Improved Probabilistic Verification by Hash Compaction

Ulrich Stern* and David L. Dill

Department of Computer Science, Stanford University,
Stanford, CA 94305

{uli@rutabaga, dill@cs}.stanford.edu

Abstract. We present and analyze a probabilistic method for verification by explicit state enumeration, which improves on the "hashcompact" method of Wolper and Leroy. The hashcompact method maintains a hash table in which compressed values for states instead of full state descriptors are stored. This method saves space but allows a non-zero probability of omitting states during verification, which may cause verification to miss design errors (i.e. verification may produce "false positives").
Our method improves on Wolper and Leroy's by calculating the hash and compressed values independently, and by using a specific hashing scheme that requires a low number of probes in the hash table. The result is a large reduction in the probability of omitting a state. Hence, we can achieve a given upper bound on the probability of omitting a state using fewer bits per compressed state. For example, we can reduce the number of bytes stored for each state from the eight recommended by Wolper and Leroy to only *five*, and still enumerate state spaces of up to 80 million reachable states while keeping the probability of missing even one state to less than 0.13%.
The new verification scheme was tried on some large, industrial examples. The results predicted by the theoretical analysis were confirmed by the outcomes of these examples. We also discuss some practical issues in choosing the number of bits for the compressed state representation, along with some of our experiences in implementing the scheme.

1 Introduction

Complex protocols are often verified by examining all reachable protocol states from a set of possible start states. This *reachability analysis* can be done using two different methods: The states can be *explicitly enumerated,* by storing them individually in a table, or a *symbolic* method can be used, such as representing the reachable state space with a binary decision diagram (BDD) [2].

The biggest obstacle of both methods is the often unmanageably huge number of reachable states – the "state explosion problem". Symbolic methods can alleviate the state explosion problem in some cases. However, in research done

* Ulrich Stern was supported during this research by a scholarship from the German Academic Exchange Service (DAAD-Doktorandenstipendium HSP-II).

in our group for some types of industrial protocols, explicit state enumeration has out-performed the symbolic approach [10].

In explicit state enumeration algorithms, a state table is maintained that will eventually hold all reachable states of the protocol under verification. This state table is usually implemented as a hash table. In practice, the total memory available for the hash table is the limiting resource in verification. Therefore, it is desirable to use the most compact representation for this table. The probabilistic verification method presented and analyzed in this paper improves on Wolper and Leroy's "hashcompact" method [16], which reduces the memory requirements for the state table. The hashcompact method was, in turn, inspired by Holzmann's supertrace algorithm [9].

The basic supertrace algorithm maintains a huge table of bits which are initially set to zero. When a state is inserted into the table, the bit corresponding to the hash value of the state is set to one. Whenever two states hash to the same location, they are assumed to be the same. For a large number of states, it is almost certain that two distinct states will hash to the same location. This results in the *omission* of the second state and – since the search algorithm backtracks when it finds a state that has been searched already – may potentially cause the omission of all successor states of the omitted state. Therefore, although supertrace intuitively seems to explore a high percentage of the reachable states of a protocol if the number of bits in the table is bigger than the number of reachable states, full coverage is extremely unlikely. Actually, Holzmann emphasizes that supertrace is only meant as a partial-search technique for protocols that cannot be analyzed exhaustively [9, page 232].

Wolper and Leroy observed that if a compressed state descriptor with, say 64 bits, instead of one bit is stored for each state (plus the information regarding which slots in the hash table are occupied), the probability of *omitting even one state* becomes very small, thus giving a bound on the probability of a false positive. The space savings of such a scheme are not as impressive as the supertrace algorithm, but still very substantial: the state descriptors for a typical protocol might be 100 bytes, which the hashcompact method could reduce to eight bytes with a very small probability of omitting states in the verification.

Wolper and Leroy implicitly assumed that the hash values for storing the compressed state descriptors are calculated using this compressed state descriptor. We show that, by deriving the hash values *independently* from the compressed state descriptor and by employing a particular hashing scheme with a low number of probes in the table, the omission probability can be reduced significantly. Our new analysis of the omission probability shows that one only needs to store *five* bytes for each state in situations where Wolper and Leroy recommend storing eight bytes.

Using this 5-byte compression, one can, for example, store 80 million states with an omission probability smaller than 0.13%. Furthermore, by re-running the verifier with different hash and compression functions, each yielding independent values from the ones of the previously employed function, one can further reduce this probability. For example, two verification runs for the above example would

yield an omission probability smaller than $(0.13\%)^2 = 1.69 \cdot 10^{-6}$.

We have extended the Murφ verification system developed at Stanford with the probabilistic verification scheme. In prior work, the "old" Murφ system was successfully applied to several industrial protocols [5, 6, 12, 14, 17]. We tried the new, probabilistic scheme on some of these protocols. No omissions occurred using 5-byte compression. Then, we looked at one of these protocols and varied the number of bits in the compressed state descriptors. For each value, we conducted multiple runs of the verifier with randomly chosen hash and compression functions. The outcomes of these experiments corresponded with the omission probabilities from the theoretical analysis. Furthermore, the mean number of omitted states at, for example, 55% omission probability was only 3.2, out of 109 080 reachable states. Thus, coverage is expected to degrade very slowly.

