algebra models written for the MCRL2 model checker [13], Petri net models from the MCC contest [27], and EventB/ProB models from LTSMIN's ProB frontend [2,28]. 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.

We compare the Compact Tree against different compressed and uncompressed data structures: a hash table, SPIN's collapse tables [19], Tries [21], Binary Decision Diagrams (BDDs) [6,8] and Multi-Valued Decision Diagrams (MDDs) [35,41].

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

5.1 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

[6] Reduce right-hand side taking $u-1 < \log(y-1)$.

[7] We show that $w/\sqrt{n} \leq 1/2$ under the assumption that $n \geq (k-4)^4$ (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 derive $\log_2(n)/\sqrt{n} \leq 1/2$. Implied by the two earlier assumptions, we have $n \geq 256$ for which the inequality indeed holds.

Table 2 Input languages and model selection criteria

DVE	All 267 benchmarks from the BEEM database [36] that completed within 1 h 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 [40] 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 [27] that are also considered by Jensen et al. [21] and complete within 10 h in (sequential) LTSMIN. (Again, this ensures a varied selection, since Jensen at al. [21] 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
EventB/ProB	All models from [28] which complete within 10 h in sequential LTSMIN

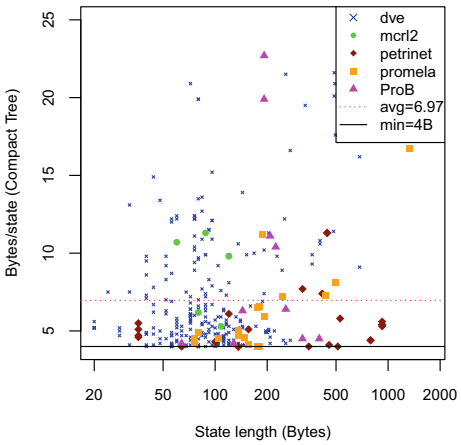

Fig. 8 Compressed sizes in Compact Tree for all benchmarks against the length k of the uncompressed state vector

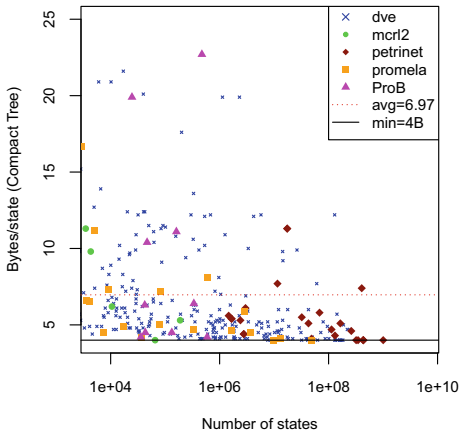

Fig. 9 Compressed sizes in Compact Tree for all benchmarks against the size of the state space n

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 6.97 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 four 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, this 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.

5.2 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.[8] 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 [27] results, we believe that LTSMIN's performance is on par with other Petri net tools as well.

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

[8] 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-cleary-dfs).

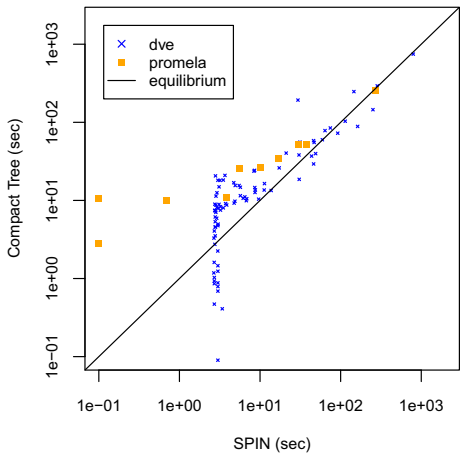

Fig. 10 Sequential runtimes of LTSMIN with Compact Tree and SPIN with optimal settings (as reported in [40]) on (translated) DVE models and Promela models

