Probabilistic Methods in State Space Analysis

Matthias Kuntz, Kai Lampka

Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Informatik
{mskuntz,kilampka}@informatik.uni-erlangen.de

Abstract. Several methods have been developed to validate the correctness and performance of hard- and software systems. One way to do this is to model the system and carry out a state space exploration in order to detect all possible states. In this paper, a survey of existing probabilistic state space exploration methods is given. The paper starts with a thorough review and analysis of bitstate hashing, as introduced by Holzmann. The main idea of this initial approach is the mapping of each state onto a specific bit within an array by employing a hash function. Thus a state is represented by a single bit, rather than by a full descriptor. Bitstate hashing is efficient concerning memory and runtime, but it is hampered by the non deterministic omission of states. The resulting positive probability of producing wrong results is due to the fact that the mapping of full state descriptors onto much smaller representatives is not injective. – The rest of the paper is devoted to the presentation, analysis, and comparison of improvements of bitstate hashing, which were introduced in order to lower the probability of producing a wrong result, but maintaining the memory and runtime efficiency. These improvements can be mainly grouped into two categories: The approaches of the first group, the so called multiple hashing schemes, employ multiple hash functions on either a single or on multiple arrays. The approaches of the remaining category follow the idea of hash compaction. I.e. the diverse schemes of this category store a hash value for each detected state, rather than associating a single or multiple bit positions with it, leading to persuasive reductions of the probability of error if compared to the original bitstate hashing scheme.

1 Introduction

It is commonplace that distributed, concurrent hard- and software systems have become part of our daily life. Because of our high dependency on these systems, it becomes more and more important to assert that they are working correctly and that they meet high performance requirements. Several methods have been developed to validate the correctness and performance of hard- and software systems. One way to do this is to model the system and generate the corresponding state space.

From some high-level description formalism like (stochastic) Petri nets, (stochastic) process algebras, etc., low level representations are derived. The (low-level) model consists of all possible states a system can be in and the set of all possible transitions between these states. The concurrency of the modelled system that can be expressed by high-level constructs must be made explicit when deriving such a state-transition model. That means that the number of states grows exponentially in the number of parallel components of the system model. Thus, a severe problem of this approach is the huge state space of concurrent system models. This problem is commonly known as *state space explosion problem*. Thus the size and complexity of systems that can be analysed is restricted, since memory space is limited.

There exist many methods to approach this problem, i.e. to accommodate very large state spaces. The existing methods can roughly be divided into two classes: Approaches that perform exhaustive state space search and methods that perform only partial state space exploration. The class of approaches that perform exhaustive state space search, can be divided into approaches that utilise mass storage and/or distributed hardware e.g. [16, 8]. and those that represent the state space in an efficient and compact way e.g. [9, 24, 1]. The methods in the second class are the so-called partial analysis methods. There exist many ways to organise a partial analysis. The two most prominent representatives of this class are partial order and probabilistic methods. Partial order techniques exploit the fact that the major contribution to the state space explosion stems from the huge number of possible interleavings of execution traces of concurrent processes. It can be observed that not all interleavings are relevant for system analysis. Generally spoken, partial order methods aim to prune away such interleavings. This can be achieved for example, by defining an equivalence relation on system behaviour [3].

It should be noted that in contrast to probabilistic methods partial order methods still guarantee full coverage of the state space to be analysed.

Using probabilistic methods, it is possible that only a fraction of all reachable states of a system being modelled is analysed, thus the probability that states are omitted is greater than zero. This paper concentrates on probabilistic methods. Probabilistic methods have been proven to be very successful in formal verification of large systems like communication protocols or cache coherency protocols. There exist several tools that employ probabilistic state space exploration. The two most prominent tools are SPIN, a model checker for functional verification [13] and DNAmaca a performance analysis tool [18]. The applicability of probabilistic methods was tested on several examples. Among the tested protocols are the IEEE/ANSI standard protocol for the scalable coherent interface (SCI) protocol, cache coherence protocols for the Stanford DASH Multiprocessor [25], the Data Transfer Protocol (DTP) and the Promela File Transfer Protocol (PFTP) [12, 29, 14]. Probabilistic methods were also applied to model flexible manufacturing systems [18].

The paper is organised as follows: In section 2 we will briefly review hashing. In section 3 state space generation in general will be covered. Section 4 introduces and analyses Holzmann's bitstate hashing, the basis of all probabilistic analysis methods. The following sections present newer variants and improvements of bitstate hashing.

2 Review of Hashing

In the sequel some principles of hashing will be reviewed. The presentation is based on [22]. (An introduction to hashing can also be found in [20]) We briefly discuss the selection of "good" hashing functions, then we introduce and analyse some hashing methods that are employed by the verification methods that are presented in this paper.

2.1 Basic terminology

Hashing is a scatter storage technique, where records that are to be stored are distributed over a prescribed address range. Hashing aims to retrieve a stored data record by computing the position where the record might be found. For this purpose a hashing function is applied. The parameter k of the hash function is called "key". The set of all keys is denoted $\mathcal{K}$, a key is a natural number such that $\mathcal{K}$ is a subset of the natural numbers. A hash table is an array with indices $0 \dots m-1$, where m is the size of the hash table. A hash function h is defined as follows:

$$h : \mathcal{K} \mapsto \{0, \dots, m-1\}$$

The value $h(k)$ is called hash address or hash value. Usually, a hash function is not injective. If two keys k and k', map to the same hash value, i.e. $h(k) = h(k')$, a so-called (address) collision occurs. Such keys are also called "synonyms".

2.2 Selection of "good" hash functions

Requirements: There are several requirements that a "good" hash function has to fulfill:

- Efficiency: It must be possible to compute the hash function easily and quickly, because it has to be computed millions of times.
- Distribution: To avoid a large number of address collisions the hash values should distribute evenly over the hash table.

Division-with-rest method: The simplest method to generate a hash address $h(k)$ for given k in a range between 0 and $m-1$ is to take the remainder of k on integer division by m:

$$h(k) = k \bmod m$$

The choice of m is decisive for the quality of the hash function. If m is even (odd), then $h(k)$ is even (odd), if k is. If m is a power of the number system in which the keys are represented, i.e. $m = r^i$, if r is the basis of the number system, then $h(k)$ has as outcome only the i least significant digits of k. A "good" choice for m is a prime, such that none of the numbers $r^i \pm j$, where r is the basis of the keys' number system, i and j small positive integers, is divisible by m [20, 22].

Multiplicative method: The given key is multiplied by an irrational number φ. The integer part of the result is cut off. So, for different keys, different values between 0 and 1 are obtained. Ideally, for keys $1, 2, 3, \ldots n$ these values are equally distributed on the interval $[0, 1)$.

$$h(k) = \lfloor m \cdot (k \cdot \varphi - \lfloor k \cdot \varphi \rfloor) \rfloor$$

Universal hashing: For hashing to produce a "good" result it is important that the keys are equally and uniformly distributed over the set of all possible keys. This assumption is somewhat problematic, as keys are often not uniformly distributed.
So, instead of making critical assumptions about the distribution of the keys, assumptions about the hash method are made, that are less problematic. Using universal hashing, one hash function is picked in a uniform and random manner out of a set $\mathcal{H}$ of hash functions. The randomisation in the selection of the hash functions shall guarantee that an unevenly distributed set of keys does not lead to many hash collisions. Note that a single function h of $\mathcal{H}$ might still lead to many collisions but the average over all functions of $\mathcal{H}$ diminishes this.
If $\mathcal{H}$ is a collection of hash functions, with $h: \mathcal{K} \mapsto \{0, \ldots, m-1\}$, for all $h \in \mathcal{H}$ then $\mathcal{H}$ is called "universal" if for any two keys k, k', $k \neq k'$

$$\frac{|\{h \in \mathcal{H} \mid h(k) = h(k')\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

holds, i.e. if at most the m-th part of the hash functions leads to a hash collision for the keys.

2.3 Collision resolution by chaining

This section introduces chaining and open addressing as collision resolution methods.

Assume that a hash table already contains a key k, at which point a further key k' is to be inserted, and that $h(k) = h(k')$, i.e. a collision occurs. The key k' has to be stored somewhere else, not on position $h(k')$. Chaining stores keys that collide with keys that are already in the hash table outside the table. For that purpose a dynamic data structure, a linked list, is associated with each slot of the hash table that contains the keys k' that collide with k.
There are two ways to realise hashing with chaining:

Separate chaining of collided keys: Each element of the hash table is the first element of a linked list containing the collided elements.

Direct chaining of collided keys: Each element of the hash table is a pointer to a linked collided list. That means that the real collided elements start with the second entry of the linked list. For a more detailed description of chaining cf. for example [22, 20].

2.4 Collision resolution with open addressing:

In contrast to chaining, open addressing stores all keys inside the hash table. If upon insertion of key k' another key k is already stored at $h(k')$, then according to a fixed rule another slot has to be searched on which k' may be accommodated. For that purpose for each key a fixed order in which slots are inspected is defined. As soon as an empty slot is found, the key is stored at this address. The sequence of slots that are inspected during the search is called "probe sequence". Such methods are called "open addressing schemes". There are many variants of open addressing of which only those are briefly introduced that are relevant for this paper. For a more thorough overview we refer to [22] and [20]. and the literature referred to there.

Clustering: Clustering is a phenomenon that may occur with open addressing. Two types of clustering can be distinguished:

- Primary clustering: If adjacent slots of the hash table are filled then primary clustering leads to the effect that these areas tend to become larger.
- Secondary clustering: Two synonyms k and k' are going through the same probe sequence, i.e. interfere with each other on insertion.

Open addressing schemes: Assuming a hash table with m slots, open addressing applies a probe sequence that is a permutation of all m possible hash values.
The permutation depends only on the actual key.

- **Uniform probing:** Under the assumption of uniform probing, each of the $m!$ possible permutations should be equally likely. Practical realisation of uniform probing is very difficult, such that alternatively a scheme called "random probing" is applied, which is almost as efficient as uniform probing [22].
- **Double Hashing:**: Here, address collisions are resolved by applying a second hash function h', which ideally should be independent of the primary hash function h. h' has to be chosen in such a way that the probing sequence forms a permutation of the hash addresses, i.e. it is required that $h'(k) \neq 0$ and $h'(k) \not| m$, if m is prime this can be granted. The function h' is used to compute the probe sequence, if $h(k)$ produces a collision. The chosen probe sequence is given as follows:

$$h(k) \bmod m, (h(k) + h'(k)) \bmod m, \ldots, (h(k) + i \cdot h'(k)) \bmod m, \ldots,$$
$$(h(k) + (m-1) \cdot h'(k)) \bmod m$$

- **Ordered Hashing:** Ordered Hashing is a method that aims to speed up during unsuccessful searches. To reach that goal, ordered hashing simulates ordered insertion, i.e. the hash keys are inserted in an ordered fashion. This method is explained by means of an example:

Assume that items 12 and 19 have been inserted into a hash table. Let the elements 12 and 19 be synonyms that have been inserted in that order. Now, assume that key 5 is searched for, which is synonymous to 12 and 19. The search for 5 starts at address $h(5)$, which is equal to $h(12)$, at this position 12 can be found. Without ordered hashing we would have to continue probing until we can decide that 5 is not in the hash table. Using ordered hashing, we could stop probing after the first probe, which yielded item 12 at position $h(5)$, as $12 > 5$.

Analysis of uniform probing: We now turn to the analysis of the efficiency of uniform probing, which is slightly better than random probing. It is assumed that the hash table has m slots, of which $m - n$ are free. Under the assumption of uniform probing, each of the $\binom{m}{n}$ possible assignments of m slots to n hash values is equally probable. The probability, that exactly i slots have to be probed when inserting a new element is computed as follows [22]:

$$p_i = \frac{\binom{m-i}{n-i+1}}{\binom{m}{n}} \tag{1}$$

Equation 1 stems from the fact, that apart from the $i-1$ fixed keys also $n-(i-1)$ keys that are still to be inserted have to be distributed on the $m-i$ free slots. The average number of probed slots on insertion, C'_n, is then computed as follows:

$$\begin{aligned}
C'_n &= \sum_{i=1}^{m} i \cdot p_i \\
&\overset{\sum_{i=1}^{m} p_i = 1}{=} m + 1 - \sum_{i=1}^{m} (m+1) \cdot p_i - \sum_{i=1}^{m} (-i \cdot p_i) \\
&= m + 1 - \sum_{i=1}^{m} (m+1-i) \cdot p_i \\
&\overset{\binom{a}{b} = \binom{a}{a-b}}{=} m + 1 - \sum_{i=1}^{m} (m+1-i) \cdot \binom{m-i}{m-n-1} \Big/ \binom{m}{n} \\
&\overset{\binom{a}{b} = \frac{b+1}{a+1} \cdot \binom{a+1}{b+1}}{=} m + 1 - \sum_{i=1}^{m} (m+1-i) \cdot \frac{m-n}{m-i+1} \binom{m-i+1}{m-n} \Big/ \binom{m}{n} \\
&= m + 1 - (m-n) \cdot \sum_{i=1}^{m} \binom{m-i+1}{m-n} \Big/ \binom{m}{n} \\
&\overset{\sum_{c=1}^{a} \binom{c}{b} = \binom{a+1}{b+1}}{=} m + 1 - (m-n) \cdot \binom{m+1}{m-n+1} \Big/ \binom{m}{n} \\
&= m + 1 - (m-n) \cdot \frac{m+1}{m-n+1}
\end{aligned}$$

$$= \frac{m+1}{m-n+1} \quad (2)$$

The average number of probed slots at successful searches using open addressing, C_n is given below:

$$\begin{aligned} C_n &= \frac{1}{n}\sum_{i=0}^{n-1} C'_i \\ &= \frac{1}{n}\sum_{i=0}^{n-1} \frac{m+1}{m-i+1} \\ &= \frac{m+1}{n} \cdot \left(\frac{1}{m+1} + \frac{1}{m} + \ldots \frac{1}{m-n+2}\right) \\ &\overset{H_n=\sum_{i=1}^{n}\frac{1}{i}}{=} \frac{m+1}{n} \cdot (H_{m+1} - H_{m-n+1}) \end{aligned}$$

This equation can be justified as follows:
Searching a key k follows the same probe sequence as it was applied, when k was inserted. If k is the $(i+1)$st element that was inserted into the hash table, then the expected number of probes, searching for k equals $\frac{m+1}{m-i+1}$. The average of all n keys in the hash table yields the factor C_n.

3 State Space Exploration

As already mentioned in the introduction, the first step in functional or temporal performance analysis is the generation of the state graph (low-level model description). Therefore a short introduction to the state space (SSp) generation will follow now. The section will be rounded by motivating and introducing coarsely probabilistic SSp exploration and clarifying the basic terminology.

3.1 Exhaustive, Explicit state space generation

The technique of generating all reachable states of a modeled system is called *exhaustive* state space exploration. If every state descriptor is explicitly represented by a record one speaks of *explicit* SSp representation. The data structure required for an exhaustive explicit state space exploration scheme is a buffer (state buffer) and a large somehow structured storage space (state table). The former holds the detected but yet not explored states, where the latter is used for storing the already visited states. Usually a state is considered to be explored once all its successor states have been determined. Depending on how the state buffer is accessed a depth-first search (dfs) or a breadth-first search (bfs) on the SSp is implemented. I.e. in case of a dfs-algorithm the buffer is accessed in a "last-in-first-out" (lifo) fashion, it thus becomes a stack. In case of a bfs-algorithm the buffer is accessed in a "first-in-first-out" (fifo) manner, it becomes a queue. The size of the state buffer is bounded by the depth, breadth respectively of the explored state graph and the number of bits required for storing the state descriptors. Since the buffer is accessed in a structured manner, the currently not used parts can be swapped onto secondary storage device (e.g. HDD). Consequently the state buffer is not the bottleneck of an exhaustive and explicit state space exploration as far as access time and memory consumption are concerned.

The state table holds the already visited states, thus it serves as a database, whose single purpose is an effective look-up of a given state. Common layouts for the state table as used under exhaustive and explicit SSp-exploration differ concerning allocation strategy (dynamic or static) and structure. Their commonness is usually the use of a hashing scheme, where the state descriptors serve as keys and where collision resolution (sec. 2.3) needs to be applied. One may note, that even though nowadays hard disk drives (HDDs) are fast and employ large buffers and block-wise access patterns, the state table needs to be kept in total in the memory. This is due to the fact, that the accesses to the state table are not structured and therefore buffer strategies may not be applied (most recently read...). Concerning the large HDD-buffers one may consider, that they can be viewed as additional memory space. This extensional RAM will only put back the boundaries, but will be easily absorbed by SSps of one or two magnitudes larger, which are not a curiosity when it comes to the modeling of large and highly complex systems. As a consequence and due to the limitation of memory the number of states to be explored and thus the size and complexity of models to be analyzed is restricted.

Figure 1 shows how an exhaustive exploration of a state space proceeds in principle:
The state buffer is here organized as stack *Stack*, thus the algorithm follows a dfs-scheme. The state table is organized as a hash table (H). Whether the algorithm follows a *direct* - or *separate* chaining, or a *probing* approach for handling collided state descriptors is hereby transparent, since a state descriptor s is inserted into the hash table by calling the function $insert(s, H)$.

