Enhanced Probabilistic Verification with 3Spin and 3Murphi

Peter C. Dillinger and Panagiotis Manolios

College of Computing, Georgia Institute of Technology
801 Atlantic Drive, Atlanta, GA 30332-0280
{peterd,manolios}@cc.gatech.edu

Abstract. 3Spin and 3Murphi are modified versions of the Spin model checker and the Murφ verifier. Our modifications enhance the probabilistic algorithms and data structures for storing visited states, making them more effective and more usable for verifying huge transition systems. The tools also support a verification methodology designed to minimize time to finding errors, or to reaching desired certainty of error-freedom. This methodology calls for bitstate hashing, hash compaction, and integrated analyses of both to provide feedback and advice to the user. 3Spin and 3Murphi are the only tools to offer this support, and do so with the most powerful and flexible currently-available implementations of the underlying algorithms and data structures.

1 Introduction

Explicit-state model checking is a popular and effective verification technique employed by numerous tools, including Murφ, TLC, Java PathFinder, and Spin. To ameliorate the memory demands of state explosion, most of these tools include algorithms that have a small probability of overlooking errors. One such probabilistic algorithm is *bitstate hashing*, developed in Spin [6]. The other major probabilistic technique is *hash compaction*, which was mostly developed in Murφ [8, 9] (building on previous work [10]). We have modified these two tools into releases called 3Spin [3] and 3Murphi [2] that each incorporate a set of features designed to maximize the effectiveness and efficiency of probabilistic verification (see Table 1). This novel set of features is

Table 1. This table shows the capabilities of four probabilistic explicit-state verification tools, Murφ 3.1, Spin 4.2.2, 3Murphi 3.2, and 3Spin 3.2.

Tool	Bitstate hashing	Hash compaction	Memory sizes	Hashing	Feedback
Murφ	no	yes, with wasted bit	any	univ.+diff.	H.C. omission analysis (probability only)
Spin	enhanced ($\geq$ 4.2.0)	yes, but inflexible	powers of 2 only	Jenkins	B.H. recommendations ($\geq$ 4.2.0)
3Spin, 3Murphi	enhanced	yes	any	Jenkins; univ.+diff.	recommendations (both algs); omission analysis (both algs)

utilized by our probabilistic verification methodology, which was introduced in [4] but we review here in Section 2. Sections 3 and 4 then discuss how the features of 3Spin or 3Murphi and our methodology enable the user to more efficiently reach her verification goal while preserving all existing functionality of Spin and Murphi (respectively).

2 Methodology

In previous work [4] we describe a methodology for utilizing both bitstate hashing and hash compaction that attempts to minimize the time to finding errors (if present) or to reaching whatever certainty the user considers adequate to concluding that the model satisfies the desired properties.

Bitstate hashing and hash compaction are probabilistic data structures used to represent sets. They support the standard *add* and *query* operations, but a query on an element that is not in the set may return *true*, yielding a false positive. Their probabilistic nature allows for memory-efficient representations of large sets, a crucial requirement for model checkers which have to keep track of very large sets of visited states. Bitstate hashing identifies states with a chosen number of addresses of a bit-vector; when a state is visited the corresponding bits are set. Hash compaction stores hashed states in a table with a fixed number of cells. Figure 1 emphasizes the key difference between the data structures, which we expand upon in our methodology description below.

