Combining State Space Caching and Hash Compaction

Ulrich Stern* and David L. Dill

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

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

Abstract

In verification by explicit state enumeration, for each reachable state the full state descriptor is stored in a state table. Two methods – state space caching and hash compaction – that reduce the memory requirements for this table have been proposed in the literature. In state space caching, "old" states are replaced by newly reached ones once the table fills up, which might increase the run-time requirements for verification. In hash compaction, introduced by Wolper and Leroy and improved upon by Stern and Dill, a compressed state descriptor is stored instead of the full one. Here, the memory savings come at the price of a small probability that not all reachable states will be explored during the state enumeration.

In this paper, we propose and analyze a new scheme to combine state space caching and hash compaction. In the new scheme, an open addressing collision resolution scheme with a limit on the number of probes in the state table is employed. The new scheme saves roughly 60% in memory in comparison with Wolper and Leroy's proposed combination of state space caching and hash compaction. Furthermore, we propose a new implementation that reduces the memory overhead for sleep sets, which are usually used to increase the efficiency of state space caching. The outcomes of experiments using the new scheme confirmed the analysis.

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) [1].

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 in our group for some types of industrial protocols, explicit state enumeration has out-performed the symbolic approach [8].

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

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. Both, *state space caching* and *hash compaction* aim at reducing the memory requirements for this table.

State space caching is based on the observation that when performing a depth-first search to enumerate the state space, the *only* purpose of the state table is to speed up the verification by preventing double work when states are re-visited. Thus, one can remove "old" states from the table and *replace* them by newly reached ones once the table fills up. Unfortunately, the run-time requirements increase dramatically when the size of the state table declines beyond a certain point; for typical protocols one can only achieve a roughly two-fold reduction in the size of the state table. However, impressive results were reported by Godefroid, Holzmann and Pirottin [6], when they combined state space caching with the sleep set method [4] which aims at reducing the number of paths to each state.

Like state space caching, the hash compaction method introduced by Wolper and Leroy [11] aims at reducing the memory requirements for the state table. However, it is not the number but the size of the slots in the table that is cut down. Instead of the full state descriptor, a compressed state descriptor – which is calculated with a hash function – is stored in the state table. This method typically saves a factor of 10 in memory but comes at the price of a small probability that not all reachable states will be explored during the search, i.e. that the algorithm will omit even one state. This probability will be called *omission probability*.

An improvement of the basic hash compaction scheme was introduced by Stern and Dill [9]. On the one hand, the improved scheme uses open addressing instead of chaining to resolve collisions in the hash table. Chaining requires storing an additional pointer besides the compressed state. By not having to store this pointer in open addressing, the memory requirements for a state decrease typically by 30%. On the other hand, the probe sequence of a state is calculated independently from the compressed value, reducing the memory requirements for a state by another 30%. In the improved scheme, the omission probability of a particular state is roughly proportional to the length of the probe sequence used during the insertion.

In this paper, we propose and analyze a new scheme to combine the most efficient versions of state space caching and hash compaction. This scheme limits the number of probes in the state table during insertions and replaces a state from the probe sequence when no empty slot is found. We will call this policy *t-limited scheme*, where t denotes the maximum number of probes during an insertion. While this scheme can reduce the omission probability since it avoids long probe sequences, the replacements now start before the table is completely full, which might increase the run-time requirements. However, practical results show that there is only a negligible increase in run-time even for small values of t. The analysis of the new scheme shows, for example, that when a 1-limited algorithm explores 800 million states with a 400-megabyte hash table using 5-byte compression, the omission probability is bounded by

0.07%. In comparison with Wolper and Leroy's proposed combination of hash compaction and state space caching [11], the new scheme saves typically 60% in memory. These savings correspond with the 60% savings of the improved hash compaction scheme in comparison with the previous one.

In addition, we address the problem of how to efficiently store sleep sets. Sleep sets are usually used in combination with a technique called "persistent sets" that aims at reducing the number of states that have to be explored for verifying certain properties [5]. When sleep sets and persistent sets are used together, one has to store the associated sleep set along with each state in the state table. In the new implementation we propose, each different sleep set is only stored once, since we observed that the number of different sleep sets is usually several orders of magnitude smaller than the number of reachable states. Thereby, the memory overhead of the sleep sets is reduced to typically one or two bytes per state.

