Swarm Verification Techniques

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce

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 and new theoretical insights, but it has also benefitted from the steady increase in processing speeds and main memory sizes on standard computers. The steady increase in processing speeds, though, ended when chip-makers started redirecting their efforts to the development of multi-core systems. For the near-term future, we can anticipate the appearance of systems with large numbers of CPU cores, but without matching increases in clockspeeds. We will describe a model checking strategy that can allow us to leverage this trend, and that allows us to tackle significantly larger problem sizes than before.

Index Terms—software engineering tools and techniques, logic model checking, distributed algorithms, software verification.

1 INTRODUCTION

Like other compute intensive applications, logic model checking techniques have benefitted from the steady increases in CPU-speeds and memory sizes in the last few decades, following Moore' s prediction from 1965 [M65]. The current trend in chip development, though, is a move away from further increases in CPU speed and is instead focused on increasing the number of CPU cores.

To continue to scale applications of logic model checking to larger problem sizes, then, we must be able to leverage the availability of potentially large numbers of processors that run at a mostly fixed and relatively low speed. These types of systems are not only increasingly available in the form of multi-core desktop systems, but also more generally as networked computers or server farms offering grid- or cloud-computing services. In this paper we report on an application of logic model checking that can build on these trends.

We focus here on the SPIN model checker [H04] as a representative tool for solving computationally expensive search problems, but the basic principles of parallelism and search diversification that we explore apply also more generally to other types of search-based processes.

The SPIN model checker can be used to locate violations of formalized correctness properties in distributed software system designs. It does so by performing a search in a carefully defined subset of the possible executions of the system. The details of the search process, and the underlying theory of logic model checking, are not important to us here. What is important are the constraints that we face when trying to handle larger and larger problem sizes. In the early days of model checking, the limited size of main memory was often the most important constraint on the size of problems that could be handled. Today, this is no longer the case. Memory sizes have increased dramatically, and are expected to continue to increase for some time to come. As we will show, though, even with the currently available memory sizes (near 10^2 GB), a SPIN model checking run that would attempt to use all of main memory will generally require more time to complete than we are normally willing to spend (i.e., days or weeks). Our goal in this study is to develop a verification strategy that allows us to obtain high quality results for large verification problems in minutes or hours, not days or weeks, and we want those results to scale with increasing numbers of processing cores or CPUs.

The paper is structured as follows. In Section 2.1 we first look at how memory size and runtime are related for SPIN based verifications. The basic observation is simple: as the data-structures we build in main memory grow, so does the time that is required to do so, until we reach a point where we can no longer afford to spend that time. In Section 2.2 and 2.3 we look at different strategies that can be, or that have been, used to tackle this problem. In Section 3 we look in detail at the different types of algorithms we can use to diversify and to randomize the model checking algorithm used in SPIN. In Section 3.1 we first study the effect of diversification alone. In Section 3.2 we then consider the added benefit of randomizations. In Section 4 we measure the performance of our proposed new strategy on a range of applications, with added detail provided in Appendix A. Section 5 shows how the setup for a swarm search can be simplified with a small preprocessor for the SPIN model checker, which we have called SWARM. Section 6 discusses related work and Section 7 concludes the paper.

2 SPIN

The time requirement of a SPIN verification run is bounded by both the size of the reachable state space and the size of available memory. If M bytes of memory are available, each state requires V bytes of storage, and the model checker on average explores S reachable states per second, then a verification run can last no longer than

• *Gerard Holzmann and Rajeev Joshi are with the Jet Propulsion Laboratory, California Institute of Technology, and can be reached via E-mail: firstname.lastname@jpl.nasa.gov. Alex Groce is with the School of Electrical Engineering and Computer Science at Oregon State University, E-mail: agroce@gmail.com.*

Manuscript received (May 2009).

M/(S*V) seconds. If, for example, M is 64 MB, V is 64 bytes, and S is 10^4 states per second, then the maximum runtime within this memory arena is 10^2 seconds. The search terminates either when all reachable states have been visited, or when memory is exhausted. If there are more states than can be stored in memory, the search will remain incomplete.

2.1 Bitstate Verification

An interesting effect occurs if we switch from an exhaustive verification mode, as outlined above, to a *bitstate* verification [H87]. In this mode, the model checker can achieve a much higher coverage of large state spaces by using just a few bits of memory per state stored. The precise number of bits that is used per state cannot be determined accurately in this case, since states can in fact overlap in their bit-positions (without ill effects on the accuracy of a search). Each bit-position is calculated with a hash-function. The current version of SPIN uses three different hash-functions by default, setting between one and three additional bit-positions for each new state explored.

Let us assume that each new state that is explored in this mode consumes 0.5 bytes of memory, and that the speed of the model checker is approximately 10^8 states per hour (about 3K states/sec). Under these assumptions, the model checker can use no more than 10^8 * 0.5 bytes of memory per hour of run time, or roughly 50 MB. It is easy to see that moving up to 8 GB then increases the maximum runtime to about a week of computation. In return, we would cover significantly more states, but both time and space are limited resources, so the increased coverage of a problem space is not always achievable in practice. To make the point perhaps more strongly, if we increase the available memory size to 64 GB, a maximal bitstate search could consume close to two months of computation, which is no longer a feasible strategy, no matter how many states are explored in the process or how much problem coverage would be improved.

We are thus faced with a dilemma. The applications that we are trying to verify with model checkers are increasing in size, especially when we start applying model checkers to implementation level code, cf. [H00], [HS00], [BR01], [VH03], [HJ04]. As state descriptors grow in size from tens of bytes to tens of kilobytes, processing speeds will also drop. As we observed in the introduction, these performance differences are no longer offset by continued CPU clock-speed increases, so they will contribute to even longer verification times. For very large applications, a bitstate search is typically the only feasible verification option available to us, as it can increase the problem coverage (i.e., the number of reachable states explored) by several orders of magnitude when compared to a standard exhaustive search attempt. Exhaustive coverage for these applications is impractical, given the enormous size of both the state descriptors and the numbers of reachable states. In these cases we have to find ways to perform the best achievable approximation of an exhaustive search, balancing both memory use *and* runtime constraints. Technically, the right solution in these cases is to apply stronger reduction and abstraction techniques to reduce the problem size as much as possible. We are assuming here that the best possible abstractions and restrictions have already been applied and that the remaining problem size still significantly exceeds available resource limits.

2.2 Multi-Core Verification

One strategy to combat the performance issue we have sketched above is to tap directly into the availability of increasing number of processing cores, communicating via shared memory. Direct (or collaborative) multi-core model checking algorithms are indeed available. For the SPIN model checker we have described such an algorithm elsewhere, [HB07]. In the best cases, the multi-core algorithms can provide near linear scaling with the number of available processing cores. Returning to our earlier example, using eight cores in parallel, each exploring 10^8 states per hour in bitstate mode, can reduce a runtime of 6.8 days on an 8 GB system to 20.4 hours, and a runtime of two months on a 64 GB system to a week. Serious limits remain though. Even if we assume optimistically that we can achieve near linear scaling on large numbers of cores, it would take about 164 cores to bring the last runtime number down to a more reasonable runtime limit of approximately one hour.

This, then, puts an interesting spin on the problem. The doubling interval for memory sizes is currently considerably shorter than the doubling interval for the number of CPU cores. This means that the performance gap we sketched will continue to grow. By the time that 164 processing cores will be available on a single system, the memory size on that system will have grown as well, and it will be much larger than the 64 GB that we assumed in our last calculation. The maximum runtime will therefore also have increased significantly. For the amount of memory that is available at any point in time, the number of CPU cores that one would *need* to reduce the runtime sufficiently is likely to exceed the maximum number of cores that is *available* at the same point in time by a significant margin.