It is now assumed that *direct chaining* is used for collision resolution and that the number of reachable states (n) is greater than the number of slots m in the hash table. Thus the average of n/m states will obtain the same hash value. If the table is full an average of n/m comparisons is necessary to decide if a state is new. With increasing number of comparisons the time efficiency of the search decreases. If an address pointer requires p bytes, then the hash table consumes $m \cdot p$ bytes of memory. Under chaining each entry of the hash table needs $mem(s)$ bytes and an additional p bytes for its referencing. To accommodate the largest possible fraction of the state space m should be small to keep the memory consumption as small as possible. On the other hand, m should be large to minimise the number of hash conflicts to keep the time efficiency of the state space exploration high.

```
Stack := ∅, H := ∅, s0 := initial state
push(s0, Stack)
insert(s, H)
DFS()

DFS() {
    while (Stack ≠ ∅)
      s = pop(Stack)
      forall successors s' of s do
        if s' ∉ H
          push(s', Stack)
          insert(s', H)
        endif
      od
    endwhile
}
```

Fig. 1. Algorithm for exhaustive state space search

The memory requirement for administering n states O, if chaining for collision resolution is employed, is given by:

$$O = m \cdot p + (mem(s) + p) \cdot n \tag{3}$$

However in order to minimise the memory requirements under *chaining* as collision resolution, one should use arrays for storing the overflows rather then lists. As a consequence the hash table must contain the starting address as well as an element counter of each dynamically allocated array. As discussed below, one should note, that the space complexity of the administrative structure is not the issue, when it comes to large state descriptors and / or huge SSps to be explored.

3.2 Introduction to probabilistic state space exploration

For exemplification think now of a bounded Petri net (PN) consisting of 1000 places. Let the maximum number of tokens per place be less than 256, yielding a memory consumption of 1 KByte per state descriptor under a non-efficient straight forward encoding scheme. Thus a SSp of $n = 10^6$ states requires 1000 MByte for storing the 10^6 distinct state descriptors only (eq. 3: $mem(s) =$ 1KByte). Furthermore one would need to spend additional memory space for the administration structure required for resolving collisions. I. e. by employing direct chaining and a non-optimal 64-bit addressing scheme and a maximal number of slots ($m = 10^6$) another 16 MByte of memory is required (eq. 3). Even under such a worst-case scenario concerning memory space, the memory requirement of the administrative structure for collision resolution is still less than 2% of the overall required memory space. Thus the space complexity of the administration structure is not the bottleneck of exhaustive and explicit state space exploration. –Alternatively one may think now of swapping parts of the SSp on to secondary storage devices. But the occurrence of states is mostly not structured, consequently an expensive look-up for each swapped state must be taken into account. Assuming a memory size of ≈ 500 MByte RAM, in the above illustrated scenario one would need to swap $5 \cdot 10^5$ states onto HDD. As a consequence one has to spend $\approx 3 hours$ only to reload $5 \cdot 10^5$ states twice into memory, assuming a HDD access time of 10 ms per read operation. Thus the employment of secondary storage devices is for its time complexity not applicable and the number of states to be explored bounded by the size of the available RAM (cf. section 3.1). For coping with the problem of limited memory on one hand and the state space explosion problem on the other hand many approaches have been developed, targeting a decrease in memory consumption of large SSps and keep the runtime at moderate levels.

A number of approaches, which have been applied successfully (mainly in the tools SPIN [11] and DNAmaca [15]), are based on the idea of not storing the full state descriptors of visited states, but storing a smaller representative for each instead. However since the mapping of the keys (state descriptors) onto the representatives (hash values) is not injective, different state descriptors may be mapped on the same representative.[1] Therefore the exploration algorithm may consider states as identical, which are in fact not. Consequently the later reached states will be misleadingly dropped and not explored, denoted as *state omission*. Furthermore their successor states might not be explored, if they are only reachable through omitted states. Thus the state table may not contain all reachable states. Consequently the modeler takes into account, that the evaluated functional properties and / or performance results are wrong.[2] I.e. the analysis can produce a positive result (e.g. deadlock free), which is in fact wrong, denoted as the production of *false positives*. Since the omission of states and therefore the production of wrong results is not predictable, nor one does know which portion of the SSp will be visited, such methods having this disadvantage are referred to as *probabilistic state space exploration* methods.
Since the probabilistic exploration techniques achieve tremendous savings in memory consumption and thus runtime and since they do neither depend on a specific high-level model description method, nor on structural properties of a model, they might produce results, where otherwise an exhaustive, explicit SSp exploration and thus an analysis of a system will fail. Consequently the flaw of producing false positives, might be accepted, where otherwise no analysis could be carried out. Therefore the consumption of memory at a fixed level of error, as well as their time complexity seems to be the most adequate benchmarks

[1] One should note, that the state descriptors may not be uniformingly distributed, thus it might be difficult determining a posteriori an adequate hash function, which will produce uniformingly distributed hash values.

[2] In case of a model's functional behaviour, this means for example, that a model might be considered of being deadlock free, which it is in fact not, due to the coincidental omission of the deadlock states during SSp exploration.

for comparing the different probabilistic approaches, where in the sequel of this survey paper the former will be emphasized.
The chance of error, which is induced by the omission of states, is denoted as probability of omission of states (p_{om}). It describes the probability that states might be skipped during SSp generation. Thus one speaks of state space coverage, if referring to the probability of visiting all reachable states, which is given by $p_{cov} = 1 - p_{om}$. The omission of a specific state depends hereby on the probability of its own collision as well as on the probability that its predecessors were omitted. For simplicity one often presumes, that each state is reachable on different paths (well connected components), thus the condition of a predecessors exploration can be dropped. As a consequence the SSp coverage is equal to the probability that no state collided ($p_{cov} = p_{no}$). But one should be aware, that this consideration does often not coincide with the reality, especially in the field of reliability analysis, since faulty behaviours leading to disastrous outcomes often only occur under very specific conditions, i.e. they are only reachable via few specific paths in the SSp. A placement of the probabilistic approaches within the different methods of SSp exploration is given in figure 2, where each method presented in this paper is briefly characterized.

4 Bitstate Hashing

In this section the principles of bitstate hashing as it was introduced by Holzmann in [10–12] shall be presented in greater detail. Bitstate hashing can be considered the father of several probabilistic state space exploration methods that will be presented in this paper.

4.1 Functionality of bitstate hashing

From section 3 we know that the standard method to maintain a state space is hashing. This traditional approach has several drawbacks:

- Memory requirements for explicitly storing a single state are very high.
- Explicit storage of states reduces the number of explorable states

The idea of bitstate hashing is as follows: Under the assumption that the number of slots in the hash table is high in comparison to the number of reachable states, then it is possible to identify a state by its position in the hash table. If this is possible, then it is not necessary to store a full state descriptor in the hash table, to indicate, whether a state was already found or not, now a single bit suffices. To decide if a state s is already present we only have to inspect the value of the bit at the index $h(s)$ to which s is mapped. It is assumed that bit value “1” indicates that state s was already found and bit value “0” means that s was not found yet.
At the beginning of the state space search all bits in all slots are set to zero. If M bytes of memory are available, we can use $8 \cdot M$ bits to maintain the state space.

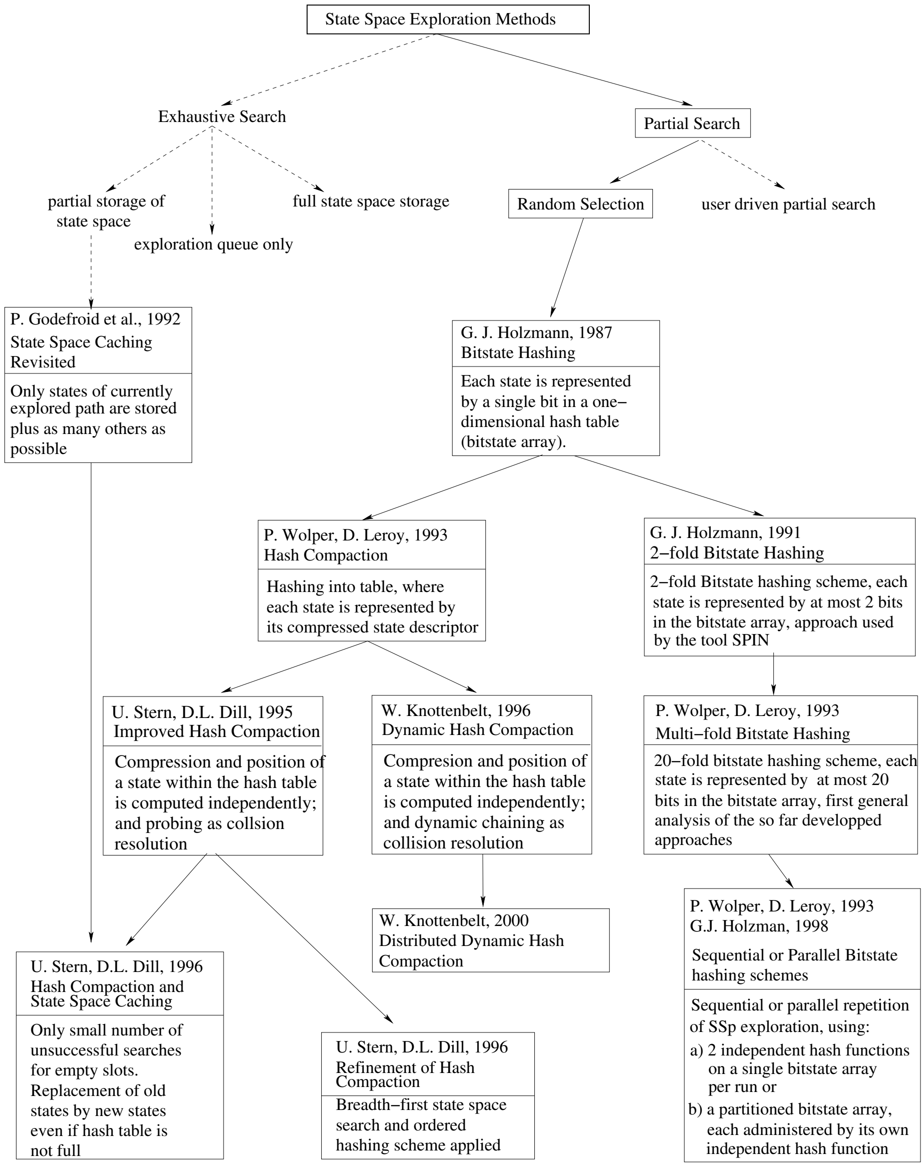


Fig. 2. Classification of probabilistic state space exploration techniques

That means, the hash table consists of a one-dimensional bit array (bitstate array), whose indices range from $0 \ldots 8 \cdot M - 1$, i.e. $m = 8 \cdot M$. From that follows, that up to m states can be analysed. The idea of bitstate hashing is illustrated in figure 3. From a given representation of the system to be analysed, for each of its states a state descriptor s is derived. For s a hash value is computed that lies between 0 and $m-1$. The hash value corresponds to the address of the hash table, where the value of the bit flag indicates whether s is already present or not.

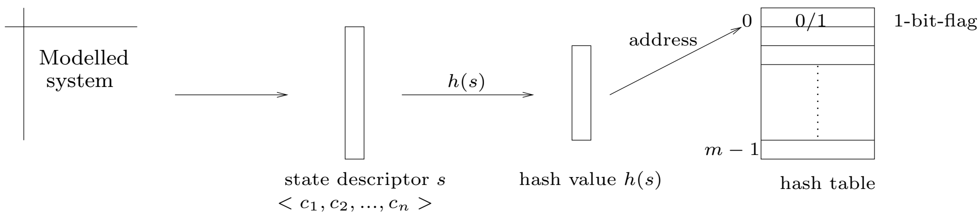


Fig. 3. Bitstate hashing

Ideally, $h(s)$ should be defined in such a way that the states are distributed uniformly over the m slots.
In figure 4 the algorithm for bitstate hashing, is given; it is assumed that the basic state space traversal is done by a depth first search:

Bitstate hashing can only be recommended if the size of the state space, and the memory requirements to represent all these states explicitly render an exhaustive search infeasible.
As the states are not stored explicitly, hash conflicts cannot be resolved. That means, if s and s' are mapped onto the same position in the hash table, s' and its successors will be ignored, because it is assumed that s' has already been analysed.
As $n/m \to 0$ the expected number of hash conflicts approaches zero. From that follows, m should be as large as possible.

Number of analysable states: For exhaustive state space search, the maximum number of analysable states given certain memory constraints, is $M/mem(s)$, given $mem(s)$ bytes are needed for storing a single state and M is the available memory space in bytes.
In contrast, using bitstate hashing this number can be incremented up to $8 \cdot M$. The hash conflicts scatter the set of analysed states over the whole set of reachable states. When comparing the coverage of traditional exhaustive search with that of bitstate hashing, we can distinguish two cases:

(1) $Stack := \emptyset$, $i := 0$ /*Lines (1) to (5) initialisation of hash table
(2) **while** $i < m$ **do**
(3) $\quad H[i] := 0$
(4) $\quad i := i + 1$
(5) **od**
(6) $\quad j := h(s_0)$
(7) $\quad H[j] := 1$
(8) $\quad$ push(s_0, $Stack$)
(9) $\quad$ DFS();

(10) DFS() {
(11) **while** ($Stack \neq \emptyset$)
(12) $\quad s = \text{pop}(Stack)$
(13) $\quad$ **forall** successors s' of s **do**
(14) $\qquad k := h(s')$
(15) $\qquad$ **if** $H[k] = 0$
(16) $\qquad\quad H[k] := 1$
(17) $\qquad\quad$ push(s', $Stack$)
(18) $\qquad$ **endif**
(19) $\quad$ **od**
(20) **endwhile**
(21) }

Fig. 4. Algorithm for bitstate hashing

1. $n \leq M/mem(s)$: In this case the traditional approach performs better than bitstate hashing, as the latter cannot guarantee a coverage of 100%.
2. $n > M/mem(s)$: In this case, an exhaustive search deteriorates into a partial search, for which the coverage might be lower than for bitstate hashing.

Finally, we compare the memory requirements of bitstate hashing with traditional hashing. For traditional hashing, the memory consumption was assessed in eq. 3 Now, O reduces to

$$O = \frac{m}{8} \tag{4}$$

bytes.

5 Analysis and Criticism of Bitstate Hashing

In this section we will derive some efficiency measures for bitstate hashing, as it was done in [14]. These measures are the collision probability, the loss probability and the coverage ratio.

5.1 Analysis

Let n be the number of reachable states, $mem(s)$ the number of bytes necessary to store a single state. Finally, let M be the size of memory in bytes that is

available for state space exploration. That means, that the size of the hash table m is equal to $8 \cdot M$.
For the analysis of the bitstate hashing method we compute the average collision probability if n states are inserted into the hash table. It should be noted that each collision in bitstate hashing leads to the omission of at least one state. The state that collides and all its successor states are omitted. The omission of the successor states is not too severe as *it is assumed that the transition graph is well connected*, i.e. several paths lead to each state such that it is likely that the state omitted due to a collision will be explored anyway.
The hashing function is assumed to distribute states uniformly over the m slots of the hash table.
For the analysis we distinguish two cases:

1. $n \leq m$: The collision probability for the first state to be inserted is clearly zero, The last state to be inserted has collision probability at most $\frac{n-1}{m}$, if no collision has occurred during the insertion of the other $n-1$ states. On average, this consideration yields the following estimation for the collision or equivalently omission probability $\tilde{p}_{om}$:

$$\tilde{p}_{om} \leq \frac{1}{n} \sum_{i=0}^{n-1} \frac{i}{m} \stackrel{*}{=} \frac{1}{n} \cdot \frac{n \cdot (n-1)}{2 \cdot m}$$

$$= \frac{n-1}{2m} \tag{5}$$

where '$*$' in equation 5 means:

$$\sum_{j=0}^{k} j = \frac{k \cdot (k+1)}{2}$$

Note carefully that the approximation in eq. 5 says only that the omission probability for state i is on average $\tilde{p}_{om}$. The coverage ratio using the approximation of eq. 5 is $1 - \tilde{p}_{om}$. (Since it is assumed, that each collision leads to the omission of only one state, we can identify coverage ratio and probability of no omissions.) This estimation is problematic because of the fact that $\tilde{p}_{om}$ only approximates an average omission probability for a particular state. A more accurate approximation of the probability of no ommissions will therefore be

$$(1 - \tilde{p}_{om})^n, \tag{6}$$

where the probability of no omissions is now taken over all n reachable states. In [14] it seems that, when comparing the coverage ratio of bitstate hashing with that of a traditional exhaustive search the approximation of eq. 6 was applied, even if never stated.

2. $n > m$: There are more reachable states than slots in the hash table. That means $n - m > 0$ is the minimal number of collisions during insertion. From that follows that the probability that collisions occur at all is certainly one. For a particular state the collision probability is normally less than one. This probability can be estimated as follows:

$$\tilde{p}_{om} = 1 - \frac{m}{n} \tag{7}$$

$\frac{m}{n}$: Fraction of states that are mapped onto different slots.

Finally, we estimate the omission probability using an exhaustive state space exploration method. The omission probability is greater than zero if the number of reachable states is larger than the memory available for state space search:

$$\tilde{p}_{om} = 1 - \frac{M}{n \cdot mem(s)} \tag{8}$$

Recall, M is the available memory in bytes. For each state $mem(s)$ bytes are required, such that the overall memory requirements for storing all n states are equal to $n \cdot mem(s)$. The percentage of states, that can be stored without omissions is thus $\frac{M}{n \cdot mem(s)}$.

5.2 Approximating the probability of state omissions

In contrast to Holzmann [14], Leroy and Wolper [29] argued, that the average probability of a state collision is not an appropriate measure for estimating the SSp coverage rate. Their estimation for the SSp coverage is based on the idea, that the SSp is completely explored, if no collision of states occurred. –In the following we consider n states to be explored and a bitstate array consisting of $m > n$ bit positions.