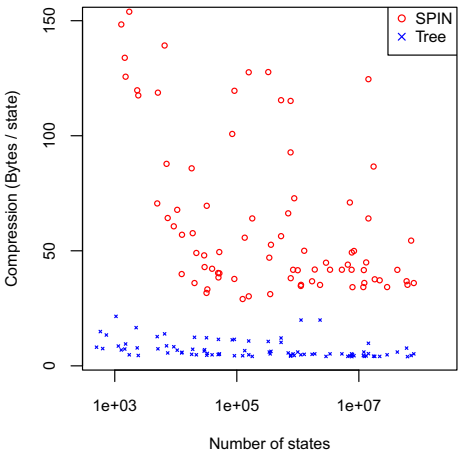

Fig. 12 Compressed sizes per state of LTSMIN with Compact Tree and SPIN with collapse compression [19] on DVE models

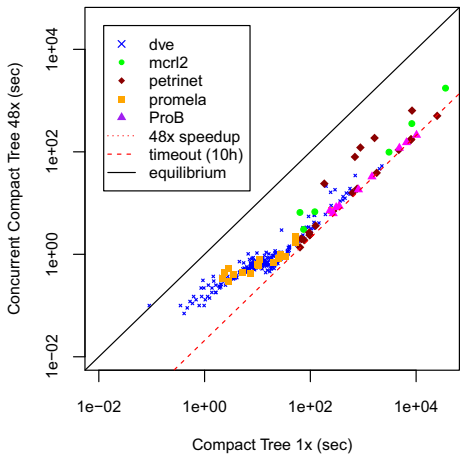

Fig. 11 Runtimes, sequentially and with 48 threads, of LTSMIN with compact tree on all models: DVE, Promela, ProB, MCRL2 and Petri nets

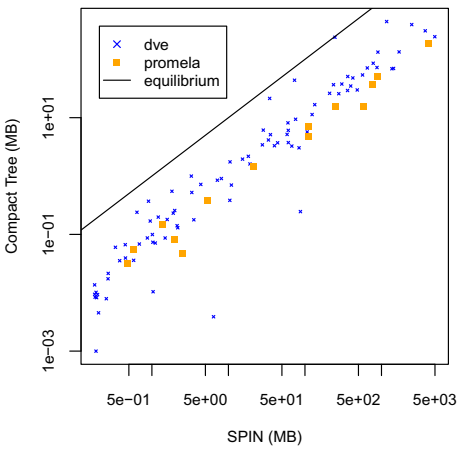

Fig. 13 Absolute memory use of LTSMIN with Compact Tree and SPIN with collapse compression [19] on DVE models

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.

5.3 Comparison with SPIN's collapse compression

SPIN's collapse compression uses the process structure in the input to fold vectors, similar as in tree compression, but with only one table per process, whereas the tree continues splitting vectors until only tuples are left. The lower bounds reported in the current paper cannot be reached with collapse due to its n-ary tree structure and limit to two levels. Our benchmarks compare the Compact Tree with SPIN's collapse compression in both per state compressed size (see Fig. 12) and total memory use (see Fig. 13). We used all DVE inputs that were translated to Promela and have the same state count in SPIN as in LTSMIN. Both the compression and the total memory use of the Compact Tree improve upon collapse by an order of magnitude.

5.4 Comparison with trie compression

Jensen et al. [21] 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. [22] figure them too slow for state space exploration. The tool from [21] implements reachability with Tries for Petri nets. We compare it to the compact tree in LTSMIN in Figs. 14 and 15. We see that its compression correlates positively with the state space size. With its near-optimal compressions for the Petri net models, the Compact Tree provides at least a factor 2 improvement over the Trie. The Trie however exhibits better runtime performance, especially for the small and large state spaces. We suspect that the Trie experiences more caching benefits for small problems and

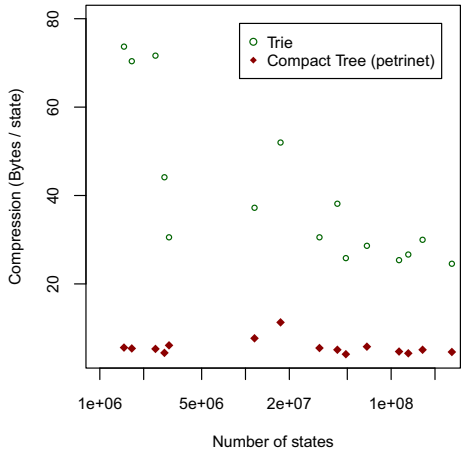

Fig. 14 Memory use per state of LTSMIN with Compact Tree and Trie from [21] on Petri net models

