What is Resource-Aware Verification?

Stavros Tripakis*

August 19, 2008

1 Why resource-aware verification?

Exhaustive verification methods such as model-checking suffer from the well-known state-explosion problem: the set of states is too large to explore exhaustively in reasonable amounts of time and space (memory). But model-checkers are often plagued with another problem, which makes state-explosion even worse: the disk-swapping problem. The latter problem manifests itself when the model-checker fills up the main memory of the computer it runs on, but without exhausting the virtual memory address space. At this point disk-swapping occurs, which is very slow and essentially makes the search stagnate: the rate of explored states (number of visited states per second) becomes practically zero.

The disk-swapping wall is usually hit pretty quickly. For example, using a model-checker that can explore 10^5 new states per second, on a model that requires 1000 bytes to represent each state, consumes memory at a rate of approximately 100 MB/sec. This means that a main memory of size 8 GB can be filled in about 2 minutes. Exploration rates in the order of 10^5 states per second are not unusual today, for an advanced model-checker such as Spin [1].

Ideally, we would like to have a verification method that scales well with resources. Informally, this could mean that the more time or the more memory we have available, "the more we can verify". In turn, "how much we verify" can be quantified as "how many distinct reachable states we explore". Ideally, we would like to have a verification tool that, say, if we let it run for 2 hours we would expect it to explore twice as many states than if we let it run for 1 hour (or it will explore the entire state space). This is an ideal goal, of course, which is hard, if not impossible, to reach. On the other hand, a method that after n hours explores only $\epsilon\%$ more states than after 5 minutes is clearly not scalable with resources.

2 A definition: memory-aware state-space exploration algorithms

Motivated by what has been said above, we introduce the class of *resource-aware*, and in particular *memory-aware*, state-space exploration algorithms. Memory-aware algorithms are meant to deal with the disk-swapping problem in a rather radical way: by simply using no disk memory, only main memory. A less strict qualification could be envisaged in the future. For now, let us consider this

*Cadence Research Laboratories, 2150 Shattuck Avenue, Berkeley, CA 94704. The author is currently on leave from CNRS-Verimag. Part of this work was done while at Verimag.

one. We also consider, for simplicity, explicit state enumeration algorithms. Symbolic algorithms could also be considered.

Assume that S_E is the number of bytes needed to represent (and store) one state of the state space of a given example E. An explicit state enumeration algorithm A is memory-aware if:

1. A takes as input a positive integer N.
2. For any example E, if A is run on a computer with main memory M bytes and N is set to $\frac{M}{S_E}$, then A will never store during its operation more than N states of the state space of E.
3. Generally the behavior of A depends on N, that is, as N varies, the behavior of A generally varies.

An algorithm is *memory-bounded* if it is guaranteed to use no more bytes than a given amount. By condition 2 above, if an algorithm is memory-aware, then it is also memory-bounded. The opposite is not generally true as we shall see.

3 Review: existing algorithms

Let us now review some existing state space exploration algorithms and try to answer the question: are they memory-aware? Most of the algorithms are well-known; for the rest, we provide references. This review is by no means attempting to be complete.

Breadth-first search (BFS): This is a deterministic and complete search algorithm. It is complete in the sense that it is guaranteed to explore the entire set of reachable states. The algorithm maintains a set of visited states (sometimes called the set of "closed" or "expanded" states). This set can grow arbitrarily large (as large as the set of reachable states). Thus the algorithm is not memory-bounded, and consequently it is not memory-aware.

Depth-first search (DFS) with a set of visited states: There are many variants of depth-first search. The one discussed here is a deterministic and complete search algorithm. It maintains not only a stack, but also a set of visited states. This set can grow arbitrarily large (as large as the set of reachable states). Thus the algorithm is not memory-bounded, and consequently it is not memory-aware.

Depth-first search (DFS) without a set of visited states: This variant is a depth-first search that only maintains a stack, but no set of visited states. There are two cases:

- The stack size is fixed or bounded. Then the search is generally incomplete, unless if the diameter of the state space (i.e., length of the longest acyclic path in the state space) is no greater than the stack size. This variant is memory-aware if the stack fits in main memory.
- The stack size is unbounded. Then the search is always complete but the algorithm is not memory-bounded because the stack can grow larger than what fits in main memory. Thus the algorithm is not memory-aware either.

In both cases the algorithm is deterministic. In both cases the time complexity of the algorithm is prohibitive: exponential in the number of states. This is because the algorithm explores all paths (up to a given depth in the case where stack size is bounded) and may revisit a state multiple times.