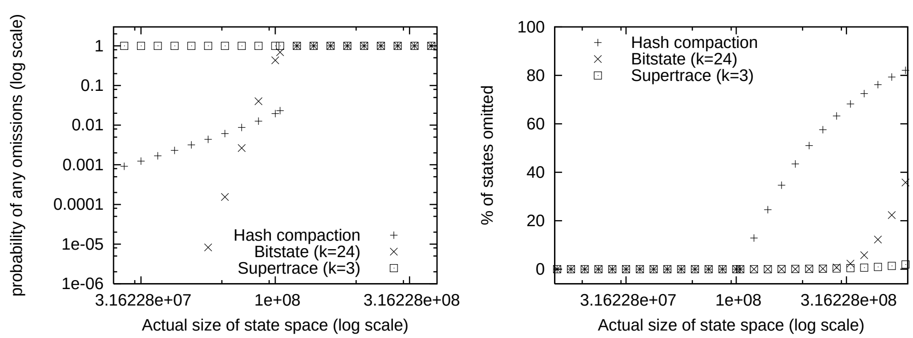

Fig. 1. These graphs show the accuracy of three probabilistic verification techniques/configurations for various state space sizes. In both graphs, lower is better. The data points for “Hash compaction” and “Bitstate (k=24)” are obtained with data structures optimized for a state space size of 10^8, using 400MB of memory. The graphs show the accuracy of the data structures as the size of the state space varies. In both graphs, lower is better according to the respective criterion. The left graph shows the probability that any omissions occur, while the right graphs shows the expected percentage of states omitted. “Bitstate (k=24)” and “Supertrace (k=3)” are instances of bitstate hashing setting 24 and 3 bits per state respectively. “Hash compaction” shows the expected results of using 32-bit hash compaction, which in this case has a maximum visitable size of about 105 million states. The results are obtained mostly analytically, as in [4].

When the state space size is completely unknown, as when first attempting to verify a model, use *supertrace*, which is bitstate hashing configured to set a small number of bits per state, such as Holzmann's recommendation of three. In such cases, supertrace is the best choice because of its low percentage of omitted states over a wide range of state space sizes (see right graph in Figure 1). Supertrace tends to find errors quickly if they exist, but is not the most productive technique for demonstrating error-freedom [5].

When we know the size of the state space rather accurately, as when iteratively building confidence of error-freedom in a model, use *hash compaction*, because of its superior accuracy when tuned for a known state space size. The left graph of Figure 1 shows that when the actual state space size is 80–100% of the maximum size, hash compaction is the most accurate. The graphs also show that if the table ends up far from full (left 1/3rd of left graph) or if it overflows (right half of right graph), hash compaction is not the best choice.

When we have a rough estimate of the state space size, as when verifying a modified version of a previously-verified model, use *bitstate hashing* configured to set a number of bits per state optimized for that estimate (shown in [4]). When that number is significantly larger than supertrace's 3 bits, this approach is likely to be much more accurate. Furthermore, it can tolerate much more deviation from the estimate than hash compaction can, because hash compaction becomes pretty useless if its table fills up. Making a small change to a model can easily change its state space size by a factor of 2 or more, which bitstate hashing tolerates *much* better than hash compaction.

In the following sections, we explain how among currently-available tools, 3Spin and 3Murphi best support this unified approach in terms of features, performance, and ease of use.

3 Feedback

3Spin/3Murphi's most notable feature that supports our methodology is feedback after a verification run fails to find an error. These two are the only tools to report both omission analysis and recommendations, and do so in both bitstate mode and hash compaction mode.

The omission analysis uses formulas and algorithms described in previous [4] and related work [8] to compute either a probability of omitting any states or an expected number of states omitted [4]. In the latter case, the estimate is rough, but looking at the connectivity of the graph allows us to report the reliability of that result. Overall, the omission analysis helps the user understand the degree to which he can be certain the model satisfies the desired properties.

Because the configuration of the algorithms can make a big impact on accuracy, 3Spin and 3Murphi also incorporate analyses for predicting the best settings for reverifying the same (or a similar) model [4]. Pursuant to our methodology, the tools give advice on whether to follow-up with hash compaction or with bitstate hashing, along with recommended settings for each. The latest versions of these tools return the recommendations not in terms of low-level settings, but in terms of visitable state space size estimates. Perhaps the greatest benefit from this new form of recommendations is that they are not closely tied to a particular memory setting, enabling users to easily benefit from the recommendations even if they change the memory settings.

As Table 1 shows, only 3Spin and 3Murphi support this rich set of feedback features. In fact, Spin's support for recommendations in bitstate mode is derived from earlier versions of 3Spin.

4 Other Improvements or Features

This section discusses the rest of 3Spin/3Murphi's core features, following Table 1 from left to right.

Bitstate hashing. The first release of 3Spin focused on improvements to bitstate hashing that have since been integrated into Spin, starting with version 4.2.0. Prior to our work, it was believed that if memory was not terribly constrained (say 8 or more bits per state) the bitstate hashing configuration with the best accuracy was inherently slow—too slow to be more productive than iteratively using a suboptimal but fast configuration [6]. Our improvement [5, 4] eliminates most of that overhead by reusing hash information in an intelligent, accuracy-preserving way.