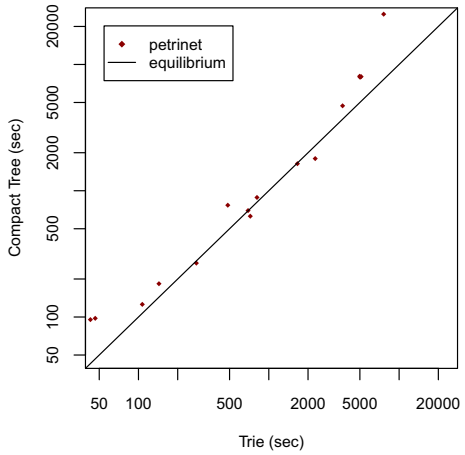

Fig. 15 Runtime (sequential) of LTSMIN with Compact Tree and Trie from [21] on Petri net models

that the hash table probes in the Compact Tree become more expensive for larger ones, as shown in [42].

5.5 Comparison with BDDs

Model checking with BDDs is done semi-symbolically in LTSMIN [3], using its partitioned next-state interface [23]. Each next-state partition represents one action in the underlying modeling formalism. (LTSMIN is language independent, but an action in turn can be, e.g., a statement in a process guarded by a program counter.) Second, the model checker creates an empty BDD representing the transition relation for each partition. (The relations are interpreted conjunctively, implementing asynchronous behavior in the input [7].) These relation BDDs are projected to the variables involved in the underlying action, which in many cases involve just a few variables, e.g., a program counter update and a variable reference/update. Third, starting from the initial state, the model checker fills the relation by calling the next-state function and adding the result to the relation BDD. Because the BDDs are (often) defined over a subset of all variables, the learning terminates after a while and the closure is computed fully symbolically inside the BDDs.

When the reachability procedure converges to a fixpoint, the BDD representing the visited set encodes all reachable states. However, intermediary (non-fixpoint) visited set BDDs might be much larger than the final visited set (as not all subsets are efficiently represented by BDDs). (For this reason, symbolic reachability with BDDs is rather sensitive to the search order used [10,41].) On the other hand, (compact) tree compression space requirements grow monotonically with the number of inserted vectors, making them insensitive to search orders. (No subset ever takes more space than storing the entire state space.) Therefore, it could be argued that symbolic BDD-based model checking is limited by the largest BDD that needs to be stored during the entire reachability procedure. However, we are strictly interested in compression here—not in investigating the smallest possible intermediary BDDs with different search orders—and hence focus on the final BDD representing all reachable states. Nonetheless, Sect. 5.7 investigates the difference in size of the final decision diagram with the peak intermediary decision diagram.

We compare both the runtime of the model checking procedure and compressed sizes of the visited set as BDDs with those of tree compression. Apart from the intermediary visited set sizes, another point should be raised about this comparison. The semi-symbolic approach might not be as efficient as a fully symbolic procedure, such as found in model checkers such as NuSMV [11], since LTSMIN learns the transition relation on the fly. On the other hand, LTSMIN allows for more freedom in the next-state function implementation, e.g., multiplication of integer variables. Therefore, we can expect similar unwieldy BDD sizes for such hard inputs in NuSMV [6]. For optimal results in LTSMIN, we ran the symbolic tool with the flags for saturation and variable reordering the following options:

```
-regroup=bcm,gs -order=chain-prev
 -saturation=sat-like -save-sat-levels
```

Figures 16 and 17 show the result of the comparisons. We observe that the compressed state sizes in BDDs are unrelated to tree compression sizes. The latter are always (slightly) larger than the minimum of four bytes per states, while BDDs can even compress better than that. This can be expected as a single BDD node; the "true" node represents all states. Other large subsets might also be represented efficiently (for example, when there is no correlation between variable values). Nonetheless, the converse is also true as the BDD can explode in size (which we see happening here for

