Explicit state model checking with Hopper

Michael Jones and Eric Mercer

Brigham Young University
Computer Science, Provo, UT
jones@cs.byu.edu and egm@cs.byu.edu

Abstract. The Murφ-based Hopper tool is a general purpose explicit model checker. Hopper leverages Murφ's class structure to implement new algorithms. Hopper differs from Murφ in that it includes in its distribution published parallel and disk based algorithms, as well as several new algorithms. For example, Hopper includes parallel dynamic partitioning, cooperative parallel search for LTL violations and property-based guided search (parallel or sequential). We discuss Hopper in general and present a recently implemented randomized guided search algorithm. In multiple parallel guided searches, randomization increases the expected average time to find an error but decreases the expected minimum time to find an error.

The Hopper[1] tool leverages the Murφ architecture to implement parallel, disk-based and heuristic model checking algorithms. The common theme in the algorithms implemented in Hopper is that they do not use abstraction. Instead, Hopper explores fundamental algorithms for reducing time and space capacity limitations in state generation and storage. Our intention is that algorithms implemented in Hopper can be combined with well-known abstraction techniques. The algorithms studied in Hopper are generic enough to be implemented in any state enumeration context–including software model checking. The current release of Hopper contains parallel and disk based algorithms published by Dill and Stern [1, 2] and heuristic search using the heuristic proposed by Edelkamp [3]. Hopper also includes parallel and guided search algorithms developed by the BYU model checking research group [4–6]. Hopper is a testbed for ideas that will be incorporated in our forthcoming C/C++ model checker built as an extension of the GNU debugger (GDB). The Hopper distribution includes a suite of 177 benchmark verification problems for Murφ.

This paper describes the architecture and algorithms implemented in Hopper along with a new randomized guided algorithm we have recently implemented. Randomization increases the variance (and the mean) of the search effort required to find a property violation. Search effort is measured by the number of transitions taken. In some problems, increasing the variance (even at the expense of increasing the mean) decreases the expected minimum number of transitions taken in error discovery using in parallel searches.

[1] Named after Edward Hopper (1882-1967), an early realist painter. Neither Hopper the artist nor Hopper the tool rely on abstraction.

1 Hopper

Hopper is a general purpose explicit state model checker built on the Murφ code base. Hopper, like Murφ, uses a *rule-based* input language for model descriptions. Although it is not process based, like `Promela` or `CSP`, it is sufficient to describe large complex transition systems [7]. Hopper adds polymorphism to Murφ's code base to implement new algorithms. The design philosophy of Hopper is to minimally alter code in the basic Murφ distribution when adding new functionality through polymorphism. The behavior of key classes is redefined in a separate code base. This design philosophy treats Murφ as an application programmer interface (API) to prototype new algorithms for empirical analysis. Hopper does not support Murφ symmetry reductions in parallel, randomized or disk based algorithms.

Hopper includes an implementation of the Stern and Dill parallel model checking algorithm [1] and the disk based algorithm from [2]. The parallel algorithm in Hopper is implemented with `MPICH` 1.2.5 for the communication layer and a modified Dijkstra's token-based termination detection algorithm. `MPICH` is a free `MPI` implementation that is portable across several different communication fabrics. The modification to Dijkstra's token termination algorithm is required because communication in the parallel algorithm is not limited to a ring topology, as required by Dikjstra's algorithm. The modification adds message count information to the token. Termination is detected when the token travels around the logical ring and both retains the correct color and indicates that the number of messages sent is equal to the number of messages received. After detection, termination is completed by passing a poison pill through the ring. The modified Dijkstra's token termination algorithm is more reliable than the algorithm based on idle time used in [1]. The Hopper implementation has been successfully tested and analyzed on two platforms with 256 processors and different communication fabrics [5].

Hopper also includes a parallel algorithm that uses dynamic partitioning to aggregate memory on multiple computation nodes. The Stern and Dill algorithm uses a static hash function to distribute known reachable states in the model across computation nodes. An imbalanced distribution, however, may not efficiently utilize the aggregated memory since it may prematurely drive a node to its maximum capacity before all reachable state have been enumerated. The dynamic partition algorithm in Hopper constructs the partition function on-the-fly. The Murφ architecture simplifies the use of either the static or dynamic partitioned hash table when running any given search algorithm.

