Swarm Verification[1]

Gerard J. Holzmann, Rajeev Joshi, Alex Groce
Jet Propulsion Laboratory, California Institute of Technology
firstname.lastname@jpl.nasa.gov

Abstract

Reportedly, supercomputer designer Seymour Cray once said that he would sooner use two strong oxen to plow a field than a thousand chickens. Although this is undoubtedly wise when it comes to plowing a field, it is not so clear for other types of tasks. Model checking problems are of the proverbial "search the needle in a haystack" type. Such problems can often be parallelized easily. Alas, none of the usual divide and conquer methods can be used to parallelize the working of a model checker. Given that it has become easier than ever to gain access to large numbers of computers to perform even routine tasks it is becoming more and more attractive to find alternate ways to use these resources to speed up model checking tasks. This paper describes one such method, called swarm verification.

1. Introduction

The search for efficient ways to implement logic model checking algorithms already started around the time that the PC was first introduced, in the late seventies and early eighties. Since then, we have seen a truly impressive rise in the computational power of desktop machines. Every doubling of speed and every doubling of the memory available on those machines translated directly into comparable increases in the reach and power of the available logic model checking tools. These increases, paired with the algorithmic improvements that could be made to improve search efficiency (e.g., the use of partial order reduction and bitstate hashing techniques in software model checkers such as SPIN [7], or symbolic model checking techniques in hardware model checkers like SMV [12]) have helped us to produce tools that can tackle problems of a size that were unimaginable before. It has made it possible for us to continue to reach further and further. For approximately the last decade, our goal has been to apply logic model checkers directly to implementation level problems, using on-the-fly abstraction techniques [8]. The hope was that our algorithms and machines would be able to catch up with the potentially very large computational complexity of these problems.

Approximately five years ago, though, chip manufacturers decided to adopt a new strategy for the fabrication of CPUs. Instead of focusing on the continued trend to shrink the size of chips, increasing clockspeed and raw performance, they switched to placing larger numbers of CPU cores onto a single chip. In principle, this strategy can bring equivalent performance increases to desktop systems. Large numbers of independent threads of execution can now all be executed in parallel, with mostly limited competition for shared resources. Instead of continuing to double the raw speed of CPUs, the chip makers now plan to double the number of cores on a chip with each new generation.

Curiously, although the raw speed of CPUs has stalled at roughly 2002 levels, the size of RAM memory that is available on standard desktop systems continues to follow Moore's curve [13]. Clearly, in multi-core systems the need for memory increases at least linearly with the number of CPU cores used, so the trend is understandable. But the growing divergence between memory size and the basic speed of a single CPU has important consequences for our work.

At a fixed speed, it takes a fixed amount of time to fill one GByte of RAM memory. If we do not change the speed, but we do change the amount of available memory, the time needed to fill that memory will also change. A fast implementation of the SPIN bitstate hashing algorithm, for instance, typically takes three

[1] The research described in this paper was carried out at the Jet propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The work was supported in part by NASA's Exploration Technology Development Program (ETDP) on Reliable Software Engineering.

hours to "fill" a bitstate hash-array of 1 GByte. The runtime will trivially increase by roughly two orders of magnitude if we upgrade our memory from 1 to128 GByte, and try to fill that. Three hours now becomes 12.5 days, which is a little more noticeable.

> We "fill" the bitstate hash-array when we try to perform model-checking on an application that has significantly more reachable states than can fit in memory. At some point during this search, all bits in the hash-array have been set, and the search will complete without having visited all states. Bitstate hashing is the fastest way we know of to search a very large statespace – other techniques can sometimes cover more reachable states but they are invariably very much slower. If therefore we can show that even the bitstate hashing algorithm will become to slow on future machines, a switch to even slower algorithms will not be an option.

If we continue our thought experiment and switch to a machine with 1 TByte of main memory, at the same speed the bitstate hashing algorithm would now take 1,024 times 3 hours to fill the memory. Twelve days now turns into roughly *three months* which makes the approach quite impractical.

There is one bright point in this outlook. When the one TByte desktop arrives on our desktops, it will likely come equipped with hundreds of compute cores as well.