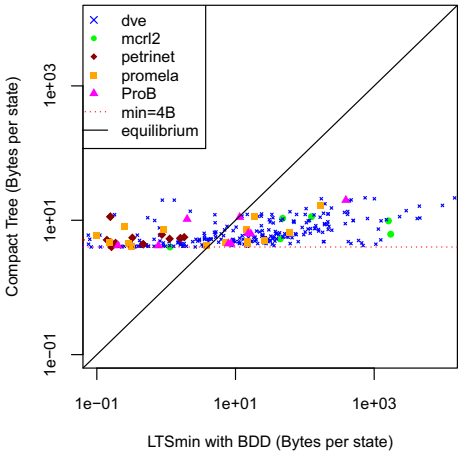

Fig. 16 Memory use per state of LTSMIN with Compact Tree and BDD [3]

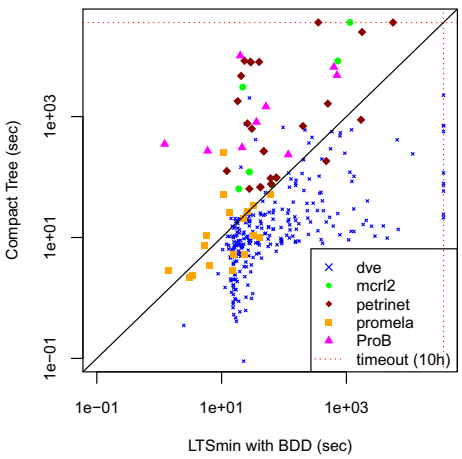

Fig. 17 Runtime (sequential) of LTSMIN with Compact Tree and BDD [3]

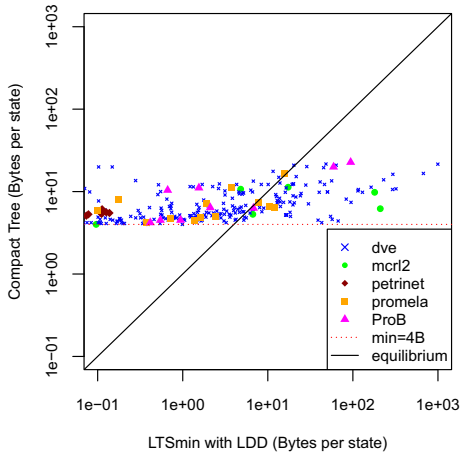

Fig. 18 Compressed sizes per state of LTSMIN with LDD [19]

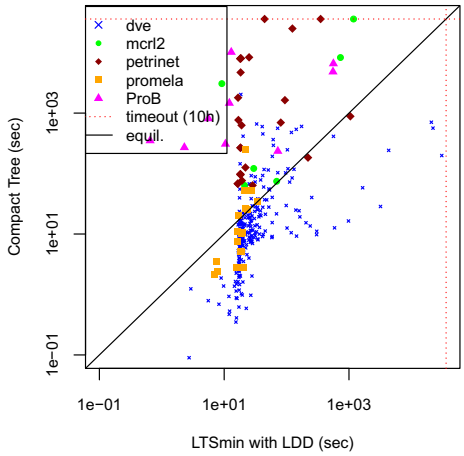

Fig. 19 Runtime (sequential) of LTSMIN with Compact Tree and LTSMIN with LDD

two Promela models and MCRL2's process algebraic models).

Runtime, however, does not follow the trend of the compression in BDDs. This is likely because the size of the intermediary BDDs is larger and/or because it may take long before the fixpoint is computed in BDDs. (The BDD operations used for image computation are polynomial in the size of the BDD, i.e., potentially exponentially faster than handling the states in BDD individually. However, many iterations might be needed before the fixpoint is found, so even when intermediary BDDs are small the reachability might take long.)

5.6 Comparison with MDDs

LTSMIN uses Sylvan as BDD/MDD implementation. Multi-valued decision diagrams are implemented in Sylvan as List Decision Diagrams (LDDs) [41]. Like MDDs, the edges in LDDs represent integer values rather than the Boolean values in BDD edges, thereby often reducing the size for state spaces of software systems. Additionally, LDDs represent the resulting n-ary tree as a binary tree, ala Knuth [25], to allow sharing between sublists at the same level. (Each level represents one totally ordered variable.)

From Fig. 18, we see that LDDs seem to provide an order of magnitude better compression ratios than BDDs. In fewer cases, the compact tree still beats the LDDs (compared to BDDs). The experiments show that runtimes of the tree and LDDs are harder to compare (see Fig. 19). Looking at the larger models (longer runtimes), we see thought that DVE problems seem better suited for tree compression, while the other input languages tend to verify faster with LDDs.