Best-first search: Best-first search is a variant of DFS, where a heuristic function is applied to every successor of a state and the successors are ordered with respect to this function. Then the successors are explored in that order. The algorithm is or is not memory-aware depending on which variant of DFS is used, as explained above.

Beam search: This is a best-first variant of BFS where at each depth level, the states visited in that level are ordered according to some heuristic function, and then only the first K states are kept, where K is an input to the algorithm. Although the breadth of the search is limited by K, the depth of the search is unlimited, thus this algorithm is not memory-bounded, and consequently not memory-aware either.

Bit-state hashing: This technique was introduced by Holzmann [2]. It essentially consists in using, instead of a set of visited states that stores entire state vectors, a large hash table T: each element of the hash table is only a few bits long. Assume for simplicity each element is a single bit. Initially all bits are set to 0. Whenever a new state s is visited, its hash value $i = h(s)$ is computed. If the i-th bit of T is 0, then it is set to 1, meaning this is a truly new state. If the i-th bit is already 1, we assume this state has been already visited. This may not necessarily be true, since there may be two states s and s' such that $h(s) = h(s')$. Then, it could be that s' was previously visited, but not s. Thus, bit-state hashing is generally incomplete. However, assuming the technique is used with a DFS of bounded stack depth, the algorithm is memory-aware.

A*: A* is a deterministic and complete search algorithm. Like BFS, it maintains a set of "closed" states that can grow arbitrarily large. Consequently A* is neither memory-bounded, nor memory-aware.

MA* and SMA*: These are two deterministic search algorithms derived from A*. MA* is proposed in [3] and SMA* is proposed in [4] as an improvement to MA*. Both algorithms share the feature that they restrict the number of states that can be accomodated during the search according to a limit set by the amount of available memory.

Random walk: This is a randomized algorithm. The algorithm maintains a single state s in memory. Initially s is set to the initial state. At each step, one of the successors of s is randomly chosen and replaces s. If s has no successors then the search stops. This algorithm is memory-bounded since it stores only a single state. However it is not memory-aware, because it does not satisfy Condition 3 above: the behavior of random walk is the same independently of the size of available main memory. See [5, 6] for modifications to standard random walk.

State caching: This term denotes a class of exploration algorithms which are based on the following idea: keep a set of visited states of bounded size (the "cache"); whenever a new state is to be inserted in the set, if the size bound is exceeded, then pick randomly an old state that is already in the set and remove it. Early experiments with different replacement policies can be found in [7, 8]. A more recent study is [9]: in this paper, a particular replacement policy called *stratified caching* is also proposed. A simple state-caching algorithm is the depth-first traversal with (random) replacement algorithm proposed in [10].

State-caching algorithms are memory-aware, provided that, in addition to the cache of visited states, all other data structures used are also memory-bounded. For example, the algorithm described in [10] is a DFS that uses a stack that can grow arbitrarily large, despite the fact that the cache remains memory-bounded. This can be easily remedied, however, by limiting the stack to a given depth.

Termination in state-caching algorithms is generally not guaranteed (for instance, imagine the behavior of such an algorithm on a single loop with a very large number of states), except if some arbitrary termination condition is enforced, which in turn compromises completeness.

References

[1] G. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[2] G.J. Holzmann. An analysis of bitstate hashing. In *Formal Methods in System Design*, pages 301–314. Chapman & Hall, 1998.

[3] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory (research note). *Artif. Intell.*, 41(2):197–222, 1989.

[4] S. Russell. Efficient memory-bounded search methods. In *In ECAI-92*, pages 1–5. Wiley, 1992.

[5] R. Pelanek and I. Cerna. Enhancing random walk state space exploration. In *In Proc. of Formal Methods for Industrial Critical Systems (FMICS'05*, pages 98–105. ACM Press, 2005.

[6] N. Abed, S. Tripakis, and J-M. Vincent. Resource-Aware Verification Using Randomized Exploration of Large State Spaces. In *SPIN'08*, number 5156 in LNCS, 2008.

[7] G. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64:2413–2434, 1985.

[8] G. Holzmann. Automated protocol validation in Argos: Assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13:683–696, 1987.

[9] J. Geldenhuys. State caching reconsidered. In *Model Checking Software, 11th International SPIN Workshop (SPIN'04)*, volume 2989 of *LNCS*, pages 23–38. Springer, 2004.

[10] C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *CAV'91*, volume 575 of *LNCS*. Springer, 1992.