The paper is organized as follows. Section 2 presents the probabilistic verification scheme in more detail. The omission probabilities obtained using this scheme are calculated in Sect. 3. In Sect. 4, the omission probabilities for 4-byte and 5-byte compression are graphed and the choice of the number of bits in the compressed values is discussed. Some experience gained in implementing the probabilistic verification scheme is elucidated in Sect. 5. We describe the results on larger example protocols in Sect. 6. Finally, Sect. 7 contains some conclusions and suggestions for future research.

2 The Probabilistic Verification Scheme

The new, probabilistic verification method is described in this section. However, first the basic algorithms for explicit state enumeration on which the new method is based and the hashing and compression schemes employed are explained in more detail. In these explanations, some definitions are made that will be used in calculating the omission probability.

Explicit State Enumeration

In explicit state enumeration, the automatic verifier tries to examine all reachable states from a set of possible start states. Either breadth-first or depth-first search can be employed for the state enumeration process. Both the breadth-first and the depth-first algorithms are straightforward. A depth-first algorithm is, for example, given in [16].

Two data structures are needed for performing the state enumeration. First, a *state table* stores all the states that have been examined so far and is used to decide whether a newly-reached state is old (has been visited before) or new (has not been visited before). Besides the state table, a *state queue* holds all active states (states that still need to be explored). Depending on the organization of this queue, the verifier does a breadth-first or a depth-first search.

The state table will eventually hold all reachable states, unless the number of states exceeds the capacity of the table, in which case the verifier halts with an error message. In practice, the total memory available for the table is the limiting

resource in verification. The probabilistic verification scheme greatly reduces the memory requirements for this table by compressing the states before they are inserted, thus increasing the size of verification problems that can be dealt with.

Hashing

In the probabilistic verification scheme, the state table is implemented as a hash table. In the following, we assume that this hash table has m slots. Thus, a maximum number of m states can be explored. One problem in hashing are collisions. A collision occurs if two states hash to the same slot in the table. This collision can be resolved by either chaining or open addressing [4]. Chaining requires storing an additional pointer besides the compressed state and with that the memory requirements increase. Therefore, open addressing was used in our probabilistic verification scheme.

In open addressing, a vectorial hash function $\underline{h}$ is applied to each state s yielding a probe sequence $h_0(s), h_1(s), \ldots, h_{m-1}(s)$. When inserting a state in the table, the slots are tested for emptiness according to this probe sequence. The state is stored in the first empty slot found during the probe sequence. Note, that each probe sequence has to be a permutation of $\{0, \ldots, m-1\}$ if we want every slot in the table to be used.

In *uniform hashing*, each state is equally likely to have any of the $m!$ permutations of $\{0, \ldots, m-1\}$ as its probe sequence. Uniform hashing has the advantage of being relatively easy to analyze and of avoiding clustering in the hash table. However, the calculation of the probe sequence is difficult in practice. Therefore, we used *double hashing* instead, where only two hash values $h'(s)$ and $h''(s)$ have to be calculated. The probe sequence is then given by $h_i(s) = [h'(s)+ih''(s)] \mathrm{mod} m$, $i = 0, \ldots, m-1$. The practical performance of double hashing is very close to uniform hashing. Gonnet states that it is practically impossible to establish statistically whether double hashing behaves differently from uniform hashing [7, page 57].

Compression

In our state compression scheme, a compression function c is applied to each state s yielding a compressed value $c(s) \in \{0, \ldots, l-1\}$. Here, l denotes the number of different compressed values. If we use b bits for these values, clearly $l = 2^b$.

Now, we introduce the notion of *uniform compression*. In uniform compression, the compressed value of each state is equally likely to be any of the l values $0, \ldots, l-1$. In practice, we can use *universal hashing* to calculate the compressed value from the state descriptor, as suggested by Wolper and Leroy [15].

Let $\Pr(A)$ denote the probability of event A and let S denote the set of all possible states. Furthermore, let H be a universal$_2$ class of hash functions [3] from S to $\{0, \ldots, l-1\}$. Details about the particular universal$_2$ class of hash functions used in our implementation can be found in Sect. 5. If we randomly and uniformly choose one function in H as our compression function c, the universal$_2$

property guarantees that the probability that two different states have the same compressed value is bound, namely

$$\Pr(c(s_1)=c(s_2)) \leq \frac{1}{l} \quad \text{for all } s_1, s_2 \in S,\ s_1 \neq s_2 \ .$$

Hence, universal$_2$ hashing is guaranteed to perform at least as good as uniform compression, where for all different $s_1, s_2 \in S$, $\Pr(c(s_1)=c(s_2)) = \frac{1}{l}$. Furthermore, it can be shown that the above bound will be tight in our application of universal$_2$ hashing. Therefore, universal$_2$ hashing will perform almost exactly like uniform compression.