5.7 Decision diagram peak sizes

As mentioned in Sect. 5.5, the size of a decision diagram does not monotonically grow with the size of the set it stores. For

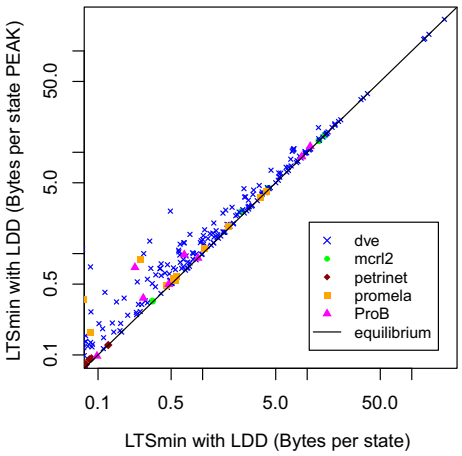

Fig. 20 Compressed state sizes in the final LDD compared to the peak intermediate LDD

this reason, the bottleneck for symbolic reachability using BDDs is the peak-sized decision diagram encountered during the entire procedure. To investigate the impact, we measured the peak decision diagram size for the MDD-based reachability and compared it against the size of the final BDD. For this experiment, we use a saturation search order as it is known to keep the size of the intermediate decision diagrams smallest [10,41]. Figure 20 shows that the peak sizes can be almost an order of magnitude larger than final sizes; however, for most inputs the impact is not that pronounced.

5.8 Comparison with parallel BDDs/MDDs

Figures 21 and 22 show the speedups of BDDs/LDDs. Good speedups are less common than with tree compression (cf. Fig. 11). We do observe however that BDDs scale better than MDDs. The difference in speedup results in a performance advantage for the tree approach, when parallel verification is used, as Figs. 23 and 24 show.

5.9 Case study: GARP

To push the envelop in enumerative model checking, we performed a case study using the GARP protocol as implemented by Konnov and Letichevsky [26]. Instantiated with one bridge and two applications, the implementation has 3.31e11 (331 billion) states according to the symbolic backend of LTSmin, which takes 3.2 h to explore the full state space. Using partial-order reduction [31], we could fully explore the model with the enumerative multi-core backend [32] for the first time. (While the model could already be explored symbolically with BDDs, this feat is still of interest as enumerative analysis methods can more efficiently verify certain temporal properties on the fly [34].)

To this end, we used another machine with 512 GB memory and 64 cores: four AMD Opteron^{TM} 6376 processors with 16 cores each. The model checker was configured to use all available memory by setting:

- $w = 36$
- $o = 7$
- $u = 30$

Table 3 shows the results of the case study. The first thing to note is that the OS-reported memory use is close to the space occupied by filled table buckets in the Compact Tree. This means that the implementation performs according to the design as predicted by the analysis in Sects. 3.2 and 4.2. (In fact, due to likely paging of all allocated table buckets, the OS-reported memory should be closer to the memory allocated by the Compact Tree. And the difference of 41 GB between tree-allocated and OS-reported memory is taken up by the five billion states we measured which were stored in the queues at peak.)

Furthermore, the compressed state size of 32.5 bits per state is close to the best case of $\approx$ 32 bits as predicted by Theorem 4 according to the above parameters. Interesting to note is that this amount of space used per state is smaller than the space required for a unique state identifier, which takes 35.6 bits given that 5.2e10 states are stored. This can be explained by the fact that the compact hash table uses the location of buckets as information as discussed in Sect. 4.1.

6 Discussion and conclusion

The tree compression method discussed here is a more general variant of recursive indexing [19], which only breaks down processes into separate tables. Hash compaction [37] compresses states to an integer-sized hash, but this lossy technique becomes redundant with the compact tree database. Bloom filters [18] still present a worthwhile lossy alternative using only a few bits per state, but of course abandon soundness when applied in model checking.

Valmari and Geldenhuys [17] present a data structure similar to Cleary's [12].

