Tackling Large Verification Problems with the Swarm Tool[1]

Gerard J. Holzmann, Rajeev Joshi, Alex Groce

Laboratory for Reliable Software (LaRS)
Jet Propulsion Laboratory, California Institute of Technology.

Abstract. The range of verification problems that can be solved with logic model checking tools has increased significantly in the last few decades. This increase in capability is based on algorithmic advances, but in no small measure it is also made possible by increases in processing speed and main memory sizes on standard desktop systems. For the time being, though, the increase in CPU speeds has mostly ended as chip-makers are redirecting their efforts to the development of multi-core systems. In the coming years we can expect systems with very large memory sizes, and increasing numbers of CPU cores, but with each core running at a relatively low speed. We will discuss the implications of this important trend, and describe how we can leverage these developments with new tools.

Introduction

The primary resources in most software applications are time and space. It is often possible to make an algorithm faster by using more memory, or to reduce its memory use by allowing the run time to grow. In the design of SPIN, a reduction of the run time requirements has almost always taken precedence.

For an exhaustive verification, the run time requirements of SPIN are bounded by both the size of the reachable state space and by the size of available memory. If M bytes of memory are available, each state requires V bytes of storage, and the verifier on average records S new states per second, then a run can last no longer than M/(S*V) seconds. If, for example, M is 64 MB, V is 64 bytes, and S is 10^4 states per second, the *maximum* runtime would be 10^2 seconds. If there are more than 10^6 reachable states, the search will remain incomplete – being limited by the size of memory.

The verification speed depends primarily on the average size of the state descriptor, which is typically in the range of 10^2 to 10^3 bytes. On a system running at 2 or 3 GHz the processing speed is normally in the range of 10^5 to 5.10^5 states per second. This means that in about one hour, the model checker can explore 10^8 to 10^9 states, provided sufficient memory is available to store them. This means that some 10^{11} bytes, or 100 GB, can be used up per hour of runtime.[2] On an 8 GB system, that means that the model checker can (in exhaustive storage mode) normally run for no more than about 5 minutes. If we switch to a 64 GB system running at the same clock-speed, the worst-case runtime increases to 40 minutes.

An interesting effect occurs if we switch from exhaustive verification mode to *bitstate* mode, where we can achieve a much higher coverage of large state spaces by using less than a byte of memory per state stored [H87]. The exact number of bytes stored per state is difficult to determine accurately in this case. The current version of SPIN by default uses three different hash-functions, setting between one and three new bit-positions for each state explored. We will assume conservatively here that each state in this mode consumes 0.5 bytes of memory, and that the speed of the model checker is approximately 10^8 states per hour. Under these assumptions, the model checker will consume maximally $10^8 * 0.5$ bytes of memory per hour of run time, or roughly 50 MB. To use up 8 GB will now take about a week (6.8 days) of non-stop computation. In return, we cover significantly more states, but both time and space should be considered limited resources, so the greater number of states is not always practically achievable. To make the point

[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.

[2] Smaller state descriptors normally correlate with higher processing speeds.

more clearly, if we increase the available memory size to 64 GB, a bitstate search could consume close to two months of computation, which is clearly no longer a feasible strategy, no matter how many states are explored in the process.

We are therefore faced with a dilemma. The applications that we are trying to verify with model checkers are increasing in size, especially as we are developing methods to apply logic model checkers directly to implementation level code [H00, HJ04]. As state descriptors grow in size from tens of bytes to tens of kilobytes, processing speed drops and is no longer offset by continued CPU clock-speed increases. For very large applications, a bitstate search is typically the only feasible option as it can increase the problem coverage (i.e., the number of reachable states explored) by orders of magnitude. Exhaustive coverage for these applications is generally prohibitively expensive, given the enormous size of both the state descriptors and the state spaces, no matter which algorithm is used. In these cases we have to find ways to perform the best achievable verification. Technically, the right solution in these cases is always to apply strong abstraction techniques to reduce the problem size as much as possible. We will assume in the remainder of this paper that the best abstractions have already been applied and that the resulting state space sizes still far exceed the resource limits in time and/or memory.