For our implementation we have chosen Holzmann's SPIN system [7] with Godefroid's add-on partial order package, which already provides sleep sets. We experimented with a subset of the protocols used by Godefroid in his PhD thesis. The observed omission probabilities in these experiments nicely matched the predictions of our theoretical analysis. Furthermore, our experiments show that one can limit the number of probes to a very small number, say 3, with almost no increase in run-time in comparison to a higher value of t (which approximates conventional state space caching).

This paper is organized as follows. Section 2 and Section 3 review state space caching and hash compaction, respectively. The new scheme is presented and analyzed – along with some remarks about choosing the parameters – in Section 4. In Section 5, we describe the new method to efficiently store sleep sets. Section 6 contains some results of the new scheme on sample protocols. Finally, in Section 7 we state some conclusions and suggestions for future work.

2 State Space Caching

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 [11].

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.

State Space Caching

State space caching is a method to reduce the memory requirements for the state table by not storing all reachable states. It is based on the observation that a *depth-first* search algorithm would still explore every reachable state if it only used the state queue and maintained no state table. However, the run-time requirements would increase dramatically, since every reachable state would be visited via every possible path to this state, and in a typical protocol there are usually many different paths to a state. Hence, in state space caching one still uses a state table that is as large as possible. Once this table fills up, every newly reached state *replaces* a state that was already in the table.

Employing state space caching, one still experiences a run-time blow-up when the size of the state table declines beyond a certain point. Unfortunately, for typical protocols, one can only achieve roughly a two-fold reduction in the size of the state table. However, there is a technique that aims at reducing the number of paths to each state, the "sleep sets" introduced by Godefroid [4]. Using sleep sets makes state space caching far more efficient [6]. For some protocols, a more than 100-fold reduction in the size of the state table is possible, while the run-time only increases 3-fold. Thus, since our goal is to combine efficient versions of state space caching and hash compaction, we also employed Godefroid's sleep sets in our implementation.

Note that when an algorithm using state space caching terminates, it can only tell us how many different reachable states of the protocol it *observed*, which is the number of insertions – with or without replacement – into the state table. We will denote this observed number of reachable states by n. Clearly, this number will be greater or equal to the (real) number of reachable states of the protocol, since every state is visited at least once when using state space caching. Furthermore, since the run-time of the state space caching algorithm is approximately proportional to the number of insertions into the state table, we can judge the increase in run-time when using state space caching by comparing the observed number of reachable states with the real number (if the latter is available). We will also say that a state space caching algorithm *performs* well if the observed number of reachable states is close to the real number.

3 Hash Compaction

Basic Scheme

The "hash compaction" technique was introduced by Wolper and Leroy [11]. Like state space caching, hash compaction aims at reducing the memory requirements for the state table. However, not the number but the size of the slots in the table is cut down. Instead of the full state descriptor, a compressed state descriptor – which is calculated with a hash function – is stored in the state table. However, these savings come at the price of a small probability that not all reachable states will be explored during the search, i.e. that the algorithm will omit even one state. This probability will be called *omission probability*.

A compression function c is used to obtain for each state s a compressed value $c(s) \in \{0, \ldots, l-1\}$ to be stored in the table. Here, l denotes the number of all possible compressed values. If we use b bits for these values, clearly $l = 2^b$.

In practice, one can use universal hashing [2] to calculate the compressed value from the state descriptor, as suggested by Wolper and Leroy [10]. Then, the probability that two different states have the same compressed value is bounded, namely $\Pr(c(s_1)=c(s_2)) \leq \frac{1}{l}$, for all $s_1, s_2 \in S$, $s_1 \neq s_2$. Here, S denotes the set of all possible states. A consequence of this bound is that the omission probability will also be bounded.

Improvements of the Basic Scheme

Typically, the state table is implemented as a hash table. One problem in hashing is the occurrence of 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 [3]. Chaining requires storing an additional pointer besides the compressed state. However, a pointer has roughly the same size as the compressed state and with that the memory requirements would approximately double. Therefore, open addressing is preferable over chaining for hash compaction.

In open addressing, a vectorial hash function $\underline{h}$ is applied to each state s yielding a probe sequence $h_0(s), h_1(s), \ldots, h_{m-1}(s)$, where m denotes the number of slots in the hash table and $h_i(s)$ is an index into the table. When inserting a state in the table, the slots are tested for emptiness according to this probe sequence. The compressed state $c(s)$ 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.

The state insertion process is depicted in Figure 1. A probe can yield three different results:

1. 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.

2. 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 $\underline{h}(s)$.