The first state has clearly the chance of $(1 - 0/m)$ of being mapped onto an occupied position, since no bit is marked as occupied at this stage. Assuming that i states have been entered collision-free so far, the $(i + 1)$'st state has the chance of $1 - (i/m)$ of being mapped onto an occupied bit position, since states are entered in the bitstate array at this stage. Analogous the n'th state to be entered has the chance of $1 - ((n - 1)/m)$ of being omitted. Since the SSp exploration is exhaustive only, if each position has got at most one state after all n states are explored (= no collision has occurred), the probability of no collision p_{no} is given by the conjunction of the probabilities of each individual state being inserted collision-free into the bitstate array. Since well-connected SSps are assumed, the individual probabilities are independent, yielding:

$$p_{no} = \left(1 - \frac{0}{m}\right) \times \cdots \times \left(1 - \frac{i-1}{m}\right) \times \cdots \times \left(1 - \frac{n-1}{m}\right) = \prod_{i=0}^{n-1} 1 - \frac{i}{m} \tag{9}$$

Since in nowadays hardware configurations one can allocate easily bitstate arrays of some billion bits[3], it can be assumed that the number of available bit positions m is not only very large, but exceeds the number of states n by far, i.e. $m \gg n$. Thus $\frac{(n-1)}{m} \ll 1$ and $1 - \frac{(n-1)}{m}$ can be approximated by $e^{-\frac{(n-1)}{m}}$. This approximation and $\sum_{i=0}^{n-1} \frac{i}{m} = \frac{n \cdot (n-1)}{2m} \approx \frac{n^2}{2m}$ for $n \gg 1$ can be employed in eq. 9, yielding the following approximation:

$$p_{no} \approx \prod_{i=0}^{n-1} e^{-\frac{i}{m}} = e^{-\sum_{i=0}^{n-1} \frac{i}{m}} = e^{-\frac{n \cdot (n-1)}{2m}} \approx e^{-\frac{n^2}{2m}} \quad , \tag{10}$$

which can be interpreted as the SSp coverage p_{cov}. Since eq. 9 is indeed the same as the one found in [30, 15, 19, 17, 18]:

$$p_{no} = \frac{1}{m^n} \cdot \frac{m!}{(m-n)!} = \prod_{i=0}^{n-1} 1 - \frac{i}{m}, \tag{11}$$

the more intuitive explanation discussed above yields the same approximation for the probability of no collision. However one should note, that Leroy et al. [30, 17], as well as Knottenbelt et al. [18] developed a more pessimistic approximation. Instead of employing $1 - \frac{(n-1)}{m} \approx e^{-\frac{(n-1)}{m}}$ the authors apply the Stirling approximation for factorials on eq. 11, yielding $p_{no} \approx e^{-n^2/m}$.

Following Knottenbelt [19], one may conclude, that eq. 10 is a lower bound in cases, where $\frac{2}{3}n^2 < m$ holds. This condition is tolerable in all cases of practical interest, since for a high probability of no collision $p_{no} \approx 1$: $n^2 \ll 2m$. Furthermore one should note, that p_{no} can be interpreted as the minimum probability of collision, if at most n states are explored. I.e. in case more states are explored, one would need to readjust m in order to maintain the SSp coverage. This gives one the maximum probability of omission $P_{om}^{max} = 1 - P_{no}^{min}$. This means in a nutshell, that the SSp exploration needs to be aborted, if more than $n = \sqrt{-2m \cdot \ln(p_{no})}$ (derived from eq. 10) states are visited, since otherwise P_{om}^{max} can not be guaranteed. The notion of P_{no}^{min} and P_{om}^{max} will become important if the hash table oriented schemes are compared, since they allow the storage of at most n states only.

5.3 Criticism of bitstate hashing

In the sequel the need for new approaches in probabilistic state space exploration is justified by presenting some reasonable criticism of bitstate hashing and its efficiency analysis, as it was reported in this section and in section 4.

In [14] Holzmann derived some approximate formulae to compute the expected coverage of bitstate hashing. This factor is equal to the average probability that

[3] Todays 32-bit architecture restricts the size of a state table to the size of 2^{32}, i.e. $m \leq 8 \cdot 2^{32}$.

a particular state is not omitted in analysis. This average is computed over all reachable states. In [14] it is assumed that one hash collision leads in average to the omission of only one successor state. This assumption is in general not valid. In addition this omission probability does not constitute a bound on the omission probability of a particular state. This is essential, since not all states have the same omission probability, states that are only reachable via one or few long paths have a much higher omission probability. It is possible that such states are error states, such that we can not give a bound of the probability that a verifier generates false positives. A further assumption that is also questionable is the assumption that the reachability graph is well-connected. This assumption reduces the omission probability only if the paths that lead to a certain state have only few states in common, but this is doubtable.

From eq. 10 one can conclude, that bitstate hashing requires a huge number of unused positions in the bitstate array in order to maintain a high level of SSp coverage. Since for achieving a high probability of no collision (≈ 1), $n^2/2m$ needs to be close to zero ($2m \gg n^2$). I.e. employing eq. 10 on the memory utilisation rate ($\mu_{mem} = n/m$) and for levels of SSp coverage toward 100% one obtains:

$$\mu_{mem} \approx \sqrt{\frac{-2 \cdot \ln(p_{no})}{m}} \quad \overset{p_{no} \to 1}{=} \quad 0,$$

Thus the major disadvantage of bitstate hashing, is the waste of memory, when it comes to SSp coverage rates close to 1. For exemplification one may assume a SSp of size $n = 10^8$. In order to maintain a $p_{no} = 1 - 10^{-6}$ one would need to allocate a bitstate array of size 625 EByte ($8 \cdot 625 \cdot 10^{18}$ Bit). Such memory space is truly beyond even tomorrows hardware configuration, since even an advanced 64-bit architecture is limited to the use of bitstate arrays of 18.5 EByte ($8 \cdot 2^{64}$ bits = 147.5 Ebit). The other way round, i.e. considering a todays (tomorrows) hardware configuration, say 500 MByte (10 GByte) of RAM, bitstate hashing allows one to explore only $\approx 29,032$ ($\approx 129,837$) states at a poor level of p_{no} of 0.9. Therefore Holzmann [12, 14] suggested to executed k runs (SSp explorations), each with its own independent hash function. Doing so will augment p_{no} to $1 - p_{om}^k$, where $p_{om} < 1$.

6 Multiple Hashing Schemes

The original probabilistic SSp exploration scheme introduced by Holzmann [10] has the disadvantage, that the number of bits m, required to maintain a low probability of state omission p_{om}, needs to be much larger than the number of states to be explored n. In order to increase SSp coverage and minimise the probability of a wrong result some proposals have been made. The suggestions range from the sequential use of 2 independent hash functions up to the repeated execution of SSp generation by employing either k bitstate hashing schemes in parallel or a multi-fold hashing scheme sequentially. This section will focus only on the two main approaches as proposed in [29]. However, before we proceed one should note that in nowadays' hardware configurations one can easily allocate bitstate arrays of some billion bits, consequently in the following only the case $n \ll m$ will be considered.

6.1 Multiple hashing on a single bitstate array

Multiple hashing is based on the application of k independent hash functions. Each hash function determines a specific bit position, which has to be checked for determining a state's first or repeated occurrence. I.e. a state is considered to be already visited, if and only if all k bit positions in the bitstate array are occupied (flagged with 1), i.e.

$$\begin{array}{c}\text{if } H[h_1(s)] \ = 1 \wedge H[h_2(s)] \ = 1 \wedge \ldots \wedge H[h_k(s)] \ = 1 \\ \text{then state } s \text{ is assumed to be already visited.}\end{array}$$

Thus each state is now represented by at most k bits, where each bit to be checked is determined by applying one of the k independent hash functions to the state descriptor. In contrast to the subsequently discussed alternative scheme, the multi-fold hashing scheme analyzed below operates on only *one* bitstate array consisting of m bits.

Holzmann [12] introduced a 2-fold, Wolper and Leroy [29] a 20-fold hashing scheme, carried out on a single bitstate array of size m. For shortening the discussion, we will now directly consider the general case of a k-fold hashing scheme, where the analysis is based on the assumption that each state inserted will occupy exactly k bits in the bitstate array. Since the hash functions are independent there are m^k ways of selecting k out of m positions. Due to the above assumption, i states will occupy $i \cdot k$ of the m positions in the bitstate array. Applying k hash functions, $(i \cdot h)^k$ combinations consisting of occupied positions only can be generated. I. e. after inserting state i into the table $m^k - (i \cdot h)^k$ combinations using at least one free bit position are available. This gives the probability of inserting state $(i + 1)$ into the bitstate array without colliding on all bit positions:

$$p_{no}^{i+1} \ = \ \frac{m^k - (i \cdot k)^k}{m^k}$$

Thus the probability of no collision for a k-fold scheme can be described by [29]:

$$p_{no} = \prod_{i=0}^{n-1} \frac{m^k - i^k \cdot k^k}{m^k} = \prod_{i=0}^{n-1} 1 - \left(\frac{i \cdot k}{m}\right)^k , \tag{12}$$

As a consequence of the assumption, that each state occupies k bits eq. 12 is pessimistic, since in reality most states will collide at least on some bit positions and even some of the k independent hash function may map a specific state descriptor onto the same bit position. In the same manner as eq. 10 was derived, one obtains as approximation for the probability of no collision under the k-fold hashing scheme for $\frac{k\cdot(n-1)}{m} \ll 1$:

$$p_{no} = \prod_{i=0}^{n-1} 1 - \left(\frac{i \cdot k}{m}\right)^k \approx \prod_{i=0}^{n-1} e^{-\frac{i^k \cdot k^k}{m^k}} = e^{-\frac{h^k}{m^k}\sum_{i=0}^{n-1} i^k}$$

For $\sum_{i=0}^{n} i^k = n^{k+1}/(k+1) + O(n^k)$ [7,30] and $n \gg 1$ one obtains:

$$p_{no} \approx e^{-\frac{n^{k+1}\cdot k^k}{(k+1)\cdot m^k}} \tag{13}$$

Thus a 2-fold hashing scheme employed on a SSp of 10^8 states requires a memory space of $\approx 144 \cdot 10^3$ GByte in order to maintain a SSp coverage rate of $1 - 10^{-6}$, which is much better than the original bitstate hashing scheme (see example at end of sec. 5.3, p. 17), but still not satisfactory. After assessing various scenarios differing in the probability of no collision and size of available memory, Wolper and Leroy [29] came to the conclusion, that the use of 20 hash function is the scheme of choice. The decision for $k = 20$ is hereby influenced by the fact, that beyond a certain level of k, the bitstate array will fill up too quickly and thus have a negative impact on the probability of no collision, whereas a small number of hash functions will require a huge bitstate array in order to achieve high coverage rates. However, as illustrated in fig. 5 and fig. 6, where in contrast to Wolper and Leroy [29] higher levels of SSp coverage and larger sizes of memory space have been considered, a choice of $k = 20$ seems only appropriate in cases where lower levels of SSp coverage ($p_{om} < 10^{-6}$) are sufficient and a large number of states ($> 8 \cdot 10^7$) needs to be explored. I.e. in nowadays hardware configurations a choice of $k = 30$ seems more appropriate, in order to keep the probability of state omission low.

We consider a scenario, where a bitstate array of 500 MByte ($8 \cdot 500 \cdot 10^6$ Bit) is available. Under the suggested 30-fold hashing approach, one would be able to explore 52.168 (65.191) million states, at a level of 99.9999% (99.99%) of SSp coverage. In case of Holzmann's 2-fold hashing scheme one would be able to explore only $22,894$ $(106,268)$ states. Finally one should note, that assuming a use of 30 bits per state the 30-fold hashing scheme yields a memory utilisation rate of $\mu_{mem} \approx 38\%$ ($\approx 49\%$) in this scenario. However 47 (31) bits per state are still unused, since for achieving a high level of SSp coverage $4000/52.168 \approx 77$ ($4000/65.191 \approx 61$) bits per state need to be allocated in the bitstate array.

6.2 Multiple hashing on a partitioned bitstate array

As an alternative to the k-fold hashing scheme, Wolper and Leroy [29] suggested a partitioning of the bitstate array into p disjoint parts. Each sub-array H_i is hereby administered by its own independent hash function h_i. Thus a state s is considered as being already visited, if all p bit positions in the p sub-arrays (H_i) are occupied ($H_i[h_i(s)] = 1; 1 \le i \le p$), which may lead to an omission.

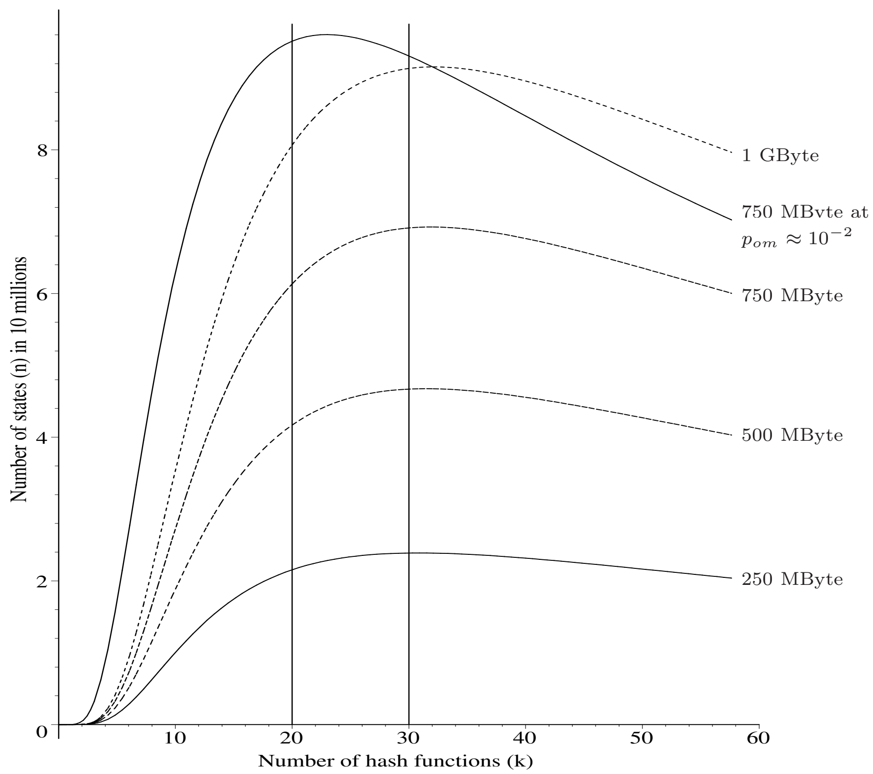


Fig. 5. Fixed SSp coverage ($p_{om} = 10^{-6}$) and varying size of memory

If one employs p independent bitstate arrays, each of size $\frac{m}{p}$ and administered by its own independent hash function h_i, there are $(\frac{m}{p})^p - i^p$ different ways of entering the $i+1$ visited state into the p bitstate arrays, yielding [29]:

$$p_{no} = \prod_{i=0}^{n-1} \frac{\left(\frac{m}{p}\right)^p - i^p}{\left(\frac{m}{p}\right)^p} = \prod_{i=0}^{n-1} 1 - \left(\frac{i \cdot p}{m}\right)^p \quad . \tag{14}$$

This is the same result as obtained for the *multi-fold hashing scheme on a single* bitstate array (k = p). This is not surprising, since under both approaches the occurrence of collisions is identical and in both cases it was assumed that each state inserted into the bitstate array(s) will occupy exactly p bits. This assumption turns out to lead to a more pessimistic estimation for the probability of no collision in case only *one* bitstate array is employed. In case of partitioning the bitstate array into p independent parts, this assumption will at least hold for

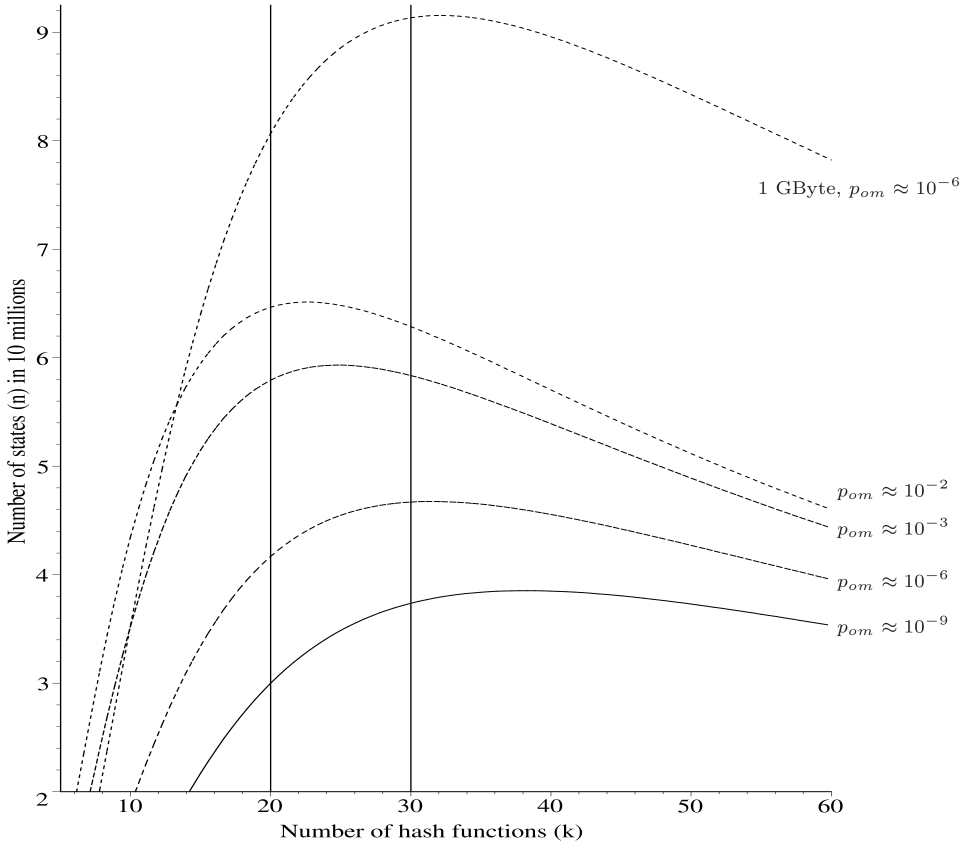


Fig. 6. Memory Space of 500 MByte and varying levels of SSp coverage

the first element. Thus one may conclude, following Wolper and Leroy [29] that a multi-fold hashing on a non-partitioned bitstate array will consume fewer bits per state, and thus have a lower probability of error and therefore must to be preferred.

