# New Methodes for Sequential Hashing

Jürgen Eckerle and Thomas Lais
Institut für Informatik, Universität Freiburg,
Am Flughafen 17, Geb. 51, D-79110 Freiburg, Germany
email: {eckerle,lais}@informatik.uni-freiburg.de

**Abstract**

Supertrace is an efficient search algorithm for the verification of protocol state spaces. Since supertrace is included in many commercial and noncommercial verification tools, it is important to know where its limitations are and where there is potential for possible improvements. To get a higher coverage in literature it is proposed to start repeatedly supertrace by using different hash functions which is called *sequential hashing.* We present both theoretical and experimental results to measure the quality of sequential hashing with supertrace. Tests made with the SPIN validator with different classes of hash functions show that the additional number of states reached in successive runs decreases rapidly. A coverage of close to 1 is practically impossible. The universal hashing seems to be much better than all the other hash functions if the state space size is greater than the size of the hash table.

We present an analysis for the expected coverage of sequential hashing which explains quite well the experimental results. Our new mathematical model shows that the overlap of successive runs is much higher than predicted by former approaches. Additionally, it predicts that simple randomization strategies which in successive runs change randomly both, the hash function and the traversal order of the states can increase the coverage sibnificantly. Experimental results show that our new algorithm, *universal supertrace*, which is based on such randomization techniques, outperforms the original supertrace and allows the coverage to be increased by a factor of up to 9-10.

**Keywords**

Protocol verification, reachability analysis, explicit state enumeration, bitstate hashing, supertrace, sequential hashing, SPIN.

# 1 Introduction

In the reachability analysis protocol validation is done by state space search. Usually, the state space explosion makes a full exhaustive search impossible. Partial search methods, like bitstate hashing or supertrace ([Hol87], [Hol88]), allow the time and space requirements to be reduced efficiently when compared with exhaustive search methods. As a result, the verification of protocol state spaces with orders of magnitude greater than that which can be verified with traditional methods is possible. As a result, supertrace, or bitstate hashing, is included in many tools for automated protocol validation.

Partial search methods risk that a protocol error remains undetected. Holzmann measures the quality of partial search methods by the coverage, where coverage is defined as the percentage of the reachable states actually explored by the verifier. The expected coverage equals the average probability that a particular state is not omitted. Therefore, it is imperative to reach the highest coverage possible.

A natural method of improving the coverage is to start the supertrace algorithm repeatedly with different hash functions. This method, referred to in ([Hol91]) as multiple hashing and in ([Hol95]) as sequential hashing, is supported by the SPIN validator of Holzmann. In theory, sequential hashing allows an expected coverage close to 1 to be achieved provided that the number of runs is sufficiently large. Up until now, it has not been determined how fast this approximation can really be done and which strategy results in the highest coverage. To get a better insight, our paper presents both theoretical and experimental results in order to measure the quality of sequential hashing.

Our tests made within the SPIN environment with different classes of hash functions show that in many cases the number of states additionally reached and checked by supertrace in successive runs decreases rapidly. While all the hash functions have the same asymptotical behaviour, universal hashing seems to be best.

We present a mathematical analysis of the combined coverage of sequential hashing which overcomes the restrictions of former approaches and is more accurate. In addition to that, we prove that the overlap of the nodes generated in the different runs is much higher than assumed in former approaches which consider the generated states as randomly chosen from the set of all reachable nodes. Our model shows that the quality of sequential hashing suffers as a result of the fixed traversal order in which supertrace generates the state space. This drawback is of principal nature and cannot be overcome by better hash functions.

Our model predicts that randomization techniques which change in each run both the hash function and the traversal order in the state space can increase the coverage by several times. We present a new algorithm, universal supertrace, which is based on such randomization techniques. Experimental results made with the SPIN validator show that randomized supertrace allows the coverage to be increased by a factor of up to 8-10 when compared with the original supertrace. We expect that additional improvements are possible using other strategies for changing the traversal order.

This paper is structured as follows. Section 2 reviews the well-known supertrace algorithm resp. bitstate/double bitstate hashing and introduces universal hashing. Our experimental results of sequential hashing with different hash functions are presented in section 3. In section 4 we mathematically analyse sequential hashing and compare our results with earlier ones. In section 5 we discuss our new algorithm, universal supertrace, its implementation and the experimental comparison with the original supertrace. Finally, the conclusion with suggestions for future research follows.

# 2 Background

In the reachability analysis the protocol is represented by an implicitly given state space, that is by a set of start states and by a set of possibly applicable transition. Protocol errors can be found by explicit enumeration of all the reachable states, that is by generating the states with either breadth-first or depth-first searches. Duplicate detection with hash tables is a widely used method of reducing the running time. The generated states are stored in a hash table which allows it to be determined whether or not a newly generated state $x$ has been reached before and can be pruned now. Unfortunately, the number of states is often unmanageably large; this is the state explosion problem. For this reason in practice the total memory available for the hash table is often the limiting resource in verification.

## 2.1 Supertrace Algorithm

The bitstate hashing method or supertrace method introduced by [Hol88] aims at reducing the memory requirements for the hash table. In contrast to conventional methods where a state descriptor is stored completely, each state is represented by a bit which is addressed by the hash function $h$. Initially, all the bits of the table are set to 0. When a state $x$ is inserted into the table the bit corresponding to the hash value $h(x)$ is set to 1. When the algorithm generates a new state $x$ and finds that the bit corresponding to this state is already set to 1, it assumes that $x$ has been visited before and prunes $x$ without verification. If the assumption is incorrect an omission of the newly generated state $x$ and potentially all of its successors occurs, since the search algorithm backtracks when it finds a state which has already been visited before. Usually supertrace is combined with a depth-first search with a typical threshold of $10^4 - 10^5$.