The Probabilistic Verification Scheme

The probabilistic verification scheme only behaves differently from the basic explicit state enumeration algorithms when a newly reached state s is entered into the hash table. This state insertion process is depicted in Fig. 1. After applying the hash function $\underline{h}$ and compression function c independently to the state s, the algorithm starts probing the table at $h_0(s)$. A probe can yield three different results:

- The probed slot may be empty. The state s has not been encountered previously in the search and its compressed value $c(s)$ is stored in the slot.
- The probed slot contains a compressed state different from the compressed state $c(s)$ being entered. This is called a *collision*. The hash table algorithm then probes the next slot given by the probe sequence $\underline{h}(s)$.
- The probed slot contains a compressed state equal to the compressed state $c(s)$ being entered. In this case, the algorithm assumes that the uncompressed states are the same, which may or may not be true. The state table is not modified, and the successors of the current state s are not generated and searched. When the uncompressed states are indeed equal, this is the desired result. When the uncompressed states are not equal, this results in an *omission* of the current state s, and the possible omission of its successors in the state graph.

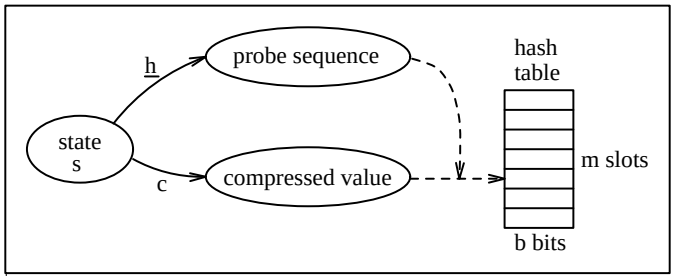

Fig. 1. The state insertion process in the probabilistic verification scheme

It is important to observe that only the slots examined using the probe sequence can lead to omissions. Usually, only a few slots are examined before an empty slot is found (which is why hashing is attractive in the first place). This positive effect of hashing on reducing the likelihood of omissions was not considered by Wolper and Leroy [16], who implicitly assumed that the probe sequence is calculated from the compressed value. Under their assumptions, states with the same compressed value *inevitably* lead to omissions.

Using our improved probabilistic verification algorithm, the choice of the hashing scheme affects the omission probability. A "good" hashing scheme for our probabilistic verification algorithm has few collisions in an unsuccessful search, i.e. while inserting a new state. We employ double hashing since its performance is very close to uniform hashing, where no clustering occurs. Clustering would increase the number of collisions in unsuccessful searches.

3 Calculation of the Omission Probabilities

In the following, we first derive an approximate formula for the omission probability in our probabilistic verification scheme. In the examples tested, this approximation formula seems to be very accurate. However, we also derive another formula that is an upper bound for the omission probability but is less accurate than the approximation formula. Finally, the results from the analytical calculations are compared with results from a simulation of the probabilistic verification scheme.

Approximate Analysis

For the following analysis, we assume that uniform hashing and uniform compression are used. Let $\mathrm{E}[X]$ denote the expected value of random variable X and $n^{\underline{k}} = \prod_{i=0}^{k-1}(n-i)$ the falling factorial power. Besides that, let ln denote $\log_e$. Recall that m denotes the number of slots in the hash table and l the number of compressed values. We assume that we insert n distinct states, numbered $1, \ldots, n$, into the table.

- We first look at a situation where the first k of the n states were inserted into the table without omissions. Thus, the table actually holds k compressed values. We now want to insert the $(k+1)$st state into the table. Let C_k be the random variable describing the number of *collisions* that occur when we insert a new state given there are k states in the table. C_k is exactly one less than the number of *probes* needed to insert the $(k+1)$st state into a conventional hash table. Therefore, one can obtain from the distribution of the number of probes – given, for example, in [13, page 527] – the probability that we have exactly j collisions inserting the $(k+1)$st state given there are k states in the table, namely

$$\Pr(C_k = j) = \frac{k^{\underline{j}}}{m^{\underline{j+1}}}(m-k) \quad . \tag{1}$$

We define N_k to be the event that there is no omission when inserting the $(k+1)$st state given that there were no omissions when the first k states were inserted. Let N'_k be the corresponding unconditional event, namely that there is no omission when inserting the $(k+1)$st state into the table. Clearly,

$$N_k = N'_k \mid N'_{k-1} \wedge \ldots \wedge N'_0 \ .$$

Observing that each collision may result in an omission, we can approximate the probability of the event N_k under the condition that there are exactly j collisions as

$$\Pr(N_k \mid C_k = j) \approx \left(\frac{l-1}{l}\right)^j \ . \tag{2}$$

Here, it is assumed that the j comparisons of the compressed value to be entered with the values in the table are mutually independent. However, remember that we conditioned on the situation where no omission occurred during the insertion of the first k states. This condition imposes some dependence on the compressed values in the table. The values have the tendency to be different from each other because the omission cases – where at least two compressed values are equal – were filtered out by the condition. Thus, (2) underestimates the resulting omission probability. In practice, we use the probabilistic verification scheme with a bound on the omission probability when inserting n states into the table. Hence, the omission probability for one of the n states will be very small – especially when n is large – and so will be the filtering effect of the condition. This is confirmed by experimental results presented later.
Using the theorem of total probability, one can easily obtain from (1) and (2) the conditional probability that there is no omission inserting the $(k+1)$st state given there are k states in the table, yielding

$$p_k := \Pr(N_k) \approx \sum_{j=0}^{k} \left(\frac{l-1}{l}\right)^j \frac{k^{\underline{j}}}{m^{\underline{j+1}}}(m-k) \ . \tag{3}$$