One may interpret the hashing scheme on p independent bitstate arrays as the parallel execution of a bitstate hashing on a single bitstate array of size $\frac{m}{p}$, where the SSp exploration is carried out p times and where for each run an independent hash function is used. However doing so would give an overall probability of state omission of:

$$p_{om} = (1 - p_{no})^p \ ,$$

where p_{no} is the probability of no collision per run. The latter is identical for each run, since independent hash functions and bitstate arrays of equal sizes ($\frac{m}{p}$) are employed. As a consequence, one can use eq. 9, where the size of the bitstate array needs now to be set to $\frac{m}{p}$, yielding:

$$p_{om} = (1 - p_{no})^p \; = \left(1 - \prod_{i=0}^{n-1} 1 - \frac{i \cdot p}{m}\right)^p \quad .$$

However further analysis seems appropriate, but goes beyond the scope of this survey paper and is left to the interested reader.

6.3 Summary

The multi-fold hashing schemes discussed so far improve the SSp coverage rate by increasing the memory utilisation rate (μ_{mem}). However, this improvement comes with a substantial increase of the runtime, if compared to the initial bitstate hashing scheme. This additional runtime overhead is due to the computation, comparison and toggling of multiple bit positions per state, which requires additional CPU time. Therefore Wolper and Leroy [29] had the idea of simulating a bitstate hashing scheme by storing the addresses of the occupied bit positions, rather than storing the whole bitstate array. This approach is called hash compaction and is the subject of the following section.

7 Hash Compaction

Wolper and Leroy [29] developed the idea of storing the addresses of the occupied bit positions in the bitstate array, rather than storing the whole array itself. Consequently, such a procedure would simulate a bitstate hashing at a given level of no collision, which would be beyond nowadays hardware configurations. Furthermore, such an approach would not suffer from the computational overhead induced by computing h different hash values. Consequently, the hash compaction scheme targets a low probability of collision by increasing the memory utilisation rate (μ_{mem}) and avoiding computational overhead as in case of the multiple hashing schemes.

7.1 Functionality of the hash compaction approach

Wolper and Leroy [29] suggested that each state descriptor is mapped onto a string of b bits by applying a hash function (h_c). This procedure can be viewed as a compression of state descriptors. The compressed state descriptors are then stored in a hash table, which is assumed to consist of as many slots as states to be explored (n). The position (slot index) of a compressed state descriptor within the hash table is hereby determined by employing another hash function (h_p). Unfortunately Wolper and Leroy [29] do not explain how the slot index for positioning a compressed state descriptor into the hash table is generated.

However, from a later work of Leroy, Wolper and others [30] one can deduce that a state descriptor's position in the hash table is not generated independently of its compressed value (see discussion below). Thus two states are wrongly considered to be identical, if their compressed descriptors are equal, since both will be mapped onto the same position within the hash table. I.e. from $h_c(s_i) = h_c(s_j)$ one concludes $s_i = s_j$, even though $s_i \neq s_j$, since for $h_c(s_i) = h_c(s_j)$ hash function h_p will generate the same slot index ($h_p(s_i) = h_p(s_j)$). In cases, where states collide on slots, but for which $h_c(s_i) \neq h_c(s_j)$, a direct chaining of the overflows for collision resolution is applied. According to Wolper and Leroy [29] the imposed overhead for chaining of overflows is negligible, since the number of slots is n and thus a collision on slots will be rarely the case. However one should note, that the collision resolution is mandatory for the analysis carried out below.

In a nutshell the hash compaction scheme can be understood as an adaptation of the common bitstate hashing approach as discussed in sec. 3.1. Where the original bitstate scheme interpreted the hash values as position within the bitstate array, the hash compaction scheme stores the obtained hash values in a hash table consisting of n slots. Due to the non-injective compression function h_c, states may be wrongly considered as identical and thus the SSp may not fully be covered.

7.2 Analysis of hash compaction

It is intuitively clear that the hash compaction scheme has a lower time complexity if compared with the 30-fold hashing, since only 2 hash functions are employed (compression and slot indexing only). The main question to be answered is the performance concerning space complexity. The analysis is hereby straightforward.

The compressed state descriptors can be interpreted as positions within a bitstate array. Since the compressed descriptors are mapped onto hash values $\in [0, 2^b - 1]$ the hash compaction method simulates a bitstate hashing scheme on a bitstate array of size $m = 2^b$. Since the probability of no collision for the latter was approximated by $p_{no} \approx e^{\frac{-n^2}{2m}}$, p_{no} under the hash compaction scheme can be approximated by

$$p_{no} \approx e^{\frac{-n^2}{2^{b+1}}}. \tag{15}$$

From Knottenbelt's work [19] one may conclude, that the above approximation is a lower bound if $2^b > 2/3n^2$. Eq. 15 gives the number of bits required per compressed descriptor for exploring a SSp of a given size and at a fixed probability of no collision:

$$b = \log_2 \left(\frac{-n^2}{2\ln(p_{no})} \right). \tag{16}$$

Since eq. 15 is based on the fact that only states are omitted which map onto the same hash value, it is mandatory that a resolution scheme for collisions on slot indices must be employed. Otherwise the approx. of eq. 15 had to be corrected, leading to a lower p_{no}. However it is also clear, that an independent generation of the slot indices will improve the SSp coverage rate, since here a collision only occurs, if slot indices, as well as the compressed state descriptors of two distinct states collide. Therefore it is not clear why Wolper and Leroy [29] did not suggest an independent generation of slot indices. It is intuitively clear that in such a case one would obtain a lower probability of omitting a state, since the probability of colliding on a slot and a compressed descriptor is given by their product, where the individual probabilities are < 1.

Under the hash compaction scheme the number of states (n) to be explored is limited by $\frac{M}{b+1}$, where M is the size of memory in bits, where b is the number of bits per compressed state descriptor and where an additional bit, indicating a slots status (occupied or not), must be spent. This flagging is hereby only required under the original scheme, since if one considers a mapping of the state descriptor onto the interval $[0, 2^b - 2]$, one value would be left for indicating a slot's emptiness. The administrative overhead for indicating a slots status is then reduced to b bits and p_{no} is almost not affected from doing so, because $2^b \gg 1$. But for the time being the original approach will be considered, where each state to be stored requires $b+1$ bits and following Wolper and Leroy [29] the overhead for chaining overflows will be neglected as well. The size of the compressed state descriptors b influences on the one hand the number of states to be stored and on the other hand the SSp coverage. From the probability of collision one can derive to choose b as large as possible for achieving a high level of SSp coverage ($\frac{n^2}{2^{b+1}} \rightarrow 0$). However $n = \frac{M}{(b+1)}$, which is the number of states to be stored in the memory, implies a small value for b. The probability of no collision (eq. 15) together with the required space in bits M for storing n states ($M = n \cdot (b+1)$) yields:

$$M \approx (b+1) \ \cdot \ \sqrt{-2^{b+1} \cdot \ln(p_{no})}. \tag{17}$$

I.e. if one uses the above eq. 17 for determining M for a given b and a fixed p_{no}, the latter can be interpreted as the minimum probability of no collision or vice versa it gives us the maximum probability of omission ($P_{om}^{max} = 1 - P_{om}^{min}$). The interpretation as minimum or maximum probability of omission is based on the following view:

1. In case the available memory M_{avail} is less than M, the number of states which can be stored n_s is smaller than $\sqrt{-2^{b+1} \cdot \ln(p_{no})}$. I.e. after the $n_s +$ 1'st state is reached, the exploration must be aborted since p_{no} is truly zero, as the $n_s + 1$'st state will produce a memory overflow.
2. In case $M_{avail} > M$ one obtains $n_s > n = \sqrt{-2^{b+1} \cdot \ln(p_{no})}$. But as soon as the $n + 1$'st state is reached, the exploration must be aborted as well, since p_{no} is violated. In such a situation one needs to recalculate b in order to reach the desired level of P_{no}^{min} and restart the exploration routine.

Fig. 7 shows the relation between memory space and size of compressed state descriptors at $P_{no}^{min} = 1 - 10^{-6}$. The graph can be interpreted as follows: A 8 to 9

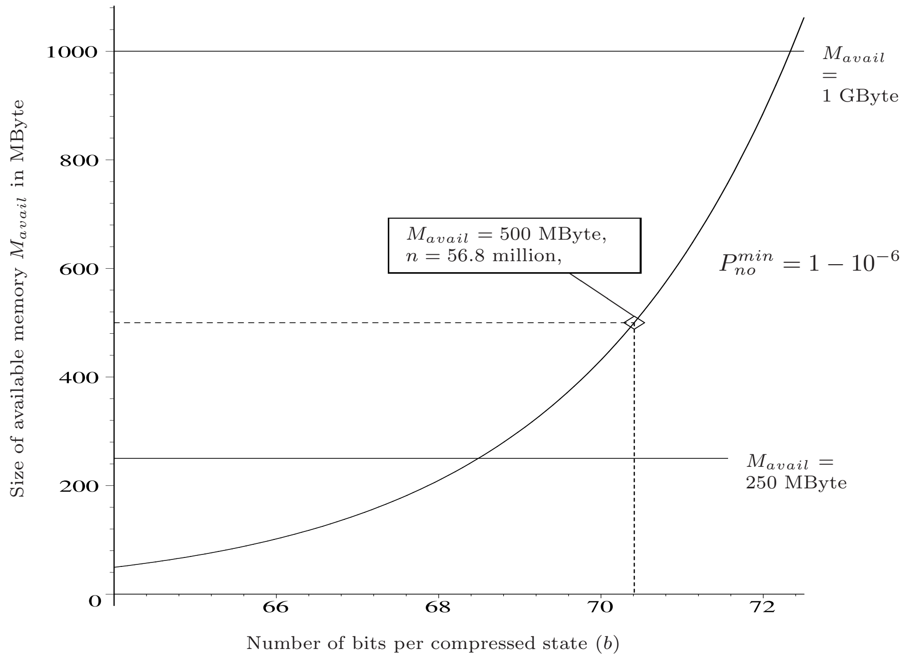


Fig. 7. Relation between size of memory space and size of hash key

Byte compression of each state descriptor requires a memory size of 49.35 MByte up to 1 GByte in order to explore a SSp consisting of 6.074002 up to 97.184040 million states at $P_{no}^{min} = 1 - 10^{-6}$. If one explores fewer states, the actual p_{no} will be larger. In case one explores more states, the exploration routine will run out of memory, yielding $p_{no} = 0$. This means that one obtains the number of states to be explored just by dividing M_{avail} by $b + 1$, where b depends on the square of the number of states n and the natural logarithm of p_{no}. Thus a 8 to 9 Byte compression of each state descriptor seems to be in the reach of today's average hardware configurations in order to guarantee a minimum coverage rate of 99.9999% and explore SSp between ~ 6 and ~ 97 million states.

For the purpose of illustration one should now imagine, that 500 MByte of memory are available. According to eq. 17 and as illustrated in fig. 7 a compression of each state descriptor onto $b = 70.41$ bits would be optimal in order to reach $P_{no}^{min} = 10^{-6}$. [4] I.e. under a 71-bit compression the hash compaction scheme

[4] One should note that a hash value of 64 (70) bit length, as suggested in [29] does not suffice, since compressed state descriptors of such a size allow one only the exploration of 4.29 (48.6) million states at a minimum coverage rate (P_{cov}^{min}) of $1 - 10^{-6}$.

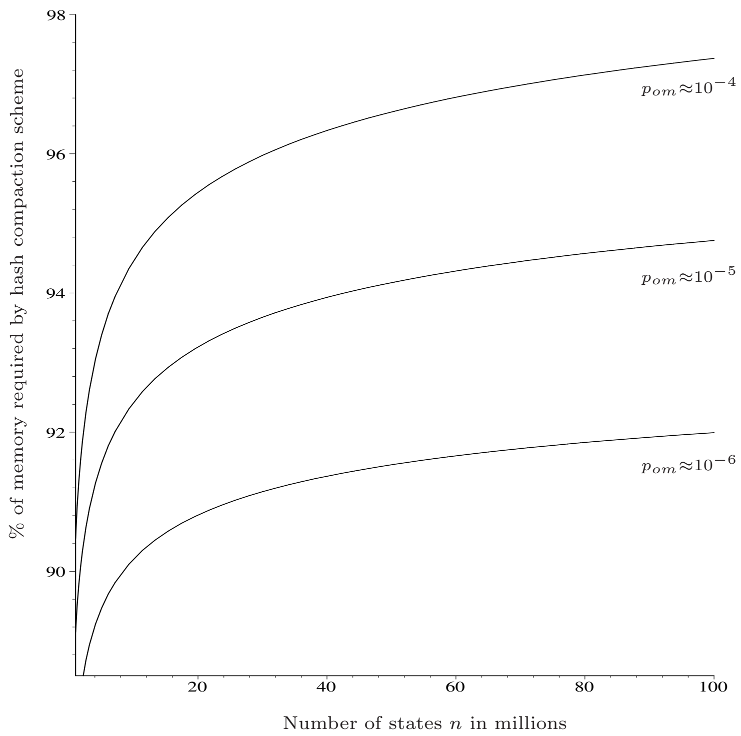


Fig. 8. Mem. requirement of HC compared to Multiple Hashing

enables one to store 55.5×10^6 states at a coverage of $0.9999993 < P_{no}^{min}$. [5] In contrast the 2-fold (30-fold) bitstate hashing scheme allows one to explore here $20,398$ (51.57×10^6) states. Thus the hash compaction method outperforms the 2-fold bitstate hashing by far, but the advantage over the 30-fold hashing scheme is not that obvious. For a more detailed analysis one just need to compare the memory requirements of the multiple hashing scheme and the hash compaction at the same levels of SSp coverage and not at the same levels of P_{min}^{no}. This is done in fig. 8, where the memory space of the multiple hashing is considered to be 100% and the ordinate shows the percentage required by the hash compaction. Since the latter is clearly below 100%, the lower space complexity of the hash compaction scheme is obvious. Furthermore the better run-time behaviour speaks to the advantage of the hash compaction method, since the latter only requires the application of two hash functions, where under the multiple hashing approach 30 hash values per states need to be computed.

[5] Each slot (state) requires the 71 bits for storing the compressed descriptor plus an additional status bit, yielding 72 bits per state.

8 Dynamic Hash Compaction

The dynamic hash compaction scheme is implemented in the performance analysis tool DNAmaca [15]. In [19, 17] and [18] the basic probabilistic SSp exploration scheme was extended for being used in a distributed computing environment. For the purpose of this survey paper we will mainly focus on the non-distributed version. But for the sake of completeness the distributed scheme will be briefly explained at the end of the section.

8.1 Functionality of dynamic hash compaction

The dynamic hash compaction scheme is based on a hash table and direct chaining of overflows. But rather than storing the full state descriptors it stores compressed state descriptors, generated by applying a second hash function, $h_c : S \mapsto \{0, \ldots, 2^d - 1\}$. The first hash function $h_s : S \mapsto \{0, \ldots, r-1\}$ determines the slot in which the state needs to be stored. Collisions of different compressed descriptors on slots are resolved by chaining.[6] As a consequence two different states ($s_i \neq s_j$) are considered equal, if they collide on *both*, indexing and compression:

$$\text{if } h_s(s_i) = h_s(s_j) \wedge h_c(s_i) = h_c(s_j) \text{ then } s_i = s_j.$$

Since h_s and h_c are not injective, this conclusion might be wrong, which results in the omission of states.[7]

8.2 Analysis of the dynamic hash compaction

The main improvement of the dynamic hash compaction scheme, in contrast to the original hash compaction approach is, that the slot indices are generated independently from the compressed values. Since the hash function h_s maps the n state descriptors onto r different slots, where the independently generated compressed descriptors are stored, the dynamic approach can be understood as the r-fold application of a hash compaction scheme. As one may recall from the hash compaction scheme, a compression of state descriptors onto bit strings of length b simulates a bitstate hashing with an array of size 2^b. Consequently the dynamic hash compaction simulates a bitstate hashing onto an array of size $m = r \cdot 2^d$. In the style of sec. 5.2 and eq. 10, by simply replacing m this yields:

$$p_{no} \approx e^{\frac{-n^2}{r \cdot 2^{d+1}}} \tag{18}$$

[6] Examples of appropriate hash functions can be found in [17].

[7] For the sake of completeness it should be mentioned that for generating a transition matrix, the states are consecutively numbered and that this number is stored with the compressed descriptors. However this paper focusses on the generation of different states and not on the subsequent analysis, thus the linear overhead induced by the element counter is neglected.

Knottenbelt [19] has shown that eq. 18 is actually a lower bound, in cases where $2/3n^2 < r \cdot 2^d$. This later criterion is normally given in practice, since a high level of state space coverage requires $\frac{n^2}{r2^{d+1}} \to 0$ and thus $n^2 \ll r2^{d+1}$ [19]. Similar to the analysis of the hash compaction p_{no} can be interpreted as the minimum probability of collision ($P_{om}^{max} = 1 - P_{no}^{min}$), if sufficient memory for storing the n states is supplied. In case the available memory is larger than required, one would be enabled to store more than n states. However, this would violate p_{no} and in such cases one would need to recalculate r and d in order to maintain the fixed p_{no}, which would induce a higher memory requirement.