The method above is referred to as single bitstate hashing, in contrast to double bitstate hashing ([Hol91]) (see figure 1). In the latter one each state corresponds to two bits calculated by two different hash functions. The probability of an omission error is reduced since a hash collision now requires a collision on both bits. Both space and time requirements are reduced efficiently by supertrace. Space is reduced, because usually the state descriptor uses about $10^3$ bits which contrasts with the one or two bits if bitstate or double bitstate hashing is used. The running time is reduced,

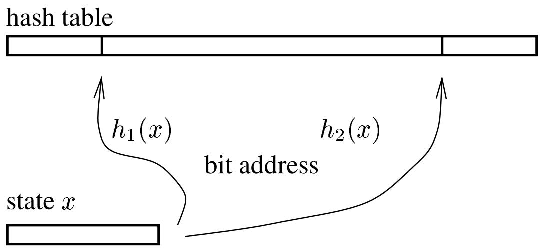


Figure 1: Double bitstate hashing.

since duplicate test and insertion is done by a simple bit operation and in addition to this, hash collisions are not resolved.

## 2.2 Sequential Hashing

The quality of partial search techniques like bitstate hashing or supertrace can be measured by the coverage which is defined as the quotient of the number of reached states and the number of states reachable by an exhaustive search. Since the higher the coverage, the lower the average probability for an omission error, partial search methods should aim at achieving the highest possible coverage.

In ([Hol91]) it is proposed that supertrace be started several times with independent hash functions. It is to be expected that different hash collisions will occur and as a result of this in each run different parts of the state space will be visited. In theory a coverage of arbitrarily close to 1 can be achieved, provided that a sufficiently large number of runs with independent hash functions are done. In the SPIN environment this method is implemented in order to increase the coverage of supertrace.

## 2.3 Universal Hash Functions

Since hash collisions are not resolved by bitstate hashing, it would be ideal if the used hash function could distribute the keys uniformly on the hash addresses. The key idea of universal hashing is to choose the hash function randomly from a suitable class of hash functions. If this class is carefully selected, a good average distribution can be guaranteed for an arbitrary set of hash keys. A state descriptor can be considered as a vector over the field of two elements. Let $\mathcal{H}$ be a class of hash functions from $\mathcal{A}$ to $\mathcal{B}$ where $\mathcal{A} = \{0, \ldots, 2^k - 1\}$ and $\mathcal{B} = \{0, \ldots, 2^l - 1\}$ for some integers $k$, $l$. Recall, $\mathcal{H}$ is *universal*, if $|\{h \in \mathcal{H} | h(x) = h(y)\}| \leq |\mathcal{H}|/|\mathcal{B}|$ for all $x$, $y \in \mathcal{A}$ . That is, $\mathcal{H}$ is universal if no pair of distinct keys collide under more than $(1/|\mathcal{B}|)$-th of the functions. Universal hashing reduces the number of hash collisions on average. More precisely, if $x \in \mathcal{A}$ and $\mathcal{S}$ is any subset of $\mathcal{A}$, then the expected number of synonyms of $x$ in $\mathcal{S}$ is lower than or equal to $|\mathcal{S}|/|\mathcal{B}|$. Note, that two keys $x$, $y$ are called *synonyms*

with respect to hash function $h$ if $h(x) = h(y)$. We present some examples used in our experiments.
**Class $\mathcal{H}_1$:** Let the class $\mathcal{H}_1$ be defined by

$$\mathcal{H}_1 = \{f_{a,b} | f_{a,b} : x \rightarrow ((ax + b)\ mod(p))\ mod(|\mathcal{B}|), a, b \in \{0, \dots, p-1\}, a \neq 0\}\}$$

where $p$ is prime, $p > |\mathcal{A}|$. $\mathcal{H}_1$ is universal ([CW79]). $\mathcal{H}_1$ is suitable for applications where the bit strings which represent the keys can conveniently be multiplied by the computer. If the bit vector is too long as occurs in the protocol domain, then in ([CW79]) the following construction is suggested .
**Class $\mathcal{H}_2$:** Let the class $\mathcal{H}_2$ be defined by

$$\mathcal{H}_2 = \{h_{f_1,\dots,f_r} | h_{f_1,\dots,f_r} : x = (x_1, \dots, x_r) \rightarrow f_1(x_1) \oplus \dots \oplus f_r(x_r), f_1, \dots, f_r \in \mathcal{H}_1\}$$

where $\oplus$ is the exclusive-or operation. If $r$ is a power of two, then $\mathcal{H}_2$ is (nearly) universal (i. e. $1/|\mathcal{B}|$ has to be replaced by $(1/|\mathcal{B}|) \cdot (1 + (|\mathcal{B}| + 1)/(p-1))$ in the definition above). If necessary, the state descriptor can be filled with 0's to increase the length to a power of two. In our implementation we divided the bit vector $x$ in integer blocks and selected $p = 2^{31} - 1$ for $\mathcal{H}_1$.
**Class $\mathcal{H}_3$:** Let $\mathcal{M}$ be the set of Boolean matrices with $l$ columns and $k$ rows. Then we define

$$\mathcal{H}_3 = \{f_c | f_c : x \rightarrow x * M_c, M_c \in \mathcal{M}\}$$

where $*$ denotes the matrix multiplication. $\mathcal{H}_3$ is also universal ([CW79]).

# 3 Experimental Results

In this section we present some experimental results made with SPIN environment to measure the coverage in dependence of the used class of hash functions, the hash table size, the number of runs and the state space size.

Some remarks to our experimetal testbed seems in place. For evaluation SPIN had to be extended. Of cause, for counting the number of states additionally reached in a second run, in a third run, etc. we added a second hash table in which each state reached in one of $q$ runs is explicitely stored with full state description and not deleted during the same test serie. This additional hash table is used in the following way: If a state $x$ is reached for the first time in a run $i$, then $x$ is inserted in addition with the run number $i$. If $x$ is reached again in a run $j$, then the hash tables remains unchanged, since $x$ is already in the hash table. The index additionally stored in this hash table for a reached state $x$ determines that run which generates and verifies $x$ firstly. In contrast to that the bitstate hash table is cleared before each new run. Note, that exclusively the bitstate hash table determines whether the search continues with traversing the successors of a state $x$ or not.

| hash class | (a) | (b) ($\times 10^{-2}$) | (c) |
|---|---|---|---|
| H2 | 25.76 | 56.06 | 2.084 |
| H3 | 19.43 | 46.84 | 1.88 |
| HC | 3.72 | 14.95 | 1.12 |
| HP | 4.57 | 18.30 | 1.13 |
| HO | 1.00 | 4.53 | 1.00 |

Figure 2: Measured time for hash computation of class X relative to class HO (originally in SPIN) (a), relative to the verification time with class X (b). Verification time with class X relative to verification time with class HO (c).