3. 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.

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). Thus, if one calculates probe sequence and compressed value independently, the same compressed value can be stored at several locations without leading to omissions.

The improved hash compaction scheme – using open addressing and an independent calculation of probe sequence and compressed value – was presented and analyzed by Stern and Dill [9]. With the improved scheme, it is possible,

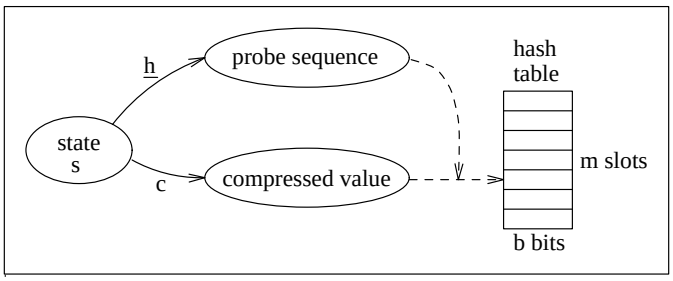

Figure 1: The state insertion process in the hash compaction scheme

for example, to store 80 million states in a hash table of size 400 megabytes with an omission probability smaller than 0.13% using 40-bit compression.

4 Combining State Space Caching and Hash Compaction

The t-limited Scheme

In conventional state space caching the replacement of states starts as soon as the state table fills up completely. However, this policy leads to some difficulties when we still want to use the open addressing scheme for resolving collisions. Open addressing performs very poorly when the table is full, since then every slot has to be probed to determine that a newly reached state is not already in the table. Additionally, the omission probability for newly reached states becomes very large.

To avoid these problems, we propose a modified insertion process, in which one probes at most t slots during the search for an empty slot. If all t slots are occupied and contain different values from the one to be entered, one replaces the compressed value in one of these t slots by the compressed state to be entered. We call this policy *t-limited scheme*. Clearly, when all insertions are performed with t-limited probe sequences, t-limited sequences are also sufficient when searching for a state.

The effect of the t-limited scheme is that replacements in the state table will also occur with some probability when the table is not completely full. The bigger the parameter t, the more unlikely are these early replacements. However, since replacements will occur in state space caching in any case, one would intuitively expect that a small number for t would be sufficient for ensuring the same performance (same observed number of states) as in conventional state space caching.

Calculating the Omission Probability

An approximation formula for the omission probability p_{om} of the t-limited scheme can be calculated (2-page derivation omitted) yielding

$$p_{\text{om}} \quad := \quad \Pr(\text{even one omission during the state enumeration})$$

$$\approx \quad 1 - \left(\frac{l-1}{l}\right)^{C} \quad , \tag{1}$$

where

$$C = \begin{cases} \dfrac{t}{t+1}\, n \left(\dfrac{n}{m}\right)^{t} + \displaystyle\sum_{j=0}^{t-1} j \left(\dfrac{n}{m}\right)^{j+1} \left(\dfrac{m}{j+1} - \dfrac{n}{j+2}\right) & \text{if } n \leq m \\ (H_{t+1} - 1)\, m + t\,(n-m) & \text{if } n > m \quad . \end{cases}$$

Here, $H_n = \sum_{i=1}^{n} \frac{1}{i}$ denotes a harmonic number. The given expression for C was simplified for the case in which $n, m \gg t^2$. Remember, that n denotes the observed number of states reported by the verifier, m the hash table size and $l = 2^b$ the number of all possible compressed values.

Choosing the Parameters

Let us assume a hash table size of 400 megabytes. Not coincidentally, this number corresponds well with the available DRAM on our largest machine. Further assume that we use 5-byte compression (b=40) and that we can hence enumerate a maximum of 80 million states with the previous hash compaction scheme without state space caching.

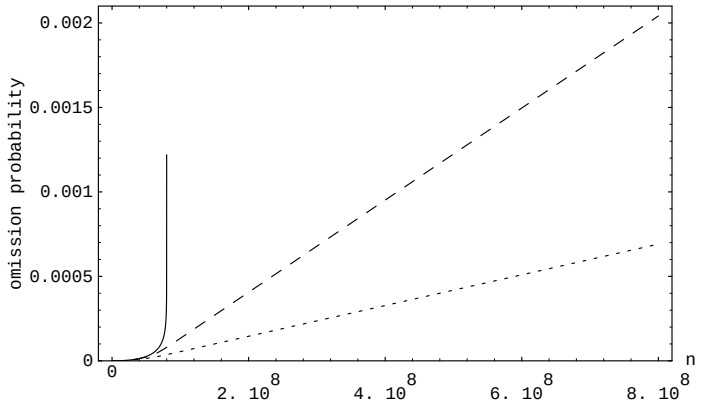

Figure 2: Omission probability for 5-byte compression for the previous hash compaction scheme (solid) and for the t-limited scheme with t=3 (dashed) and t=1 (dotted)