Concerning the memory consumption, one should note, that the dynamic hash compaction scheme stores the slot overflows not in lists but makes use of dynamic allocated arrays. Since each array is referenced by a pointer and equipped with a counter in order to track the number of overflows per slot individually the space complexity of the dynamic hash compaction is described by:

$$M = n \cdot d + \rho \cdot r, \tag{19}$$

where ρ is the number of bits per slot (row) for administering the at most r different dynamic allocated arrays. This gives an administration overhead of $\frac{\rho \cdot r}{n}$ bits per state. Since $r \ll n$, collision on slots will occur quite frequently, i.e. a substantial number of overflows may be stored per slot. As a consequence the dynamic hashing scheme can be understood as a dynamic *2-dimensional* hashing scheme, where h_s determines the row index and where h_c computes the column index for each state descriptor to be inserted. Therefore slots are in the following referred to as rows.

The next question to be dealt with, is about the choice for good pairs of r and d, in order to minimise the memory requirement at a given level of state omission and size of SSp.

For simplification let r be described by a power of 2, i.e. $r = 2^k$ and thus $p_{no} \approx e^{\frac{-n^2}{2^{k+d+1}}}$. If the sum of k and d equals the number of bits used under the hash compaction scheme for storing a compressed state descriptors, one cannot notice any difference concerning SSp coverage. From formula 19 one can easily obtain, that an increment of d yields an additional memory consumption of n bits. However an increment of k comes along with an additional memory consumption of

$$\rho \cdot (2^{k+\epsilon} - 2^k) = \rho \cdot 2^k(2^\epsilon - 1),$$

per increase of k by ϵ. In order to keep the probability of collision now constant, it is clear, that an increment of k requires a decrement of d and vice versa. The idea is now, to increase k and decrease d as long as the additional row overhead per state is ≤ 1, since otherwise an increment of d, in order to maintain the probability of no collision and minimise the additional memory space required ought to be favoured. Since d can only be decremented by whole numbers, the smallest change of k is given by $\epsilon = 1$. Together with the bound of 1 bit increase in memory consumption per state one obtains:

$$\frac{\rho \cdot 2^k}{n} \leq 1$$

$$=> k \leq \log_2\left(\frac{n}{\rho}\right). \tag{20}$$

Together with formula 18 one finally obtains:

$$d \approx \log_2\left(\frac{-n \cdot \rho}{2 \cdot \ln(p_{no})}\right) \tag{21}$$

Thus for a given number of states n and bits per row ρ, one obtains a value of k, where another doubling of r (increment of k by 1) would lead to an additional memory consumption of more than one bit per state. I.e. in order to minimise memory consumption and maximise SSp coverage, it would be more sensible to increase the number of bits per compressed state descriptor by one instead. –One should note, that Knottenbelt et al. [19, 17, 18] make use of a different approach. In order to minimise M they deploy eq. 18 in eq. 19 and derive the resulting term with respect to d. Since a minimum for M is targeted, the resulting eq. can be set equal to 0 and finally resolved for d. This procedure yields almost the same result as the straight forward approximation followed here.

For exemplification one should now consider the exploration of 10^8 states at a coverage rate of $1 - 10^{-6}$. Under the original hash compaction method one has to use $b = 72.08$ per compressed state descriptors in order to maintain the required coverage rate. As a consequence the original hash compaction method requires 912.5 MByte of memory in order to explore 10^8 states and maintain a SSp coverage rate of 99.9999%.[8] According to eq. 20 and 21 the dynamic hash compaction scheme requires here only the use of $d = 51$ bit. The additional allocation of $r = 2^{21}$ rows guarantees the desired level of SSp coverage. Thus the dynamic hashing scheme requires only 625 MByte in order to store 10^8 different states at a coverage rate of $P_{cov}^{min} = 1 - 10^{-6}$. This increase in memory space is quite obvious if one compares eq. 16 and 21, which discloses $d \leq b$ and considers that the allocation of r rows maintains the desired level of P_{no}^{min} for less memory space as required for the additional $d - b$ bits per state descriptor as required in case of the original hash compaction scheme. Furthermore, one should note that the overhead for collision resolution on slots under the hash compaction scheme was neglected so far, which speaks once more for the dynamic hash compaction scheme.

[8] A use of 73 bits per state was assumed, i.e. the status bit as required under the original approach was considered, but $P_{no}^{min} = 1 - 10^{-6}$ is therefore violated slightly.

8.3 The Distributed Dynamic Hash Compaction

The basic dynamic hash compaction approach was extended for being used in a distributed computing environment [19, 17, 18]. The distribution of a state to a partial SSp exploration routine running on a specific machine is realized by a hash function. I.e. by consecutively numbering the target machines $(0, ...P-1)$ and employing a hash function $h_d : S \rightarrow \{0, ...P-1\}$, each state is mapped non-ambiguously to a machine.[9] As a consequence two distinct states $(s_1 \neq s_2)$ are considered identical if they not only collide on the slot index and compressed descriptor values, but are *distributed to the same machine*:

$$\text{if } h_d(s_1) = h_d(s_2) \wedge h_p(s_1) = h_p(s_2) \wedge h_s(s_1) = h_s(s_2) \text{ then } s_1 = s_2.$$

As a consequence the distributed dynamic hashing scheme simulates a bitstate hashing scheme on an array of size $p \cdot r \cdot 2^{d+1}$. Thus the probability of no collision is given by:

$$p_{no} \approx e^{-\frac{n^2}{p \cdot r \cdot 2^{d+1}}}. \tag{22}$$

It is clear, that the enormous increase of memory space (by factor P)and a parallelisation of the SSp exploration enables one to explore SSps whose size is clearly beyond 10^8. But one has to accept an increase in runtime due to the communication overhead. A profound analysis of the latter can be found in [17, 18].

9 Improvement of the Hash Compaction Scheme

In this section a new scheme, introduced by Stern and Dill in [26] will be presented. The method is inspired Wolper's and Leroy's [29] hash compaction method.

9.1 Improved hash compaction scheme

In general terms, this method proceeds as follows:

1. From a given state descriptor s, using a compression function h_c, a compressed value, $h_c(s)$, which is b bits long, is computed.
2. From the same state descriptor as in 1, applying a hash function h_p, a probing sequence $h_p(s)$ is generated.
3. The compressed value $h_c(s)$ from 1 is inserted at the address, computed in 2.

Note, that the order of 1 and 2 is irrelevant, i.e. 1 and 2 are independent. In figure 9 the functionality of the improved hash compaction method is shown pictorially. Figure 9 (a) shows step 2, figure 9 (b) shows step 3 of the description

[9] It is clear, that the hash values can be injectively mapped to IP-, or MAC-addresses.

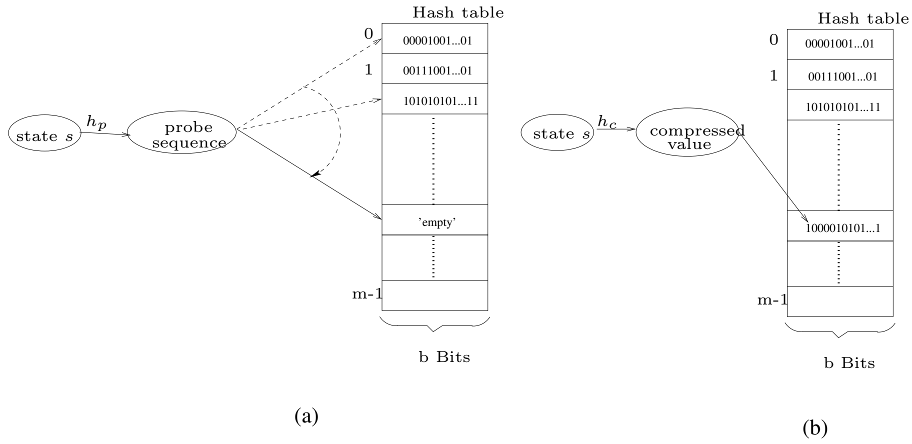


Fig. 9. Improved hash compaction

above.
The compression of the state descriptor was already applied in Wolper's and Leroy's original hash compaction method (cf. section 7). The differences between the original hash compaction method and Stern and Dill's improved scheme are as follows:

- In the improved scheme open addressing instead of chaining is employed to resolve collisions.
  The advantage of open addressing is a lower memory consumption, because chaining requires the additional storage of pointers, whose size corresponds roughly to the size of a compressed entity that is to be stored in the hash table. The improved hash compaction scheme requires approximately only half of the memory that is needed by the method presented in section 7.
- In the new scheme, the hash value that determines the address at which the state is to be inserted and the compressed value are computed independently. That means that the hash value is computed from the uncompressed state descriptor. This has the advantage that the same compressed value can appear at several different places in the hash table. As different states may be mapped onto the same compressed value and the same address this might result in less omissions than in Wolper's and Leroy's original hash compaction scheme.

Hashing and compression:

- The hashing scheme employed here, is uniform probing. A description of uniform probing was given in section 2. The employment of uniform probing is an assumption only used for the analysis. In the implementation some approximation of uniform probing has to be used.

– In improved hash compaction, the compressed state descriptor $h_c(s)$ is stored in the hash table. We assume that the state descriptor s is compressed down to b bits. Thus, the compression function can take $l = 2^b$ different values, so, h_c has a range from 0 to $l - 1$.
Of particular interest is the notion of uniform compression. In uniform compression each state s is equally likely to be one of the values $0, \ldots, l-1$. To achieve this goal, universal hashing can be employed. If the compression function h_c, is selected as described for universal hashing (cf. section 2) then it is guaranteed that the probability for two states s_1 and s_2 to be mapped onto the same compressed value is not greater than $1/l$:

$$\forall s_1, s_2 \in S,\ s_1 \neq s_2 :\ Pr(h_c(s_1)) = h_c(s_2)) \leq \frac{1}{l}$$

State insertion: When inserting a new state into the hash table, three cases can be distinguished:

1. The probed slot is empty, the state is inserted.
2. The probed slot is not empty and contains a state whose compressed value is different from the one that is to be inserted. In this case the probing has to be continued.
3. The probed slot contains a compressed value that is equal to the one that is to be inserted. In this case it is concluded that the two uncompressed states are equal. If this conclusion is wrong, this leads to the omission of the probed state and possibly also to the omission of one or more successor states.

9.2 Analysis

A thorough derivation can be found in section 13.
As usual, n is the number of reachable states, m is the number of slots in the hash table.

Omission probability: The omission probability of hash compaction can be approximated as follows:

$$\begin{aligned} p_{om} &= Pr(\text{even one omission during state enumeration}) \\ &\approx 1 - \left(\frac{l-1}{l}\right)^C \end{aligned} \tag{23}$$

where

$$C = (m+1)[H_{m+1} - H_{m-n+1}] - n$$

The approximation 23 has the following explanation:
C is equal to the total expected number of hash collisions on h_c. As we employ uniform compression, the probability that two compressed values are not equal

is $(l-1)/l$, assuming that the comparisons are independent, the probability of zero omissions, i.e. zero collisions can be approximated as $[(l-1)/l]^C$.
The upper bound $\hat{p}_{om}$ on the omission probability is

$$p_{om} \leq \hat{p}_{om} = \frac{1}{l}C \tag{24}$$

Maximum omission probability: A thorough derivation can be found in section 13.
An approximation of the maximum omission probability in the case, when the hash table fills up is given, using the approximation of eq. 23:
We assume that $l \gg 1$, then, using that for small values of x, $1-x \approx e^{-x}$, $(l-1)/l = 1-1/l \approx e^{-\frac{1}{l}}$. This yields:

$$p_{om} \approx 1 - e^{-\frac{1}{l}\cdot C} \tag{25}$$

H_n, which appears in the definition of C can be approximated by $\ln(n)+\gamma$, where γ ist the Euler-constant ($\gamma \approx 0.57722$), e^x is approximated by $1+x$, so we obtain by assuming $m=n$, $l \gg 1$ and also $n \gg 1$:

$$\begin{aligned}
C &= (m+1)\cdot[H_{m+1} - H_{m-n+1}] - n \overset{m=n}{=} (n+1)\cdot[H_{n+1} - H_{n-n+1}] - n \\
&= (n+1)\cdot[H_{n+1} - H_1] - n \\
&= (n+1)\cdot[\ln(n+1)+\gamma-1] - n = (n+1)\cdot(\ln(n+1)+\gamma) - n - 1 - n \\
&= (n+1)\cdot(\ln(n+1)) + \gamma\cdot(n+1) - 2n - 1 \overset{n\gg 1}{\approx} n\cdot\ln(n)) + \gamma\cdot n - 2n \\
&\approx n\cdot(\ln(n) - 1.4)
\end{aligned}$$

Using eq. 25 and the approximation of the natural exponential function from above this yields:

$$P_{om}^{max} \approx \frac{1}{l}n(\ln(n) - 1.4) \tag{26}$$

Again, note that this approximation is only applicable for $m=n$.

Number of bits per state: The question, how many bits for the compressed value is addressed. Two different bit values are investigated in greater detail: 5-byte and 4-byte compression.
In the sequel it is assumed that 400 MByte of memory space is available for the state space search. Using a 5-byte compression scheme, the hash table can accommodate $m = 80$ million states, whereas assuming a 4-byte compression scheme the table can hold up to 100 million states. Figure 10 depicts the omission probabilities for the different sizes of the compressed state descriptors.

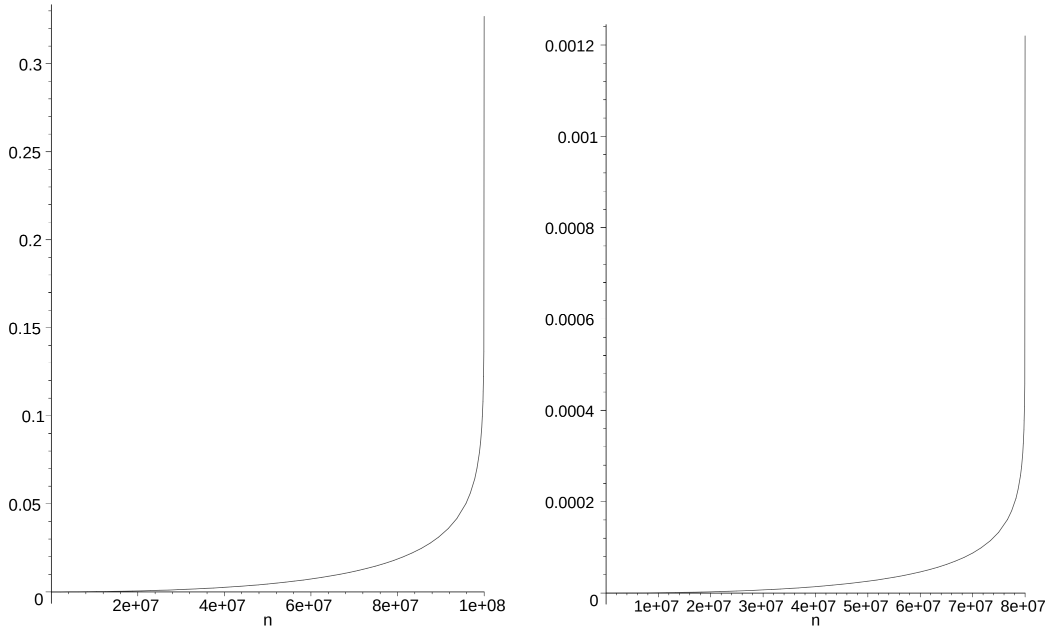


Fig. 10. Omission probabilities, using a 4 and 5 byte compression scheme

9.3 Comparison of improved hash compaction with hash compaction and dynamic hash compaction

In this subsection we will compare Stern's and Dill's improved hash compaction scheme with that of Wolper's and Leroy's original hash compaction and Knottenbelt's dynamic hash compaction scheme.
As dynamic hash compaction realizes compression and positioning of a state within the hash table independently, it is very interesting whether it performs better or not in comparison to improved hash compaction, both with regard to memory requirements and omission probability.

Hash compaction and improved hash compaction: We now briefly compare the original hash compaction scheme of Wolper and Leroy with the improved scheme of Stern and Dill. It is assumed that 10^8 states shall be analysed. The size of the compressed state will vary between 40 and 72 bits. From that follows, that the memory requirements lie between $\approx$ 500MByte for a 40-bit compression and $\approx$ 1 GByte for a 72-bit compression. From table 1 it becomes obvious that the improved hash compaction scheme outperforms the original one by orders of magnitude. This is clear from the fact that the omission probability in the improved scheme is the product of the collision probability on compression and the collision probability on hashing, whereas the omission probability in the original approach is equal to the collision probability on compression. For the improved

Bit key length	40	48	56	64	72
HC	$0.11 \cdot 10^{-1974}$	$0.19 \cdot 10^{-7}$	0.9329	0.9997	0.99999
IHC	0.99845	0.99999	0.9999999	0.9999999999	0.999999999999

Table 1. Probability of no omissions for hash compaction and improved HC

hash compaction approach we have taken the maximum omission probability (P_{om}^{max}), such that the probability of no omissions is the minimum probability of no omissions.