We would like to make use of as many resources as are actually available to us at any point in time to get the best possible coverage of large verification problems, but only *within a predefined time limit*. It is undesirable to start a verification run on a large machine and wonder after a few days have passed if the run is about to complete, or might continue for another month, with no indication of the actual coverage of the search problem that has been realized at any given point in time.

2.3 Parallelism and Search Diversity

For the remainder of this paper we will assume that there is an upper bound on the time that is available for any verification run, especially for large problem sizes. To make the challenge specific, we will assume an upper-bound of *one hour* of computation. With a fixed exploration rate this means that we cannot use more than a few Gigabytes of memory in an exhaustive verification and no more than about 50 to 500 MB in a bitstate exploration.

```
int val;

inline check() {
            if
            :: (val == -1196372740)
            || (val == -222966779)
            ...
                -> c_code { printf("assertion violated %d\n", now.val); }
            :: else /* no match */
            fi
}

active [8] proctype word()
{           /* _pid = 0..7 -- each proc owns 4 bits */

end:        do
            :: d_step { val = val | 1 << ((4 * _pid) + 0); check() }
            :: d_step { val = val | 1 << ((4 * _pid) + 1); check() }
            :: d_step { val = val | 1 << ((4 * _pid) + 2); check() }
            :: d_step { val = val | 1 << ((4 * _pid) + 3); check() }
            od
}
```

Fig. 1. Spin model for generating all 32-bit numbers. Test model to illustrate the effect of search diversification strategies.

For very large verification problems we have to accept that a time-bounded search for errors will generally remain incomplete. It is therefore important that we do not expend all our resources on a single search strategy. Within the time available, we should approach the search problem from a number of different angles – each with a different chance of revealing errors.

Our strategy, therefore, is to leverage both *parallelism* and *search diversity*. To study the effectiveness of candidate strategies to solve this problem, we will begin by using a relatively simple model that can generate a large state space. The model is defined in such a way that we can easily identify every reachable state and measure the individual and cumulative effectiveness of a range of different search strategies. The example, written in the specification language of SPIN, is shown in Figure 1.

The test model shown in Figure 1 defines the behavior of eight asynchronously executing processes, each of which executes a loop with four possible execution steps that can be selected non-deterministically. Each process has a predefined id number, named *_pid*, with a value between zero and seven in this case. At each execution step, an arbitrary process is selected by the model checker, and that process will select one of its four possible execution steps at random. We have defined the model in such a way that each process " owns" four bits from the 32-bit global integer variable *val*. A process can either set one of these bits, or leave them zero, but each time it sets a bit it performs a check to see if a particular 32-bit target value was reached. The check is defined as an inline function *check()* that checks for a match with a predefined set of one hundred 32-bit target values, generated with the help of a random number generator when we defined the model. If one of the target numbers is matched, a line is printed. We can assess the quality of a search attempt by counting how many of the one hundred numbers are matched in a run. The target numbers are used here to represent generic search targets, or in terms of logic model checking " property violations" that we would like to be able to locate in large search spaces.

Clearly, there will be 2^{32} (over 4 billion) possible assignments to the 32-bit interger *val*. Each state descriptor for the model as a whole is relatively small at 76 bytes. Storing all reachable states exhaustively, however, would require more than 300 GB of memory.

If, therefore, we perform a traditional search on a machine with no more than 3 GB, an exhaustive search cannot reach more than 1% of the state space, and statistically we may expect just one match within the set of target numbers. A bitstate search could in principle store all states in this amount of memory. For this example, the model checker explores approximately 6.10^4 states per second on a 2.3 GHz system, which means that exploring the full statespace sequentially in bitstate mode would take about 20 hours, assuming a sufficient amount of memory is available to record all states.

We are interested, though, in the case where we are forced to use less memory than what would suffice for a full search – even in bitstate mode. We will, therefore, limit the amount of memory that we make available to the search to just 32 MB (or 0.01% of the 300 GB required for an exhaustive search) and study what can be achieved in terms of state coverage by exploiting parallelism and search diversification techniques. Note that 32 MB corresponds to 8*32*1024*1024 = 268,435,456 bits (or 6.25% of the 4 Billion states that are generated by our test model).
A default run of the model checker using a bitstate memory arena of 32 MB can be done as follows, using a standard Linux command shell:

```
$ spin –a model.pml
$ cc –DBITSTATE –o pan pan.c
$ ./pan –w28 |
  grep "assertion violated" |
  sort –u |
  wc –l
4
```

This search reaches 1.56 10^8 states, or about 3.6% of the

2^{32} reachable states. It locates 4% of the randomly seeded target numbers in that search space. Which numbers are found will depend on the search order that is used in the model checker. Normally, this search order is irrelevant, when all states are covered in the end. In partial or incomplete searches though, the search order used can bias the search in an undesirable way, by systematically always missing the same parts of the search space.

The challenge that we will consider next is to increase the number of matches from 4% to 100%, without changing the memory constraint of 32 MB that we imposed.

3 Search Diversification

Our strategy will be to use a range of different search methods, and to run as many small verification jobs as possible in parallel, using all available CPUs (local CPU cores and/ or networked machines). Because we will use no more than 32 MB per search, on a multi-core system with 8 GB of memory, we could in principle run up to 256 jobs in parallel without exhausting memory.

There are several methods that we can use to diversify the search process in the SPIN model checker. We can change, for instance:

- the hash-polynomials that Spin uses to compute the bitstate locations during a search (with run-time parameter –h),
- the number of hash-functions used, i.e., the number of bitpositions set per state (with run-time parameter –k), or
- the search algorithm that is used to perform the search itself (we will consider this option in more detail later).

As a first experiment, we can check the effect of just varying the hash-polynomials, leaving everything else fixed. (The use of hash-polynomials in bitstate hashing is discussed in detail in [H87] and can also be found in [H04].) Here is the result of that experiment:

```
$ for h in 0 5 11 17
do
       ./pan –w28 –h$h
done |
  grep "assertion violated" |
  sort –u |
  wc –l
9
```

We performed four runs (which can all be done in parallel and therefore take no more time than a single run) and the number of unique matches increased from 4 to 9, more than doubling our coverage.

We can expand the search further by also varying the number of hash functions (-k). The default number of hash-functions in SPIN verifications is three, but we can use other numbers as well. We can execute this set of runs with a nested for-loop in the Bourne (or *bash*) shell:

```
$ for k in 1 2 3 4
  do
       for h in 0 5 11 17
       do
          ./pan –w$w –k$k –h$h
       done
  done |
    grep "assertion violated" |
    sort –u |
    wc –l
24
```

The number of unique matches increased from 9 to 24, by performing 16 small independent searches that can still all be executed in parallel. None of the individual searches uses more than 32 MB of memory.

3.1 Adding Randomization

We mentioned the possibility to increase diversification further by varying the search algorithm that we use in the state space exploration itself. One of the methods we can use is a randomization of the search order. In a SPIN-based model-checker we can introduce randomization at two specific points in the search where non-determinism is resolved, i.e. in

- process scheduling decisions, and
- transition selections within processes.

The use of randomization has the advantage that it can support a large variety of behaviors, merely by selecting different seeds for the random number generator. Each separate search can be expected to have approximately the same runtime performance, being constrained in the same manner by our self-imposed time and/ or memory limits. But each search variant can also be expected to explore a different subset of states, and locate different types of defects (or search targets). Cumulatively, all search variants combined, executed in parallel, can thus outperform any one variant used separately.

To perform a proof of concept of this strategy, we perform a separate experiment using just the search randomization technique, with a fixed hash-function and using only single-bit hashing (–h0 –k1). We start by generating a file with a hundred random numbers to be used as seeds. For each small run we take a different number from this file and use it to seed the random number generator.