Hopper includes a visualization toolkit for postmortem analysis of parallel model checking algorithm behavior through time. This Java based toolkit reads time stamped entries from Hopper log files. The time series data is then reconstructed and animated. The default configuration shows, for each computation node, the size of its state queue, the total number of states in its hash table, the number of states sent, and the number of states received as dynamic bar charts.

Hopper implements a cooperative parallel search algorithm for finding LTL violations. The bee-based error exploration (BEE) algorithm is designed to operate in a non-dedicated parallel computing environment. It does this by employing a decentralized forager allocation scheme exhibited as a social behavior by honeybee colonies. Forager allocation involves identifying flower patches and allocating foragers to for-

age for resources at the patches. In LTL search, flower patches map to accept states and foraging maps to finding cycles that contain accept states. The resulting algorithm searches for accept states, then allocates workstations to forage for cycles beginning at accept states. A complete presentation and analysis of the BEE algorithm is given in [4]

Hopper implements property-based guided search in either parallel or sequential modes. The Hopper distribution uses admissible and inadmissible versions of property-based heuristics given by Edelkamp et. al. in [3]. Hopper also implements a Bayes heuristic search (BHS) to improve the expected accuracy of estimates treated as random variables (i.e., functions that assign a real valued probability between 0 and 1 to each possible outcome of an event). A probability density function characterizes the distribution of confidence in the heuristic. If the heuristic is accurate, then most of the probability is close to the actual distance to the target. The BHS algorithm minimizes mean squared error in heuristic estimates using an empirical Bayes [8] meta-heuristic. This is done using sets of sibling states to derive the confidence that should be attributed to each individual estimate. The confidence level is then used to proportionally revise the original estimate toward the mean of the sibling estimates. The theoretical and empirical validation of the approach using a Bayesian model is given in [6]. The analysis shows that the resulting improved heuristic values have smaller total expected mean squared error.

A model database for empirical testing is a final piece of Hopper. The primary obstacle to designing and comparing state enumeration algorithms is a lack of performance data on standardized benchmarks. This lack of data obscures the merits of new approaches to state enumeration. Hopper includes a set of 177 benchmark models with an web portal to add new models and report new benchmark results. The web portal for the database is located at `http://vv.cs.byu.edu`.

2 Randomized Guided Search

Randomizing the guided search algorithm intends to improve the decentralized parallel search for LTL violations. The decentralized parallel searches will cover more of the search space if they do not all share the same deterministic behavior. Random walk is a trivial, but surprisingly effective, way to distribute the searches. In terms on the expected number of states explored before finding an error, randomized guided search aims to achieve a variance near that of random walk with a mean near that of guided search.

The guided search is randomized by selecting the next state to expand randomly from the first n states in the priority queue. Randomizing next state selection increases the variance of the expected number of states expanded before finding an error. Suppose X is a random variable that represents the *number of states expanded before reaching an error* in a given model for some amount of randomization n. In our experiments, X follows a normal distribution with a mean μ and a variance σ^2. Increasing randomization increases both σ^2 and μ. Randomization can improve search performance because the probability of observing a small value of X increases logarithmically in σ^2–if μ remains unchanged.

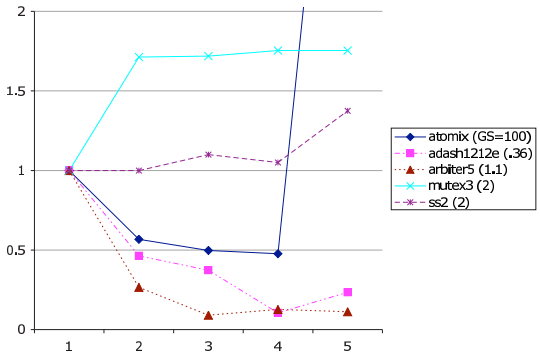

Fig. 1. Ratio of minimum value in 100 samples of randomized guided search and best deterministic search. A ratio less then one indicates that parallel randomization improved search performance.

Unfortunately, randomization of guided search can increase μ. In other cases, randomization *decreases* μ. In general, increasing randomization in guided best-first search drives μ toward the number of states expanded by breadth-first search. If breadth-first search expands fewer states than non-randomized guided search, then randomization decreases μ. Otherwise, randomization increases μ.