Leveraging Search Diversity

To focus the discussion, we will assume that there is always an upper bound on the time that is available for a verification run, especially for large problem sizes. We will assume that this bound is ***one hour***. With a fixed exploration rate this means that we cannot use more than a few Gigabytes of memory in exhaustive verification and no more than about 50 to 500 MB in bitstate exploration. Given that for very large verification problems we have to accept that the search for errors within a specific time constraint will generally be incomplete, it is important that we do not expend all our resources on a single strategy. Within the limited time available, we should approach the search problem from a number of different angles – each with a different chance of revealing errors.

A good strategy is to leverage both *parallelism* and *search diversity*. The types of applications that then become most promising fall in the category of "embarrassingly parallel" algorithms.

To illustrate our approach, we will use a simple model that can generate a very large state space, where we can easily identify every reachable state and predict when in a standard depth-first search each specific state will be generated. The example is shown in Figure 1.

.
```
byte pos = 0;
int val = 0;
int flag = 1;

active proctype word()
{
end: do
     :: d_step { pos < 32 -> /* leave bit 0 */ flag = flag << 1; pos++ }
     :: d_step { pos < 32 -> val = val | flag; flag = flag << 1; pos++ }
     od
}

never { do :: assert(val != N) od } /* check if number N is reached */
```

Fig. 1. Model to generate all 32-bit values, to illustrate the benefits of search diversification

The model executes a loop with two options, from which the search engine will non-deterministically select one at each step. Each option will advance an index into a 32-bit integer word named *val* from the least significant bit (at position one) to the most significant bit (at position 32). The first option leaves the bit pointed to set to its initial value of zero, and merely advances the index. The second option sets the bit pointed at to one. Clearly, there will be 2^{32} (over 4 billion) possible assignments to *val*. Each state descriptor is quite small, at 24 bytes, but storing all states exhaustively would still require over 100 GB.

If we perform the state space exploration on a machine with no more than 2 GB, an exhaustive search cannot reach more than 2% of the state space. A bitstate search on the other hand could in principle store all states, but only under ideal conditions. For this model, with a very small state descriptor, we reach a processing speed of close to 2 million states per second, on a standard 2.3 GHz system. We will, however, artificially limit the amount of memory that we make available for the bitstate hash array to 64 MB and study what we can achieve in terms of state coverage by exploiting parallelism and search diversification techniques. In terms of SPIN options, this means the selection of a *pan* runtime flag of maximally –w29. In practice this means that for this example only about 148 million states are reached in bitstate mode, or no more than 3.5% of the total state space.

For this example it is also easy to check if a specific 32-bit value is reached in SPIN's exploration, by defining the corresponding value for N when the model is generated. For instance, checking if the value negative one is reached in the maximal bitstate search can be done as follows:

```
$ spin –DN=-1 –a model.pml
$ cc –DMEMLIM=2000 –DSAFETY –o pan pan.c
$ ./pan –w29
```

This particular search fails to produce a match. It is easy to understand why that is. Note that the value negative one is represented in two's complement as a series of all one bits, which in the standard depth-first search is the *last* number that would be generated by the verifier. Performing the same search for the value positive one will produce an almost immediate match, for the same reason. If we reverse the order of the two options in the model of Figure 1, the opposite effect would occur: the search for negative one would complete quickly and the search for positive one would fail.

If the number to be matched is randomly chosen, we could not devise a search strategy that can optimize our chances of matching it, which is more representative of a real search problem. After all, if we know in advance where the error might be, we would not need a model checker to find it.

In the experiments that we will describe we will use a list of 100 randomly generated numbers, and compare different methods for matching as many of them as possible. If the random number generator behaves properly, the random numbers will be distributed evenly over the entire state space of over 4 billion reachable states. Statistically, the best we could expect to do in any one run would be to uncover no more than 3 or 4 of those states (given that with runtime flag –w29 we can explore at most 3.5% of the reachable state space). We will see that with a diversified search strategy, we can do significantly better and identify 49% of the randomly generated states. We will also show that even when using only 4 MB (a tiny fraction of the 100 GB that would be required to store the full state space) we can already identify 10% of the target numbers.

Algorithms

To make our method work we have to be able to use as many different search methods as there are processing cores available to us. If each search is setup to use only a small fraction of the total memory that is available on our system, we can run all searches in parallel. In the description that follows we describe a number of different search algorithms. Several of these algorithms can be modified to form any number of additional searches, each of which able to search a different part of the state space. The base algorithms we use can be described as follows.