Figure 2 depicts the omission probability as a function of the observed number of states n. Note that the previous hash compaction scheme shows a sharp increase in the omission probability when the table fills up completely. This increase is avoided by the t-limited scheme. A 1-limited scheme can, for example, explore 800 million (observed) states with only 0.07% omission probability.

In [9] it was shown that adding one bit to the compressed values roughly halves the omission probability. The corresponding result holds also for the t-limited scheme. Thus, when using 4-byte compression the graphs would have approximately the same shape as the ones in Figure 2. The main difference would be roughly 256 times bigger omission probabilities.

Further note, that the omission probability is roughly proportional to the parameter t. The trade-off in choosing t will be discussed in Section 6.

5 Implementation

The calculation of the probe sequence and the compressed value out of the state descriptor was performed using the H_3 universal$_2$ class of hash functions [2]. More details about this calculation can be found in [9].

Another interesting part of the implementation is the storage of the sleep sets. A sleep set is a set of transitions associated with a state. Sleep sets are usually used in combination with a technique called "persistent sets" that aims at reducing the number of states that have to be explored for verifying certain properties [5]. When sleep sets are employed alone, it is sufficient to store them in the state queue. In combination with persistent sets, however, for every state in the state table the associated sleep set has to be stored. Hence, the sleep sets might significantly contribute to the memory requirements.

Table 1 compares the memory requirements per state for the sleep sets in the previous implementation[1] with a new implementation for three sample protocols. PFTP is a file transfer protocol presented in [7], Mulog is a mutual exclusion algorithm and Abra is a protocol in which four processes communicate via FIFO channels. In the previous implementation, for each transition in a sleep set associated with a state, a pointer with 32 bits is stored. Thus, the memory requirements for a state vary dependent on the size of the associated sleep set. In the new implementation, however, a constant slot size in the state table is required since open addressing is used.

Table 1: Memory requirements per state for sleep sets in sample protocols

		previous implementation		new implementation	
protocol	reachable states	transitions in sleep sets	bits/state	different sleep sets	bits/state
PFTP	250 514	250 398	32.0	89	7
Mulog	17 984	30 755	54.7	1 037	11
Abra	36 913	4 392	3.8	12	4

A solution to this problem can be found by observing that there are usually several orders of magnitude fewer *different* sleep sets than reachable states. This observation suggests storing each different sleep set only once in a separate (small) hash table and assigning the integer identifiers $0, 1, 2, \ldots$ to the sets in the table. Then, only this small integer identifier has to be stored for each state in the state table. Observe, that this method achieves constant memory requirements for each state and furthermore saves space in comparison with the previous implementation.

[1] Partial-Order Package for SPIN developed at the University of Liège (ULg)

6 Results on Sample Protocols

In our first set of experiments, we used the Mulog protocol (38181 states) with a 3-limited hash compaction scheme. We varied the number of bits b in the compressed values. For each value of b, we conducted 100 runs of the verifier and counted the number of runs in which omissions occurred. Results for the cases where $n \approx 5m$ and $n \approx m$ are given in Figures 3 and 4, respectively. While in Figure 3 the outcomes of the experiments accurately match the theoretical analysis, they seem to be slightly smaller in Figure 4. The difference between the two curves in the latter case can be explained by observing that the (omitted) analysis is conservative in this case.

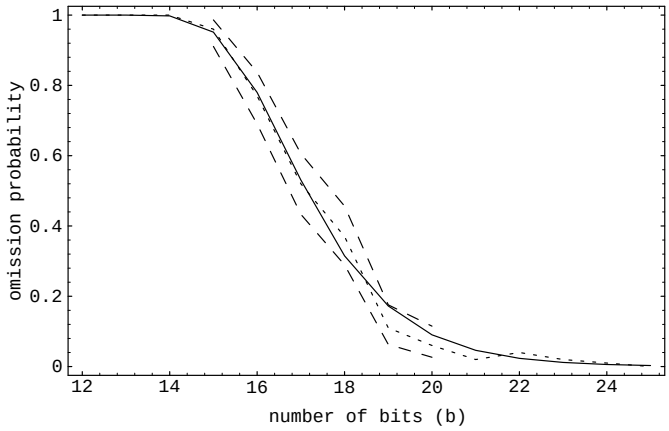

Figure 3: Omission probabilities in the case m=8 009 observed in the experiments (dotted, with 90% confidence intervals) and obtained from the approximation formula (solid)