Evangelista et al. [16] report on a hash table storing incremental differences of successor states (similar to the incremental data structure discussed in Sect. 3). Partial vectors in the table contain a pointer to one predecessor, and only the initial vector is stored fully. Their partial vectors take $2u + \log(E)$ bits, where E is the set of (deterministic) actions in the model. To look up vectors in this database, a state is hashed to a table bucket. Defying our requirement of constant time for lookups, Evangelista et al. reconstruct full states by reconstructing all ancestors (in the worst case, there might be as many ancestors as reachable states). We

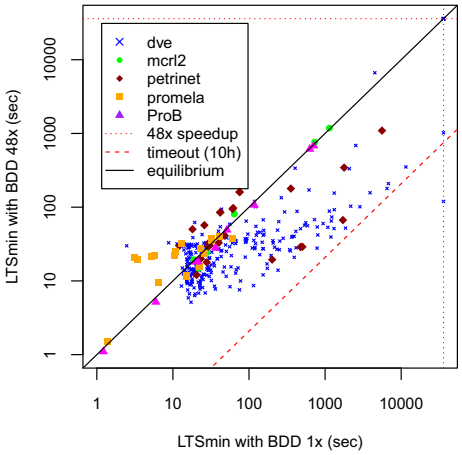

Fig. 21 Speedup using 48 cores of LTSMIN with BDD (cf. Fig. 11)

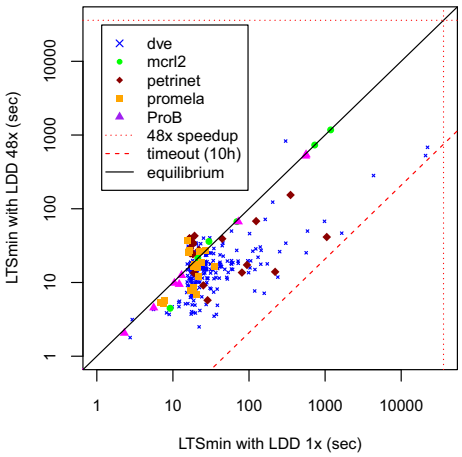

Fig. 22 Speedup using 48 cores of LTSMIN with LDD (cf. Fig. 11)

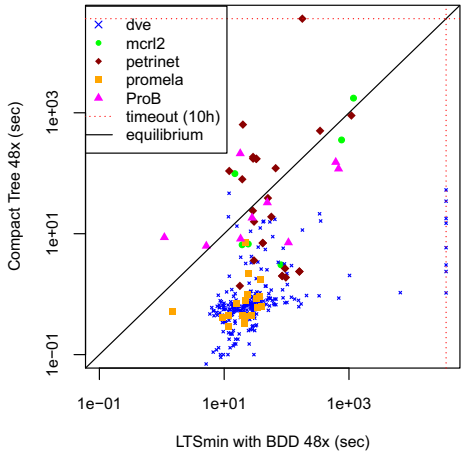

Fig. 23 Runtime (48 cores) of LTSMIN with Compact Tree versus LTSMIN with BDD

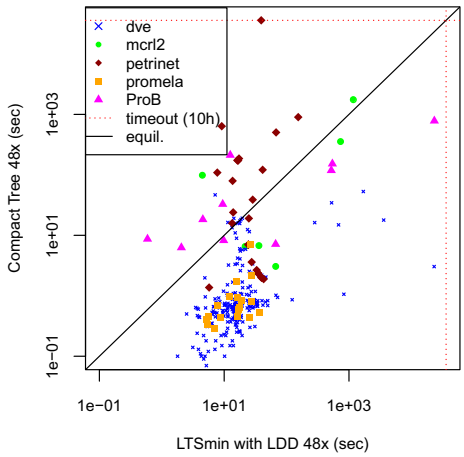

Fig. 24 Runtime (48 cores) of LTSMIN with Compact Tree versus LTSMIN with LDD

could not compare to this approach due to lack of an available implementation.

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, an analysis is needed that takes into account the entire space, much like the analysis presented in [17] In the case of model checking, we could also assume structural, global properties to describe the nonlinear compression of BDDs (e.g., the input's decomposition into processes, symmetries, etc). Valmari [39] presents a similar approach for storing Rubik's cube states.

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 $\{\mathbf{i, i, j, j} \mid i, j \in [1 \ldots N]\}$: 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 $\{\mathbf{i, j, i, j} \mid i, j \in [1 \ldots N]\}$. Preliminary research [14] revealed that the tree's optimum can be reached in most cases for DVE models, but we were unable to find a heuristic that consistently realizes this.