1. *(dfs)* The first method is the standard depth-first search that is the default for all SPIN verifications.
2. *(dfs_r)*The next method reverses the order in which a list of non-deterministic choices within a process is explored, using the compiler directive –D_TREVERSE (new in SPIN version 5.1.5).
3. *(r_dfs)* The next method uses a search randomization strategy on the order in which a list of transitions is explored, using the existing compiler directive –DRANDOMIZE (first introduced in SPIN version 4.2.2). With this method the verifier will randomly select a

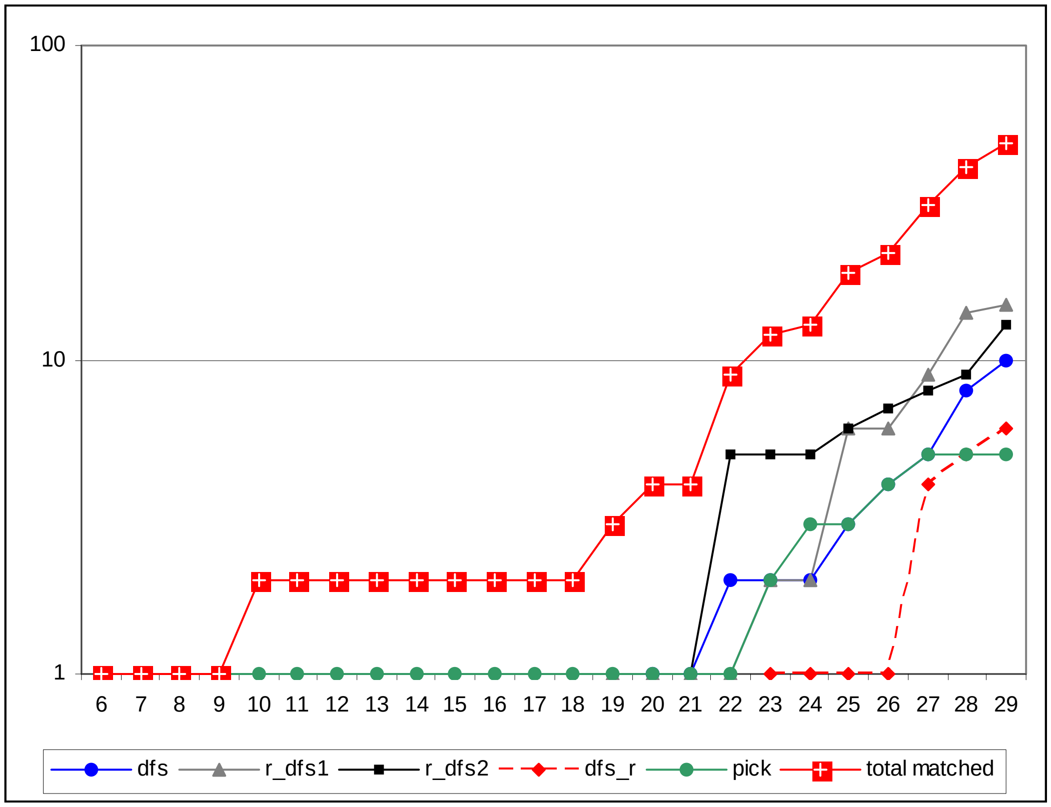

Fig. 2. Results of 2,400 Verification Runs for the Model in Figure 1 (logscale).

starting point in the transition list, and start checking transitions for their executability in round-robin order from the point that was randomly selected.

4. *(pick)* The next method uses a user defined selection method to permute the transitions in a list.
5. The last method reverses the order in which process interleavings are explored, using the compiler directive –DREVERSE (introduced in SPIN version 5.1.4).

Because our example uses just a single process, we will not use the last variation of the search. Alternative methods for modifying process scheduling decisions during a search can be found in [MQ08].