Observe that, in general, omissions change the distribution of C_k, since the insertion algorithm stops when a compressed value in the table equals the compressed value of the $(k+1)$st state. However, we can use the unaltered C_k distribution as given in (1) in the above calculation of p_k. To see why this is correct, imagine that we altered our insertion algorithm so that it always probes until an empty slot is found and so that the final insertion is only made if none of the probed slots contained the compressed value of the $(k+1)$st state. Clearly, this algorithm behaves exactly the same way as the "old" insertion algorithm except an increased run-time and except that the number of collisions is distributed according to the unaltered C_k distribution.

– Now, we consider the situation where we insert n states into the (initially empty) table. Using the definitions of N_k and p_k, the probability that we

have no omissions inserting all n states is obtained as

$$\begin{aligned} p_{\text{no}} &:= \Pr(\text{no omission inserting } n \text{ states}) \\ &= \Pr(N'_{n-1} \wedge N'_{n-2} \wedge \ldots \wedge N'_0) \\ &= \Pr(N'_{n-1} \mid N'_{n-2} \wedge \ldots \wedge N'_0) \Pr(N'_{n-2} \wedge \ldots \wedge N'_0) \\ &= p_{n-1} \Pr(N'_{n-2} \wedge \ldots \wedge N'_0) \ . \end{aligned}$$

Using the above argument recursively yields

$$p_{\text{no}} = \prod_{k=0}^{n-1} p_k \ . \tag{4}$$

By substituting (3) into (4) and making the falling factorial powers explicit, we obtain the following formula for the probability that there are no omissions inserting n states into the table to be

$$p_{\text{no}} \approx \prod_{k=0}^{n-1} \left[\sum_{j=0}^{k} \left(\frac{l-1}{l} \right)^j \frac{m-k}{m-j} \prod_{i=0}^{j-1} \frac{k-i}{m-i} \right] \ . \tag{5}$$

Straightforward evaluation of this formula requires $O(n^3)$ operations. In practice, we need to calculate the omission probability for state spaces on the order of 10 million reachable states. Therefore, we now derive an approximation formula that only needs $O(1)$ operations.

For this approximation, we re-consider (3) in a slightly different form, namely

$$p_k \approx \sum_{j=0}^{k} \left(\frac{l-1}{l} \right)^j \Pr(C_k = j) \ . \tag{6}$$

The right-hand side is an instance of the theorem of total probability. The "expected" exponent of the term $(\frac{l-1}{l})$ is the expected value of C_k. Thus, we approximate

$$p_k \approx \left(\frac{l-1}{l} \right)^{\mathrm{E}[C_k]} \ . \tag{7}$$

Plugging this approximation into (4) yields

$$p_{\text{no}} \approx \left(\frac{l-1}{l} \right)^{\sum_{k=0}^{n-1} \mathrm{E}[C_k]} \ . \tag{8}$$

This formula has a simple, intuitive explanation. The sum equals the expected value of the total number of collisions inserting n states into a conventional hash table. For each of these collisions, we have a probability of $(\frac{l-1}{l})$ that no omission occurs and the compressed value comparison is approximately independent of all others.

A closed form for $\mathrm{E}[C_k]$ can, for example, be obtained from the formulas in [13, page 528], yielding

$$\mathrm{E}[C_k] = \frac{k}{m+1-k} \quad . \tag{9}$$

Using this formula, the sum in (8) can be written in closed form as

$$\sum_{k=0}^{n-1} \mathrm{E}[C_k] = (m+1)[H_{m+1} - H_{m-n+1}] - n \quad , \tag{10}$$

where H_n denotes a harmonic number. Using the asymptotic approximation for H_n given, for example, in [8, (9.28)] we obtain

$$H_n \approx \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} \quad . \tag{11}$$

Substituting (10) and (11) into (8), we obtain the final result of our approximation effort, namely

$$p_{\mathrm{no}} \approx \tilde{p}_{\mathrm{no}} := \left(\frac{l-1}{l}\right)^{(m+1)\,\ln(\frac{m+1}{m-n+1}) - \frac{n}{2(m-n+1)} + \frac{2nm+2m-n^2}{12(m+1)(m-n+1)^2} - n} \quad . \tag{12}$$

Obviously, the evaluation of this formula only requires $O(1)$ operations and it is thus useful for handling our "arbitrarily" large values of n.

In the remainder of this paper, we will only consider the *omission* probability of our probabilistic verification scheme. Clearly, this probability can be calculated as

$$p_{\mathrm{om}} := \Pr(\text{one or more omissions inserting } n \text{ states}) = 1 - p_{\mathrm{no}} \quad .$$

Upper Bound

An upper bound for the omission probability is derived in the appendix yielding

$$p_{\mathrm{om}} \leq \hat{p}_{\mathrm{om}} := \frac{1}{l}\,\left[(m+1)\,[H_{m+1} - H_{m-n+1}] - n\right] \quad . \tag{13}$$

Simulation Results

We wrote a simulator for the probabilistic verification scheme to assess if the theoretical results for the omission probability can be achieved in a practical implementation. Here, we only show results for an example with a small number of slots in the hash table. Results running the real verifier on larger examples can be found in Sect. 6.