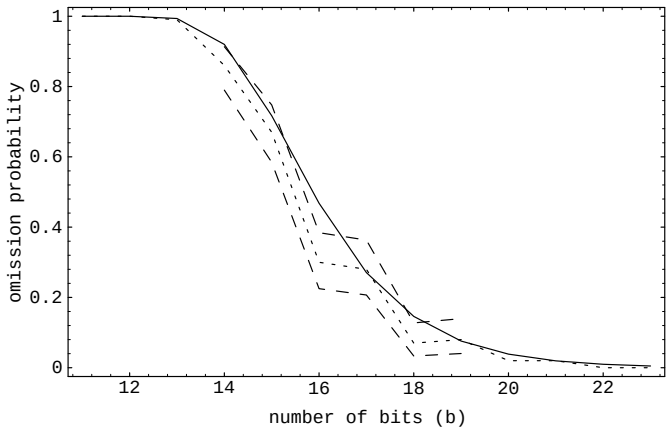

Figure 4: Omission probabilities in the case m=38 183 observed in the experiments (dotted with 90% confidence intervals) and obtained from the approximation formula (solid)

In our second set of experiments, we studied the observed number of reachable states n while varying the parameter t and the size of the hash table m for two different protocols. In the case of the Mulog protocol shown in Table 2, there is only a negligible difference in performance for the three different values of t. This behavior can be explained intuitively by observing that the state space caching works extremely well in this case and thus the state table seems to be not really needed. The situation for the Abra protocol shown in Table 3 is different. Here, state space caching performs worse than in the case of the Mulog protocol – a factor 3.74 in memory savings comes already at the cost of roughly a factor of 1.2 in "time". Note that for this protocol the performance of the t-limited scheme improves with t.

Table 2: Observed numbers of reachable states n for the Mulog protocol (38 181 states)

m	16 001	8 009	2 003	503	251
1-limited	38 185	38 198	38 248	38 829	39 448
3-limited	38 184	38 188	38 210	38 846	39 403
10-limited	38 182	38 183	38 233	38 782	39 471

Table 3: Observed numbers of reachable states n for the Abra protocol (149 816 states)

m	149 827	80 021	60 013	40 009
1-limited	152 536	157 570	163 293	190 885
3-limited	150 359	153 295	158 218	178 758
10-limited	149 909	152 573	157 023	176 814

Selecting t=3 seems to be a good compromise. It yields almost the same performance as the 10-limited scheme and leaves the choice of three elements for the replacement. This choice can be exploited in the replacement strategy [6], which may result in further improvements. On the other hand, t=3 is small enough to not increase the omission probability too severely.

7 Conclusion and Future Work

The new t-limited scheme avoids the sharp increase in the omission probability that is experienced in the previous hash compaction scheme [9] when the table fills up completely. Furthermore, a limit of t=3 induces only negligible increase in run-time, especially when the size of the state table roughly equals the number of reachable states. Therefore, it seems advantageous to employ the new scheme to avoid the final increase in the omission probability even in situations in which state space caching was not originally intended.

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. This shift is particularly noticeable when combining different

techniques that aim at reducing the memory requirements in explicit state enumeration, which increases the priority of research into accelerating explicit-state verification methods.

Acknowledgements

We are grateful to Pierre Wolper and Denis Leroy for sharing the unpublished revision [10] of their paper with us. We would also like to thank Patrice Godefroid for providing us with the example protocols we presented and for his help in understanding his partial-order implementation and Ross Rosen for his comments on a draft of this paper.

References

[1] 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.

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

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[4] P. Godefroid. Using partial orders to improve automatic verification methods. In *Computer Aided Verification. 2nd International Workshop*, pages 176–185, 1990.

[5] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, University of Liège, 1994.

[6] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Computer Aided Verification. 4th International Workshop*, pages 178–191, 1992.

[7] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[8] 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.

[9] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[10] P. Wolper and D. Leroy. Reliable hashing without collision detection. Unpublished revised version of [11].

[11] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.