A shell-script for performing one hundred runs using SPIN version 5.2 or later, can be written as follows.

```
$ spin –a model.pml
$ cc -DBITSTATE -DT_RAND -DP_RAND \
       -o pan pan.c
$ seq=1
$ while [ $seq -lt 100 ]
  do
       r=`sed -n ${seq}p seeds`
       seq=`expr $seq + 1`
       ./pan –w28 –RS$r -k1 -h0 \
        > out_${r}
  done
```

```
$ grep "assertion violated" out_* |
  sort -u | wc -l
100
```

In this example all verification jobs are executed sequentially on a single CPU. When using a multi-core machine we can execute as many of these jobs in parallel as there are available cores. In a larger network, all jobs can be executed in parallel for a fast turn-around. At the end of the runs, *all* targets in the test model were reached. This is especially interesting because we used only search randomization to define these runs, and no other type of diversification. None of the runs used more than 32 MB or 0.01% of the 300 GB of memory that is required to complete a standard exhaustive verification of the problem.

Clearly, the number of runs that are needed to reach a specific level of coverage of a large search space must increase when the amount of available memory for each search is reduced, and vice versa when memory is increased, the number of runs may be reduced. For a memory arena of 32 MB, 81 runs suffice to reach full coverage of the example problem. If we repeat the experiment with a memory arena that is twice as large (using 64 MB of memory, corresponding to 0.02% of 300 GB), then we can reach full coverage of the sample problem in 32 runs. If we reduce the memory arena to 16 MB (0.005% of 300 GB) it takes 236 randomized runs to reach full coverage for our sample problem, as summarized in Table 1.

Table 1 -- Coverage of Randomized Runs

Hash-Array Size (MB)	% of 300 GB	Nr of runs to reach 100% coverage
64	0.020	32
32	0.010	81
16	0.005	236

These simple experiments suggest that we can significantly increase our coverage of large search spaces by using search diversification and randomization techniques, and by running many small jobs in parallel. The incremental effect of additional runs decreases, as more and more runs are added. The maximum number of runs that can be performed within a given time limit will also be bounded by the number of available CPUs. Within the available resource bounds, though, we can develop search strategies that can give us the best possible results.

4 Application

The test model from Figure 1 illustrates the potential of swarm verification techniques. No single example can of course be representative of large search problems in general. In this case, the test model defined a search problem with a relatively shallow search depth of 32 steps. This could limit the effectiveness of hash-diversification and benefit search randomization techniques. We have therefore also performed an extensive series of measurements with more typical verification models, of various sizes and search depths.

4.1 Methodology

Our goal in performing a range more detailed measurements is to study the behavior of swarm verification when compared with standard model checking. The primary metric in the comparisons we perform is *coverage*: the fraction of the reachable statespace that is reached with each method. A second metric is *resource use*: the time and memory needed to complete each type of run.

To be able to make accurate comparisons we use models with precisely known structure and size. The models selected for these measurements are non-trivial, yet small enough to be exhaustively verifiable.

We can reproduce the effect of a resource constrained system by varying the amount of memory that we make available for each run. In each test we compare the coverage that is realized by a swarm run with the one realized by single verification runs (the reference). By disabling search options that can cause unpredictable differences in the statespace sizes explored (e.g., partial order reduction) we can measure accurately what fraction of a state space is visited and what fraction is missed in each test. To show that partial order reduction in itself is not an impediment to the coverage improvement of swarm verifications, we also performed teste with it enabled (Fig 4.).

We use five different test models. The first four models can be exhaustively explored with standard techniques, and serve as our main target for comparisons. The fifth model is added as an example of a very large application that cannot be verified fully with traditional means. In this last case we can still compare the number of states that are reached with each method, but we cannot determine what fraction of the statespace these numbers correspond to. In this one case, we only know that a large increase is in the number of states reached, within the *same* resource constraints, the greater the search improvement that is realized will be (cf. Fig. 4).

4.2 Models

We study the following three medium and two large size verification models:

- a data transfer protocol model, *dtp*,
- a file transfer verification model, *pftp*,
- a model of the Cambridge ring network protocol, *cambridge*,
- a large model of an operating system kernel developed at Honeywell, called *DEOS*, and
- a very large a model of an experimental network architecture design, called *fleet*.

The first three models are taken from the standard SPIN distribution and and are frequently used in performance measurements. The *DEOS* model was also discussed in [P05] and used in [HB07]. The *fleet* model was provided to us by Sanjit Sehia from UC Berkeley.

Table 2 summarizes some key characteristics of each model. The search depth is the maximum number of steps that the model checker takes in the depth-first search before reaching a previously visited state. The size of each state is given in bytes.

Table 2 – Applications

Verification Model	Reachable States	Search Depth	State Size
dtp	394,182	334	172
pftp	439,895	5,780	144
cambridge	532,532	12,428	56
DEOS	22,452,390	177,911	584
fleet	Unk. (>10^{11})	705	1440

In our tests of the first three (small) models, we performed the verification *without* using partial order reduction [H04]. This allows us to eliminate a potential source of confusion in the generally unpredictable effect of a partial order reduction on statespace sizes. For each run performed, we must know precisely how many states should be reached, to allow us to accurately measure the fraction that was effectively reached. For the pftp verification model we also disabled several other features in the verifier, such as statement merging, dataflow optimization and dead-variable elimination, by generating the model checker as follows:

```
$ spin –a –o1 –o2 –o3 –o4 pftp.pml
```

The default optimizations normally decrease the number of reachable states. Because our objective in these experiments is to measure the fraction of all system states that is reached by different search strategies, a somewhat larger state count helps to make the measurements more meaningful. (An alternative could have been to use slightly larger models, that can still be exhaustively verified.)

We can calculate how much memory would be required to perform an exhaustive verification for the first four models by multiplying the number of reachable states with the size of each state. The first two applications, then, require approximately 64 MB of memory for exhaustive exploration, the third application 30 MB, and the *DEOS* application 13 GB. The total state space size for the fleet model is at least 10^{11} states, but otherwise unknown. This means that exhaustive exploration with traditional search algorithms would require at least 130 TB. This last search problem, then, is representative of the type of very large applications for which we would like to develop improved verification strategies.

To check how many *unique* states are reached in searches that necessarily remain incomplete due to resource constraints (imposed by the time and/or memory bounds we use), we perform the verifications in a mode where the model checker records *every* state visited in a binary file. For a swarm run, we combine the data from all individual runs, to count the cumulative number of unique states covered in all runs combined. We then compare these totals with the number of reachable states given in Table 2, which were obtained with standard exhaustive runs (as a reference), where possible.

We run a large number of small verification jobs for each application, each using a randomly different search strategy. Any number larger than *one* can illustrate the effect. There is no preset maximum to the number of swarm runs, but clearly, at some point there will be no added benefit from using still larger numbers of swarm runs. For our test models, 100 runs are sufficient to illustrate the gains convincingly: a notable increase in problem coverage. In practice, the optimal number of runs to be performed will depend on the available resources: the amount of memory and the length of time available, and the number of cores or CPUs that is available.

For search diversification we varied the number of hash-functions randomly between 1 and 3, and used a different type of hash-function for each of the one hundred runs. SPIN version 5.2 has one hundred different hash-functions predefined that are selectable with the runtime –h parameter, which sufficed for these experiments. For randomization we used a different seed for the random number generator for each run performed, using a file with one hundred random numbers. For all measurements on the models from Table 2 we further randomly chose one of the following four pre-compiled executables for each run:

1. the standard depth-first search, with randomization of both process scheduling decisions and transition selection decisions,
2. the same search as in 1, but with default process scheduling choices reversed (using pan.c compiler directive –DREVERSE),
3. the same search as in 1, but with the default transition selection choices reversed (using pan.c compiler directive –DT_REVERSE)
4. the same search as in 1, but with both process scheduling choices and transition selection choices reversed by default.

Each set of one hundred runs for each application was repeated several times, using different memory constraints, performing thousands of separate runs. Table 3 first shows the fraction as a percentage of all system states that were reached for each of the first three models, for three sample sets of 100 runs.