The experimental evaluation is based on the PFPT-protocol which was also used in ([Hol95]) and is provided by the SPIN environment. To evaluate sequential hashing, we measured the coverage in a sequence of tests in which the hash class and the parameters like state space size $n$, hash table size $m$ and the number of runs $q$ has been varied. In our experiments we used supertrace with the partial-order reduction methods supported by SPIN. This reduces the search effort, but has no effect on the evaluation.

The following hash classes were used: The hash functions originally included in SPIN which are based on an efficient implementation of a CRC-calculation (here denoted by **HO**) (for more details see ([Hol91])), class $\mathcal{H}_2$ (**H2**), class $\mathcal{H}_3$ (**H3**), a class based on hashing by cyclic polynomials (**HC**) ([Coh97]) and a class based on hashing by general polynomial division (**HP**) ([Coh97]). In our release of SPIN (version 2.9.5) the number of CRC-polynomials is restricted to 32. Thus, we extended SPIN in order to supply a sufficiently large set of randomly generated CRC-polynomials (**HZ**).

Figure 2 gives an overview of the running time necessary for the different hash functions. It shows that universal hashing doubles the running time when compared with CRC hashing (HO). We observed that the measured coverage of a single run differs only by a few percent between the classes, whereas when the number of runs is increased class $\mathcal{H}_2$ works significantly better than all the others, especially if $n \gg m$ (this is the relevant case in practice). In our tests $n$ is increased by simply increasing the depth-first threshold. The improvement by $\mathcal{H}_2$ grows as $(n/m)$ or $q$ grows (compare fig. 3-up and fig. 3-down). For $n/m \approx 1/250$ and $q = 32$, the coverage could be increased of up to 4 (see fig. 3-up). This can be explained as follows: The higher $n/m$ is, the more important it is that the hash functions selected in successive runs are independent, and this is best achieved by the universal class $\mathcal{H}_2$. This improvement cannot be increased indefinitely by increasing the search depth $d$. If $n$ and $m$ are fixed and $d$ grows, the improvement first increases, but then gradually decreases (see fig. 3). Figure 5 reflects how coverage is improved by increasing $m$. If $n \gg m$, then doubling the hash table approximately doubles the measured coverage. This factor decreases rapidly with a decrease in $n/m$.

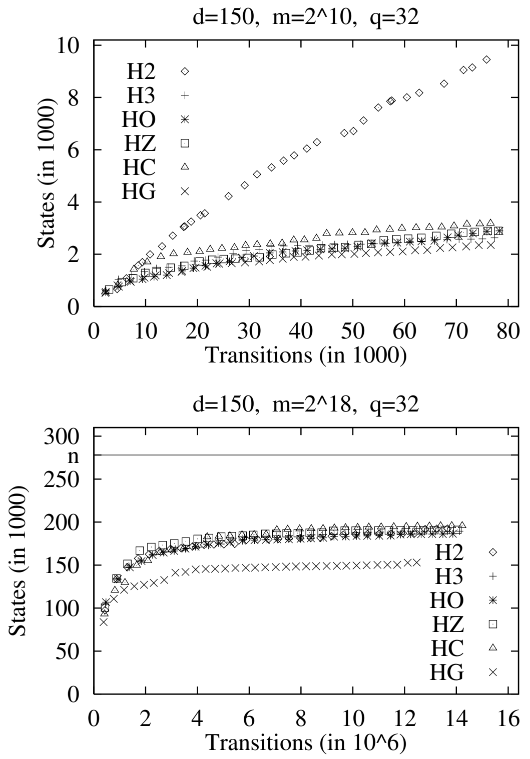


Figure 3: Measured coverage in dependence of performed transitions.

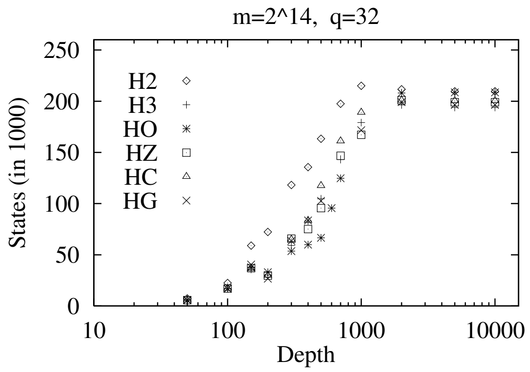


Figure 4: Measured coverage in dependence of search depth.

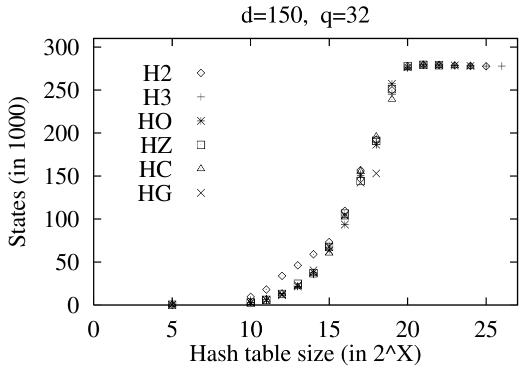


Figure 5: Measured coverage in dependence of hash table size.

Note: In our experiments the coverage could not be increased by a simple strategy of using alternatively a high (about $10^4 - 10^5$) and a small (about $10^2 - 10^3$) threshold in successive runs with the intention of traversing different parts of the search tree.

# 4 Analysis

In this section we present an analysis of bitstate and sequential hashing. Let $m$ be the size of the hash table (measured in the number of bit addresses), $q$ the number of runs and $n$ the size of the state space.