One *experiment* of the simulator consists of generating n states with a random number generator and inserting the compressed values into a hash table. The experiment is repeated N times with different states. Finally, the omission probability is calculated as the number of experiments where omissions occurred divided by N.

The simulation results discussed in the following were obtained by repeating each experiment $N = 10$ million times, using a hash table with $m = 199$ slots and using $b = 10$ bits for the compressed values. The number of states n inserted into the table was used as a parameter. Simulation runs were performed for $n = 19, 39, \ldots, 199$.

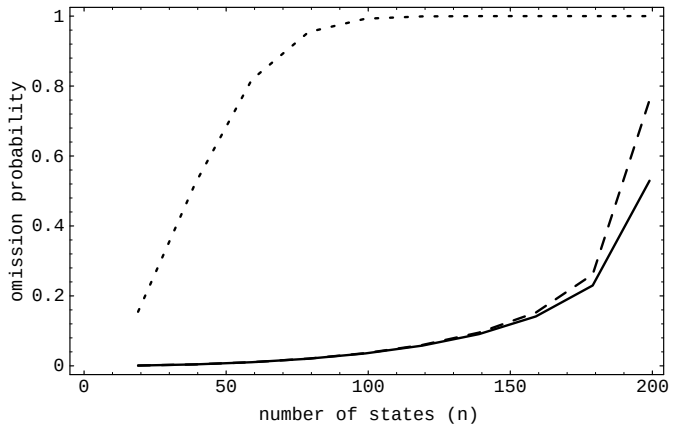

Fig. 2. Omissions probabilities obtained from the simulation (solid), the upper bound (dashed) and Wolper and Leroy's formula (dotted)

Fig. 2 compares the omission probabilities for Wolper and Leroy's formula and our upper bound (13) with the simulation results. Observe that the omission probability rises as the table fills up. Wolper and Leroy's formula clearly overestimates the omission probability, and the upper bound becomes less accurate for larger values of the omission probability. Both, our final approximation (12) and the $O(n^3)$ formula (5) yield values that are so close to the simulation values for the omission probability that the graphs coincide in Fig. 2.

Therefore, in Fig. 3 the *relative errors* of the omission probabilities obtained by the $O(n^3)$ formula and the final approximation are shown. Furthermore, the relative errors of the 90% confidence intervals of the omission probabilities are graphed. All relative errors were calculated using the simulation results as exact values. Both, final approximation and $O(n^3)$ formula yield relative errors smaller then 0.8%.

Observe that the values from the final approximation are always bigger than the ones from the $O(n^3)$ formula while still being very close to the exact values. As discussed before, the $O(n^3)$ formula slightly underestimates the omission probability.

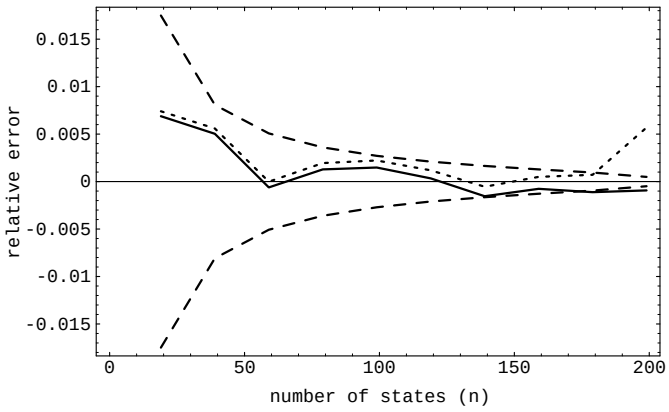

Fig. 3. Relative errors of the omission probabilities obtained by the $O(n^3)$ formula (solid) and the final approximation (dotted) and of the 90% confidence intervals of the omission probabilities (dashed)

4 Choosing the Number of Bits per State

We now address the question of how one should choose the number of bits b in the compressed values. First, we depict the omission probabilities for 5-byte (i.e. 40-bit) and 4-byte compression using a large hash table of size 400 million bytes and present a table showing the number of bits in the compressed values dependent on hash table size and maximum omission probability. Finally, a rule of thumb for the maximum omission probability is derived.

5-byte and 4-byte Compression

As mentioned before, for the following two examples a hash table size of 400 million bytes was chosen. Not coincidentally, this number corresponds well with the available DRAM on our largest machine. Our hash table of size 400 million bytes has $m = 80$ million slots using 5-byte compression and $m = 100$ million slots employing 4-byte compression.

Figures 4 and 5 show the omission probabilities for 5-byte and 4-byte compression, respectively, while the number of states n inserted into the table was varied. Even if the table fills up completely, the omission probability stays smaller than 0.13% for 5-byte compression and below 33% for 4-byte compression.