Improved hash compaction and dynamic hash compaction: In fig. 11 we see the omission probabilities of improved and dynamic hash compaction, where in part (A) the slight disadvantage of the latter concerning space complexity was neglected. Under dynamic hash compaction one can allocate a hash table of size $r = 2220445$ slots and map the state descriptors onto hash values of length of 51 bit, leading to a memory consumption of ≈ 645 MByte. In case of the improved hash compaction scheme a memory consumption of 637.5 MByte was assumed, i.e. the hash table consists of 10^8 slots and the state descriptors are mapped onto 50 or 51 bits. The different compressions are due to the use of either a flagging bit per slot or a dedicated flagging value. I.e. in case one allows only $2^b - 1$ different hash values the overhead for flagging can be reduced to b bits instead of employing n flagging bits. This reduction of the memory requirement enables one to use an additional bit for the compression (50 + 1), where the loss of one hash value is negligible, since $2^b \gg 1$. As a consequence the omission probability, even when it comes to fill-ups close to the size of the employed hash table, is clearly smaller, i.e. the improved hashing scheme performs better than the dynamic scheme concerning omission probability. However in cases were the flagging bit per slot is considered, the p_{om} and especially P_{om}^{max} increase, leading to an advantage of the dynamic scheme (fig. 11(A)). This result is very interesting, since in Knottenbelt's thesis [19] only this less advantageous case of flagging each slot was considered. But we cannot follow this since assuming $2^b - 1$ different hash values, the overhead for flagging is reduced to b bits and a compression onto 51 bits is possible. I.e. assuming an appropriate flagging scheme the improved hash compaction scheme outperforms the dynamic hashing scheme clearly concerning p_{om} and especially P_{om}^{max}. If we go one step further and compare both approaches at the same memory size, say $M_{avail} \approx 645$ MByte, the maximum number of states which can be stored in the memory varies slightly between the approaches. As a consequence the advantage of the improved hash compaction scheme over the dynamic one is clearly not to be neglected, whether one uses a flagging bit or not (fig. 11(B)). This effect is due to the circumstance, that the hash table under the improved scheme consists of $(645 \cdot 10^8 \cdot 8)/51 = 1.011764706 \cdot 10^8$ slots. I.e. one is actually enabled to store more than 10^8 states in the memory, however the SSp consits at most of 10^8 states. Thus the hash table won't be filled up totally, improving P_{om}^{max} considerably. However, one should note that the length of the probing sequences in case of the improved scheme is problematic and may increase the runtime dramatically, where as in case of the dynamic hash compaction the comparison is bounded by the number of compressed state descriptors per slot, which is on average $\frac{n}{r}$ states per slot.

10 Combining State Space Caching and Improved Hash Compaction

In this section we will show how an exhaustive state space exploration method, state space caching, can be combined with a probabilistic scheme, namely the improved hash compaction scheme, introduced in section 9. This is done in order to overcome some problems and inefficiencies of both state space caching and improved hash compaction. This approach was first reported by Stern and Dill in [28].

10.1 State space caching

We briefly introduce another state space exploration method, namely state space caching, which was investigated in [5].
State space caching performs an exhaustive exploration of all reachable states of a system that is to be analysed. It is a kind of depth first search. We will distinguish two "extreme" positions in performing a state space search as a depth first search:

- A traditional depth first search employs two data structures, a state buffer and a state table. This kind of analysis has very high memory requirements and the size of state spaces that can be analysed is limited by the memory space that is available for the state table (sec. 3.1). If the size of the state space exceeds the memory space the search might deteriorate into an uncontrolled partial search.
- It is possible to perform an exhaustive search by storing only the states of the path that is currently under investigation. This "store nothing" approach [5] has the advantage of an enormous decrease in memory requirements but comes along with a blow up in run time. The reason for the run time blow up is, redundant work has to be done, as the already visited states are not stored, each state and consequently the subgraph associated with it has to be explored as many times as different paths lead to that particular state.

As a compromise between these two extreme positions, state space caching takes the following approach:
The stack stores only the states of the path that is currently analysed. In the state table or cache only as many states are stored as memory is available. If the cache fills up, old states must be replaced by newly reached ones. This method pushes the limits of the size of protocols that are verifiable but still has to deal with the problem of an increase of the run time. To overcome the run time problem a partial order reduction method is employed that fully avoids

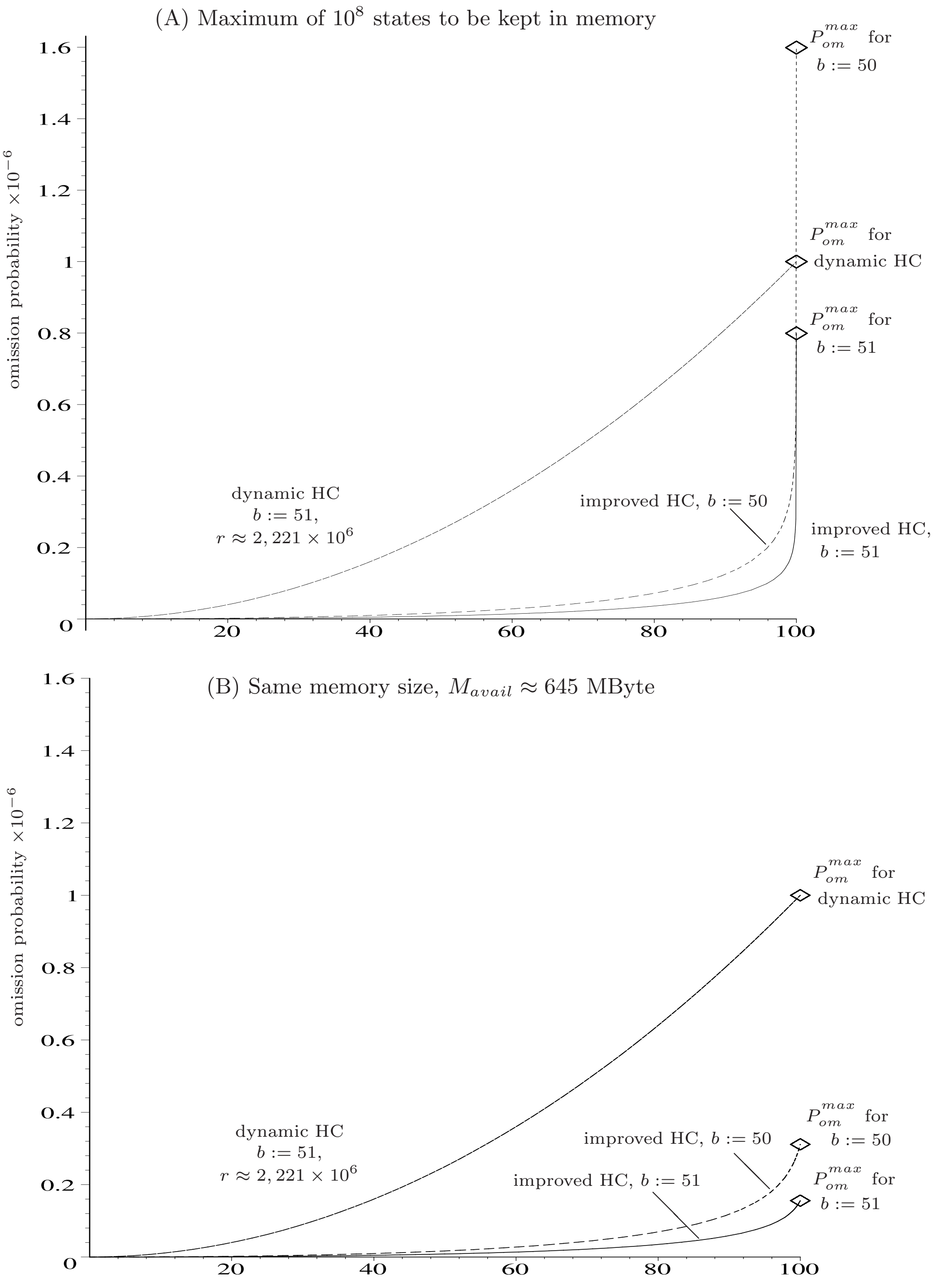


Fig. 11. Comparison of dynamic - and improved HC

redundant work, but still guarantees full state space coverage. So the method of state space caching is combined with sleep sets cf. [2, 6]. This approach has been successfully used in the model checker "VeriSoft" [4].

Introduction to sleep sets: Using classical depth first search all transitions that are enabled in a certain state are explored. But it can be observed that in concurrent systems all reachable states can be explored without taking all enabled transitions. Sleep sets constitute a means to overcome the state space explosion problem that stems from the huge number of possible interleavings. This is achieved by describing a system's behaviour by partial order methods, here by using Mazurkiewicz traces [21]. Traces are defined as equivalence classes of sequences of transitions. Two sequences are in the same equivalence class if they are obtained from each other by successively exchanging adjacent transitions that are independent. is irrelevant, which can be expressed by the following two conditions:

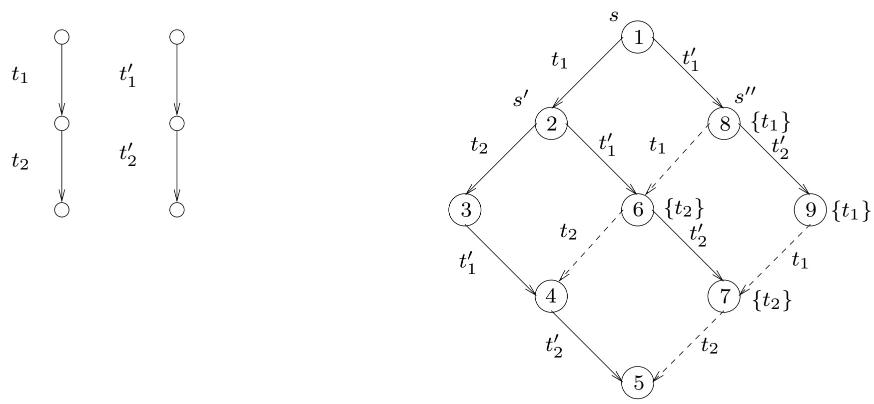
1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'.
2. Let t_1, t_2 be enabled in s, then there is a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$

The independence of transitions is decidable on purely syntactical level e.g. Petri net or PROMELA ([13]) syntax. A sleep set is a set of transitions, whose purpose is to avoid the exploration of a possibly huge number of permutations of independent transitions.
The concept of sleep sets will be clarified by the small example in figure 12.
We assume that t_1 and t_1' are independent transitions, both emanating from state s. The execution of t_1 leads to s'. When all (immediate) successors of s' are explored, the search algorithm goes back to state s. The execution of t_1' leads to s''. As t_1 and t_1' are independent, t_1 is still enabled in s''. However, it is not necessary to explore the sequence $t_1' t_1$, as the sequence $t_1 t_1'$ has already been investigated and leads to the same state as $t_1' t_1$, cf. definition of independence. Thus, to prevent the exploration t_1 is put into the sleep set (indicated by $\{t_1\}$ in figure 12) that is associated with state s''. Sleep sets do not affect the coverage of the state space search, i.e. all reachable states are visited. This is proven in theorem 1 in [5]. Also, the aim of avoiding to visit a particular state more than once is met, as proven in theorem 2 in [5].

The algorithm above shows, how sleep sets can be introduced in a depth first search algorithm.

In lines 9, 12 and 18 to 20 it is described how to compute sleep sets associated with the successor states of the current state and its sleep set.

– Line 9: All transitions from $s.Sleep$, pointing to a state on the stack are removed from $s.Sleep$.
– Line 12: All transitions of $s.Sleep$, which are independent of t, are added to the sleep set of the successor state s' after t.



(a) (b)

Fig. 12. Sleep sets: (a) two independent sequential processes, (b) their combined behaviour, with sleep sets associated with the states

```
Stack := ∅;
  s0.Sleep := ∅
  H[h(s0)] := s0 /*enter s0 in hash table*/
  push(s0, Stack)
  DFS();

DFS() {
  s := top(Stack)
  T := enabled(s)\s.Sleep
  s.Sleep := {t ∈ s.Sleep | succ(s, t) ∉ Stack}
  forall t ∈ T do
    s' := succ(s, t) /*execution of t */
    s'.Sleep := {t' ∈ s.Sleep | t' and t are independent }
    if s' ∉ H then
      H[h(s')] := s'
      push(s', Stack)
      DFS()
    endif
    if s' ∉ Stack then
      s.Sleep := s.Sleep ∪ {t}
    endif
  endfor
  pop(s, Stack)
}
```

Fig. 13. Depth-first search with sleep sets

– Lines 18 - 20: Each time a transition t from state s to s' is backtracked, t is added to $s.Sleep$, if s' is not in $Stack$.

10.2 Combination of state space caching and improved hash compaction

The replacement strategy of state space caching, replacing old states by new ones, when the hash table is full, is not very efficient, since a long probe sequence might be necessary before it can be decided whether a newly reached state is in the table. To overcome this inefficiency one now probes at most t slots before, if necessary at all, the new state is inserted into the table. If all probed t slots are non-empty and if all entries at these t slots are different from the state to be inserted then one value of any of the t slots is chosen (at random) to be replaced by the new value. Stern and Dill [28] call this approach t-limited scheme. It is assumed that uniform hashing for the state insertion and uniform compression is employed.
This combination of state space caching and improved hash compaction has the following consequences:

– Replacement now might also take place if the table is not full.
– State space caching itself performs an exhaustive search of the state space, the combination with the improved hash compaction method as presented in this section, turns state space caching into a partial search:
  Assume two states s and s', $s \neq s'$ are given, then it might be the case that the two states are hashed onto the same slot and also the compressed values for s and s' are equal, in this case it might be erroneously concluded that $s = s'$, such that an omission with probability greater zero occurs, which makes the combination of hash compaction and state space caching a probabilistic verification method.

Analysis: For the analysis, there is one important point to be noticed:
In case of state space caching it is only possible to give the number of different states that were inserted in the table, this number, n', is greater or equal to the real number of reachable states. This is obvious, since every reachable state may be reached by more than one path and is so visited at least once when state space caching is employed.
From 23 we have the approximation of the omission probability of state space caching. the derivation of this factors can be found in section 13

$$p_{om} \approx 1 - \left(\frac{l-1}{l}\right)^C \tag{27}$$

with

$l = 2^b$, the number of different compressed keys, each b bits long.

$$C = \begin{cases} \frac{t}{t+1} n' \left(\frac{n'}{m}\right)^t + \sum_{j=0}^{t-1} j \left(\frac{n'}{m}\right)^{j+1} \left(\frac{m}{j+1} - \frac{n'}{j+2}\right) & \text{if } n' \leq m \\ (H_{t+1} - 1)m + t(n' - m) & \text{if } n' > m. \end{cases}$$

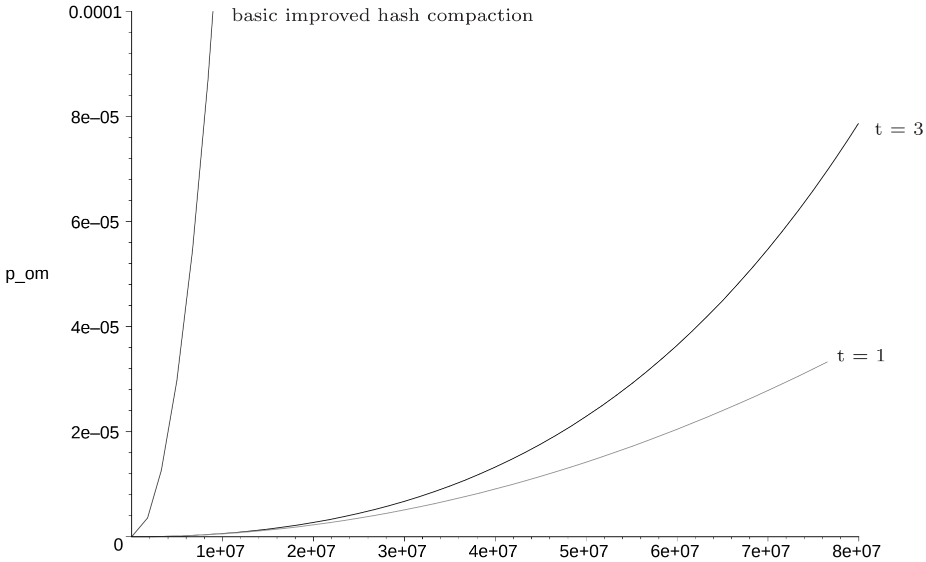


Fig. 14. Omission probabilities, using a 4 byte compression and $t = 1$ and $t = 3$

The improved hash compaction scheme experienced the problem that the omission probability increased sharply, when the hash table is nearly full, as it can be seen in figure 10. The combination with state space caching overcomes this problem. The effect of state space caching is shown pictorially. Again it is assumed that the state table is of size 400 MByte. The compressed values are of size 4 bytes (figure 14) resp. 5 bytes (figure 15). The t limited scheme is employed for $t = 1$ and $t = 3$.