## 4.1 Related Work

In ([Hol91]) it is questioned whether sequential bitstate hashing can increase the coverage with the same system constraints. He answers with yes and gives an estimation of the coverage under certain conditions. He assumes that in each run exactly $s$ states are generated and that the states generated in each run can be considered as randomly selected from the pool of reachable states. The combined coverage of two runs can then be computed as follows. A first run with a first hash function finds $s$ from $n$ states. A second run with a second hash function finds again $s$ states, but makes a different selection. The combined coverage is greater than $s$, but lower than $2s$ since there is an overlap between both state sets. Under the above assumption the probability that a state $x$ is generated in both runs is $(s/n)^2$, hence the expected overlap is $n \cdot (s/n)^2$ and the combined coverage after two runs is $2s - s^2/n$. More generally, the probability that an arbitrary state $x$ is not generated in a selected run is $1 - s/n$, hence the probability that $x$ is not generated in $q$ independent runs is $(1 - s/n)^q$, and hence $x$ is reached with probability $1 - (1 - s/n)^q$. The expected number of reached states after $q$ runs is $n \cdot (1 - (1 - s/n)^q$ and thus the expected coverage is given by

$$1 - (1 - s/n)^q. \tag{1}$$

Equation (1) assumes that there are "ideal circumstances" which occur if the generated states can be considered as randomly chosen from the set of all reachable states. In ([Hol95]) the reason for this is stated to be the fact that a graph representing a protocol state space is well-connected. We will show theoretically below that this is not sufficient. The overlap of successive runs is much higher and the combined coverage much smaller than predicted by (1).

In ([Hol95]) there are presented lower and upper bounds for the expected coverage of a single run with bitstate and double bitstate hashing. These bounds are given by general constraints. Unfortunately, there are many pairs $(n, m)$ for which either the lower bound or the upper bound is trivial or not defined. Our analysis does not suffer from such restrictions and works well even in the case $n > m$ which is the relevant case.

## 4.2 Our Approach

### 4.2.1 Analysis of Bitstate Hashing

In this subsection we present an analysis of the expected coverage of sequential hashing based on double bitstate hashing. Single-bitstate hashing can be analysed in a similar way (see [EL98]).

Equation (1) does not reflect the fact that the states are traversed in each run in depth-first order. There is an enumeration of the reachable states, say $(x_i | 1 \leq i \leq n)$, such that for each run and for all pairs $i, j$, $1 \leq i < j \leq n$, it is true: If both $x_i$ and $x_j$ are generated (reached), then $x_i$ is generated before $x_j$. Note, since supertrace is a partial search algorithm it is not guaranteed that $x_i$ and $x_j$ are both generated. The following analysis takes into account that the states are traversed with respect to a fixed order.

In a traversal of a protocol state space, many states are generated twice. Since we compute an upper bound of the number of generated states, w. l. o. g. we may assume that duplicates have been pruned in the sequence above. Duplicates are states $x_i, x_j$ such that $i \neq j$, but $x_i = x_j$. Such a shorter sequence is achieved by removing repeatedly in the case of two duplicates the one with the higher index. Note that this shortened sequence generates the exact same set of states as the original sequence, since only duplicates have been pruned. However, using this shortened sequence the ratio of reached states to the number of transitions is estimated too optimistically.

In case of double bit-state hashing, two different independent hash functions are used to address two bits. Let $h_1$ and $h_2$ be these hash functions. We assume, as in ([Hol91], [Hol95]), that the hash values are distributed randomly and independently. Then we may consider the insertion operation as a random experiment which will be applied to each state with respect to the order $x_1, \ldots, x_n$. This is done in the following way. First, we compute $h_1(x_i)$ and $h_2(x_i)$ for state $x_i$. If the bit addressed by $h_1(x_i)$ or the bit addressed by $h_2(x_i)$ is 0, then $x_i$ is newly generated and inserted into the hash table by setting bits $h_1(x_i)$ and $h_2(x_i)$ to 1. Otherwise, $x_i$ is considered as already generated and the hash table remains unchanged. If the insertion is successful, we say that the state has been reached, otherwise not. This reflects the fact that in the latter case state $x_i$ is considered as a duplicate and no verification test is done. This simplification ignores the fact that in the case of $h(x_i) = 1$ (i. e. $x_i$ is considered as a duplicate) the search tree is pruned and none of the successors of $x_i$ are generated. As a result the expected coverage of a single run is possibly too high. This is probably also true for the combined coverage of sequential hashing.

Let $E(i,k)$ represent the event that the hash table has $i$ entries after the $k$-th insertion try (of state $x_k$). Then event $E(i,k)$ follows from event $E(i-2,k-1)$, if the bits addressed by $h_1(x)$ and by $h_2(x)$ are both 0, before $x$ is inserted, or from event $E(i-1,k-1)$, if either bit $h_1(x)$ or bit $h_2(x)$ is 0 and the other 1, or from event $E(i,k-1)$, if both bits are set to 1. In the latter case the insertion of $x_k$ fails and in the other cases the insertion of $x_k$ has success. This gives the following recurrence equation.

$$\begin{aligned} p\{E(i,k)\} &= p\{E(i,k)|E(i,k-1)\} \cdot p\{E(i,k-1)\} \\ &\quad + p\{E(i,k)|E(i-1,k-1)\} \cdot p\{E(i-1,k-1)\} \\ &\quad + p\{E(i,k)|E(i-2,k-1)\} \cdot p\{E(i-2,k-1)\} \end{aligned} \tag{2}$$

for $i \geq 0, k > 0$ and with initial values $p\{E(0,0)\} = 1$, $p\{E(i,0)\} = 0$, for $i > 0$, and $p\{E(-1,k)\} = p\{E(-2,k)\} = 0$ for $k \geq 0$. We may assume that for a given state $x$ first, it is tested whether $h_1(x)$ is 0 and then, it is tested whether $h_2(x)$ is 0. It is $p\{E(i,k)|E(i,k-1)\} = (i/m)^2$ (since there are $i$ occupied hash positions), $p\{E(i,k)|E(i-2,k-1)\} = (m-i+2)(m-i+1)/m^2$ (since there are $(m-i+2)$ free positions for $h_1$ and $(m-i+1)$ free positions for $h_2$). Further, we have $p\{E(i,k)|E(i-1,k-1)\} = (m-i+1)i/m^2 + (i-1)(m-i+1)/m^2$, since we have two possibilities: either bit $h_1(x)$ is 0 and bit $h_2(x)$ is 1 (including the case that $h_1(x) = h_2(x)$) or bit $h_2(x)$ is 1 and bit $h_2(x)$ is 0. Thus, (2) is replaced by

$$\begin{aligned} p\{E(i,k)\} &= (i/m)^2 \cdot p\{E(i,k-1)\} \\ &\quad + ((m-i+1)i + (i-1)(m-i+1))/m^2 \cdot p\{E(i-1,k-1)\} \\ &\quad + (m-i+2)(m-i+1)/m^2 \cdot p\{E(i-2,k-1)\} \end{aligned} \tag{3}$$