For both figures, the omission probabilities were calculated using the final approximation (12). The upper bound (13) yields values for the omission probabilities that are usually quite close to the ones from the final approximation. The biggest difference occurs when the table fills up completely. Then, the upper bound values have relative errors of 0.06% and 21% for 5-byte and 4-byte compression, respectively.

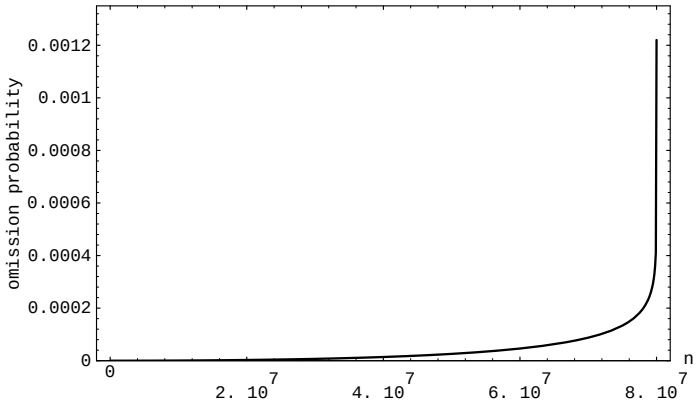

Fig. 4. Omission probability for 5-byte compression

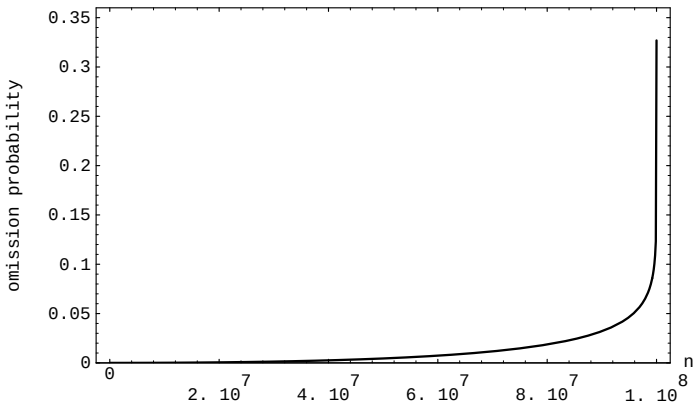

Fig. 5. Omission probability for 4-byte compression

In both, 5-byte and 4-byte compression, the omission probability rises sharply immediately before the hash table fills up completely. Intuitively, this can be explained by observing that when we insert a compressed state into an almost full table, it is compared against many compressed values until an empty slot is found. Omissions may occur in any of these comparisons. In practice, the characteristic shape of the omission probability curve yields a nice behavior of our probabilistic verification scheme. If the verifier terminates without a hash table overflow, the table is usually not almost completely full and we have a much lower omission probability than in the worst case, where it fills up completely. On the other hand, if there is an overflow in the hash table, we are sure to miss states. So the final increase in the omission probability will not matter.

Table 1 shows the required number of bits for the compressed values – given the size of the hash table in bytes and the maximum omission probability P_{om}.

The maximum omission probability occurs when the table fill up completely (i.e. $n = m$). Remember that typically the omission probability will be much smaller. Table 1 was generated by solving (12) numerically.

For example, if one is willing to tolerate a maximum omission probability of 50%, one could use 4-byte compression until the hash table size exceeds roughly 700 million bytes.

Table 1. Required number of bits for the compressed values. In this table, $1\,\mathrm{M} = 10^6$ and $1\,\mathrm{G} = 10^9$.

	hash table size (bytes)						
P_{om}	100 M	200 M	500 M	1 G	2 G	5 G	10 G
0.1%	38.2	39.3	40.6	41.6	42.6	44.0	45.0
1%	35.0	36.1	37.4	38.4	39.4	40.8	41.8
10%	31.8	32.8	34.2	35.2	36.2	37.5	38.5
50%	29.2	30.2	31.6	32.6	33.6	34.9	35.9
99%	26.6	27.6	29.0	30.0	31.0	32.3	33.3

A Rule of Thumb

In the following, we derive a rule of thumb for the maximum omission probability P_{om}, that occurs when the table fills up completely.

Assuming that $l \gg 1$, we can approximate $(\frac{l-1}{l}) = 1 - \frac{1}{l} \approx e^{-\frac{1}{l}}$ since $e^x \approx 1 + x$ for $|x| \ll 1$. By plugging this approximation into (8), we obtain

$$1 - P_{\mathrm{om}} \approx e^{-(\frac{1}{l})\sum_{k=0}^{m-1} \mathrm{E}[C_k]} \quad .$$

Using (10), $H_n \approx \ln n + \gamma$, $e^x \approx 1 + x$ (assuming a sufficiently large l) and assuming $m \gg 1$, we get

$$P_{\mathrm{om}} \approx \frac{1}{2^b}\, m(\ln m - 1) \quad . \tag{14}$$

Observe, that adding just one bit to the compressed values approximately halves the maximum omission probability.

5 Implementation

The interesting part in the implementation of the probabilistic verification algorithm is the calculation of the hash and compressed values out of the state descriptor. This calculation was done in two steps. First, a 96-bit key value (natural number) was calculated from the "arbitrary" long state descriptor using universal hashing. Second, the two hash values for double hashing and the compressed value were determined from this key.