Algorithms 3 and 4 can be used to define a range of search options, using different seeds for the random number generator. To illustrate this, we will use two versions of algorithm 3, called *r_dfs1* and *r_dfs2*. Each algorithm from the set can be used in a series of runs. In our tests we repeated each run 100 times, once for each number from the list of 100 target numbers to match. We then repeat each of these 100 runs 24 times, while varying the size of the hash-array used from our limit value of –w29 (64 MB) down to a minimum of –w6 (64 bytes). This is an application of *iterative search refinement*, as first discussed in [HS00].

About 2,000 runs from this series of experiments, for hash array sizes from –w6 up to and including –w23 take less than a second of runtime each, so despite the large number of runs, they can be completed very quickly. For the larger hash array sizes (2MB and up) the runtime and the number of states covered becomes more notable, with the longest runs taken 84 seconds each on a 2.4 GHz system. All 24 runs combined take no more than about 3 minutes of real time when run sequentially, which means that all 2,400 runs can be completed in about 5 hours on a single CPU core, or in about 37 minutes total on the 8-core machine that was used for these experiments. The results are shown in Figure 2.

When used separately, and performing one verification run alone, none of the search methods identify more than about **9%** of the target values set for this experiment. If we look at the cumulative effectiveness of the iterative search refinement method, using 24 runs of a each algorithm and increasing the size of the hash array step by step, this coverage increases, with the best performing search method (*r_dfs2*) identifying **15%** of all targets. The performance of all five search strategies combined in our proposed diversified multi-core search strategy increases the coverage to the identification of **49%** of all targets.

The top curve in Figure 2 shows the *cumulative* number of matches (out of 100) as the memory arena is increased from 64 bytes (-w6) to 64 MB (-w29). The other curves show the performance of the individual search algorithms.

Each different search method identifies a different set of targets, thus boosting the overall effectiveness of the use of all methods combined. Adding more variations of the searches could increase the problem coverage still further. It is evident from these data that the performance of the diversified multi-core search is significantly better than any one search method in isolation.

The diversified multi-core search promises to be a very valuable addition to the range of techniques that we can use to tackle very large software verification problems. It builds directly on the availability of systems with increasing numbers of CPU cores and rapidly growing memory sizes, that we can expect in the coming years and perhaps decades. There may also be a direct application of this strategy of swarm verification in grid computing, using large numbers of standard networked computers. In this paper, though, we focus on the application to multi-core systems only.

The Swarm Tool

Based on the observations above, we developed a tool that allows us to leverage the effect of search diversification on multi-core machines. The Swarm tool, written in about 500 lines of C, can generate a large series of parallel verification runs under precise user-defined constraints of time and memory. The tool takes parameters that define a time-constraint, the number of available CPUs, and the maximum amount of memory that is available. (For a manual page see: http://spinroot.com/swarm/).

Swarm first calculates how many states could maximally be searched within the time allowed, and then sets up a series of bitstate runs. By limiting the hash arena in a bitstate run, Swarm controls what the maximal time for each run will be. Parallelism is used to explore searches with different search strategies to provide diversity. The commands that are generated include standard, randomized, and reverse depth-first search orders, varying depth-limits, and using a varying numbers of hash-functions per run. In a relatively small amount of time, hundreds of different searches can be performed, each different, probing different parts of an oversized state space.

A typical command line invocation of the Swarm tool is as follows:

```
$ swarm –c4 –m16G –t1 –f model.pml > script
swarm: 33 runs, avg time per cpu 3593.8 sec
```

In this case we specify that we want to use 4 CPU cores for the verifications, and have up to 16 GB of memory available for these runs. The –t parameter sets the time limit for all runs combined to one hour. Swarm writes the verification script onto the standard output, which can then be written into a script file. Executing the script performs the verification.

Application: An example of a large problem that cannot be handled with standard search methods is a SPIN model of an experimental Fleet processor architecture. The details of the design itself are not of interest to us here, but the verifiability of the model is.[3] One version of this model has a known assertion violation that can be triggered through a manually guided simulation in about 350 steps.

The model is over one thousand lines of PROMELA.[4] Each system state is 1,440 bytes. An attempt to perform a full verification on a machine with 32 GB of memory runs at roughly 10^5 states per second, and exhausts memory in 195 seconds without reporting the error. At this point the search explored 23.4 million states, corresponding to an unknowable fraction of the reachable state space. A search using -DCOLLAPSE compression (a lossless state compression mode) reaches 327.6 million states before running out of

[3] The Spin model of the Fleet Architecture Design was built by Rhishikesh Limaye and Narayan Sundaram under the guidance of Sanjit Sehia from UC Berkeley.