Figure 2 gives more detailed coverage numbers for the *pftp* application, over a broader range of memory sizes. In this case, we varied the memory size from 8KB (-w16) through 256 KB (-w21) for the swarm runs (top, solid curve), and from 8 KB through 4MB (-w25) for a series of single bitstate runs (bottom, dashed curve). The detailed results of all three sets of measurements can be found in Tables 5 and 6 in Appendix A.

Table 3 – Coverage Using Diversification

Model	64KB	128KB	256KB
dtp	99.37%	99.97%	100%
pftp	99.32%	100%	100%
cambridge	92.97%	99.50%	100%

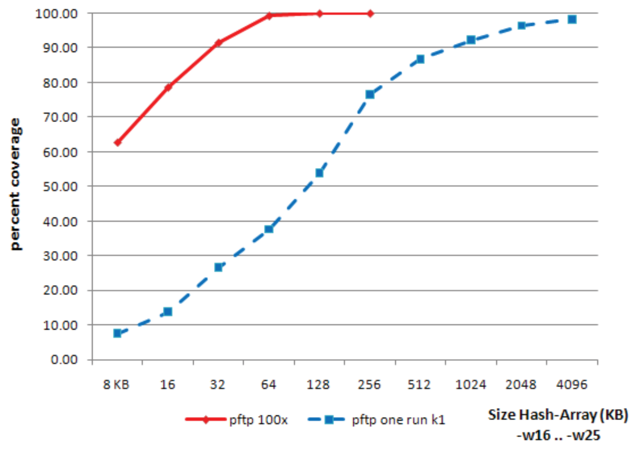

Fig. 2 Relative coverage realized for the *pftp* application by a set of one hundred swarm runs (solid line, top), compared with single bit-state runs (dashed line, bottom) for a range of different memory constraints (see also Appendix A).

Remarkably, when using just 8 KB (**1.2%** of what is required for an exhaustive verification run), the swarm run for the *pftp* model already reaches **63%** of all system states, while a single bitstate run in the same amount of memory covers no more than **7.6%.** Viewed differently, by reading the chart horizontally instead of vertically, to realize the same coverage as the 8 KB swarm run, we would have to use 32 times as much memory with a single bit-state run (increasing the memory arena that is used from –w16 to –w21).

When memory is restricted, as it will be in our target domain of application, this means that we can increase coverage by about an order of magnitude by performing a set of swarm runs. The results for the *cambridge* and *dtp* models are similar, and not separately shown.

For the *DEOS* model we performed six sets of 100 runs, varying the available memory size from 512 KB to 16 MB. We compared the results with the performance of eleven single bitstate runs, ranging from a memory arena of 512 KB through 512 MB. Three sets of bitstate runs were done, using three differrent settings of the number of hash-functions (using -k1, -k2, and -k5). Finally, the swarm runs were also peformed twice, once with all the randomizations described before, and once without the –DREVERSE and –DT_REVERSE options (variants 2, 3, and 4 from our list). The results are shown in Figure 3. All measurement detail for Figure 3 is also included in Tables 6 and 7 in Appendix A.

For this large application we used all available optimizations, including partial order reduction, to reduce the large statespace size as much as possible. To be able to compare the cumulative effectiveness of the randomized runs with that of a single bitstate runs in the same memory arena, we again recorded all states reached into files and counted unique states across all swarm runs with a post-processing step.

The time taken by each bitstate run depends on the size of the memory arena, and thereby the maximum number of states that is explored per run. Because all randomized runs in a swarm set are independent, they can all be performed in parallel, and cumulatively need to take no longer than a single run. If we compare the coverage of the swarm runs with the bitstate runs, therefore, it is important to note that each individual bitstate run takes more and more time as we relax the memory constraints, while moving to the right in Figures 2 and 3, making these search alternatives less and less attractive.

In this study we are primarily interested in cases where for a given problem size externally imposed constraints on memory and runtime prevent us from completing an exhaustive verification. This means that we are most interested in the data shown on the left-hand side of Figures 2 and 3.

If we consider the left-most point in Figure 3, we see that the best single bitstate run realized a problem coverage of **4.3%** (reaching 960,743 out of 22,452,390 states). The swarm runs, completing one hundred randomly different runs, each using no more memory or time than the single run, realizes coverage near **41%** (reaching 9,126,333 states). Neither run realizes exhaustive coverage, but the swarm method improved our coverage of the search problem by nearly an order of magnitude. Adding more runs can improve coverage further, but we can expect that it will take an exponentially increasing number of runs to continue to expand coverage in a meaningful way.

As intended, this approach benefits from massive parallelism, as we expect it to increasingly become available for routine use. At –w26 the swarm runs reach **100%** coverage of the search space, using just 8 MB of memory per run, or **0.075%** of the 13GB that would be required to complete an exhaustive verification. The best bitstate run realizes only 50% coverage in the same memory arena (see the top dotted line in Figure 3). The cumulative number of states in the swarm runs can be seen to slightly exceed the number that is reached in an exhaustive run. This effect is caused by the use of the partial order reduction, which could cause a slightly different number of states to be reached, depending on where truncations in the bitstate runs occur.

One more set of measurements was performed for the *fleet* model, as an example of a model that cannot be exhaustively verified within reasonable resource constraints. One version of this model has a known assertion violation that can be triggered through a manually guided simulation in about 350 execution steps.

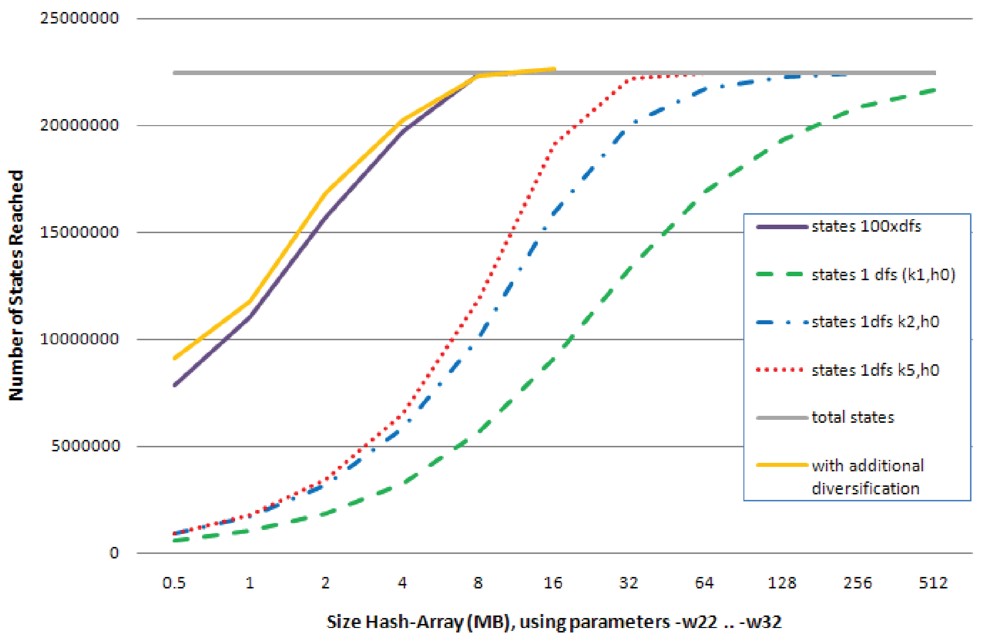

Fig. 3 Relative coverage realized for the *DEOS* model by two sets of one hundred randomized swarm runs (solid lines, top) compared with three standard bitstate hashing runs using one (dashed, bottom), two (dash-dotted, middle), and five (dotted, top) bits set per state, in memory arenas that range from 512 KB (-w22) to 512 MB (-w32). The gray horizontal line indicates the total number of reachable states (22,452,390) for this model. The top solid line includes full diversification of swarm runs. The one slighty below it does not include the reversed scheduling and transition selection options. See also Appendix A, Tables 6 and 7.