This improvement has allowed our methodology to utilize fast *and* accurate bitstate hashing configurations, when one has a rough estimate of the state space size.

Hash compaction. Spin's implementation of hash compaction is very limited. It only supports compacted state sizes of 32 to 64 bits per state in 8 bit increments, and the size of the table must be a power of 2 (discussed in **Memory sizes** below). Both limitations inhibit Spin's ability to take advantage of available memory in minimizing the possibility of overlooking an error.

As of version 2.0, 3Spin has its own implementation of hash compaction, which has also been put in to 3Murphi. Our implementation and Murφ's support all compacted state sizes from 4 bits to 64 bits. The extent of this range is justified as follows: when fewer than about 10 bits per state are available, bitstate hashing is superior to hash compaction; 3Spin/3Murphi makes recommendations accordingly. On the high end, using a compacted state size of 64 bits is so accurate that, even if the table is almost full, the probability of *any* omissions is on the order of one in trillions. At this level of accuracy, random hardware errors are probably more likely to cause error omission than algorithmic losses.

Our implementation actually improves upon Murφ's (which is better than Spin's) in the way it determines whether a cell in the table is occupied. In addition to the memory dedicated to the compacted state, Murφ allocates a single-bit flag with each cell of the compacted table to indicate whether the cell has a state stored in it. We instead reserve the compacted state "0" to indicate that a cell is not used. As a result, we can use the bit saved to increase the compacted state size and nearly cut in half the probability of omitting an error when using the same amount of memory as Murφ.

Memory sizes. An informed choice to use a probabilistic technique is motivated by memory constraints with respect to state space size, but Spin limits the user to only power-of-2 sizes for its probabilistic data structures. 3Spin and 3Murphi allows their data structures to be of any size addressable on a 32-bit machine. Keep in mind that allocating more memory to either data structure *always* makes it more accurate—and the impact is significant. For example, when using 1024MB of memory for $k = 21$ bitstate hashing on 300 million states, the search expects to omit about 20 states. Using 3Spin with 1750MB instead leads to less than a 1% chance of omitting *any* states. If you also increase k to 35, there is less than 1 in 1000 chance of omitting any states.

Hashing. In the first release of 3Spin, we showed how to get more hash information from the hash function used by Spin, the Jenkins LOOKUP2 hash function [7], with no observable impact on coverage/accuracy. Incorporating this improvement into Spin reduced its execution time by about 25% [5] in common scenarios, because it could make just one call to Jenkins where it used to make two. 3Murphi adds support for this hash function, which can be faster than Murφ's.

Murφ uses a different hash function that enables an optimization called *differential hashing* [1]. The hash function, H_3, also has the advantage of being a *universal* hash function. Since version 3.0, 3Spin has included this hash function and similar optimizations as an alternative to Jenkins, and in many cases, the differential hashing optimization makes the universal hash function *faster* than Jenkins.

We include both hash functions as options in both tools because they are fundamentally different: Jenkins is always competitively fast but only heuristically accurate; H_3 is only heuristically fast but provably accurate.

5 Conclusion

Earlier versions of 3Spin have already made an impact by introducing features that eventually made their way into Spin itself, and here we have introduced version 3.2 of 3Spin and its new cousin 3Murphi. These tools offer a more effective verification environment to users of Spin and Murphi.

References

1. B. Cousin and J. Hélary. Performance improvement of state space exploration by regular and differential hashing functions. In *6th CAV*, pages 364–376, 1994.
2. P. C. Dillinger. 3Murphi Home Page. http://www.cc.gatech.edu/˜peterd/3murphi/.
3. P. C. Dillinger. 3Spin Home Page. http://www.cc.gatech.edu/˜peterd/3spin/.
4. P. C. Dillinger and P. Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*. Springer-Verlag, November 2004.
5. P. C. Dillinger and P. Manolios. Fast *and* accurate bitstate verification for SPIN. In *11th SPIN Workshop*, volume 2989 of *LNCS*. Springer-Verlag, April 2004.
6. G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 301–314. Chapman & Hall, 1995.
7. B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal*, September 1997.
8. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *CHARME*, volume 987 of *LNCS*, pages 206–224. Springer-Verlag, 1995.
9. U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. In *FORTE/PSTV*, pages 333–348, 1996.
10. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *5th International Conference on Computer Aided Verification*, pages 59–70, 1993.