[4] The specification language of the SPIN model checker.

memory after 3,320 seconds. A run with hash-compact (a stronger, but not lossless, form of compression) runs out of memory after 1,910 seconds and increases the coverage to 537 million states. A bitstate run, using all 32 GB of memory, runs for 34 days, and explores over 10^{11} system states. None of these search attempts succeed in locating the assertion failure.

The full reachable state space for this problem is likely to be orders of magnitude larger than what can be searched or stored by any verification method. The bitstate run can be performed in parallel on 8 CPUs, shrinking the run time from 34 days to about 5 days [HB07], but without change in result. An alternative would be to run the verification with –DMA compression, which is lossless and often extremely frugal in memory use. Such a run could in principle be able to complete the verification and reveal the error, but it would likely take at least a year of computation to do so.

A Swarm run for this application is quickly setup. Swarm generates 74 small jobs in 8 scripts that can be executed in parallel on the 32 GB machine, when given a time limit of one hour (the default). Executing the script finds the assertion violation within a few seconds. In this case by virtue of the inclusion of the reversed depth-first search. The assertion violation, as it turns out, normally happens towards the end of the standard depth-first order– which means that it is encountered near the very beginning of the search if the depth-first search order is reversed.

For a different test of the performance of Swarm we also studied a series of large verification models from our benchmark set, most of which were also used in [HB07]. EO1 is a verification model of the autonomous planning software used on NASA's Earth Observer-1 mission [C05]. The Fleet architecture model was discussed above. DEOS is a model of an operating system kernel developed at Honeywell Laboratories, that was also discussed in [P05]. Gurdag is a model of an *ad hoc* network structure with five nodes, created by a SPIN user at a commercial company. CP is a large model of a telephone switch, extracted from C source code with the Modex tool. DS1 is a large verification model with over 10,000 lines of embedded C code taken from NASA's Deep Space 1 mission, as described in [G02]. NVDS is a verification model of a data storage module developed at JPL in 2006, with about 6,000 lines of embedded C code, and NVFS is non-volatile flash file system design for use on an upcoming space mission, with about 10,000 lines of embedded C.

For each of these models we first counted the number of local states in the automata that SPIN generates for the model checking process. This is done by inspecting the output of command "pan –d." Next, we measured the number of these local states that is reported as *unreached* at the end of a standard depth-first search with a bitstate hash-array of 64 MB (using runtime flag –w29, as before). Next, we used the Swarm tool to generate a verification script for up to 6 CPUs and 1 hour of runtime. We then measured how many of the local states remained unreached in *all* runs.

Table 1. Swarm Coverage Improvement for Eight Large Verification Models

Verification Model	Number of Control States			Percent of Control States Reached	
	Total	Unreached Control States			
		standard dfs	dfs + swarm	standard dfs	dfs + swarm
EO1	3915	3597	656	8	83
Fleet	171	34	16	80	91
DEOS	2917	1989	84	32	97
Gurdag	1461	853	0	41	100
CP	1848	1332	0	28	100
DS1	133	54	0	59	100
NVDS	296	95	0	68	100
NVFS	3623	1529	0	58	100

The last two columns of **Table 1** show the percentage of control states reached, respectively in the original depth-first search using a 64 MB hash-array, and in that search plus all Swarm verification runs. In all cases the coverage increases notably. For the EO1 model performance increases from 8% to 83%. In the next two cases, coverage increases to over 90%. In the last five large applications we see coverage by this metric reach 100% percent of the control states. It should be noted that this last result does not mean that the full reachable state space was explored. For the models considered here achieving the latter would be well beyond our resource limitations, which is precisely why we selected them as candidates for the evaluation of Swarm verification.

All measurements were performed on a 2.3 GHz eight-core desktop system with 32 GB of main memory, of which no single run consumed more than 64 MB in these tests. The state vector size for the models ranges from a180 (NVDS) to 3426 (DS1) bytes of memory. The number of Swarm jobs that can be executed within our 1 hour limit ranged from 86 (EO1) to 516 (NVDS)..

