On Checking Model Checkers

Gerard J. Holzmann
gerard@research.bell-labs.com

Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Abstract. It has become good practice to expect authors of new model checking algorithms to provide not only rigorous evidence of the algorithms correctness, but also evidence of their practical significance. Though the rules for determining what is and what is not a good proof of correctness are clear, no comparable rules are usually enforced for determining the soundness of the data that is used to support the claim for practical significance. We consider here how we can flag the more common types of omission.

1 Introduction

Most of us will have experienced the phenomenon that a 'Friday afternoon discovery' falls apart when reconsidered more carefully in the early morning light. Not all compelling ideas are logically sound, not all sound ideas are also relevant, and few ideas that survive these two filters can actually make a significant difference in practice. One could say that the purpose of science is to help us perform this filtering process in a reliable and systematic manner. We know how to discover the logical flaws in our reasoning, to filter out the ideas that are not correct. To intercept the ideas that are (perhaps temporarily) not relevant, we rely on program committees, editorial boards, and grant committees. That leaves practical significance.

How do we convince ourselves that an idea can have practical impact? This is, of course, not a new problem, and it is not without solution. What makes this problem of interest is that its solution is so seldomly used.

In this paper we will look at experiments that are meant to demonstrate practical significance and critique them. Drawing examples for this purpose from the literature would make for a far too enjoyable paper. The strawman example used here is therefore strictly made-up for this purpose.

2 A Strawman Algorithm

Let us look at a simple example of a proposed improvement of the search algorithm used in the model checker SPIN. SPIN implements an on-the-fly procedure for LTL model checking that is based on explicit state enumeration by a nested depth-first search, as detailed in, for instance, [6]. The improvement we will consider here is to perform a semi-stateless search, maintaining only the depth-first stack as a temporary holding place for visited states, but no statespace. This was called a *stack-search* or *Type 3* algorithm in the taxonomy of [3].

2.1 Correctness and Relevance

The correctness of the algorithm follows from the fact that a classic depth-first search will visit all the states in a graph that are reachable from given start node, independent of whether the reached nodes are marked as visited or not. If the marking is used, the successors of each reachable state are expanded only once. If no marking is used this may happen more than once, but the search is still guaranteed to terminate. Since SPIN's model checking procedure relies only on reachability, its scope is unaffected by such a change. Furthermore, SPIN's partial order reduction strategy [5] can be expected to reduce the overhead introduced by possibly multiple visits to the same states.

The new algorithm is meant to reduce the memory requirements of a search. Memory is reduced to the requirements for the stack alone. In many cases, the maximum depth of the stack needed to traverse a graph is considerably smaller than the number of nodes in that graph. It is possible, though, that all reachable states appear in a single execution sequence, and hence would all appear in sequence on the stack. Even in this case we can expect a small memory savings because we avoid the need for the hashtables that are normally used to store the states that are removed from the stack.

2.2 Practical Value

We now have a description of a new algorithm, a persuasive argument for its logical correctness and relevance. With that, we have arguably passed two of the three filters. Next, we will try to show that the algorithm is also of practical value.

The data from a comparison of the behavior of the algorithm compared to a standard search is given in Table 1. The test data is for SPIN models of a leader election protocol [2], the classic alternating bit protocol [1], and a model of the X.21 protocol, e.g., [10], all verified using partial order reduction [5] combined with either a standard search algorithm, or the stack search algorithm using only the depth-first search stack but no statespace to store previously visited states.

Application	**Standard Search**		**Stack Search**		**Ratio (see text)**	
	States	Depth	States	Depth	Memory	Time
Alternating Bit	11	9	11	9	0.45	1.0
Leader Election	108	125	108	125	0.54	1.0
X21 model	29	21	79	31	0.42	2.7

Table 1. Comparison between Standard Search and Stack Search

With a conventional storage discipline, using hashed table-lookup, the memory requirements for the standard search are determined by the number of states

reached plus the maximum depth of the stack. In the stack search it is determined only by the latter. The entry in the one-before last column is therefore calculated as the ratio of the number of states reached with the stack search and the sum of the number of states reached in the standard search and the depth of the stack. For the relative time requirements, the ratio given in the last column is calculated as the number of states visited in the stack search divided by the number of states visited in the standard search. In both cases, a ratio less than one indicates a gain, and a ratio greater than one indicates overhead.