Let $p(i,k)$ denote the probability that $x_k$ is inserted (with the $k$-th insertion operation) under the precondition that the hash table has $i$ entries after the $k$-th insertion. These values can be computed by

$$
\begin{aligned}
p(i,k) &= p\{E(i,k)|E(i-1,k-1)\} \cdot p\{E(i-1,k-1)\} \\
&\quad + p\{E(i,k)|E(i-2,k-1)\} \cdot p\{E(i-2,k-1)\} \\
&= ((m-i+1)i + (i-1)(m-i+1))/m^2 \cdot p\{E(i-1,k-1)\} \\
&\quad + (m-i+2)(m-i+1)/m^2 \cdot p\{E(i-2,k-1)\} \qquad (4)
\end{aligned}
$$

Similarly as above, the probability $p(k)$ that $x_k$ is inserted is therefore given by the sum of all probabilities $p(i,k)$, that is $p(k) = \sum_{i=1}^{k} p(i,k)$. The expected number of reached elements, provided that the elements are inserted with respect to the order $(x_1, \ldots, x_n)$, is $\sum_{k=1}^{n}(1 \cdot p(k))$ and the expected coverage after $n$ insertions, denoted by $\alpha(n)$, is given by:

$$
\alpha(n) = \frac{1}{n} \cdot \sum_{k=1}^{n} p(k). \qquad (5)
$$

A comparison between the expected coverage of a single run computed with Holzmann's equations and (5) is given in figure 6. While both Holzmann's upper bound and our estimation measure the coverage (of DTP-protocol) too optimistically ([Hol95]), our estimation is more accurate. Note that our estimation predicts that bitstate hashing is better than double bitstate hashing if the state size is greater than the table size (more precisely, if $n \geq 0.965 \cdot m$). This reflects quite accurately the empirical results given for DTP-protocol in ([Hol95]).

#### 4.2.2 Analysis of Sequential Hashing

Let us compute the combined coverage of $q$ runs. The probability that state $x_k$ is not reached in $q$ runs is $(1 - p(k))^q$ (provided that the hash values are independent). Hence, the probability that $x_k$ is reached in $q$ runs is given by $p_q(k) = 1 - (1 - p(k))^q$. As a result, the expected coverage is given by

$$
\alpha_q(n) = \frac{1}{n} \sum_{k=1}^{n} p_q(k) = \frac{1}{n} \sum_{k=1}^{n} (1 - (1 - p(k))^q) \qquad (6)
$$

Of course, each state $x$ is generated with probability $1 - \epsilon$ if a large enough value of $q$ is chosen. But in practice this can require an unacceptable amount of time.

The following diagrams summarize a comparison between the expected coverage of $q$ runs computed with (1) and with (10). For fair comparison, we have to replace $s/n$ in (1) by $\alpha(n)$ the expected coverage of a single run. Figure 7 and 8 shows how the number of reached states grows in dependence of the state space size and the number of runs. The coverage measured with (1) is much higher than when computed with (6). This contradicts the assumption that the states can be considered as selected

| | $m$ | $n$ | $\alpha(n)$ (single bitstate hashing) Holzmann | Our approach | $\alpha(n)$ (double bitstate hashing) Holzmann | Our approach |
|---|---|---|---|---|---|---|
| 1. | 1024 | 256 | 0.8754,— | 0.8851 | 0.9173,— | 0.9419 |
| 2. | 1024 | 511 | 0.7509,— | 0.7872 | 0.6613,— | 0.8321 |
| 3. | 1024 | 1023 | 0.5009,— | 0.6325 | 0.5014,— | 0.6197 |
| 4. | 1024 | 2047 | —,0.5002 | 0.4323 | 0.2501,0.5002 | 0.3660 |
| 5. | 1024 | 4095 | —,0.2500 | 0.2454 | 0.1250,0.2500 | 0.1874 |
| 6. | 1024 | 8191 | —,0.1250 | 0.1249 | 0.0625,0.1250 | 0.0937 |
| 7. | 8192 | 511 | 0.9688,— | 0.9695 | 0.9948,— | 0.9952 |
| 8. | 8192 | 1023 | 0.9376,— | 0.9401 | 0.9792,— | 0.9827 |
| 9. | 8192 | 2047 | 0.8751,— | 0.8848 | 0.9168,— | 0.9418 |
| 10. | 8192 | 4095 | 0.7501,— | 0.7870 | 0.6669,— | 0.8319 |
| 11. | 8192 | 8191 | 0.5001,— | 0.6321 | 0.5000,— | 0.6193 |

Figure 6: Comparison of the expected coverage of a single run with supertrace estimated with Holzmann's equations and our approach ("—" indicates when either no upper or no lower bound is defined and is to be replaced by the trivial bound 1 or 0 resp.).

randomly from the set of all reachable states. Figure 7 corresponds with figure 3 since increasing the threshold increases the number of reachable states. The figure shows that coverage grows slowly while the state space grows exponentially with the search depth. In the best case ($n \gg m$) doubling the table size doubles the coverage (as measured for class $\mathcal{H}_2$).

#### 4.2.3 Relation to SPIN's Hash-factor

SPIN uses a hash-factor which provides some information about how reliable bitstate hashing works. This hash-factor is given by

$$\frac{m}{N}.$$

If SPIN is compiled with the option –DCOVEST, then SPIN prints an estimated coverage after the end of each supertrace run. This value is given by

$$N \cdot \frac{\log(1 - 1/m)}{(\log(1 - N/m)},$$

where $N$ is the number of states actually reached and $m$ the hash table size. The hash-factor of SPIN is used to indicate how reliable this estimation is.