Table 3 Results for the GARP case study

Maximum memory used as reported by OS	321 GB	
Memory allocated by Compact Tree	260 GB	
Memory *occupied* in Compact Tree	198 GB	
Run time	2.4 h	
States explored (reduced to 16% by POR)	5.2e10	$\approx 2^{35.6}$
Compressed state size	4.06 B	$= 32.5$ bits

Note that this includes queues and other data structures in the model checker

Acknowledgements The author thanks Yakir Vizel for promptly pointing out the natural number as a limit, Tim van Ervsen for a fruitful discussion and Jaco van de Pol for the use of the multi-core cluster of his Formal Methods & Tools group at University of Twente.
This work is part of the research program VENI with Project Number 639.021.649, which is (partly) financed by the Netherlands Organization for Scientific Research (NWO).

References

1. Baranová Z, Barnat J, Kejstová K, Kučera T, LaukoJan H, Mrázek J, Ročkai P, Štill V (2017) Model checking of C and C++ with DIVINE 4. In: D'Souza D, Narayan Kumar K (eds) Automated technology for verification and analysis. Springer, Cham, pp 201–207
2. Bendisposto J, Körner P, Leuschel M, Meijer J, van de Pol J, Treharne H, Whitefield J (2016) Symbolic reachability analysis of b through prob and ltsmin. In: Ábrahám E, Huisman M (eds) Integrated formal methods. Springer, Cham, pp 275–291
3. Blom SCC, van de Pol J, Weber M (2010) LTSmin: distributed and symbolic reachability. In: CAV, volume 6174 of LNCS. Springer, pp 354–359
4. Blom S, Lisser B, van de Pol J, Weber M (2009) A database approach to distributed state space generation. J Logic Comput 21(1):45–62
5. Bollig B, Wegener I (1996) Improving the variable ordering of OBDDs is NP-complete. IEEE Trans Comput 45:993–1002
6. Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput 35(8):677–691
7. Burch JR, Clarke EM, Long DE (1991) Symbolic model checking with partitioned transition relations. North-Holland, pp 49–58
8. Burch JR, Clarke EM, McMillan KL, Dill DL (1991) Sequential circuit verification using symbolic model checking. In: Proceedings of the 27th ACM/IEEE design automation conference. ACM, New York, NY, USA, pp 46–51. https://doi.org/10.1145/123186.123223
9. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1990) Symbolic model checking: 10^{20} states and beyond. In: LICS, pp 428–439
10. Ciardo G, Lüttgen G, Siminiceanu R (2001) Saturation: an efficient iteration strategy for symbolic state-space generation. Margaria T, Yi W (eds) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, Heidelberg, pp 328–42
11. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV version 2: an opensource tool for symbolic model checking. In: Proceedings of international conference on computer-aided verification (CAV 2002), volume 2404 of LNCS, Copenhagen, Denmark, July 2002. Springer
12. Cleary JG (1984) Compact hash tables using bidirectional linear probing. IEEE Trans Comput C–33(9):828–834
13. Cranen S et al (2013) An overview of the mCRL2 toolset and its recent advances. Springer, Berlin, pp 199–213
14. de Vries SHS (2014) Optimizing state vector compression for program verification by reordering program variables. In: 21st Twente SConIT, 21(June 23)
15. Emerson EA, Wahl T (2005) Dynamic symmetry reduction. Springer, Berlin, pp 382–396
16. Evangelista S, Kristensen LM, Petrucci L (2013) Multi-threaded explicit state space exploration with state reconstruction. Springer, Berlin, pp 208–223
17. Geldenhuys J, Valmari A (2003) A nearly memory-optimal data structure for sets and mappings. In: Ball T, Rajamani SK (eds) Model checking software. Springer, Berlin, Heidelberg, pp 136–150
18. Holzmann GJ (1996) An analysis of bitstate hashing. Springer, Berlin, pp 301–314
19. Holzmann GJ (1997) State compression in SPIN: recursive indexing and compression training runs. In: Proceedings of the third international SPIN workshop
20. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23:279–295
21. Jensen PG, Larsen KG, Srba J (2017) PTrie: data structure for compressing and storing sets via prefix sharing. In: Hung DV, Kapur D (eds) Theoretical aspects of computing–ICTAC 2017. Springer, Cham, pp 248–265
22. Jensen PG et al (2014) Memory efficient data structures for explicit verification of timed systems. In: Badger JM, Rozier KY (eds) NASA formal methods. Springer, Cham, pp 307–312
23. Kant G, Laarman A, Meijer J, van de Pol J, Blom S, van Dijk T (2015) LTSmin: high-performance language-independent model checking. Tools and algorithms for the construction and analysis of systems. Springer, Berlin, Heidelberg, pp 692–707
24. Katz S, Peled D (1988) An efficient verification method for parallel and distributed programs. In: REX workshop, volume 354 of LNCS. Springer, pp 489–507
25. Knuth D (1968) The art of computer programming 1: fundamental algorithms 2: seminumerical algorithms 3: sorting and searching. Addison-Wesley, Boston, p 30
26. Konnov I, Letichevsky OA Jr (2010) Model checking GARP protocol using spin and VRS. In: Algorithms, and information technologies, international workshop on automata
27. Kordon F et al (2016) Complete results for the 2016 edition of the model checking contest. http://mcc.lip6.fr/2016/results.php. Accessed 14 May 2019
28. Körner P, Leuschel M, Meijer J (2018) State-of-the-art model checking for b and event-b using prob and LTSmin. In: Furia CA, Winter K (eds) Integrated formal methods. Springer, Berlin, pp 275–295
29. Laarman A, van de Pol J, Weber M (2011) Parallel recursive state compression for free. In: Groce A, Musuvathi M (eds) Model checking software. Springer, Berlin, Heidelberg, pp 38–56
30. Laarman A, van de Pol J, Weber M (2011) Multi-core LTSmin: marrying modularity and scalability. In: Bobaru M, Havelund K, Holzmann GJ, Joshi R (eds) NASA formal methods. Springer, Berlin, Heidelberg, pp 506–511
31. Laarman AW, Pater E, van de Pol JC, Hansen H (2014) Guard-based partial-order reduction. Int J Soft Tools Technol Transf 18(4):427–448. https://doi.org/10.1007/s10009-014-0363-9
32. Laarman A (2014) Scalable multi-core model checking. Ph.D. thesis, UTwente
33. Laarman A (2018) Optimal storage of combinatorial state spaces. In: NASA formal methods symposium. Springer, pp 261–279
34. Laarman A, Langerak R, van de Pol J, Weber M, Wijs A (2011) Multi-core nested depth-first search. In: Bultan T, Hsiung P-A (eds) Automated technology for verification and analysis. Springer, Berlin Heidelberg, pp 321–335