Taking multiple samples of X logarithmically increases the probability that any one sample will be less than a given threshold. This produces a logarithmic speedup when performing independent randomized guided searches in parallel. This is similar to the amplification of stochastic advantage in a one-sided probabilistic algorithm.

We have implemented the randomized guided search algorithm in Hopper, and we have conducted a series of experiments to asses the impact of randomization on guided search using the BHS algorithm. The amount of randomization was controlled by varying n, the *pickset* which is the number of states in the prefix of the priority queue from which the next state to expand was chosen. Each search result is computed by observing the outcomes of 100 trials and taking the outcome with the smallest number of states expanded. This is done for each model/pickset combination.

We chose five models for which guided search had a wide range of effects. The effect of heuristic guidance on a search problem can be measured using the *guided-speedup* (GS) which is the BFS transition count divided by the guided search transition count. The models used in the experiments have GS ratios ranging from 100 (meaning guided search was 100 times faster) to 0.36 (meaning guided search was almost 3 times slower). The GS ratios for each model are included later in the legend for Figure 1. Each model contains at least one violation of its invariant.

Two sets of experiments were conducted: one to determine the effects of randomization on the adash1212e model (for which guided search was particularly ineffective) and one to determine the effects of randomization on the five models. The adash1212e model was tested with picksets ranging from 2 to 2000 states. As the pickset size increased, the mean number of states expanded decreased. For every pickset size, the

minimum number of states explored by any one node was less than both the deterministic BFS and parallel pure random walks.

Each of the 5 models were tested with pickset sizes of 2, 3, 4 and 5. The effect of randomization on the minimum sample drawn from 100 experiments is shown in Figure 1. Figure 1 plots the ratio of the number of transitions taken in the minimum of the 100 samples and the smaller of either the BFS or deterministic guided search. A ratio less than 1 indicates that randomization lead to faster error discovery.

The series of experiments with adash1212e demonstrate that for models in which the heuristic performs extremely poorly increasing randomization results in steadily decreasing search times. For all 5 models, choosing randomly from the first 2 to 4 states in the priority queue gives almost all of the reduction in transition count while avoiding state explosion. These results taken together suggest that choosing from the first four states in the priority queue balances randomization and guidance.

3 Conclusion

Hopper is general purpose model checker built on top of the Murφ code base. It uses Murφ as an API that provides low-level building blocks for new algorithms. Hopper includes several published and unpublished algorithms, as well as a visualization tool and a model database. Recent algorithms in Hopper are a BHS and a parallel randomized guided search. Future work for Hopper includes the implementation of novel disk-based, shared memory and multi-agent search algorithms. Hopper is a testbed for algorithms for increases capacity that can be incorporated into any state enumeration tool, including SPIN.

References

1. Stern, U., Dill, D.L.: Parallelizing the Murϕ verifier. In Grumburg, O., ed.: Computer-Aided Verification, CAV '97. Volume 1254 of Lecture Notes in Computer Science., Haifa, Israel, Springer-Verlag (1997) 256–267
2. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Murϕ verifier. In Hu, A.J., Vardi, M.Y., eds.: Computer-Aided Verification, CAV '98. Volume 1427 of Lecture Notes in Computer Science., Vancouver, BC, Canada, Springer-Verlag (1998) 172–183
3. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with HSF-SPIN. In: 8th International SPIN Workshop on Model Checking Software. Number 2057 in Lecture Notes in Computer Science, Springer-Verlag (2001)
4. Jones, M.D., Sorber, J.: Parallel search for LTL violations. Software tools for technology transfer (2004) To appear.
5. Jones, M., Mercer, E.G., Bao, T., Kumar, R., Lamborn, P.: Benchmarking explicit state parallel model checkers. In: Workshop on Parrallel and Distributed Model Checking. (2003)
6. Seppi, K., Jones, M., Lamborn, P.: Guided model checking with a Bayesian meta-heuristic. Technical Report VV-0401, Dept. of Computer Science, Brigham Young U. (2004)
7. German, S.M.: Formal design of cache memory protocols in IBM. Formal Methods in System Design **22** (2003) 133–141
8. Bradley P. Carlin, T.A.L.: Bayes and Empirical Bayes Methods for Data Analysis. Chapman & Hall (1996)