A good confidence level is given by a hash-factor greater than 100 in case of double bitstate hashing and by a hash-factor greater than 1000 in case of single bitstate hashing [Hol95]. In protocols of practical interest, however, a high confidence level

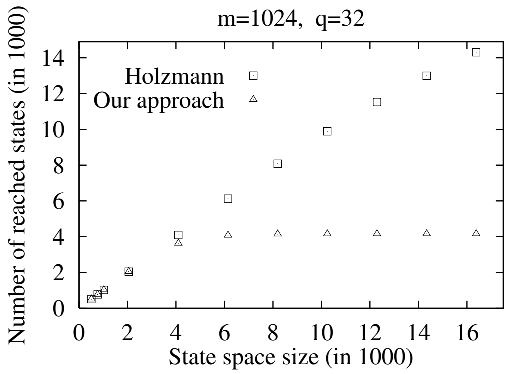


Figure 7: Holzmann's versus our approach: Expected coverage of sequential hashing in dependence of $n$.

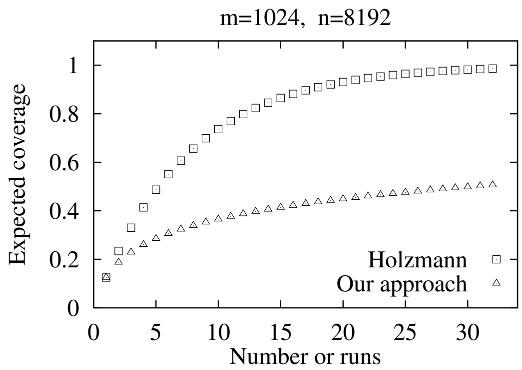


Figure 8: Holzmann's versus our approach: Expected coverage of sequential hashing in dependence of $q$.

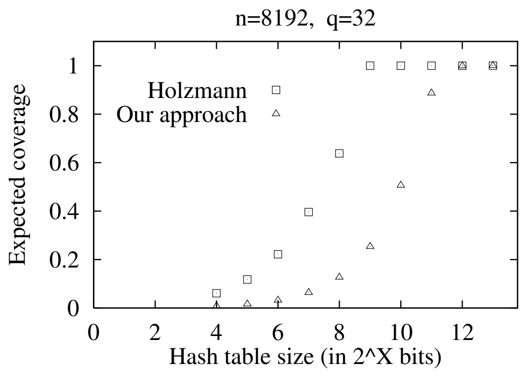


Figure 9: Holzmann's versus our approach: Expected coverage in dependence of hash table size.

| $m$ | $N$ | $m/27183$ | real cov. | estimated cov. | hash-factor | confidence |
|---|---|---|---|---|---|---|
| $2^{12}$ | 2460 | 0,15 | 9,04% | 65,4% | 1,66 | low |
| $2^{15}$ | 15734 | 1,20 | 57,88% | 73,4% | 2,08 | low |
| $2^{17}$ | 25531 | 9,64 | 93,9% | 89,9% | 5,13 | low |
| $2^{19}$ | 27054 | 19,28 | 99,5% | 97,4% | 19,37 | medium |
| $2^{20}$ | 27111 | 36,57 | 99,7% | 98,7% | 38,67 | medium |
| $2^{21}$ | 27146 | 77,14 | 99,8% | 99,4% | 77,25 | medium |
| $2^{22}$ | 27174 | 154,29 | 99,9% | 99,7% | 154,34 | good |

Figure 10: Real coverage versus estimated coverage by SPIN. The supertrace run was performed with threshold 100. ($N$ denotes here the number of states actually reached by SPIN. The size of the state space is 27183.)

is seldomly achieved. For example, in case of the PFTP-protocol a hash table size of $2^{22}$ (measured in bits) is needed a good coverage to be achieved for a double-bitstate run with threshold 100. That is, hash table size $2^{22}$ contrasts with 27183 reachable states. The table in figure10 shows the hash-factor, the coverage estimated by SPIN and the real coverage. It seems that if the hash-factor indicates a high confidence level then the coverage is actually close to 1 and then the coverage estimated by SPIN is too pessimistic. If the state space size is greater or equal than the table size, then the hash-factor indicates that coverage estimated by SPIN is not to be trusted and indeed, then the coverage is much too optimistic. Our model confirmes this tendency. Figure 11 shows how the hash-factor is correlated with coverage estimated by SPIN. For comparison we have to replace $N$ by the expected coverage computed with (5).

| $m$ | $n$ | $m/n$ | expected cov. | estimated cov. | hash-factor | confidence |
|---|---|---|---|---|---|---|
| 100 | 1000 | 0,1 | 7,50% | 54,37% | 1,33 | low |
| 100 | 100 | 1 | 62,04% | 64,36% | 1,61 | low |
| 500 | 100 | 5 | 96,05% | 90,14% | 5,20 | low |
| 1000 | 100 | 20 | 98,86% | 95,01% | 10,11 | medium |
| 2000 | 100 | 20 | 99,69% | 97,51% | 20,06 | medium |
| 4000 | 100 | 40 | 99,92% | 98,75% | 40,03 | medium |
| 8000 | 100 | 80 | 99,97% | 99,38% | 80,01 | medium |
| 16000 | 100 | 160 | 99,99% | 99,69% | 160,00 | good |

Figure 11: Expected coverage versus estimated coverage by SPIN. ($n$ denotes here the number of reachable states.)

# 5 Universal Supertrace

## 5.1 Theory

The hash function is used as a parameter which is changed for each run of sequential hashing with supertrace while the traversal order in which the states are generated remains unchanged. Thus, the question is: Can the combined coverage of sequential hashing be increased if in each run both the hash function and the traversal order of the states are changed. The answer is Yes.

Recall that universal hashing, for example, follows a general principle in order to get a good performance, that being a good distribution of the inserted keys. This principle suggests not to use a single algorithm for solving a class of problems, but rather a suitable set of algorithms should be provided from which one particular algorithm is chosen randomly in order to solve a given problem instance. This method is able to give a bound, which is independent of the input, for the average performance of the class of algorithms. Such a bound is valuable when it is better than the performance of any known single algorithm in that algorithm's worst case. We follow a similar principle to get a better coverage of bitstate hashing.