The model is over one thousand lines of text and each system state takes 1,440 bytes to store in exhaustive verification mode. An attempt to perform a full verification on a 2.3 GHz 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 has explored 23.4 million reachable system states, which corresponds to an unknowable fraction of the total reachable state space. A search using -DCOLLAPSE compression (a predefined lossless state compression mode in SPIN) reaches 327.6 million states before running out of memory after 3,320 seconds of runtime, also without revealing the error. A run with the hash-compact algorithm (a stronger, but not lossless, form of compression) runs out of memory after 1,910 seconds and increases the coverage to 537 million states. The most aggressive bitstate run we can perform under the given constraints, using all 32 GB of memory, runs for 34 days, and explores well over 10^{11} system states. None of these search attempts succeed in locating the assertion failure.

The 34-day bitstate run finishes with a low hash-factor (meaning that most bits in the hash-array were set), still providing little guidance on the fraction of the reachable state space was explored. Likely, the full reachable state space for this problem is much larger than what can be searched or stored by any verification method. The bitstate run can be performed in parallel on eight CPUs, shrinking the run time from 34 days to about 5 days, but without change in coverage.

To explore the coverage that can be realized with swarm verification runs for this problem, we performed a series of measurements similar to the ones described for the *DEOS* model, using the full set of search randomization and diversification options we have described. We performed seven sets of one hundred swarm runs, varying the available memory size from 512 KB to 32 MB. The swarm runs can be extended to also larger memory sizes, but the number of reached states becomes too large to store, sort, and count with the method we are using, so no further datapoints were obtained. In these swarm runs, the assertion violation is readily found. We also performed eleven single bitstate runs for comparison, repeated twice, using respectively one bit per state and five bits per state. The results of the measurements are shown in Figure 4, and should be compared with the curves shown for the more complete dataset for the *DEOS* model from Figure 3. The measurement detail is included in Appendix A, Table 8.

The part of the curve that we are exploring for the *fleat* application is clearly in the range that we have defined to be our primary target for large search problems. To obtain the right-most datapoint on the curve shown in Figure 4 required a bitstate run that took roughly 10 hours and 22 minutes on our 2.3 GHz machine, using a memory arena of 512 MB. The largest swarm run we performed in a memory arena of just 32 MB can be completed 15.3 times faster, in 41 minutes, while covering 2.6 times more states (close to 8 billion compared to 3 billion).

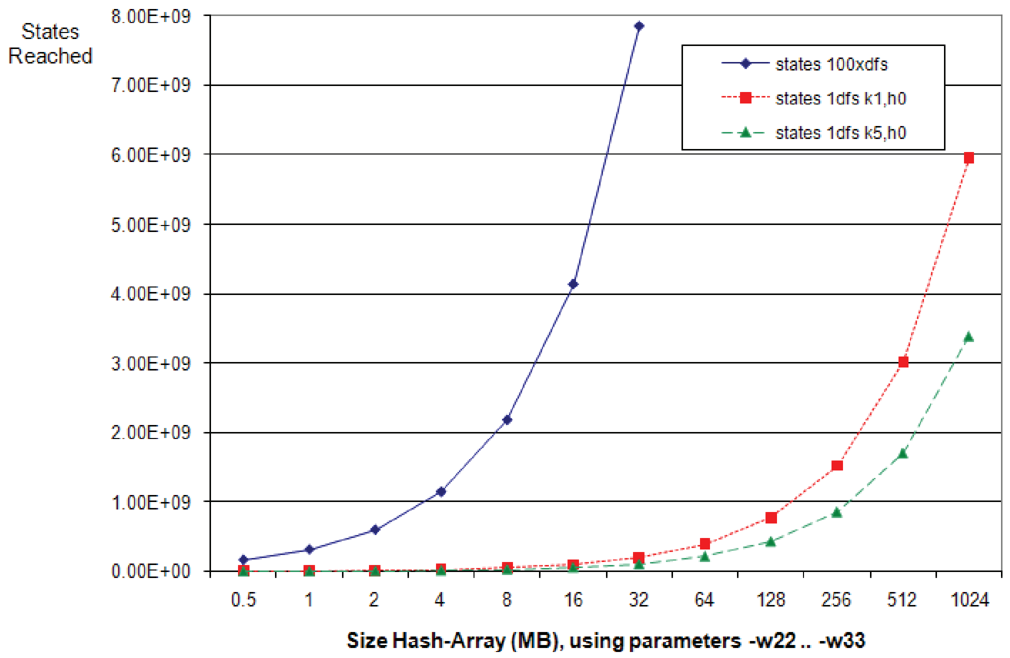

Fig. 4 Relative coverage realized for the *fleet* application with one hundred randomized swarm runs (solid line, top) compared with standard bitstate hashing runs using one (dotted, middle), and five (dashed, bottom) bits per state, in memory arenas that range from 512KB (-w22) to 512MB (-w32). The total number of reachable states for this model is unknown, but was estimated it to be >10^{11} reachable states. This is the problem size *after* the application of partial order reduction (which can itself reduce the overall state space size by an exponential amount). See also Appendix A, Table 8.

Seen another way, the 3 billion states that require over 10 hours to compute with a traditional sequential bitstate run can be reached in about 16 minutes with a parallel and randomized swarm run using 100 cores, or about 40 times faster. The parallel runs can be performed on networked computers, in a cloud or grid arrangement, with each individual run using the maximum amount of memory available to maximize the coverage that could be obtained with this technique.

4.3 Scaling Behavior

In applications of distributed algorithms we are especially interested in studying how a particular algorithm or methodology will scale with the use of increasing numbers of processes. The ideal scaling behavior, then, would be to achieve linear or near-linear scaling. With the swarm technique we have described here, we would like to see how problem coverage (measured as the fraction of the cumulative number of all reachable states covered in a swarm run) changes with the use of increasing numbers of CPUs or CPU-cores. For these measurements we chose the *DEOS* model. It is sufficiently large to make meaningful measurements, but not so large that we can no longer determine what the full state space size is. We measured how coverage changes with increasing numbers of CPUs for four different memory sizes: 1, 2, 4, and 8 Mbyte. These sizes correspond to, respectively, 0.01%, 0.02%, 0.03%, and 0.06% of the 13 GByte of memory that would be required to complete a traditional exhaustive search. The results are shown in Figure 5.

The top curve, corresponding to the use of 8 MByte of storage with runtime parameter –w26, reaches to 100% coverage. The other three curves correspond to an increasing disparity between problem size and amount of memory used. We can expect that these curves too will reach 100% coverage if extended towards the right, but they require larger numbers of swarm runs to do so.

The effect of reducing the amount of memory used, then, is two-fold:

- The runs complete *faster* (since each run will explore a smaller number of states). For instance, the runs for –w25 take half as long as those for –w26, etc.
- *More* runs are required to reach full coverage: the *slope* of the cumulative coverage curve decreases. All these runs can be performed in parallel, so all can in principle complete in the same time it takes to perform one single run, provided a sufficient number of CPUs is available.

Towards the left side of the range shown in Figure 5, the relative effect of adding additional swarm runs is quite significant, beating linear increases. Towards the right, as we approach full coverage, the effect of additional runs diminishes, as can be expected.

For very large applications, which are the focus of this paper, we are most likely to operate in a range where exhaustive coverage is out of reach, i.e., the bottom curve in Figure 5, where increases in the number of swarm runs performed is most effective.

