Proving Properties of Concurrent Programs
(Extended Abstract)

Gerard J. Holzmann*

Jet Propulsion Laboratory, California Institute of Technology
gholzmann@acm.org

Abstract. How do you prove the correctness of multi-threaded code? This question has been asked since at least the mid-sixties, and it has inspired researchers ever since. Many approaches have been tried, based on mathematical theories, the use of annotations, or the construction of abstractions. An ideal solution would be a tool that one can point at an arbitrary piece of concurrent code, and that can resolve correctness queries in real-time. We describe one possible method for achieving this capability with a logic model checker.

Keywords: software verification, logic model checking, statistical model checking, model extraction, bitstate hashing, swarm verification, multi-core, cloud computing.

Spin is a logic model checking tool that is designed to help the user find concurrency related defects in software systems.[6] Originally the tool was designed to analyze models of concurrent, or multi-threaded, software systems, but today it is increasingly used to analyze implementation level code directly, without the need to construct a design model first. The benefit of this approach is an increase in convenience, but the penalty can be a notable increase in computational complexity.

Formal methods tools are generally designed to produce reliable results: they should be able to reveal true defects without omissions, and they should not report non-defects, i.e., they should not allow either *false negatives* or *false positives*. Ideally, they should also be fast and easy to use.

This sets three requirements for the design of an effective verification tool: reliability, ease-of-use, and efficiency. There are many software development tools that satisfy all three, and that developers rely on daily. A good example is a language compiler. Most modern compilers harness sound theory, yet they are easy to use, fast, and reliable.

Although formal methods tools are generally *reliable*, they are rarely accused of being fast or easy to use. On the contrary, a frequent complaint about these tools is that they can require extensive training, and consume excessive, and

* This research 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 by NSF Grant CCF-0926190.

E. Bartocci and C.R. Ramakrishnan (Eds.): SPIN 2013, LNCS 7976, pp. 18–23, 2013.

often unpredictable, amounts of time. Although there have been many successes in the application of formal methods, the impact of formal methods tools on general software development practice remains very limited.

Against this background it is interesting to note that in the last five to ten years at least one verification technology did successfully emerge from relative obscurity to evolve into a significant new force in industrial software development. This technology is static source code analysis. After a long period of development, static source code analysis tools have become very successful commercial products, and they are now used broadly. The tools in this class require little or no explanation to use well, and can execute reasonably quickly. But curiously, they make no claim to be *reliable* as described above. The tools can, and do, miss reporting true defects (false negatives), and they can, and so, mis-report non-defects (false positives). Yet, it is clear that these tools have reached more users, and have a greater impact on software development practice, than the technically far more *reliable* formal methods tools.

In some cases then, ease of use and speed can outrank precision and reliability. After all, perfect knowledge that is inaccessible has less practical value to the end-user than partial knowledge that is easy to obtain. Phrased differently, it can be better to get *some* results from a partial method that is easy to use, than *no* results from a complete method that is too difficult to use.

We will describe how we can use the Spin model checker in a way that allows it to replicate some of the speed and usability of static analyzers. When used in the way we will describe, the model checker will be fast and easy to use, while still retaining its accuracy by never reporting non-defects. We can call this mode of allowing only false negatives but not false positives *robustness*, to distinguish it from the informal definition of *reliability* we gave earlier. In return for increased speed and ease of use, we must yield only the certainty of complete coverage. No new algorithms are needed to deploy the model checker in this way. All that is needed is to leverage the availability of already existing multi-core or cloud computing networks. When used in this way, the model checker can return robust and actionable results in seconds, even for large applications.

An Example. In 1999 we used the Spin model checker for the formal verification of the call processing code from a new commercial switching system that was marketed by Lucent Technologies, called the PathStar access server [5]. The target of this work was to automate as much as possible of the logic verification process, and to maximize its performance by parallelizing the verification tasks. The system we developed consisted of 16 small networked computers, each running at 500 MHz.

A few hundred correctness requirements were captured as part of this project, and a first model extraction technique was developed that allowed us to mechanically extract Spin verification models from the implementation level code, which was written in C. Once the requirements were formalized, the entire verification process could then be automated, and executed as part of a regression test

suite without user intervention. At the time, the verification process took roughly 40 minutes to verify all requirements, running independent verification tasks on all computers in parallel.[1]

Today, integrated multi-core desktop systems larger and faster than the network of standalone computers from 1999 have become ubiquitous. In an experiment we repeated the same verification task from before on a single desktop system with 32 cores (i.e., well below what is currently available), with each core running at 2.5 GHz.