Let $S_n$ be the permutation group of $n$ elements and the sequence $x_1, \ldots, x_n$ be an explicit enumeration of all the reachable states. Let $S \subset S_n$ be a subset. Then we assign each $\sigma \in S$ a sequence $A_\sigma$: $x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ which represents the order in which the states are traversed. Note that there are permutations $\sigma \in S_n$ which do not correspond to a possible traversal order of an exploration tree, since a state $x$ is always visited before its successor states will be visited. Let $\mathcal{A}_S$ be the class of all the sequences $A_\sigma$, $\sigma \in S$.

A natural way of changing the traversal order is as follows: Fix a subset $S$ and a class of hash functions $\mathcal{H}$ and for each new run randomly choose both the hash function $h \in \mathcal{H}$ and the permutation $\sigma \in S$ (with equal probabilities) and explore the state space with respect to traversal order $A_\sigma$ and hash function $h$. We compute below the expected coverage when this strategy is applied to sequential hashing.

First, let $q = 1$: It is easy to see that for a particular run the expected coverage, denoted by $\alpha_{S,1}$, is equal to the coverage of the original supertrace. The coverage is given by the sum $\sum_{k=1}^{n} p(\sigma(k))$ provided that $\sigma \in S$ is selected. The expected coverage averaged over all permutations is

$$\alpha_{S,1}(n) = \frac{1}{|S|}\sum_{\sigma\in S}\frac{1}{n}\sum_{k=1}^{n} p(\sigma(k)) = \frac{1}{|S|}\sum_{\sigma\in S}\frac{1}{n}\sum_{k=1}^{n} p(k) = \frac{1}{|S|}\sum_{\sigma\in S}\alpha(n) = \alpha(n) \quad (7)$$

Note that the expected probability that the state $x_k$ is visited is given by

$$\frac{1}{|S|}\sum_{\sigma\in S} p(\sigma(k)). \quad (8)$$

Let $q > 1$: Let us compute the expected coverage for $q$ runs. Since we assume that the hash functions are independent, the probability that a selected state $x_k$ is reached in at least one run, if the permutations $\sigma_1, \ldots, \sigma_q$ are selected, is given by

$$1 - (1 - p(\sigma_1(k))) \cdot (1 - p(\sigma_2(k))) \cdot \ldots \cdot (1 - p(\sigma_q(k))) \quad (9)$$

The expected coverage of $q$ runs if the sequence $\sigma_1, \ldots, \sigma_q$ is selected is

$$\frac{1}{n}\sum_{k=1}^{n}\left(1 - (1 - p(\sigma_1(k))) \cdot (1 - p(\sigma_2(k))) \cdot \ldots \cdot (1 - p(\sigma_q(k)))\right). \quad (10)$$

The expected coverage averaged over all sequences of $q$ permutations $(\sigma_1, \ldots, \sigma_q) \in S^q$ provided that each sequence is chosen randomly with equal probability, here denoted by $\alpha_{S,q}$, is given by

$$\alpha_{S,q}(n) = \frac{1}{n}\sum_{k=1}^{n}\left(1 - (1 - \frac{1}{|S|}\sum_{\sigma\in S} p(\sigma(k)))^q\right). \quad (11)$$

that is, (11) is gained from (6) when $p(k)$ is replaced by (8). We omitt the proof and refer the interested reader to [EL98].

*Optimization problem:* The problem now is to find a suitable subset $S$ such that the combined coverage of $q$ runs would be maximized. An upper bound for the expected coverage is given by the following optimization problem under constraints. Consider the system

$$f(y_1, \ldots, y_n) = \frac{1}{n}\left(\sum_{j=1}^{n} y_j\right) - c \quad (12)$$

$$g(y_1, \ldots, y_n) = \frac{1}{n}\sum_{j=1}^{n}\left(1 - (1 - y_j)^q\right) \quad (13)$$

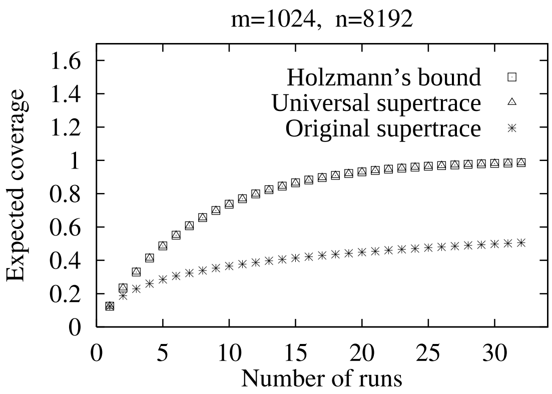


Figure 12: Expected coverage of universal supertrace versus original supertrace.

where $y_j \in [0,1]$, $1 \leq j \leq n$. A maxima of $g$ under the condition $f = 0$ is given by $y_j = c, 1 \leq j \leq n$. This optimization problem applied to

$$f(y_1, \ldots, y_n) = \frac{1}{n}(\sum_{j=1}^{n} y_j) - \frac{1}{n}(\sum_{j=1}^{n} p(j)) \tag{14}$$

$$g(y_1, \ldots, y_n) = \alpha_{S,q}(n) \tag{15}$$

where $y_j = (1/|S|)(\sum_{\sigma \in S} p(\sigma(j)))$, $1 \leq j \leq n$ yield a maxima for

$$y_j = (1/n)(\sum_{j=1}^{n} p(j)) = \alpha(n) = \alpha_{S,1}(n),\ 1 \leq j \leq n.$$

This maxima corresponds with the (impossible) choice $S = S_n$. Then $\alpha_{S,q}(n)$ is equal to (1). We can conclude that (1) is an upper bound for the expected coverage to be achieved by such randomization methods.

But how to select $S$ in practice? A natural way of changing the traversal order is as follows: Keep the depth-first order, but for each reached state $x$ change randomly the order in which the immediate successor states of $x$ are visited. Let us call this strategy combined with (double) bitstate hashing *universal supertrace*. We have computed the expected coverage in the case that the exploration tree has a constant branching factor $b$ (that is, that each state has exactly $b$ immediate successor states). Surprisingly, the expected coverage comes very close to the upper bound (1). The higher the branching factor the higher the coverage, but even for $b = 2$ the coverage is very close to the upper bound (see figure 12).

| depth | measured coverage | coverage/transitions |
|---|---|---|
| 100 | 0.9989 | 0.9849 |
| 150 | 2.1452 | 2.0614 |
| 200 | 4.3391 | 4.6981 |
| 300 | 8.7905 | 12.7175 |
| 400 | 8.1253 | 12.0768 |
| 500 | 8.3390 | 14.2580 |

Figure 13: Comparison of expected coverage of universal supertrace versus original supertrace (for $q = 64$, $m = 2^{16}$). The measured coverage of universal supertrace relative to that of original supertrace.