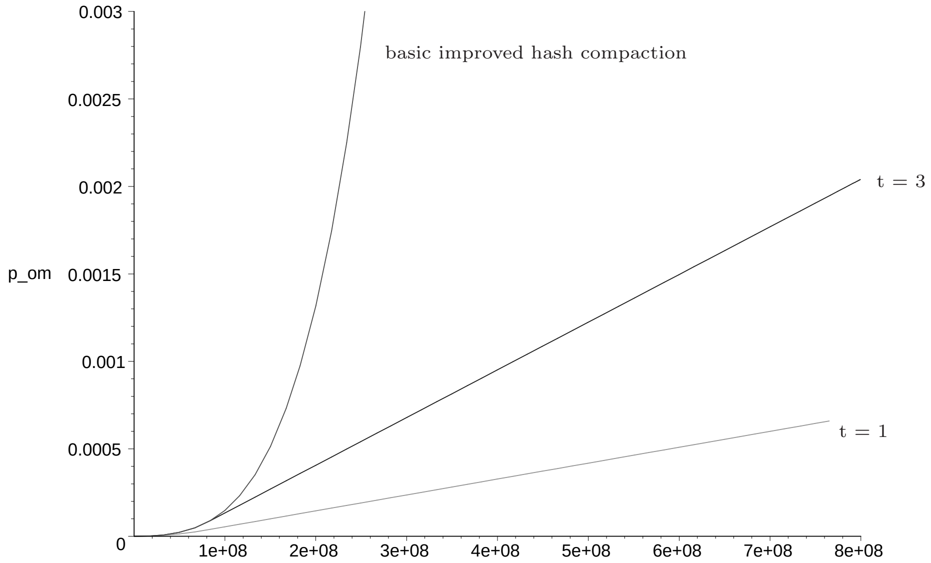


Fig. 15. Omission probabilities, using a 5 byte compression and $t = 1$ and $t = 3$

11 Refinement of Improved Hash Compaction

In this section the hash compaction method that was introduced in section 9 is further refined. The material presented in this section is based on [27] by Stern and Dill. The refinement consists of changing the following points in comparison with improved hash compaction from section 9: Firstly breadth first search is employed and secondly another type of hashing is applied. Both changes will be explained in detail in the next section. The goal of this refinement is to obtain a tighter bound for the probability of false positives. As a result of this new approach a further reduction of memory requirements is achieved in comparison with the improved hash compaction method of section 9, i.e. given a maximum omission probability, the same value can now be obtained by spending less bits for the compressed state descriptor than the improved hash compaction scheme.

11.1 Refined method

In the sequel we will describe the differences between this new approach and the improved hash compaction method from section 9.

- Here breadth first search (BFS) for the state space traversal is employed. The omission probability is computed over a longest BFS-path. Let d be the diameter of the reachability graph, i.e. the longest BFS-path from the initial state to any reachable state.
- In the refined method ordered hashing instead of uniform hashing for the state insertion algorithm is employed. Ordered hashing aims to reduce the average number of probes during an unsuccessful search (cf. section 2). Thus, the omission probability can be reduced, as it is dependent on the length of the probe sequence.

11.2 Analysis

Let d be the diameter of the reachability graph. Let s_d be the state on level d on a BFS-path. State s_d is omitted, if any of the states on the path $s_0, \ldots, s_d$ is omitted, thus the omission probability for s_d is bounded by the path's omission probability.
To derive an upper bound for s_d's omission probability resp. a lower bound for the probability that s_d is not omitted, we proceed as follows:
Let N_i be the event that s_i is not omitted during its search. State s_i can be omitted for two reasons:

1. A state being necessary for s_i's reachability has been omitted, i.e. s_i is not reached at all.
2. s_i can be reached but is omitted, due to a hash collision and the fact that the compressed values of s_i and the state that is already present are equal.

Thus, N_i can formally be defined as follows:

$$N'_0 \Leftrightarrow N_0$$
$$N'_i \wedge N_{i-1} \Rightarrow N_i \quad 0 < i \leq d,$$

where N'_i is the event that s_i is not omitted during its insertion. For the probability of the events N_i and N'_i this yields:

$$\begin{aligned} &Pr(N'_0) = Pr(N_0) \\ &Pr(N_i) \geq Pr(N'_i \wedge N_{i-1}) = Pr(N'_i \mid N_{i-1}) \cdot Pr(N_{i-1}) \end{aligned}$$

Applying this recursively yields:

$$Pr(N_d) \geq Pr(N'_0) \cdot \prod_{i=1}^{d} Pr(N'_i \mid N_{i-1}) \tag{28}$$

It is obvious that the probabilities on the right hand side of eq. 28 depend on the hashing method that is applied. $Pr(N_i' \mid N_{i-1})$ can be assessed for ordered hashing as follows:
Let p_k be the probability that, given k states already in the hash table, during insertion of a new state no omission occurs.

$$p_k = 1 - \frac{2}{l}(H_{m+1} - H_{m-k}) + \frac{2m + k(m-k)}{ml(m-k+1)} \tag{29}$$

A derivation of eq. 29 can be found in [25]
p_k can be used to approximate $Pr(N_i' \mid N_{i-1})$:
Let k_i be the total number of states in the hash table if breadth first search has completed level i. As k_i is taken over all paths of length at most i, k_i is at least as large as the number of states on a single path of length i, thus

$$Pr(N_0') \geq p_{k_0-1}$$
$$Pr(N_i' \mid N_{i-1}) \geq p_{k_i-1}$$

This approximation is justified by the observation that with increasing number of entries in a hash table also the omission probability increases. Using eq. 28 one obtains for $Pr(N_d)$:

$$Pr(N_d) \geq \prod_{i=0}^{d} p_{k_i-1} \tag{30}$$

Note carefully that the last level analysed is not necessarily the actual last level, as it might be the case that states on higher/deeper levels are all omitted. In this case the bound for the probability of no omissions is too optimistic.

12 Conclusion

The discussion so far has shown, that the historic development, as far as space complexity and SSp coverage are considered, has led to a step wise improvement of the probabilistic SSp exploration schemes. However concerning run-time, it is clear, that the original bitstate hashing scheme outperforms the latter developments. But its enormous memory requirements in case of high SSp coverage levels contradict its practical usefulness. Considering the developments from bitstate hashing to the approaches where chaining of slot overflows was employed, the dynamic hash compaction seems to be the most advantageous scheme. However if the latter is compared to the improved hash compaction approaches, which are based on probing for resolving collisions on slots, a different pictures has to be drawn, which will follow below.

The four inner columns of table 2 show the memory requirements of the various approaches, assuming a fixed number of 10^8 states and different levels of omission probability. In contrast to that, the leftmost column shows a varying number of states (n), where a fixed omission probability of $p_{om} \approx 10^{-3}$ and a fixed size of memory of 500 MByte was considered. The respective values were obtained by using the equation introduced in the analysis section of each respective approach. From the table one can obtain, that the improved hash compaction scheme is the most space efficient approach. In table 3 the so far presented schemes are listed

Scheme	$p_{om} \approx 10^{-9}$	$p_{om} \approx 10^{-6}$	$p_{om} \approx 10^{-4}$	$p_{om} \approx 10^{-2}$	unit	n
BSH	$624 \cdot 10^3$	625	6.25	$62.17 \cdot 10^{-3}$	EByte	2829
2-fold BSH	$14.5 \cdot 10^3$	144	14.44	1.4	TByte	$228,981$
30-fold BSH	1230	980	840	721	MByte	$1.676 \cdot 10^6$
HC	1087	912.5	825	737	MByte	$67,93 \cdot 10^6$ [a]
dynamic HC[b]	770	645	559	487	MByte	$96 \cdot 10^8$ [c]
improved HC	762.5	637.5	550	475	MByte	$97.5 \cdot 10^8$ [d]

[a] $b = 61$
[b] $\rho = 28$ bit
[c] $r = 2,272,599, d = 41$ bits
[d] $b = 41$ bits

Table 2. Comparison of probabilistic SSp exploration schemes

and assessed concerning run-time and space complexity. As shown there, the (extended)improved hash compaction approaches as well as the dynamic hash compaction scheme are the most appropriate for being employed in practice, where a slight advantage of the former methods concerning space complexity is given. However it should be noted that in case of the improved hash compaction the length of the probe sequence heavily influences the runtime behaviour of the approach. This dependency may turn out to be a clear disadvantage in comparison to the dynamic hash compaction scheme. Since in cases, where it comes to high fill-up rates of the hash table the runtime of the improved scheme may increase dramatically, where the run time of the dynamic hash compaction scheme is bounded by the maximum number of overflows per slot. Concerning the work of Stern and Dill [28, 26] in combination with breadth first search a comparison is problematic. On one hand the effectiveness of the t-limited scheme cannot be assessed correctly, since one does not have a bound on the number of states that are visited and inserted more than once. On the other hand a similar problem arises in the case of the improved hash compaction scheme. Its efficiency depends on the diameter of the reachability graph, which might be much smaller than the number of states, but cannot be assessed a priori.

Scheme	Runtime	Space Complexity[a]	Remark
BSH	++	− − −	very low mem. utilisation (μ_{mem})
2-fold BSH	+	− −	slight improved μ_{mem}
30-fold BSH	− −	−	requires h independent hash function, improved μ_{mem}
HC	+	+	no independent slot indexing
dynamic HC	+	++	search on arrays of overflows is not efficient
improved HC (IHC)	− −	++	probing sequences interfere with runtime
SSp Caching and IHC	−	+ +	reduced P_{om}^{max} compared to IHC
BFS and IHC	−	++	reduced memory req. for same P_{om}^{max} compared to IHC

[a] Refers to the theoretical memory requirement concerning high-levels of SSp coverage

Table 3. Ranking of probabilistic SSp exploration schemes

13 Analysis Revisited

13.1 Approximate analysis of improved hash compaction

In the sequel a very thorough derivation of the probabilities that were presented in section 9.2 will be given. We will give an approximation of the maximum omission probability of a particular state. This approximation constitutes also an estimation of the probability that the verifier produces false positives. We assume that a uniform compression and hashing scheme is applied. For the analysis the reader is reminded of the following notation:

- $n^{\underline{k}} := \prod_{i=0}^{k-1}(n-i)$: falling factorial powers
- m: number of slots in the hash table
- l: number of different hash values
- n: number of (different) states to be inserted in the hash table

We make the following assumptions: The hash table accommodates already k states. On insertion of these k states no omissions occurred.
We want to insert the $(k+1)$st state in the hash table. Let C_k be the random variable that describes how many collisions occur, if the $(k+1$st state is to be inserted into a table that already contains k states. The value of C_k is exactly one less than the number of probes that are needed on the insertion of the $(k+1)$st state. Equation 31 gives the probability that on insertion of the $(k+1)$st state, j collisions occur. We use equation 1 for the derivation:

$$
\begin{aligned}
Pr(C_k = j) &= \frac{\binom{m-(j+1)}{k-(j+1)+1}}{\binom{m}{k}} = \frac{\binom{m-j-1}{k-j}}{\binom{m}{k}} \\
&= \ldots = \frac{(m-j-1)! \cdot (m-k)! \cdot k!}{(m-k-1)! \cdot (k-j)! \cdot m!} = (m-k) \cdot \frac{(m-j-1)! \cdot k!}{(k-j)! \cdot m!} \\
&= (m-k) \cdot \frac{k \cdot (k-1) \cdot (k-2) \cdot \ldots \cdot (k-j-1)}{m * (m-1) \cdot (m-2) \cdot \ldots \cdot (m-j)} =
\end{aligned}
$$

$$
= (m-k) \cdot \frac{k^{\underline{j}}}{m^{\underline{j+1}}} \tag{31}
$$

Note that equation 1 is valid for $i-1$ collisions.
Let N_k be the event that on insertion of the $(k+1)$st state no omissions occurred given that during the first k insertions no omissions occurred. Let N'_k be the corresponding unconditional event, i.e. the event that on insertion of the $(k+1)$st state no omissions occurred. Obviously: $N_k = N'_k \mid (N'_{k-1} \wedge \ldots \wedge N'_0)$.
The probability of N_k given there are j collisions can be approximated as follows:

$$
Pr(N_k \mid C_k = j) \approx \left(\frac{l-1}{l}\right)^j, \tag{32}
$$

where $(l-1)/l$ is obtained from the probability that the hash function maps two keys on the same address is equal $1/l$, so $(l-1)/l$ is its complementary probability that two different keys are mapped on different addresses. Equation 32 is justified from the observation that each collision might lead to an omission.
Using the theorem of total probability [10] the probability of the event N_k is derived as follows:

$$
p_k = Pr(N_k) = \sum_{j=0}^{k} Pr(N_k \mid C_k = j) \cdot Pr(C_k = j)
$$

$$
\overset{equ.\ 32}{\approx} \sum_{j=0}^{k} \left(\left(\frac{l-1}{l}\right)^j \cdot \frac{k^{\underline{j}}}{m^{\underline{j+1}}} \cdot (m-k) \right) \tag{33}
$$

Note carefully, that omissions lead to an altered distribution of C_k. But, if we slightly adopt the hashing algorithm in the following sense, then we can use the unaltered distribution for C_k:
The algorithm probes until an empty slot is found, insertion only, if none of the probed slot contains a value that is equal to the one that is to be inserted. The

[10] The theorem of total probability: $Pr(A) = \sum_{i=1}^{\infty} Pr(A \mid B_i) \cdot Pr(B_i)$

probability of no omissions, when inserting n states into an initially empty hash table can be computed as follows:

$$Pr(\text{no omissions inserting } n \text{ states}) = Pr(N'_{n-1} \wedge \ldots \wedge N'_0) \overset{P(A \wedge B) = P(A|B) \cdot P(B)}{=}$$
$$Pr(N'_{n-1} | (N'_{n-2} \wedge \ldots \wedge N'_0)) \cdot Pr(N'_{n-2} \wedge \ldots \wedge N'_0)$$

This probability will be referred to as p_{no}.
Applying the scheme given above recursively we obtain for p_{no} the following equation:

$$p_{no} = \prod_{k=0}^{n-1} p_k \tag{34}$$

Substituting equation 33 into equation 34 yields the following approximation for the probability that there are no omissions during the insertion of n states into a hash table:

$$p_{no} \approx \prod_{k=0}^{n-1} \left(\sum_{j=0}^{k} \left(\left(\frac{l-1}{l} \right)^j \cdot \frac{m-k}{m-j} \cdot \prod_{i=0}^{j-1} \frac{k-i}{m-i} \right) \right) \tag{35}$$

Above, the definition for falling factorial powers was substituted, to make clear, that this equation does not constitute efficient means to compute p_{no}, as computing p_{no} here requires $O(n^3)$ operations.
In the sequel it is shown, how p_{no} can be computed requiring only $O(1)$ operations. For that purpose equation 34 can be rewritten as:

$$p_k = \sum_{j=0}^{k} \left(\frac{l-1}{l} \right)^j \cdot Pr(C_k = j) \tag{36}$$

Interpreting equation 36 as a generation function, it can be rewritten as follows:

$$\begin{aligned} p_k &\overset{(l-1)/l=z}{=} G_{C_k}(z) = \sum_{j=0}^{k} z^j \cdot P_k(C_k = j) \\ &= \quad E\left(\left(\frac{l-1}{l} \right)^{C_k} \right) \end{aligned}$$

Using Jensen's inequality

$$E\left(\left(\frac{l-1}{l} \right)^{C_k} \right) \geq \left(\frac{l-1}{l} \right)^{E(C_k)}$$

we obtain:

$$p_k \approx \left(\frac{l-1}{l} \right)^{E(C_k)} \tag{37}$$

Substituting equation 37 into equation 34 yields:

$$p_{no} \approx \prod_{k=0}^{n-1} \left(\frac{l-1}{l}\right)^{E(C_k)} = \left(\frac{l-1}{l}\right)^{\sum_{k=0}^{n-1} E(C_k)} \tag{38}$$

A closed form for $E(C_k) = \sum_{j=1}^{k} j \cdot Pr(C_k = j)$ can be derived as follows:

$$\begin{aligned}
E(C_k) &= \sum_{j=1}^{k} j \cdot Pr(C_k = j) = \sum_{j=1}^{k} j \cdot \left(\frac{\binom{m-j-1}{k-j}}{\binom{m}{k}}\right) \\
&\overset{\binom{a}{b}=\binom{a}{a-b}}{=} \sum_{j=1}^{k} j \cdot \left(\frac{\binom{m-j-1}{m-k-1}}{\binom{m}{k}}\right) \\
&\overset{cf.[7]p.175}{=} \frac{k}{m-k+1} \cdot \frac{\binom{m}{m-k}}{\binom{m}{k}} \\
&= \frac{k}{m-k+1} \cdot \frac{m! \cdot (m-k)! \cdot k!}{m! \cdot (m-k)! \cdot k!} \\
&= \frac{k}{m-k+1}
\end{aligned} \tag{39}$$

Equation 39 can be used to obtain a closed form for the sum in the exponent of equation 38:

$$\begin{aligned}
\sum_{k=0}^{n-1} E(C_k) &= \sum_{k=0}^{n-1} \frac{k}{m-k+1} = \sum_{k=0}^{n-1} \frac{k-(m+1)+(m+1)}{m-k+1} \\
&= \sum_{k=0}^{n-1} \left(\frac{k-(m+1)}{m-k+1} + \frac{(m+1)}{m-k+1}\right) = \sum_{k=0}^{n-1} \left(\frac{k-(m+1)}{m-k+1}\right) + \sum_{k=0}^{n-1} \left(\frac{(m+1)}{m-k+1}\right) \\
&= \sum_{k=0}^{n-1} \left(\frac{-[-k+(m+1)]}{m-k+1}\right) + (m+1) \cdot \sum_{k=0}^{n-1} \left(\frac{1}{m-k+1}\right) \\
&= \sum_{k=0}^{n-1} -1 + (m+1) \cdot \sum_{k=0}^{n-1} \left(\frac{1}{m-k+1}\right) \\
&= -n + (m+1) \cdot \sum_{k=0}^{n-1} \left(\frac{1}{m-k+1}\right) \\
&= -n + (m+1) \cdot \left(\frac{1}{m+1} + \ldots + \frac{1}{m-n+2}\right)
\end{aligned}$$

$$= (m+1) \cdot (H_{m+1} - H_{m-n+1}) - n \tag{40}$$

H_n can be approximated by

$$\ln(n) + \gamma + 1/2n - 1/12n^2 \tag{41}$$

Equation 40 combined with 38 yield:

$$p_{no} \approx \left(\frac{l-1}{l}\right)^{\sum_{k=0}^{n-1} E(C_k)} = \left(\frac{l-1}{l}\right)^{(m+1)*(H_{m+1}-H_{m-n+1})-n} \tag{42}$$