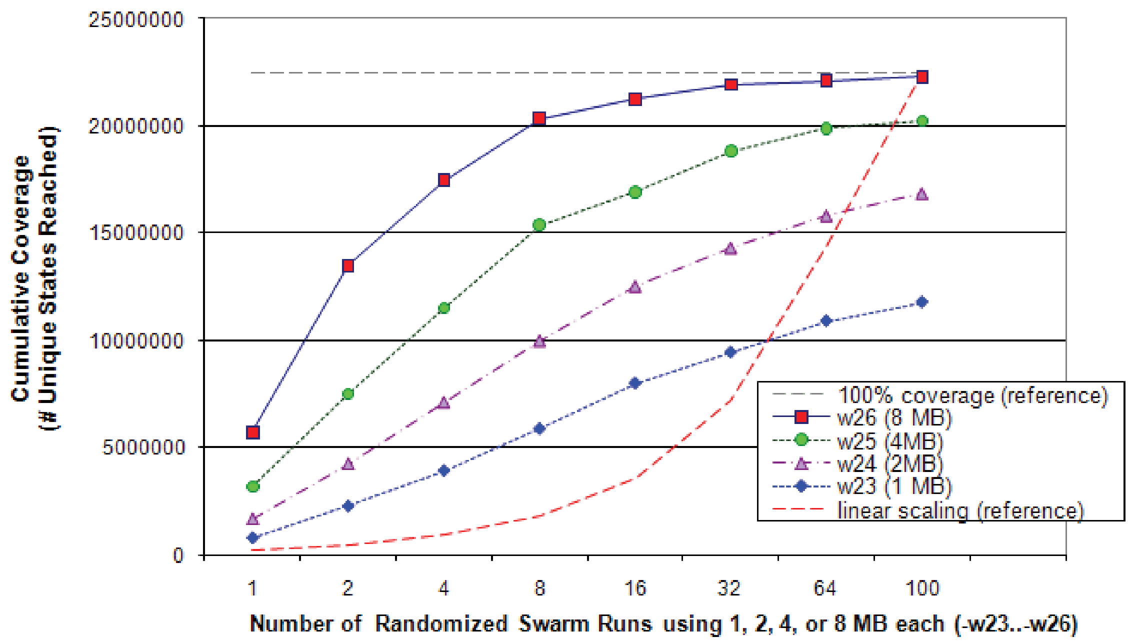

Fig. 5 Cumulative coverage for *DEOS*, realized by 1, 2, 4, .., 100 swarm runs, using between one (w23) and eight MByte (w26) of storage, corresponding to between 0.01% and 0.06% of the memory required for exhaustive search. The horizontal dashed line indicates 100% coverage. The bottom curve corresponds to what would be linear scaling with the number of CPUs (i.e., swarm runs).

5 The Swarm Tool

Even though all search variants that we have described here are supported as standard search options in SPIN version 5.2 and later, it may not always be easy to remember the entire set, and there is some work involved in setting up the execution of large numbers of small verification jobs for a swarm run. We have therefore developed a swarm configuration tool that can automate these tasks. The tool has a range of predefined seach options builtin, exploiting both randomization and diversification, and can be updated as new search variants are identified.

The Swarm tool allows the user to take full advantage of search randomization and diversification options when large numbers of CPUs or CPU-cores are available to tackle a large search problem. The user is asked to specify just three key constraints to a verification task: the amount of memory that is available per run, the number of CPUs or CPU-cores that can be used, and the maximum amount of runtime that may be used to complete the search. Using these parameters, the tool configures a swarm run that can provide significantly greater coverage of the given search problem within the stated constraints when compared to single bitstate run. The tool allows the user to define also additional parameters, such as the state size and the average state exploration speed, but these parameters are mostly used for fine-tuning the performance of a swarm run when needed.

The tool is built as a verification script generator, and is written in about 800 lines of C. The swarm tool generates a shell script that performs as many different types of verification runs as possible without exceeding user-defined constraints on time and memory use.[1] The essence of the configuration and script generation algorithm used in the Swarm tool is shown in Figures 6 and 7.

Given that the time and memory constraints are tightly connected, the tool only needs to take the minimum of these two constraints into consideration. For a given time limit, for instance, Swarm can derive and estimate for the maximum amount of memory that can be used. Swarm first calculates how many states could maximally be searched within the time and memory constraint that is specified. It then sets up a series of bitstate runs within that limit, using the variations we have discussed. Swarm further adds variations of the maximum search depth, to increase the diversification somewhat more.

The commands that are generated include standard, randomized, and reverse depth-first search orders, using varying numbers and types of hash-functions per run. In a small amount of time, hundreds of different searches can thus be performed, each slightly different, probing different parts of an oversized search space.

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

```
$ swarm -c4 -m1G -t1 -f model.pml
```

For this run we specifed the availability of 4 CPU cores, and up to 1 GB of memory per run. The –t parameter was used to set the time limit for all runs combined to one hour (which is also the default). The swarm tool writes the verification script into a file with the same basename as the verification model, but replacing the extension .pml with extension .swarm, e.g., for the example above the

[1] The tool is available from: http:/ / spinroot.com/ swarm/ .

```
for (width = fct(mem); width > 0; width--)
{ for (steps = 4; steps >= 1; steps--)
  { if (TimeUse(width, steps) < Tmax)
    { depth = (max_d - min_d)/steps;
      depth = min(1, depth);
      return width and depth # success
} } }
configuration fails
```

Fig. 6 Sample configuration for Swarm verification script generation. Tmax is the total time available on all CPUs. The configuration tries to find the best values for the size of the hash-array and the smallest incremental step for varying depth-limits. If no runs can be scheduled, the configuration attempt fails. Each pass through the outer-loop reduces the size of the hash-arena, creating shorter verification runs, using all available time.

result is written into a file named model.swarm. The verification can now be performed by simply executing the script.

As a simple example, a swarm run for the *fleet* model can be setup for an eight-core system and a one-hour time limit, as follows:

```
$ export CCOMMON="-DVECTORSZ=1500"
$ swarm -b1440 -s35000 -c8 -t1 \
      -f fleet.pml
Swarm: 96 runs, avg per cpu 3599.6 sec
Swarm: script written to fleet.swarm
$ ./fleet.swarm
```

In this example, we first used an environment variable to define compilation directives we would like to use for all verification jobs. The invocation of the swarm command then defines the state-size to be 1440 bytes, and gives an estimated processing speed (measured in earlier verification attempts) of 35,000 states per second. In this case, swarm generated a script with 96 randomized runs, with an estimated completion time of 3,599 seconds – within one hour as requested.

Executing the script finds the assertion violation we have described for the *fleet* model within a few seconds. In this case this is by virtue of one of the search variations that is a standard part of Swarm's mix: a reverse depth-first search. The assertion violation, as it turns out, is normally encountered only towards the very end of the standard depth-first order used by SPIN, but resource limits normally prevent us from reaching that point in the search. The error is trivially found near the start of the search if the depth-first search order is reversed, and as we saw earlier the chances of finding are significantly increased if we use randomized search orders as well.

The Swarm tool can read configuration parameters from the command line and from environment variables, as we have shown in the examples above, but it can also read them from a configuration script that is stored as an plain text file. A default configuration file can be generated by the tool itself with a runtime option: " swarm –I". This default configuration file is shown in Appendix B.

```
do {
  for (d = Dmax; d >= Dmin; d -= depth)
  { for (m = 0; m < modes; m++)
    {  AddRuns(width, m, d);
  } }
} while (--width > 0);
```

Fig. 7 Sample script generation. Each call to AddRuns adds a run into the script to be executed by each CPU, if possible within the time limit. For each addition, a different compilation mode (m) and search depth-limit (d) is used. Random choices are made for the type and number of hash-functions, and the seed for the random number generator.

The user can edit the configuration file to adjust the parameter settings for a specific application. The default file defines eight different ways to compile the model checking engine itself, using forward, reversed, randomized process and transition orderings. As new search modes are defined, they can be added to the set, and when specific variants are not desired in a particular application they can be removed. The methodology is therefore not restricted to leveraging diversified verification runs, as we have discussed so far, but could also be applied more generally.