Our back-of-the-envelope calculation shows that it will be quite unattractive, one could even say infeasible, to run the old trusted sequential model checking algorithms on the new systems.

Multi-core model checking techniques have been an active area of research for at least a decade (e.g., [14], [9], [2]). The early indications are, though, that these algorithms do not always give predictable performance, and they do not necessarily scale well to the use of very large numbers of CPUs (e.g., hundreds or thousands). There are several reasons for the difficulty in scaling. A first factor is the communication overhead that is incurred in coordinating a search across multiple machines. Sometimes significant amounts of data must be exchanged between machines to avoid overlapping work. This data exchange can be expensive. In shared memory machines, the very access to shared memory can become costly once the machine reaches a certain size. The root cause of this lies in the use of NUMA (non-uniform memory access) machine architectures, that make the internals of a large distributed machine with thousands of processing nodes look like a mini computer network with substantial overhead required for cache synchronization and relatively slow data transfer times between compute nodes [1], [3].

2. Swarm Verification

Our work on swarm verification has grown out of attempts to use aggressive test randomization techniques to verify the correctness of mission critical software modules we developed for use in a spacecraft. Initially, a direct application of logic model checking techniques to this problem seemed infeasible, due to the impressive size of the search space for even abstract versions of these modules. One of these modules, for instance, implements a robust POSIX-compatible file-system on flash hardware, with guarantees for file-system integrity across sudden reboots, loss of power, and arbitrary bit-flips caused by cosmic radiation. The problem itself is sufficiently challenging that we have also proposed it as an element of the 'Grand Challenge in Verification' project [11].

The state of a flash disk can be abstracted as the number of current, obsolete, and free pages, and the number of free, used, and bad blocks. Both current and obsolete pages carry version numbers and information that ties them to specific files or directories in the file system hierarchy. Even if we abstract from all data contents, and restrict to small numbers of blocks and pages, the number of abstract disk states can be overwhelming. Our first verification attempts were based on the use of a regimen of randomized and differential testing [5]. In differential testing one uses a reference system to check the behavior of the system under test. For the implementation of a file system, reference systems were of course not hard to find. Differences in behavior between the new module and the reference system that are revealed by randomized testing can quickly be diagnosed and remedied.

The differential testing approach also allowed us to test the robustness requirements of our new system, without requiring that the reference system (e.g., a standard Linux file system) satisfied the same requirements. In the event of power loss or sudden reboot, the behavior of our flash file system has to preserve the state of the system as it existed before the interrupted operation is initiated, even for complex multi-step operations, such as moving an entire sub-tree in the file system to a new place in the directory hierarchy. We can perform these checks by verifying that the file system under test restores its disk after the reboot in a way that matches the state of the reference file system *before* the start of the interrupted operation.

The randomization strategy proved effective in finding defects in our initial module implementation. Once the test regimen *stops* finding errors, though, we lose faith that sufficient coverage is realized. At this point, we reinvestigated the possibility to use a model checker to increase the thoroughness of our testing.

Starting with SPIN version 4, the model checkers generated by SPIN can be linked to the compiled C code of an external application to verify hybrid models that consist of high-level behavior fragments specified in PROMELA, and low-level behavior fragments from the application itself [7,8]. The PROMELA fragments in such hybrid models are normally used to formalize assumptions about the environment against which the application is to be verified. In our application, the PROMELA fragments capture possible user behavior as well as the assumed functioning and malfunctioning of the flash hardware (i.e., the upper and lower interfaces to the code being verified). The C code is the actual code from the flash file system we built.

As noted, exhaustive verification of even an abstracted version of this type of system cannot be completed in any reasonable amount of time, no matter how much memory is available to perform the search. Given a runtime limit of say one day, it is also not attractive to simply cut-off a standard search once the time limit is reached. The model checker would explore only a relatively insignificant initial portion of the search tree: the top few layers if a breadth-first search is used, or the left-most part of the search tree if a depth-first search is used. In both cases, critically important behavior would not be explored at all, and in that sense even random testing techniques can have a better chance at finding the errors that are hiding there.