Swarm unexpectedly succeeded in uncovering previously unknown errors in both the CP model and the NVFS applications. The NVFS application is relatively new, but the CP verification model was first subjected to thorough verification eight years ago, and has since been used in numerous tests without revealing any errors. In our own applications, Swarm has become the default method we use for large verification runs.

Conclusion

It is often assumed that the best way to tackle large verification problems is to use all available memory in a maximal search, possibly using multi-core algorithms, e.g., [HB07], to reduce the runtime. As memory sizes grow, most search modes that would allow us to explore very large numbers of states take far too much time (e.g., months) to remain of practical value. In this paper we have introduced a new approach that allows us to perform verifications within strict time bounds (e.g., 1 hour), while fully leveraging multi-core capabilities. The Swarm tool uses parallelism and search diversity to optimize coverage.

We have measured the effectiveness of Swarm in several different ways. In each case we could determine that the new approach could defeat the standard method of a single depth- or breadth first search by a notable margin, both by dramatically reducing runtime and by increasing coverage.

A similar approach to the verification problem was explored in [D07], where it was applied to the verification of Java code with the Pathfinder tool, though without considering run-time constraints.

The use of embarrassingly parallel approaches, like Swarm, becomes increasingly attractive as the number of processing cores and the amount of memory on desktop systems continues to increase rapidly.

An often underestimated aspect of new techniques is the amount of training that will be required to fully leverage them. This is perhaps one of the stronger points in favor of the Swarm tool. It would be hard to argue that the use of Swarm requires more training than a cursory reading of the manual page.

Acknowledgements

The authors are grateful to Sanjit Seshia, Rhishikesh Limaye, Narayan Sundaram, for providing access to and insight in the Fleet Architecture models, and to Madan Musuvathi and Klaus Havelund for inspiring discussions about search strategies. Doron Peled proposed the introduction of the –DRANDOMIZE option in SPIN version 4.2.2.

References

[C05] S. Chien, R. Sherwood, D. Tran, et al, Using Autonomy Flight Software to Improve Science Return on Earth Observing One (EO1), *Journal of Aerospace Computing, Information, and Communication*, April 2005.

[D07] M.B. Dwyer, S.G. Elbaum, et al., Parallel Randomized State-Space Search, *Proc. ICSE 2007*, pp. 3-12.

[M69] G.E. Moore, Cramming more components onto integrated circuits, *Electronics*, 38, (8), April 9, 1965.

[G02] P. R. Gluck and G. J. Holzmann, Using Spin Model Checking for Flight Software Verification, *Proc. 2002 Aerospace Conf.*, IEEE, Big Sky, MT, USA, March 2002.

[H87] G.J. Holzmann, On limits and possibilities of automated protocol analysis. *Proc. 6th Int. Conf. on Protocol Specification, Testing, and Verification*, INWG IFIP, Eds. H. Rudin and C. West, Zurich, Switzerland, June 1987.

[H00] G.J. Holzmann, Logic verification of ANSI-C Code with Spin, *Proc.7th Spin Workshop*, Stanford University, CA, August 2000, Springer Verlag, LNCS 1885, pp. 131-147.

[HS00] G.J. Holzmann and M.H. Smith, Automating software feature verification, *Bell Labs Technical Journal*, Vol. 5, No. 2, pp. 72-87, April-June 2000.

[HJ04] G.J. Holzmann and R. Joshi, Model-driven software verification, *Proc. 11th Spin Workshop*, Barcelona, Spain, April 2004, Springer Verlag, LNCS 2989, pp. 77-92.

[HB07] G.J. Holzmann and D. Bosnacki, The design of a multi-core extension to the Spin model checker, *IEEE Trans. On Software Engineering*, 33, (10), pp. 659-674, Oct. 2007.

[MQ08] M. Musuvathi and S. Qadeer, Fair stateless model checking. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, (PLDI) Tucson, AZ, June 7-13, 2008.

[P05] J. Penix, W. Visser. C. Pasareanu, E. Engstrom, A. Larson and N. Weininger, Verifying Time Partitioning in the DEOS Scheduling Kernel, *Formal Methods in Systems Design Journal*, Volume 26, Issue 2, March 2005