Equation 41 could be used to approximate the exponent in the formula above, we will skip this substitution.
Finally, we are ready to give the omission probability p_{om}:

$$p_{om} := Pr(\text{at least one omission during the insertion of } n \text{ states}) = 1 - p_{no}$$

Upper bounds for the omission probability: We want to give an upper bound, $\hat{p}_{om}$ for the omission probability. Equation 32 will be replaced by

$$Pr(N_k \mid C_k = j) \geq \frac{l-j}{l}$$

The formula given above is valid under the following assumption:
All j values to which the state to be inserted is compared to, must be different. There are still $l-j$ compression values left, to which the new state can be mapped. Each collision can lead to an omission. So, the probability, that the state that is to be inserted is mapped onto one of those values is equal to $(l-j)/l$. So the probability of no omission, when insertion $(k+1)$st given j collisions is as given above.
To obtain $\hat{p}_{om}$ the following intermediate result is needed:

$$\begin{aligned}
p_k := Pr(N_k) &\overset{\text{total probability}}{=} \sum_{j=0}^{k} Pr(N_k \mid C_k = j) \cdot Pr(C_k = j) \\
&\geq \sum_{j=0}^{k} \frac{l-j}{l} \cdot Pr(C_k = j) \\
&= \sum_{j=0}^{k} \frac{(l-j) \cdot Pr(C_k = j)}{l} = \sum_{j=0}^{k} \frac{l \cdot Pr(C_k = j)}{l} - \sum_{j=0}^{k} \frac{j \cdot Pr(C_k = j)}{l} \\
&= \sum_{j=0}^{k} Pr(C_k = j) - \frac{1}{l} \sum_{j=0}^{k} j \cdot Pr(C_k = j) \\
&= 1 - \frac{1}{l} E(C_k)
\end{aligned}$$

Using equation 34, we deduce from above:

$$p_{no} = \prod_{k=0}^{n-1} p_k \geq \prod_{k=0}^{n-1} \left(1 - \frac{1}{l} E(C_k)\right)$$

$$= 1 - \frac{1}{l} \sum_{k=0}^{n-1} E(C_k) + \left(\frac{1}{l}\right)^2 \sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} E(C_k) \cdot E(C_j)$$

$$- \left(\frac{1}{l}\right)^3 \sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} \sum_{i=j+1}^{n-1} E(C_k) \cdot E(C_j) \cdot E(C_i) + \ldots \quad (43)$$

Equation 43 is clarified by means of an example. Let $n = 3$:

$$\begin{aligned}
p_{no} &\geq (1 - 1/l \cdot E(C_0)) \cdot (1 - 1/l \cdot E(C_1)) \cdot (1 - 1/l \cdot E(C_2)) \\
&= 1 - 1/l \cdot (E(C_0) + E(C_1) + E(C_2)) \\
&+ (1/l)^2 \cdot (E(C_0)E(C_1) + E(C_0)E(C_2) + E(C_1)E(C_2)) \\
&- (1/l)^3 \cdot (E(C_0)E(C_1)E(C_2) \\
&= 1 - 1/l \cdot \sum_{k=0}^{2} E(C_k) \\
&+ (1/l)^2 \sum_{k=0}^{2} \sum_{j=k+1}^{2} E(C_k)E(C_j) \\
&- (1/l)^3 \sum_{k=0}^{2} \sum_{j=k+1}^{2} \sum_{i=j+1}^{2} E(C_k)E(C_j)E(C_i)
\end{aligned}$$

The triple sum in 43 can be rewritten as:

$$\sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} \sum_{i=j+1}^{n-1} E(C_k) \cdot E(C_j) \cdot E(C_i) = \sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} E(C_k) \cdot E(C_j) \sum_{i=j+1}^{n-1} E(C_i)$$

That means that this sum (= fourth term of 43) has in comparison to the third term of 43 the non-constant factor $1/l \cdot \sum_{i=j+1}^{n-1}$ in addition.
Using the assumption that $l \geq \sum_{i=0}^{n-1} E(C_i)$. where l is the number of different compressed values, we can give an upper bound for this factor as follows:

$$\sum_{i=j+1}^{n-1} E(C_i) < \sum_{i=0}^{n-1} E(C_i)$$

as we add only non-negative values this is clearly valid, then using the assumption from above, we can deduce:

$$(\sum_{i=j+1}^{n-1} E(C_i) < \sum_{i=0}^{n-1} E(C_i) \leq l) \cdot \frac{1}{l}$$

$$\frac{1}{l}\sum_{i=j+1}^{n-1} E(C_i) < \frac{1}{l}\sum_{i=0}^{n-1} E(C_i) \leq 1,$$

Using the fact that $\frac{1}{l}\sum_{i=j+1}^{n-1} E(C_i) < 1$, makes clear that the fourth term in equation 43 is less than the third one, thus their difference is greater than zero. From that we conclude:

$$p_{no} \geq 1 - \frac{1}{l}\sum_{k=0}^{n-1} E(C_k) \tag{44}$$

This is correct because, the omission of a factor greater than zero, that is added to the first term in the equation above doesn't change the validity of the equation 44. If this is applied iteratively to all higher numbered terms, we finally obtain the upper bound of the omission probability p_{om}:

$$\hat{p}_{om} \leq p_{om} = \frac{1}{l} \cdot (m+1) \cdot ((H_{m+1} - H_{m-n+1}) - n)$$

13.2 Analysis of the t-limited scheme

Let C_k as defined in the section of the approximate analysis of the hash compaction scheme. If $t \to \infty$ we have for the probability of C_k:

$$Pr(C_k \geq j) = \frac{k^{\underline{j}}}{m^{\underline{j}}}$$

The equation above can be derived as follows:

$$\begin{aligned} Pr(C_k \geq j) &= 1 - Pr(C_k < j) = 1 - \sum_{i=0}^{j-1} \frac{\binom{m-(i+1)}{k-i}}{\binom{m}{k}} \\ &= \ldots = \frac{k^{\underline{j}}}{m^{\underline{j}}} \end{aligned}$$

The probability of exactly j collisions in the t-limited scheme distinguishes three cases:

$$Pr(C_k = j) = \begin{cases} \frac{k^{\underline{j}}}{m^{\underline{j+1}}} * (m-k), & 0 \leq j < t \\ \frac{k^{\underline{j}}}{m^{\underline{j}}}, & j = t \\ 0, & j > t \end{cases} \tag{45}$$

The estimation of the second probability from above stems from $Pr(C_k \geq j)$, as the limitation of the number of probes means that even more collisions might occur, so here is only a lower bound to be given.

To compute p_{no} we use the same approximation as in equation 38. Let C be the expected value of the number of collisions, if n states ought to be inserted into the table, then p_{no} is given as follows:

$$p_{no} = \left(\frac{l-1}{l}\right)^C = \left(1 - \frac{1}{l}\right)^C$$

Using equation 45 we obtain for C:

$$\begin{aligned} C &= \sum_{k=0}^{n-1} E(C_k) = \sum_{k=0}^{n-1}\sum_{j=0}^{t} j \cdot Pr(C_k = j) \\ &= \sum_{k=0}^{n-1}\left(\sum_{j=0}^{t-1} j \cdot \frac{k^{\underline{j}}}{m^{\underline{j+1}}} \cdot (m-k) + t \cdot \frac{k^{\underline{t}}}{m^{\underline{t}}}\right) \end{aligned}$$

To derive a closed form for the sum from above, we will change at first the order of summation:

$$\begin{aligned} C &= \left(\sum_{j=0}^{t-1}\sum_{k=0}^{n-1} j \cdot \frac{k^{\underline{j}}}{m^{\underline{j+1}}} \cdot (m-k)\right) + \sum_{k=0}^{n-1} t \cdot \frac{k^{\underline{t}}}{m^{\underline{t}}} \\ &\stackrel{*}{=} \left(\sum_{j=0}^{t-1}\left(\sum_{k=0}^{n-1} j \cdot \frac{k^{\underline{j}}}{m^{\underline{j+1}}} \cdot (m-k)\right)\right) + \frac{t}{m^{\underline{t}}} \cdot \frac{n^{\underline{t+1}}}{t+1} \end{aligned}$$

The last step, referred to as * can be justified as follows (cf: [7], equation 2.50):

$$\sum_{k=0}^{n-1} k^{\underline{m}} = \frac{n^{\underline{m+1}}}{m+1}$$

Continuing, using Maple, yields the following intermediate result for C:

$$C = \sum_{j=0}^{t-1}\sum_{k=0}^{n-1} j * \frac{k^{\underline{j}}}{m^{\underline{j+1}}} \cdot (m-k) + \frac{t}{m^{\underline{t}}} \cdot \frac{n^{\underline{t+1}}}{t+1} = \tag{46}$$

$$\sum_{j=0}^{t-1} j \cdot \frac{n^{\underline{j+1}}}{m^{\underline{j+1}}} \cdot \left(\frac{m+1}{n+1} - \frac{n+1}{j+2}\right) + \frac{t}{t+1} \cdot \frac{n^{\underline{t+1}}}{m^{\underline{t}}} \tag{47}$$

At first, we will determine C for $n = m$, which leads to some simplifications of equation 47:

$$\sum_{j=0}^{t-1} j \cdot \left(\frac{m+1}{j+1} - \frac{m+1}{j+2}\right) + \frac{t}{t+1} \cdot (m-t)$$

The sum from above will be solved as follows:

$$\sum_{j=0}^{t-1} j \cdot \left(\frac{m+1}{j+1} - \frac{m+1}{j+2}\right) = \sum_{j=0}^{t-1} j \cdot (m+1) \cdot \left(\frac{1}{j+1} - \frac{1}{j+2}\right)$$

$$
\begin{aligned}
&= (m+1)\cdot\sum_{j=0}^{t-1} j\cdot\left(\frac{1}{j+1}-\frac{1}{j+2}\right)\\
&= (m+1)\cdot\left(\sum_{j=0}^{t-1}\frac{j}{j+1}-\sum_{j=0}^{t-1}\frac{j}{j+2}\right)\\
&= (m+1)\cdot\left(\sum_{j=0}^{t-1}\frac{j+1-1}{j+1}-\sum_{j=0}^{t-1}\frac{j+2-2}{j+2}\right)\\
&= (m+1)\cdot\left(\sum_{j=0}^{t-1}1-\sum_{j=0}^{t-1}\frac{1}{j+1}-\left(\sum_{j=0}^{t-1}1-\sum_{j=0}^{t-1}\frac{2}{j+2}\right)\right)\\
&= (m+1)\cdot\left(-\sum_{j=0}^{t-1}\frac{1}{j+1}+t-t+\sum_{j=0}^{t-1}\frac{2}{j+2}\right)\\
&= (m+1)\cdot\left(-\sum_{j=0}^{t-1}\frac{1}{j+1}+2\cdot\sum_{j=0}^{t-1}\frac{1}{j+2}\right)\\
&= (m+1)\cdot\left(2\cdot\sum_{j=0}^{t-1}\frac{1}{j+2}-\sum_{j'=1}^{t}\frac{1}{j'}\right)\\
&= (m+1)\cdot\left(2\cdot\sum_{j=0}^{t-1}\frac{1}{j+2}-H_t\right)\\
&= (m+1)\cdot\left(2\cdot\sum_{j'=1}^{t}\frac{1}{j'}-2-H_t\right)\\
&= (m+1)\cdot(2\cdot H_{t+1}-2-H_t)\\
&= (m+1)\cdot(H_{t+1}+(1+1/2+\ldots+1/t+1/(t+1))-\\
&\quad (1+1/2+\ldots+1/t)-2)\\
&= (m+1)\cdot(H_{t+1}+1/(t+1)-2)
\end{aligned}
$$

Finally,

$$
C=(m+1)\cdot\left(H_{t+1}+\frac{1}{t+1}-2\right)+\frac{t}{t+1}\cdot(m-t) \tag{48}
$$

If the number of states n that are to be inserted is bigger than the number of slots m in the hash table, then each new entry after the first m entries leads to exactly t collisions, so equation 48 becomes:

$$
C=(m+1)\cdot\left(H_{t+1}+\frac{1}{t+1}-2\right)+\frac{t}{t+1}\cdot(m-t)+t\cdot(m-n) \tag{49}
$$

Under the assumption $n, m \gg t^2, m \approx m+1$ equation 49 can be approximated as follows:

$$\begin{aligned}
& m \cdot (H_{t+1} + 1/(t+1) - 2) + \frac{m \cdot t - t^2}{t+1} + t \cdot (n-m) \\
\approx\, & m \cdot H_{t+1} + \frac{m}{t+1} - 2 \cdot m + \frac{m \cdot t}{t+1} + t \cdot (n-m) \\
=\, & m \cdot H_{t+1} + \frac{m + m \cdot t}{t+1} - 2 \cdot mt \cdot (n-m) \\
=\, & m \cdot H_{t+1} + \frac{m \cdot (1+t}{t+1} - 2 \cdot mt \cdot (n-m) \\
=\, & m \cdot (H_{t+1} - 1) + t \cdot (n-m)
\end{aligned}$$

For $n \leq m$, with some further simplifications, we obtain:

$$C = \frac{t}{t+1} \cdot n \cdot \left(\frac{n}{m}\right)^t + \sum_{j=0}^{t-1} j \cdot \left(\frac{n}{m}\right)^{j+1} \cdot \left(\frac{m}{j+1} - \frac{n}{j+2}\right) \tag{50}$$

The simplifications, made in 50 yield a more pessimistic approximation, because of the use of powers.
The omission probability is computed as follows:

$$p_{om} := 1 - p_{no} = 1 - \left(\frac{l-1}{l}\right)^C \tag{51}$$

References

1. P. Buchholz. Structured Analysis Approaches for Large Markov Chains. *Applied Numerical Mathematics*, 31:375–404, 1999.
2. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of 2nd International Workshop on Computer Aided Verification (CAV'90)*, volume 531 of *LNCS*, pages 176 – 185. Springer, 1990.
3. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem.* PhD thesis, Université de Liege, 1995.
4. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.
5. P. Godefroid, G. Holzmann, and D. Pirottin. State Space Caching Revisited. In *Proceedings of CAV'92*, volume 663, pages 178–191. Springer, LNCS, 1992.
6. P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design*, 2(2):149 – 164, April 1993.
7. R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2 edition, 1994.
8. B.R. Haverkort, A. Bell, and H. Bohnenkamp. On the Efficient Sequential and Distributed Generation of very Large Markov Chains from Stochastic Petri Nets. In *Proc. of IEEE Petri Nets and Performance Models*, pages 12–21, 1999.
9. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In Plateau et al. [23], pages 188–207. Proc. of the 3rd Int. Workshop on the Numerical Solution of Markov Chains (NSMC'99).
10. G. Holzmann. On Limits and Possibilities of Automated Protocol Analysis. In *Proc. 7th IFIP WG6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, pages 339–344. North-Holland Publishers, 1987.
11. G. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software, Practice and Experience*, 18(2):137–161, 1988.
12. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
13. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, 1997.
14. G. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.
15. W. Knottenbelt. Generalized Markovian Analysis of Timed Transition Systems, 1996. Master's Thesis, University of Cape Town.
16. W. Knottenbelt and P. Harrison. Distributed Disk-based Solution Techniques for Large Markov Models. In Plateau et al. [23], pages 58–75. Proc. of the 3rd Int. Workshop on the Numerical Solution of Markov Chains (NSMC'99).
17. W. Knottenbelt, P. Harrison, M. Mestern, and P. Kritzinger. Probability, Parallelism and the State Space Exploration Problem. In R. Puigjaner, N. Savino, and B. Serra, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 1469 of *LNCS*, pages 165–179. Springer, 1998. Proc. of the 10th International Conference, Tools '98, Palma de Mallorca, Spain, September 14-18, 1998.

18. W. Knottenbelt, P. Harrison, M. Mestern, and P. Kritzinger. A Probabilistic Dynamic Technique for the Distributed Generation of Very Large State Spaces. *Performance Evaluation Journal*, 39(1-4):127–148, February 2000.
19. W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models.* PhD thesis, University of London, Imperial College, Dept. of Computing, 1999.
20. D. Knuth. *The Art of Computer Programming: Vol 3: Sorting and Searching.* Addison-Wesley, 2 edition, 1998.
21. A. Mazurkiewicz. Trace Theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *LNCS*, pages 279 – 324. Springer, 1986. Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986.
22. Th. Ottman and P. Widmayer. *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag, 3. revised edition, 1996.
23. B. Plateau, W.J. Stewart, and M. Silva, editors. *Numerical Solution of Markov Chains.* Prensas Universitarias de Zaragoza, 1999. Proc. of the 3rd Int. Workshop on the Numerical Solution of Markov Chains (NSMC'99).
24. M. Siegle. Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties. Habilitationsschrift, Institut für Informatik, Friederich-Alexander-Universität Erlangen-Nürnberg, 2002.
25. U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Technische Universität München, 1997.
26. U. Stern and D. L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Proceedings of CHARME '95*, volume 987, pages 206–224. Springer, LNCS, 1995.
27. U. Stern and D. L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, application and tools*, volume 96 of *IFIP Conference Proceedings*, pages 333–348. Kluwer, 1996. Proc. of IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI, Kaiserslautern Germany, October 8-11.
28. U. Stern and D. L. Dill. Combining State Space Caching and Hash Compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90, 1996.
29. P. Wolper and D. Leroy. Reliable Hashing Without Collision Detection. In *Proceedings of the 5th International Computer Aided editor = Costas Courcoubetis, Verication Conference (CAV'93)*, volume 697 of *LNCS*, pages 59–70. Springer, 1993.
30. P. Wolper, U. Stern, D. Leroy, and D. L. Dill. Reliable probabilistic verification using hash compaction. Submitted for publication, downloaded from http://www.citeseer.nj.nec.com/32915.html on 23.08.2002.