We can improve our odds of finding errors by combining three basic ideas to modify the search process:

1. search randomization,
2. search diversification, and
3. search parallelization.

The parallelization of the search is meant to exploit the availability of both small multi-core systems, and potentially vast numbers of computers in a larger network. To do so, we choose an "embarrassingly parallel" approach, rather than the more sophisticated strategies normally used in the multi-core application of model checking, e.g. [9]. To gain optimal advantage from the parallelism, the algorithms we use require no communication between processors at all, thus avoiding one of the main bottle-necks in the scaling of multi-core algorithms.

Each of the three basic techniques is very simple when used separately, but the combination with a model checking engine can turn their joint use into an unexpectedly powerful approach.

2.1. Randomization and Diversification

Search randomization is relatively simple to implement in a model checker. At each point where the model checker must resolve a non-deterministic choice, there normally is a pre-determined order in which it will do so. In very large applications, the search typically cannot be performed exhaustively, being limited by either a time bound (e.g., a day of computation) or a memory bound (e.g., 32 GByte). Since in these cases not all available search options can be explored, it becomes important to choose the ones that *are* explored more fairly. Randomization techniques can come to the rescue here.

We can distinguish between two basic sources of non-determinism in verifications with the SPIN model checker: (1) process scheduling decisions and (2) non-deterministic choices made within processes. The first type of non-determinism is fundamental to the modeling of asynchronous process behavior in a distributed system. The latter type is typically the result of the use of abstractions in the model itself. In our case, these abstractions are captured in the PROMELA portion of the verification models (and not in the embedded C code portion). The current version of SPIN (version 5.1.6) has support for modifying the default behavior of the model checker for both types of decisions.

For process level non-determinism, i.e., transition selection within a process, a first method to change the search order is to simply perform the search for executable transitions in reverse – starting with what otherwise would be the last transition inspected. This is done by compiling the pan.c model checker generated by SPIN with compiler directive -DT_REVERSE. Another method is to randomize that part of the search, which can be more effective if there are larger numbers of enabled transitions available for execution. The latter search mode is enabled by using the compiler directive –DRANDOMIZE=N, where N is an arbitrary integer that is used to seed the random number generator that SPIN will now use to perform the search.

By varying N, and performing searches forwards or in reverse, we can already vary the search strategy over a fairly broad range. There is a second level of the search that we can modify though, and that is at the level where process scheduling decisions are made.

SPIN supports a third new directive, –DREVERSE, to reverse search over processes in the resolution of top-level scheduling decisions. Randomizing the search engine at this point as well could provide a fourth option, but this capability is not quite as straightforward to implement and remains pending.

We now have three ways to influence the search order, plus the capability to vary the seeding of the random number generator in one of these options. This, in combination with other existing search options such as breadth-first search, , and standard depth-first search, hash-compact, depth-limited search, and bitstate hashing with different hash-arrays sizes, gives us the capability to perform a wide diversity of searches, each with a different chance of finding errors in very large statespaces.

2.2. Diversification and Parallelization

The capabilities we have described could in principle be used on a single CPU, by performing multiple searches sequentially. The true power of search diversification and randomization comes, though, if we can perform the searches in parallel. All searches we have described can be performed completely independently, on separate hardware. No access to shared memory is required and no communication overhead will be involved in running these 'verification tests.'

We use the term 'verification test' deliberately here, to distinguish this approach from exhaustive verification, which can provide a stronger guarantee of the correctness (or incorrectness) of an application. In a verification test our aim is to use a model checker to increase the coverage of a test suite significantly, when we compare this approach with standard testing. While doing so, we accept that within available constraints of memory and time our verification effort cannot be truly exhaustive.