Generating the model checking code for all properties with Spin is virtually instantaneous. A straight compilation of the generated code, without optimization, takes a little under 5 seconds. With -O1 optimization that increases to 10 seconds, with -O2 it becomes 16 seconds, and with -O3 it reaches 45 seconds. The use of compiler optimization affects how fast the model checking runs can be executed, but there is of course a tradeoff that can be made between preparation time and execution time.

The verification process itself is based on bitstate hashing with iterative search refinement, to optimize the chances of finding errors fast [5]. This works remarkably well on the multi-core system. The first 11 counter-examples are generated in just 1 second, and after 7 seconds a total of 38 counter-examples have been generated.[2]

At this point the verification could be stopped, having yielded enough evidence for developers to act on. If we allow the search process to continue, though, it can find another 38 counter-examples in the next 11 minutes, with the number of errors found per minute quickly decreasing. The total number of issues reported is slightly larger than what was obtained in 1999. Using more cores, or faster CPUs, could trivially increase the performance further. More specifically, a total of 234 different properties were verified in this experiment, with each property checked up to 5 times with the iterative search refinement method that we discuss in more detail below. This means that up to 1,170 verification runs are performed on 32 cores in parallel. If 1,170 CPUs were available, for instance with brief access to the capacity of a cloud-computing platform, the performance could trivially improve still further.

The minimum time that is required to locate the first counter-examples in this experiment is measured in seconds, with the time dominated by compilation, and not verification. It is also interesting to note that in this experiment we achieve performance that is on par with, if not better than that of static analysis.

As we discuss in more detail below, the set of counter-examples that is generated with this system is not necessarily complete: there is no guarantee that *all* errors are found, or even can be found. But this is also not necessary. The speed and ease with which counter-examples are generated can add significant practical value especially in the early phases of software development.

[1] `http://cm.bell-labs.com/cm/cs/what/feaver/`

[2] For this example we compiled the verifiers with -O3. The verification times can double at lower optimization settings.

Key Enablers. There is a relatively small number of enabling technologies that we used in this example, and we believe that each of these is essential to the success of the method. They are:

- Model extraction [3],
- Bitstate hashing [2],
- Iterative search refinement [5], and
- Swarm verification [8].

Below we briefly discuss each of these, already existing, techniques, and consider the potential of their combined use.

Model Extraction. The mechanical extraction of verification models directly from implementation level source code would be fairly straightforward if it wasn't for one single complicating factor: the need to define and apply sound logical abstractions. The abstractions can help to render complex verification tasks computationally tractable, which is needed to secure logical soundness. If we yield on soundness, though, the burden of finding strong abstractions is lessened, and sometimes removed. The Modex[3] tool, for instance, that can be used as a front-end to the Spin model checker to extract verification models from C source code, allows the user to define abstractions, but it can also operate without it.

The problem is that if we try to use model checking to exhaustively solve a complex verification task without the benefit of prior abstraction, the tool can take an unpredictable amount of time, and will likely eventually exhaust available memory and abandon the search without completing the task. We then get a partial answer to the verification task: the tool renders an incomplete result. But the result is not just incomplete, it is also highly biased by the search algorithm to fully explore only one part of the search space and ignore all of the remainder. This type of incompleteness[4] is not comparable to the incompleteness that is inherent in static source code analyzers, precisely because it is systematically biased. There is literally zero chance that an error in the unexplored part of the search space can be reported.

All types of software analyses do of course face the same intractability issues as logic model checkers do, but they handle it differently. A tool that performs an incomplete analysis but provides a meaningful sampling of the *complete search space* can still provide useful feedback to the user, as illustrated by the success of static source code analyzers. So if we could modify the model checker to make it work in a fixed amount of memory and provide a true *sampling* of the entire search space, instead of a detailed search of one small unknown portion of it we should be able to make a similar improvement in the practical value of model checking of complex applications. This, though, is precisely the type of behavior that is available through the use of bitstate hashing algorithms.

[3] `http://spinroot.com/modex/`

[4] For completeness: we mean the incompleteness of the set of results that could in principle be provided, not the *logical incompleteness* that is shared by all program analysis systems. Logical incompleteness refers to the impossibility to design a program that could prove all true properties (e.g., halting) for any program.