35. Miner AS, Ciardo G (1999) Efficient reachability set generation and storage using decision diagrams. In: International conference on application and theory of petri nets. Springer, pp 6–25
36. Pelánek R (2007) BEEM: benchmarks for explicit model checkers. In: Bošnački D, Edelkamp S (eds) Model checking software. Springer, Berlin, Heidelberg, pp 263–267
37. Stern U, Dill DL (1995) Improved probabilistic verification by hash compaction. In: Camurati PE, Eveking H (eds) Correct hardware design and verification methods. Springer, Berlin, Heidelberg, pp 206–224
38. Valmari A (1988) Error detection by reduced reachability graph generation. In: Proceedings of the 9th european workshop on application and theory of petri nets, pp 95–112
39. Valmari A (2006) What the small Rubik's cube taught me about data structures, information theory, and randomisation. STTT 8(3):180–194
40. van der Berg F, Laarman A (2013) SpinS: extending LTSmin with Promela through SpinJa. ENTCS 296:95–105
41. van Dijk T, van de Pol J (2017) Sylvan: multi-core framework for decision diagrams. Int J Softw Tools Technol Transf 19(6):675–696
42. van der Vegt S, Laarman AW (2012) A parallel compact hash table. In: Kotásek Z, Bouda J, Černá I, Sekanina L, Vojnar T, Antoš D (eds) Mathematical and engineering methods in computer science. Springer, Berlin, Heidelberg, pp 191–204
43. Wahl T, Donaldson A (2010) Replication and abstraction: symmetry in automated formal verification. Symmetry 2(2):799–847