To support the generation and parallel execution of a broad diversity of search jobs, we built a relatively simple scripting tool called 'swarm.'[2] [10]

4. Tool Support and Application

The swarm tool is a relatively simple front-end to SPIN. The tool reads a configuration file that details the specific constraints under which we would like to execute the verification test and uses it to generate a script that can run hundreds of small verification jobs on one or more CPUs. A sampling of options from the default swarm configuration file is as follows. Lines starting with a pound sign are comments.

```
# ranges
w   20    32
d   100   10000
k   2     5

# limits
cpus      2
memory    512M
time      1h
```

The first group of options shown here defines the range of search depths (d), hash-array sizes (w) and number of hash-functions that can be used to define alternate search modes. The depth range can be used to define a series of depth-bounded searches. The has-array size range can be used to define a series of bitstate hash runs with varying coverage, and hence with varying runtime limits.

The second group of options defines a few more global constraints: the number of CPUs that is available to perform the searches, the amount of memory that can maximally be used per job, and the maximum time that can be used for the entire sweep of verification tests that will be generated by swarm based on these settings. For the memory settings, the suffixes M and G are recognized, and for the time settings the suffixes d (days), h (hours), m (minutes), and s (seconds) are recognized. Swarm uses the bounds to generate a script that can be executed by the user to perform the verification test. Optionally, the user can also specify additional search modes and ways to compile the pan.c model checking code, to expand the range of options that swarm can use.

For larger applications the swarm tool typically generates several hundred different search jobs, that can be completed within a one or two hour time limit. In our applications, the model checking code in pan.c itself is compiled in nine different ways by swarm, using the directives we discussed in the previous section. Multiple runs are performed with each of those nine executables, to boost the level of search diversification. Some runs, e.g., those performed with bitstate hashing for small hash-array sizes, complete in seconds. Others can take a more significant portion of the allocated time. Because the slower running jobs can be executed on separate processors or processing cores, they do not interfere with the faster ones, and no time is lost.

For the verification of our mission-critical file system modules, we used two 8-core machines for the verification tests, each with 32 GByte of memory, and

[2] http://spinroot.com/swarm/

we performed each set of verification tests within a time bound of one hour.

The effectiveness of this approach has surprised us. Counter-examples are typically generated within the first few minutes of a search, and often the result of the one hour of running swarm jobs is a range of different failure scenarios, rather than one single trail-file. Defects that reveal themselves only after hours or days of standard exhaustive verification attempts with SPIN on a single processor now show up in minutes with the swarm approach. The results of our applications have been strong enough that the swarm testing has become the primary mode of testing that we perform on all large applications.

It should be noted that the applications that we apply this type of verification testing to are indeed large enough that exhaustive verification would be prohibitively expensive. Each application includes a significant amount of embedded C code. The file system module, for instance, defines a state-vector of 3,623 bytes (with a worst-case number of $2^{10,000}$ system states). The model checking system for this application is setup to perform a broad range of randomized differential tests, under the control of the model checker to avoid repeatedly visiting the same states (which is much more difficult to prevent in a standard test run).

To still support the differential test approach also when using the model checker to perform the search for errors, we include *two* full software modules into the SPIN-driven verification tests, as embedded C code. One module contains the file system code being tested; the second module contains the reference model against which its operation is being compared. In our case, both modules execute completely in core (e.g., for obvious reasons we do not use an external file system for the reference system when executing in this mode). For the file system application, each of the two modules used currently consists of roughly 5,000 lines of C. SPIN uses the c_track primitive [7] to track the memory used in these modules. Using these primitives, the model checker can set and restore each module to its proper state during the search.

We have also applied this method to re-verify several large applications that we, and others, studied in the past. For two of these applications the swarm verification tests revealed errors that had been missed in the earlier verification attempts with the standard sequential model checking algorithms [10]. One of these two applications was a model extracted from the call processing code of a commercial voice and data-switch [6], the other was a model of an experimental Fleet processor architecture design, provided by R. Limaye and N. Sundaram from UC Berkeley. In both cases, the randomized search process could quickly home in on errors that were too deep in the search tree for a standard run to find.

4. Conclusion

The relatively recent switch of chip makers from the development of single-core to multi-core chips, as used for desktop systems, is matched by a similar trend that applies to the internet as a whole. Several names have been used to describe this new trend. The term "grid computing" first appeared in the nineties (e.g., [4]). More recent are terms such as "cloud computing" and "network centric computing." Companies such as Amazon, Microsoft, and Google are all working on services that can make thousands of computers available to customers for a defined period of time, thus enabling the user to run massively parallel jobs.