The focus in evaluating the stack search method is on the amount of overhead it could introduce in revisits to old states, that are avoided in the standard algorithm. The measurements in Table 1 indicate that it is indeed not unusual for the stack to gather all the reachable states in a single execution: in two of the three tests performed this phenomenon is observed. The measurements also show that the concern about the overhead introduced by the stack search is less serious than feared: in the first two tests the overhead was absent. Only in the third test did the overhead cause the number of states reached that is reported to increase.

2.3 Test Quality

The measurements appear to confirm the central assumption about the behavior of the new algorithm. But do they really? In most cases we cannot judge the validity of a conclusion by other means, as in this case, so we have to rely on the data. But even if we could not tell by other means that the conclusion was invalid, could we at least see that the experiments reported here do not constitute a valid test of the stack search algorithm? We can. The strawman demonstration above indeed has many flaws.

- **Reproducibility** We have not stated how the experiments were performed. Can a peer reproduce the results given the data? Are all the models used in the public domain and accessible to colleagues? Which system was used to perform the experiments? How was the standard search modified into a stack search, or was a new search engine written from scratch for these experiments?
- **Test Selection** How were the tests chosen? Are the results representative? What precautions were taken to make sure that this is so? What would trigger worse behavior? Is it predictable in which cases we get good and in which cases we get bad behavior from the new algorithm? Is the mechanism that accounts for the outcome of the tests fully understood and reported?
- **Scope and Controls** The test provides isolated data points, but does not give the context needed to properly interpret them. Can the parameter that was changed between the two algorithms also be varied more gradually? (The parameter is arguably the number of states that is saved in a statespace cache, and it could be varied from all to none.) The test fails to provide all results for the controlled parameter.

- **Interpretation** Is the data clearly separated from the tester's interpretation? In the example, the computation of the ratios from the last two columns in Table 1 conceals the real measurements of memory and time. How can we be sure that no data was lost or accidentily misrepresented? Note that in a standard search the stack only needs to maintain a pointer to a state that appears elsewhere in the statespace. The cost of maintaining a stack in the standard search, therefore, is less than in a stack search. The memory ratios in Table 1, therefore, are probably too optimistic.

A rigorous testing procedure has the same purpose as a formal correctness proof: it is intended to help us catch bugs in our reasoning. Without any discipline or controls, we are generally motivated to collect only supporting evidence for the quality of a proposed new algorithm. A meaningful test asks us to go against our best judgement and to undertake a targeted attempt scrutinize the validity of our work. A meaningful test is, in a way, the documentation of a serious, scrupulously objective, verifiable, and hopefully failed, attempt to **find fault** with our work. Clearly, the data reported in Table 1 has none of these qualities.

3 The Test Hypothesis

A good test is not the random collection of data from measurements: it is a targeted attempt to check a specific claim, or *hypothesis*. In model checking applications the hypothesis is typically about the relative performance of algorithms. The hypothesis should be specific enough that it can predict the outcome of the measurements before they are done. The prediction must be specific enough that it can clearly distinguish random luck in the outcome of a test from a match of the theory.

So where does the hypothesis itself come from? It can be based on a critical insight into the nature of a prior algorithm, or, it can be based on measurements, which, in this case only, are more or less randomly performed. The first type of measurements are not 'tests' since they are not intended to test anything in particular. There is a fundamental difference between a *measurement* and a *test*. [1]

- **Measurements** are intended to collect empirical data without bias. The data generally suggests a hypothesis that can explain the data that was obtained. The quality of such a hypothesis is determined by its predictive power: the hypothesis should predict the outcome of a new set of tests in which some parameter is changed. The hypothesis should also firmly identify specific results as a contradiction to the hypothesis itself, which, if observed, would cause the hypothesis itself to fall. That is: in advance of new experiments it should be possible to identify clearly what specific results would contradict the hypothesis itself.

[1] In this paper we use the term *experiment* if either a *measurement* or a *test* is meant.