Bitstate Hashing. The bitstate hashing technique was introduced in 1987 [2], and as we later discovered, can be theoretically founded in the still older theory of Bloom filters from 1970.[1] Since the basic algorithms have in the last two decades been described in detail in many sources, the method itself will need no detailed explanation here. The key characteristics of the method are, though, that it (1) allows us to define a fixed upper-bound on the amount of memory that the model checking algorithm will use, and (2) that it allows us to predict with some accuracy what the maximum runtime of a verification will be. (After a fixed period of time all bits in the hash-array must have flipped from zero to one, which limits the maximum search time.) The algorithm further has the desirable feature that any counter-examples that are generated are necessarily accurate (i.e., the algorithm permits no false positives). The fundamental properties of the hashing method that is used further guarantees that in an incomplete search the part of the search space that is verified will be a random sampling of the entire search space: the search is not systematically biased. This means that all parts of a large search space will be considered, though not necessarily exhaustively. By selecting the size of the bitstate hash-array we can control both the accuracy of the search and its speed. The two are always correlated, with accuracy increasing as speed decreases.

Iterative Search Refinement. As noted, there is an inverse relation between precision (or coverage of the search space) and speed. The size of the bitstate hash array determines the maximum runtime for a bitstate hashing run. We can now increase the probability of locating errors early by performing a series of bitstate runs, starting with very small bitstate hash array sizes, and repeating as necessary with larger sizes. The first runs performed will typically complete in a fraction of a second. If they succeed in generating a counter-example, the search has been successful and can stop. If not, we double the size of the hash array to sample a larger part of the search space, and repeat. Each time the hash array size is increased, the probability of locating defects also increases. In the process we adopted in [5] the doubling of the hash array size continues until a physical memory limit is reached, or a preset upperbound on the runtime that can be used is reached. It would also be possible to terminate the search process once the the time between new error reports drops below a given limit.

A series of bitstate searches can of course be executed purely sequentially on a single CPU, but it would then consume more time than necessary. Since all runs are in principle independent, we can also perform all these runs in parallel. For the last key enabling technology we will now look at methods that further leverage available parallelism by increasing the coverage of the sampling method still further. We can achieve this effects with a swarm verification method.

Swarm Verification. If we have access to large numbers of CPU-cores, or a cloud network of computers with potentially hundreds or thousands of compute engines available, we can deploy large numbers of independent verification tasks that jointly can solve a single large problem. Each task can, for instance, be

defined by a small bitstate hashing run of the model checker. To increase the quality of the statistical sampling of the search space, we can now configure each of the verifiers to perform a slightly different type of search. We can achieve this search diversity with the Swarm[5] tool, a front-end to Spin, by:

- Using different types hash functions for each search,
- Using different numbers of hash functions for each search,
- Using different search orders in each search,
- Using randomized search methods, with different seed values.
- Using different search algorithms in each search (e.g., breadth-first, depth-first, context-bounded search [7], depth-bounded search, different types of heuristic search methods, etc.)

The variety thus added gives us the search diversity we need. It is not difficult to define as many different search variants as there are CPUs available to perform the search on, even for very large numbers.

All independent searches can be executed in parallel, using a fixed and predetermined amount of memory per CPU and completing in a known amount of time, which can now be limited to a few seconds. As we saw in the example application, the joint effectiveness of all these independent, small, and individually incomplete searches can be impressive, and can contribute true practical value.

Based on the above, we believe to be near a tipping point in the application of software model checking techniques that may be comparable in its effects on multi-threaded software development to the introduction of static source code analyzers in the last decade.

References

1. Bloom, B.H.: Spacetime trade-offs in hash coding with allowable errors. Comm. ACM 13(7), 422–426 (1970)
2. Holzmann, G.J.: On limits and possibilities of automated protocol analysis. In: Rudin, H., West, C. (eds.) Proc. 6th Int. Conf. on Protocol Specification Testing and Verification, INWG IFIP, Zurich Switzerland (June 1987)
3. Holzmann, G.J., Smith, M.H.: Software model checking – extracting verification models from source code. In: Formal Methods for Protocol Engineering and Distributed Systems, pp. 481–497. Kluwer Academic Publ. (October 1999), also in: Software Testing Verification and Reliability 11(2), 65–79 (June 2001)
4. Holzmann, G.J.: Logic Verification of ANSI-C Code with Spin. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 131–147. Springer, Heidelberg (2000)
5. Holzmann, G.J., Smith, M.H.: Automating software feature verification. Bell Labs Technical Journal 5(2), 72–87 (2000)
6. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
7. Holzmann, G.J., Florian, M.: Model checking with bounded context switching. Formal Aspects of Computing 23(3), 365–389 (2011)
8. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification techniques. IEEE Trans. on Software Eng. 37(6), 845–857 (2011)

[5] `http://spinroot.com/swarm/`