We employed the H_3 universal$_2$ class of hash functions [3] to calculate the key value from the state descriptor. Assume the key values have k bits and the state descriptors have d bits. Then, each function $h \in H_3$ corresponds to a d by k Boolean matrix m and H_3 corresponds to the set of all those matrices (which has 2^{dk} elements). Let m_i be the ith row of m, s_i the ith bit of state s and let $\oplus$ denote the exclusive or operator. Then, the hash function h is calculated as

$$h(s) = s_1 m_1 \oplus s_2 m_2 \oplus \ldots \oplus s_d m_d \quad ,$$

i.e. the bits in the state descriptor select those rows of m that are to be combined with exclusive or.

Note that traditional methods for calculating keys from long state descriptors contain some pitfalls. Simple adding or "exclusive oring" all words in the state descriptor yields the same key value for several kinds of symmetries. For example, if only sums and "exclusive ors" are used, any permutation of the words in the state descriptor results in the same key value and therefore *necessarily* leads to omissions in the verification if "permuted" states occur. This is especially severe, since many systems contain several kinds of symmetries. These problems do not occur in the universal hashing-based calculation.

We used b of the k bits of the key directly as compressed value. The hash values were calculated with the division method [4] using a subset of the remaining k–b bits of the key to achieve independence of hash and compressed values. Note that the number of slots m in the hash table can be any prime number since we employed the division method.

6 Results on Larger Examples

We tested the probabilistic verification scheme on three examples – Peterson's algorithm for mutual exclusion (*Peterson*), an industrial cache coherence protocol (*cache3*) and the cache coherence protocol of the Scalable Coherent Interface [11] (*SCI*). The sizes of all of the three examples are scalable. That explains the differing numbers of reachable states for the SCI example in the following.

In Table 2, the three examples were scaled to yield huge numbers of reachable states. We used 5-byte compression for all of the three examples when using the probabilistic verification scheme. In the table, the run-time of the classic scheme is compared to that of the probabilistic one. Observe that there is only a small increase in the run-time requirements for the probabilistic scheme due to the calculation of the compressed values. As expected, no omissions occurred.

In the next set of experiments, we used the SCI example with $n = 109\,080$ states and a hash table with $m = 116\,531$ slots. We varied the number of bits b in the compressed values. For each value of b, we conducted 100 runs of the verifier with randomly chosen universal hash functions and counted the number of runs in which omissions occurred. Figure 6 compares the outcomes of these experiments with the omission probabilities from our final approximation (12). Furthermore, the 90% confidence intervals of the omission probabilities

Table 2. Verification with 5-byte compression

example	state descriptor (bits)	states	run-time		omissions
			classic	probabilistic	
Peterson	69	6 698 326	3 187 s	3 232 s	none
SCI	320	1 179 942	4 254 s	4 396 s	none
cache3	179	2 093 231	4 303 s	4 436 s	none

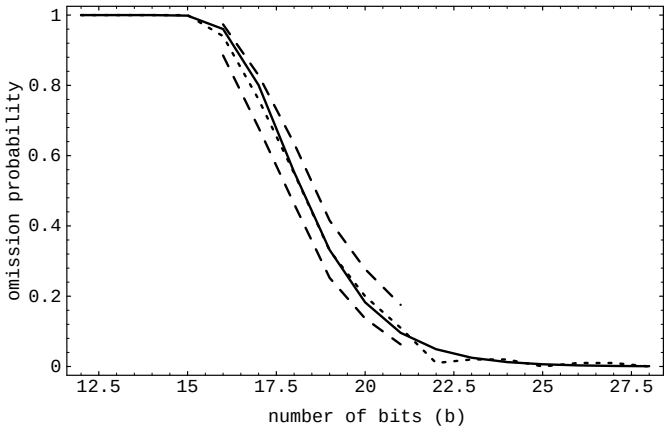

Fig. 6. Omission probabilities obtained from the experiments (dotted) and the final approximation (solid) and 90% confidence intervals of the omission probabilities (dashed)

are shown. The experimental results in the figure nicely match the theoretical analysis.

The mean number of omitted states at, for example, 55% omission probability was only 3.2, out of 109 080 reachable states. The 90% confidence interval for the mean number of omitted states in this case was $(2.6, 3.8)$. Thus, coverage is expected to degrade very slowly.

7 Conclusion and Future Research

The probabilistic verification scheme discussed in this paper worked reliably for several large verification examples. A small probability of omitting states during the verification process seems negligible in comparison with the other inaccuracies incurred by making verification of large systems feasible. Usually, one has to simplify systems of industrial size (e.g. by down-scaling and by omitting system parts) before they are amenable to formal verification [14]. This simplification is more likely than a small omission probability to lead to failures in detecting all errors in the system under verification.

Furthermore, the omission probability can be controlled by the user of the verification system. Our rule of thumb (14) shows that adding just one bit to the

compressed values halves the omission probability. Thus, it seems advantageous – whenever possible – to add a few bits to the compressed values rather than to re-run the verifier with different hash and compression functions. In practice, for large hash tables with on the order of 500 million bytes, either 5-byte or 4-byte compression is recommended – depending on the maximum omission probability one is willing to tolerate.