The line in the configuration script that defines the number of available cpus, e.g.:

```
cpus        4
```

can also specify the use of remote computers, provided that they are setup to allow password-less ssh connections. For instance, if we want to define a run using 4 CPUs on the local machine, 8 on a remote machine called nada, and 6 more on a remote machine called niks, for a total of 4+8+6 = 18 CPUs, we would specify this as:

```
cpus        4 nada:8 niks:6
```

and swarm will take care of the rest.

This setup gives us a flexible and general methodology for tackling large search problems that we expect will be increasingly common.

6 Related Work

Randomization is a well-known method for the partial exploration of large search spaces. One of the first descriptions of a random walk technique for protocol validation is, for instance, [W89]. This method was applied with an extension of the Murphi model checker in [SG03] and used in combination with a breadth-first search discipline. Stateless search methods such as random testing and random simulation methods have an even longer history, cf., [BM83]. Heuristic and random pruning of statespaces in model checking tools also has a very rich and long history, from the scatter searching method in *Trace* [H85] to the random search methods used in *Lurch*

[OM03]. Parallelization methods and diversification strategies have also been applied in closely related fields, such as SAT solving [HJ08], [OU09], and SMT solving [WH09]. A detailed overview of attempts to develop distributed algorithms for solving the model checking problem directly (an approach that is orthogonal to the one described in this paper) can be found elsewhere, e.g., in [HB07].

The closest method to our approach is the one that was described in [D07], which focused on the verification of Java code with the Java Pathfinder tool, also using parallelism and randomization, but not search diversification.

Different from this earlier work, we set a firm upper-limit on the amount of time that can be used for a run, and use a tool to find the best configuration of runs that takes advantage of the available resources within given time constraints. The swarm method uses the available information to configure a large set of diversified and randomized parallel runs. Swarm is defined as a relatively simple front-end to SPIN that requires no changes in SPIN itself to leverage the new capabilities. Extensions for newly developed search algorithms, furthermore, are trivial to add, requiring no change in either SWARM or SPIN.

7 CONCLUSION

It is often assumed that the best strategy for tackling large verification problems with SPIN is to use all available memory in a maximal bitstate search. The number of system states reached in such a search can be significantly larger than what is covered in a standard exhaustive verification run, which quickly exhausts memory in these cases. Technically, one could cover still more states with aggressive compression techniques, such as SPIN' s builtin minimized automaton verification algorithm, but the runtime penalty for doing so can be prohibitive in the cases of interest here.

We have noted that as memory sizes continue to grow, at fixed processor clock-speeds, the runtime cost of a maximal bitstate run can also become prohibitively expensive. Once the time required for performing a verification run increases to weeks or months, the capability loses most of its appeal, no matter how many reachable states it can cover. The same is true for all other known approaches to the model checking problem.

We have described a method that allows us to perform verifications for very large problem sizes within user-defined time or memory bounds, while exploiting the multi-CPU and multi-core capabilities in a more fundamental way. The method we have described uses *parallelism* and *search diversity* to optimize verification coverage.

All search algorithms must address the problem of finding the proverbial *needle in a haystack*. The odds of finding the needle trivially depend on both the size of the haystack and the size of the needle. Swarm verifications use the principle that we can increase the odds of finding the needle by using more workers, all searching the large haystack in parallel, provided that they do not all look in the same place. The fundamentals of the search problem do of course remain unaltered. If the haystack is infinitely large, the needle infinitely small, and both available time and the number of workers is finite, basic statistics will tell us that it would be unwise to bet that the needle will always be found. Yet, as we have shown, we can increase the odds of finding it by close to an order of magnitude. Swarm verifications use the one element that has so far been under-utilized in applications of model checking: and that is to use large numbers of *parallel* workers in a *diversified* search strategy.

We have measured the effectiveness of the swarm approach in different ways. We first used a simple model to generate all 32-bit word values, and used diversified search to match randomly placed targets in this very large search space. In these measurements we saw the effectiveness of the search increase from 4% to 100%, using a relatively small number of different search algorithms. In a second experiment we looked for hard-to-find assertion violations in both medium size and very large SPIN models. For the very large problem sizes, standard bitstate searches in the maximal amount of memory available can still fail to locate the errors in weeks of computation. The swarm approach succeeds even when strict time bounds are imposed, e.g., of one hour. We compared the number of states reached in swarm runs with comparable individual bitstate runs and again saw improvements in coverage of over an order of magnitude.

Large verification problems should be expected to become increasingly common in the application of logic model checkers to software verification. The use of diversified parallel approaches becomes more attractive as the number of processing cores and memory sizes (but not clock-speeds) on standard desktop systems continues to increase.

The search method we have described can be extended in many other ways, for instance by adding context-bounded search options as described in [MQ08], [QR05], [HF09]. Such extensions can be made by editing a Swarm configuration file, and require no changes in the Swarm tool itself. We expect many other search variants to be added in years to come to enrich the set of available choices for search diversification.

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 this tool requires more training than a cursory reading of the manual page.

ACKNOWLEDGMENT

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.

REFERENCES

[1] [BM83] D.L. Bird, C.U. Munoz, Automatic generation of random self-checking test cases, *IBM Syst. J.*, 1983, Vol. 22, No. 3, pp. 229-245.

[2] [BR01] T. Ball, S. K. Rajamani, Automatically validating temporal safety properties of interfaces, *Proc. Spin Workshop on Model Checking Software*, LNCS 2057, May 2001, pp. 103-122.

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

[4] [M65] G.E. Moore, Cramming more components onto integrated circuits, *Electronics*, 38, (8), April 9, 1965.

[5] [H85] G.J. Holzmann, Tracing Protocols, *AT&T Technical Journal*, Vol. 64, No. 10, pp. 2413-2433, Dec. 1985.

[6] [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.

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

[8] [H04] G.J. Holzmann, *The Spin Model Checker: primer and reference manual*, Addison-Wesley, 2004.

[9] [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.

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

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

[12] [HF09] G.J. Holzmann, M. Florian, Model checking with bounded context switching. JPL LaRS Tech.Report, Oct. 2009.

[13] [HJ08] A.E.J. Hyvärinen, T. Junttila, and I. Niemelä. Strategies for solving SAT in grids by randomized search. *Intelligent Computer Mathematics*, LNCS Vol. 5144, Springer, 2008, pp. 125–140.

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

[15] [OU09] K. Ohmura, K. Ueda, C-sat: a parallel SAT solver for clusters, *Proc. 12th Int. Conf. on Theory and Applications of Satisfiability Testing*, Swansea, UK, Springer, 2009, pp. 524-537.

[16] [OM03] D. Owen and T. Menzies, Lurch: A Lightweight Alternative to Model Checking. *Proc. 15th Int. Conf. on Software Engineering and Knowledge Engineering*, SEKE 2003, July 2003.

[17] [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.

[18] [W89] C. West, Protocol validation in complex systems, in: *Proc. Symp. On Comm. Architecture and Protocols*, Austin, Texas, USA, 1989, pp. 303-312.

[19] [QR05] S. Qadeer and J. Rehof, Context-bounded model checking. Proc. TACAS 2005, LNCS 3440, pp. 93-107.

[20] [SG03] H. Sivaraj, and G. Gopalakrishnan, Random walk based heuristic algorithms for distributed memory model checking, *Proc. 2nd Int. Workshop on Parallel and Distributed Model Checking (PDMC'03)*, Boulder, Colorado, USA, July 2003.

[21] [VH03] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, Model Checking *Programs Int. Journal on Automated Software Engineering* 10(2), April 2003.

[22] [WH09] C.M. Wintersteiger, Y. Hamadi, and L. de Moura, A concurrent portfolio approach to SMT solving, Proc. Int. Conf. on Computer Aided Verification, Grenoble, France, 2009, pp. 715-720.

Gerard J. Holzmann received a B.Sc. (1973) and M.Sc. (1976) degree in EE, and a Ph.D. degree in Technical Sciences from Delft University in The Netherlands in 1979.