## 5.2 Implementation and Experimental Results

We have implemented our new algorithm, *universal supertrace*, in the context of the SPIN validator. Fortunately, only slight extensions of the original supertrace were necessary. The main extension consisted of a function which allowed the order in which the immediate successor states are visited to be changed randomly. In SPIN each state refers to a chain of possibly applicable transitions. This chain reflects the order in which the successor states are generated. We included in SPIN a function which generates pointers to the transitions of the actual transition chain and changes the order randomly. In contrast to original supertrace, universal supertrace generates the successors in respect to this changed order.

It must be guaranteed that the order in which the successors of a state $x$ are generated can be restored if the search backtracks to $x$ in order to continue the search with the next successor of $x$. In the original supertrace, this order is fixed during the search. Thus it is sufficient to store only a pointer to the transition to be performed next on the depth-first stack. In contrast to this, universal supertrace changes for each state $x$ the original order of the transitions to be applied to $x$. For this reason, for each state $x$ the new transition order has to be stored on the depth-first stack. This can be done in a space efficient way. We use a random function which is called up with a randomly generated initial value for each state $x$ on its first visit in order to generate a new order in its immediate successors: equal initial values lead to equal transition orders. It then suffices to store only those initial values with which the random function has been called up in order to generate the order of the successors to be applied to $x$. This leads to an additional space requirement of $O(d)$ where $d$ is the search depth.

SPIN allows to use partial-order-search method based on persistent-sets [God96]. Note, it is possible to combine the method for changing the traversal order as described above with the partial-order method used in SPIN, since not the order in which the processes are selected, but only the order in which the transitions of a selected process to be performed is changed.

| depth | measured coverage | coverage/transitions |
|---|---|---|
| 100 | 0.9989 | 0.9913 |
| 150 | 1.3741 | 1.2904 |
| 200 | 2.9180 | 2.9588 |
| 300 | 8.4193 | 10.8444 |
| 400 | 6.6697 | 9.5129 |
| 500 | 6.7667 | 10.8152 |

Figure 14: Comparison of expected coverage of universal supertrace versus original supertrace combined with partial-order reduction (for $q = 64$, $m = 2^{16}$). The measured coverage of universal supertrace relative to that of original supertrace.

For evaluation, we made several tests with different parameters by using CRC-hashing without partial-order reduction. It seems that the maximal factor of improved coverage achieved with universal supertrace grows when the number of runs and the search depth are increased. For example, universal supertrace improves the coverage by a factor of up to 4.9095 compared with the original one for $q = 32$, $m = 2^{14}$ and by a factor of up to 8.7905 for $q = 64$, $m = 2^{16}$ (see figure 13). If we compare the ratio of reached states to performed transitions (coverage/transitions) as a measure of how quickly new states are generated, the improvement is still higher. For example, in the first case this factor is increased of up to 5.9739 and in the second case by up to 14.2580 (see figure 13). Similar results are obtained if additionally the partial-order reduction method in SPIN is combined with universal supertrace.

Similar tests have been made with the partial-order-reduction method used in the SPIN-tool. Figure 14 corresponds with presents figure 13 where the results are given for the same parameters.

*Hash compaction with randomization:* An iteresting question is how well the randomization of the traversal order works if combined with other partial search methods, e. g. hash compaction. We made some test with hash compaction combined with open hashing (double hashing) and found that randomization is also useful. But the improvement factor is much smaller than given in case of bitstate hashing. This question may be the subject of our future research.

# 6 Conclusion

This paper is intended to give better insight into the quality of bitstate and sequential hashing. We have presented an analysis of the expected coverage of bitstate and sequential hashing which is more applicable and much more accurate than former approaches ([Hol91], [Hol95]). Our experimental tests and our analysis shows that the quality of the original supertrace suffers as a result of the fixed order traversal used in supertrace. We have presented a new algorithm, universal supertrace, which randomly changes in successive runs both the hash function and the traversal order. Theoretical and experimental results show that universal supertrace outperforms the original supertrace ([Hol87], [Hol88]) and improves the measured coverage by a factor of up to 8-10.

We expect that other strategies for changing the traversal order will have potential for additional improvement which may be the subject of our future research. The mathematical analysis developed in this paper can be applied in a similar way to other partial search methods, such as hashcompact ([SD96], [UD96]), and would help to clarify under which circumstances hash compact is to be preferred.

## Acknowledgment

This research is supported by the DFG-project "Heuristische Suche und Anwendung auf die Protokollvalidation", No. Ot-64/11-1.

## References

[Coh97] Jonathan D. Cohen. Recursive hashing functions for $n$-grams. *ACM Transactions on Information Systems*, 15(3):291–320, jul 1997.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[EL98] J. Eckerle and Th. Lais. Limits and possibilities of sequential hashing with supertrace. In S. Budkowski and A. Cavalli, editors, *Proceedings of the 11th Formal Description Techniques & 18th Int. Conf on Protocol Specification, Testing, and Verification (FORTE/PSTV'98)*, Paris, France, november 1998. Chapman & Hall. To appear.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996.

[Hol87] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C. West, editors, *Proc. 6th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Zurich, Sw., June 1987. (Invited paper).

[Hol88] Gerard J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2):137–161, February 1988.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Hol95] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314, Warsaw, Poland, jun 1995. Chapman & Hall.

[SD96] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In R. Gotzheim and J. Bredereke, editors, *Proc. 16th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 333–348, Kaiserslautern, Germany, october 1996. Chapman & Hall.

[UD96] U. Stern and D.L. Dill. Combining state space caching and hash compaction. In Bernd straube and Jens Schoenherr, editors, *4. GI/ITG/GME Workshop zur Methoden des Entwurfs und der Verifikation Digitaler Systeme*, Berichte aus der Informatik, pages 81–90, Kreischa, March 1996. GI/ITG/GME, Shaker Verlag, Aachen.