We believe that this trend will continue. Once it solidifies it will dramatically change the way software verification tasks can be performed. Still, despite many years of work in this area, we do not know of multi-core model-checking algorithms that can scale effortlessly to the use of thousands of loosely connected computers in a network, so existing technology does not yet allow us to take full advantage of the vast array of compute power that may soon become available.

In this paper we have outlined a relatively simple approach that will allow us to leverage this trend to some extent. On multi-core systems with large memories, swarm can produce scripts that execute a large diversity of verification jobs that complete within a user-defined time bound, whether is a minute, an hour, or several days. The same principle can be used for using swarm verification scripts on the network at large. In a larger network with thousands of available computers, swarm may generate hundreds of thousands of small verification jobs, all randomly different, to search different parts of a very large statespace and thus optimize our chances of finding bugs.

The 'swarm' part in the term swarm verification means that we both 'swarm' the statespace of a large model checking application, and we 'swarm' the set of computers available to perform the verification as efficiently as possible. A good extension of this work would be to find a way to divide a large verification job into smaller pieces that can all be verified separately, so that we can derive more solid guarantees from a swarm verification effort. If sufficiently many computers are available, it may not even matter much if there is redundancy in such an effort.

5. References

[1] J. Appavoo, V. Uhlig , D. da Silva, "Scalability: The Software Problem," *Proc. Second Workshop on Software Tools for Multi-Core Systems*, San Jose, CA, March 2007.

[2] J. Barnat, L. Brim, and P. Rockai, "Scalable Multi-Core LTL Model-Checking," *Proc. 14th Spin Workshop*, Berlin, Germany, July 2007, Springer,LNCS 4595.

[3] R. Bryant and J. Hawkes, "Linux Scalability for Large NUMA Systems," *Proc. Linux Symp.*, June 2003, Ottawa, Canada, pp.83-95.

[4] Foster, I., and C. Kesselman, *The Grid: Blueprint for a new computing infrastructure*, Morgan Kaufmann Publ., San Francisco, CA., 1998 (1st ed.).

[5] A. Groce, G.J. Holzmann, R. Joshi, "Randomized differential testing as a prelude to formal verification," *Proc. Int. Conf. on Software Engineering* (ICSE 2007). Minneapolis, Minnesota, May 2007, pp. 621-631.

[6] G.J. Holzmann and M.H. Smith, "An automated verification method for distributed systems software based on model extraction," *IEEE Trans. on Software Engineering*, Vol. 28, No. 4, pp. 364-377, April 2002.

[7] G.J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004.

[8] G.J. Holzmann and R. Joshi, "Model-driven software verification." *Proc. 11th Spin Workshop*, Barcelona Spain, April 2004, Springer, LNCS 2989, pp. 77-92.

[9] G.J. Holzmann, and D. Bosnacki, "The design of a multi-core extension of the Spin model checker," *IEEE Trans. On Software Eng.*, Vol. 33, No. 10, Oct. 2007, pp. 659-674.

[10] G.J. Holzmann, R. Joshi, and A. Groce, "Tackling large verification problems with the Swarm tool," *Proc. 15th Spin Workshop*, Los Angeles, US, August 2008, Springer, LNCS 5156, pp. 134-143.

[11] R. Joshi and G.J. Holzmann, "A mini challenge: build a verifiable file-system," Grand Challenge in Verification, Verified Software: Theories, Tools, Experiments, Zurich, Sw., October 2005, *Formal Aspects of Computing*, 2007, Vol. 19, 4 pgs.

[12] K.L. McMillan: *Symbolic Model Checking*. Kluwer Academic Publishers, 1993

[13] G.E. Moore, Cramming more components onto integrated circuits, *Electronics*, 38, (8), April 9, 1965.

[14] U. Stern and D. Dill. "Parallelizing the Murφ verifier," *Proc. 9th Int. Conf. on Computer Aided Verification*, Haifa, Israel, June 1997, Springer, LNCS 1254, pp 256–278.