He joined the Computing Sciences Research Center of Bell Laboratories in Murray Hill, NJ, USA in 1980 as a Member of Technical Staff, was promoted to Distinguished Member of Technical Staff in 1995, and to Director of Computing Principles Research in 2001. In 2003 he joined NASA's Jet Propulsion Laboratory in Pasadena, CA. USA as Principal Computing Scientist, and was appointed JPL Fellow in 2007. He also serves as Faculty Associate in the Computing Science Department of the California Institute of Technology in Pasadena, CA, USA.

Dr. Holzmann is a member of the Association for Computing Machinery (ACM) and a member of the US National Academy of Engineering (NAE). He was the recipient of the 2001 ACM Software Systems Award, the 2002 ACM SIGSOFT Outstanding Research Award, and a co-recipient of the 2006 ACM Paris Kanellakis Theory and Practice Award.

Rajeev Joshi received his B.Tech degree in Computer Science from the Indian Institute of Technology in Bombay, India, an M.Sc. degree in 1994, and a PhD in Computer Science in October 1999, both in Computer Science from the University of Texas at Austin. From November 1999 through August 2003 he worked at the Compaq/HP Systems Research Center (formerly DEC-SRC). Since October 2003 he is with the Laboratory for Reliable Software (LaRS) at NASA's Jet Propulsion Laboratory in Pasadena, CA.

Alex Groce received a BSc degree in Computer Science from North Carolina State University in 1999, and a PhD degree in Computer Science from Carnegie Mellon University in Pittburgh, PA in 2005. In April 2005, Dr. Groce joined the Laboratory for Reliable Software at NASA's Jet Propulsion Laboratory in Pasadena, CA. Since June 2009 he is with the Computer Science Department at Oregon State University.

APPENDIX A: MEASUREMENT DETAIL

The following tables detail the numbers of states reached in each measurement reported in Figures 2, 3, and 4 and Table 3.

Table 4 – Swarm Runs (Fig. 2 and Table 3)

100 runs	dtp	pftp	cambridge
-w 16 (8 KB)	224198	275624	233696
-w 17 (16 KB)	308369	345728	320038
-w 18 (32 KB)	364250	402585	413284
-w 19 (64 KB)	391709	436886	495084
-w 20 (128 KB)	394076	439891	529869
-w 21 (256 KB)	394182	439895	532529

Table 5 – Single Bitstate Runs (Fig. 2)

single runs	dtp	pftp	cambridge
-w 16 (8 KB)	23861	33258	37329
-w 17 (16 KB)	31105	60555	74516
-w 18 (32 KB)	110395	117122	134287
-w 19 (64 KB)	204778	165765	229623
-w 20 (128 KB)	302131	237065	343325
-w 21 (256 KB)	350131	336867	428286
-w 22 (512 KB)	372653	381832	478664
-w 23 (1 MB)	382996	405679	505950
-w 24 (2 MB)	388242	424016	518912
-w 25 (4 MB)	391222	431913	522487

Table 6 – DEOS Model, Swarm Runs (Fig. 3)
Randomization Only (l) and With Diversification (r)

100 runs	randomization	w.diversification
-w 22 (512 KB)	7843426	9126333
-w 23 (1 MB)	11009609	11773978
-w 24 (2 MB)	15724931	16833483
-w 25 (4 MB)	19724102	20239659
-w 26 (8 MB)	22345857	22277612
-w 27 (16 MB)	22486315	22654789

Table 7 – DEOS model, Single Bitstate Runs (Fig. 3)

single runs	-k1	-k2	-k5
-w 22	538050	873288	912201
-w 23	1017783	1683773	1743155
-w 24	1807077	3183257	3412892
-w 25	3261134	5817156	6478031
-w 26	5630508	10028176	11827967
-w 27	9118826	15907709	19127215
-w 28	13296249	20001910	22139576
-w 29	16869092	21721577	22437522
-w 30	19318514	22262421	22451655
-w 31	20802816	22407351	22452390
-w 32	21608959	22442930	22452390

Table 8 – Fleet Model (Fig. 4)

HashArray	100 runs	single –k1	single –k5
-w 22 (512 KB)	160092887	3398644	1779123
-w 23 (1 MB)	309139672	6737602	3526140
-w 24 (2 MB)	594329913	13242575	7007478
-w 25 (4 MB)	1141673661	25883720	13905585
-w 26 (8 MB)	2176803189	51183710	27657072
-w 27 (16 MB)	4132283171	100521980	54924346
-w 28 (32 MB)	7851992006	197517950	109225430
-w 29 (64 MB)		393052570	216462870
-w 30 (128 MB)		771986150	431536010
-w 31 (256 MB)		1520757500	857823270
-w 32 (512 MB)		3019252900	1706369200
-w 33 (1 GB)		5953472200	3392343400

For the five applications used for the measurements in section 4 of this paper, the model checking code was generated as follows:

```
$ spin –a dtp.pml
$ spin –a –o1 –o2 –o3 –o4 pftp.pml
$ spin –a cambridge.pml
$ spin –a DEOS.pml
$ spin –a fleet.pml
```

In each case, the model checking code generated was compiled for the single bitstate runs with the following command:

```
$ gcc –O2 –DSAFETY –DBITSTATE –o pan pan.c
```

As discussed in the paper, for the *dtp*, *pftp*, and *cambridge* applications we further includeed the compilation directive –DNOREDUCE to increase the size of the statespace to a more meaningful value.

For the swarm runs, we added the compilation directives –DP_RAND and –DT_RAND to enable randomization, and we used the four possible uses/ non-uses of directives –DREVERSE and –DT_REVERSE, as discussed in the paper, and as also illustrated in the Swarm configuration file shown in Appendix B.

APPENDIX B: SWARM DEFAULT CONFIGURATION FILE

```
## Swarm Version 2.2 -- 15 October 2009
#
# Default Swarm configuration file
#
# there are four main parts to this configuration file:
#       ranges, limits, compilation options, and runtime options
# the default settings for each are shown below -- edit as needed
# comments start with a # symbol
# this version of swarm requires the use of Spin Version 5.2 or later

# See the documentation for the additional use of
# environment variables CCOMMON and RCOMMON
# http://spinroot.com/swarm/

# range
k       1       4       # optional: to restrict min and max nr of hash functions

# limits
d       10000           # optional: to restrict the max search depth
cpus    2               # nr available cpus (exact)
memory  512M            # max memory per run; M=megabytes, G=gigabytes
time    60m             # max time for all runs; h=hours, m=min, s=sec, d=days
hash    1.5             # hash-factor (estimate)
vector  512             # nr of bytes per state (estimate)
speed   250000          # nr states explored per second (estimate)
file    model.pml       # file with the spin model to be verified

# compilation options (each line defines one complete search mode)
-DBITSTATE -DPUTPID                  # standard dfs
-DBITSTATE -DPUTPID -DT_REVERSE      # reversed transition ordering
-DBITSTATE -DPUTPID -DREVERSE        # reversed process ordering
-DBITSTATE -DPUTPID -DREVERSE -DT_REVERSE # both
-DBITSTATE -DPUTPID -DP_RAND -DT_RAND     # randomized versions of the same set
-DBITSTATE -DPUTPID -DP_RAND -DT_RAND -DT_REVERSE
-DBITSTATE -DPUTPID -DP_RAND -DT_RAND -DREVERSE
-DBITSTATE -DPUTPID -DP_RAND -DT_RAND -DREVERSE -DT_REVERSE
-DBITSTATE -DPUTPID -DBCS            # bounded context switching
-DBITSTATE -DPUTPID -DBCS -DREVERSE
-DBITSTATE -DPUTPID -DBCS -DT_REVERSE
-DBITSTATE -DPUTPID -DBCS -DREVERSE -DT_REVERSE

# runtime options (one line)
-c1 -x -n
```