While space was for a long time considered to be the major limiting factor in verification, we currently experience a shift to run-time as the new major limiting factor, which increases the priority of research into accelerating explicit-state verification methods.

Finally, one improvement in the probabilistic verification scheme presented in this paper could be made. If a scheme like ordered hashing [1] that aims at reducing the complexity of the unsuccessful search in a hash table is employed, the omission probability is further reduced. Some preliminary results indicate that thereby the the number of bits in the compressed values could be reduced by a small fraction.

Acknowledgements

We would like to thank the members of our formal verification group at Stanford, especially Alan Hu, for discussions on the subject of this paper. Furthermore, discussions with Richard Chatwin helped in clarifying some aspects of the analysis of the probabilistic verification scheme. We are also grateful to Irene Shen and Gerard Holzmann for their comments on a draft of this paper and to Pierre Wolper and Denis Leroy for sharing the unpublished revision [15] of their paper with us.

References

1. O. Amble and D. E. Knuth. Ordered hash tables. *Computer Journal*, 17(2):135–42, 1974.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.
3. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
5. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.
6. D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.
7. G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley Publishing Company, 2nd edition, 1991.

8. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Publishing Company, 1988.
9. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
10. A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–82, 1994.
11. *IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI).*
12. C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, 1993.
13. D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, 1973.
14. U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.
15. P. Wolper and D. Leroy. Reliable hashing without collision detection. Unpublished revised version of [16].
16. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.
17. L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC™-I. In *32nd Design Automation Conference*, pages 7–12, 1995.

Appendix

In this appendix, an upper bound $\hat{p}_{\mathrm{om}}$ for the omission probability p_{om} is derived. Clearly, this upper bound for the omission probability yields a lower bound for the no omission probability p_{no} and vice versa. We first calculate the latter bound.

The first approximation in our approximate analysis was (2). We try to replace this approximation by a worst case formula. This worst case occurs, when each of the j compressed values the compressed state to be entered is compared with is different from all the others. Hence, we obtain

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

By using the theorem of total probability, the definition of p_k and the definition of the expected value, we obtain

$$\begin{aligned} p_k \ &\equiv \ \Pr(N_k) \ = \ \sum_{j=0}^{k} \Pr(N_k \mid C_k = j) \Pr(C_k = j) \\ &\geq \ \sum_{j=0}^{k} \frac{l-j}{l} \Pr(C_k = j) \ = \ 1 - \frac{1}{l} \mathrm{E}[C_k] \ . \end{aligned}$$

Substituting this result into (4) and multiplying out the product yields

$$\begin{aligned} p_{\mathrm{no}} \ \geq \ 1 \ &- \ \frac{1}{l} \sum_{k=0}^{n-1} \mathrm{E}[C_k] \ + \ \left(\frac{1}{l}\right)^2 \sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} \mathrm{E}[C_k] \, \mathrm{E}[C_j] \\ &- \ \left(\frac{1}{l}\right)^3 \sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} \sum_{i=j+1}^{n-1} \mathrm{E}[C_k] \, \mathrm{E}[C_j] \, \mathrm{E}[C_i] \ + \ \ldots \end{aligned} \tag{15}$$

The triple sum in the fourth term on the right-hand side can be rewritten by changing the order of summation, namely

$$\sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} \sum_{i=j+1}^{n-1} \mathrm{E}[C_k] \, \mathrm{E}[C_j] \, \mathrm{E}[C_i] \ = \ \sum_{k=0}^{n-1} \sum_{j=k+1}^{n-1} \mathrm{E}[C_k] \, \mathrm{E}[C_j] \sum_{i=j+1}^{n-1} \mathrm{E}[C_i] \ .$$

Thus, the fourth term has the additional (non-constant) factor $\left(\frac{1}{l} \sum_{i=j+1}^{n-1} \mathrm{E}[C_i]\right)$ in comparison to the third term. If we assume that $l \geq \sum_{i=0}^{n-1} \mathrm{E}[C_i]$, the value of this factor can be bounded above as

$$\frac{1}{l} \sum_{i=j+1}^{n-1} \mathrm{E}[C_i] \ < \ \frac{1}{l} \sum_{i=0}^{n-1} \mathrm{E}[C_i] \ \leq \ 1 \ .$$

Then, the difference of the third and the fourth term of the right-hand side of inequality (15) is greater than zero. Using the above argument for all higher numbered terms, we obtain

$$p_{\mathrm{no}} \;\geq\; 1 - \frac{1}{l} \sum_{k=0}^{n-1} \mathrm{E}[C_k] \quad . \tag{16}$$

Observe, that this inequality also holds for $l < \sum_{i=0}^{n-1} \mathrm{E}[C_i]$ since p_{no} is a probability and thus it holds for all l. Plugging formula (10) into (16), we finally obtain our upper bound for the omission probability, namely

$$p_{\mathrm{om}} \;\leq\; \hat{p}_{\mathrm{om}} \;:=\; \frac{1}{l} \left[(m+1)\left[H_{m+1} - H_{m-n+1}\right] - n\right] \quad .$$