- **Tests** are meant to challenge the hypothesis. They are intended to be a serious attempt to determine if results that would contradict the hypothesis can be obtained [8, 9]. They are **not** attempts to further support the hypothesis: that support is already available in the data that inspired the hypothesis to begin with.

We frequently tend to use a *measurement* when a *test* is called for. The sample measurements for the stack search algorithm are an example of such an omission.

What are the properties of a true test? A test can offer support for a hypothesis by recording the outcome of a serious attempt to discover contradictions to the results it predicts or implies. In a sense, a test therefore is intended to work just like a model checker: it attempts to find a counter-example to a correctness claim, and from its failure to do so we may draw some tentative (though rarely definitive) conclusions.

We can only test a hypothesis if it can indeed be contradicted: Popper's classic criterion for distinguishing a scientifically meaningful statement [8, 9] from a non-scientific one. A good test, then, must have the following identifiable components:

- **Hypothesis** – The hypothesis is based on easily obtainable supporting data, which are obtained by more or less random, but certainly unbiased, measurements. The hypothesis must have a predictive power and should, if it can survive meaningful challenges, provide insight or knowledge. The hypothesis can, for instance, postulate a specific cause of complexity in model checking applications. Removing that cause then should result in a visible reduction of the complexity. The failure to observe such a relation would contradict the hypothesis.
- **Test Setup** – A clear description of how the tests were performed with sufficient detail that the tests can be repeated, and *challenged*, by peers.
- **Test Data** – The test data must give an uninterpreted description of the results of a series of controlled experiments that are meant to seriously and thoroughly challenge the validity of the hypothesis.
- **Interpretation** – Data and the interpretation of data must be clearly separated, so that in principle it would be possible for peers to reach different conclusions based on the same raw data. The interpretation concludes whether the hypothesis survived or failed the tests, or it could conclude that the outcome is inconclusive and requires a different set of challenges.

To evaluate the validity of a conclusion drawn from the tests, all four pieces of the test description are generally needed.

4 Application

Let us now return to the stack search example and test its performance, rather than measure it. We can take the data from Table 1 as an initial set of measurements. We now first describe a motivation for a hypothesis to be tested, and then the hypothesis itself.

Motivation – The statespace used in a reachability analysis can be seen as a way to optimize the search process by preventing the renewed exploration of the successors of previously explored states. Partial order reduction reduces the number of times that previously explored states are revisited, and therefore the reliance on the statespace is also diminished. The statespace increases the memory requirements in order to decrease the time requirements of the search. By examining the properties of a stack search in combination with partial order reduction, we can check if this trade-off needs to be re-assessed.

Hypothesis – The stack search saves memory over a standard search, at the expense of potentially increasing the time requirements. A larger increase in time requirements is predicted to be correlated with a larger decrease in memory requirements. Specifically, reductions in memory requirements of one order of magnitude or more are predicted to be linked to an increase in the time requirements by no more than an order of magnitude.

This hypothesis fails if, for instance, we can establish that there are cases where the stack search uses more memory than the standard search, or (more likely) if we can show that in a set of reasonable applications a reduction in memory of less than an order of magnitude is combined with an increase of time by more than an order of magnitude.

Test Setup – In the following, a "standard search" is the classic depth-first search algorithm with both a search stack and a statespace. The search stack contains (pointers to) states on the path that leads from the initial system state to the currently explored state. The statespace is assumed here to be implemented with hashed table-lookup. A "stack search" is the same algorithm that operates without the statespace store. This can be implemented, at least for the purposes of the tests that follow, by adapting the statespace storage routine (`hstore` in SPIN) to pretend that states that were found in the statespace store (and outside the search stack) where newly created. The Appendix shows the specific code fragment that is modified for these tests.

To test the hypothesis, we will first check if we really understand the mechanism that controls the tradeoff between the two resources that are at stake here: memory and time. We will try to create a simple model for which the stack search algorithm is predicted to perform optimally, giving memory savings without time penalty. We will also try to create a model for which the stack search should exhibit worst case performance, giving minimal memory reduction in return for maximal time penalty. To accomplish this we must (according to the hypothesis) control the number of revisits to previously visited states. If we can avoid revisits entirely we should observe best case performance, if we can secure a maximally connected statespace, or something close to it, we should observe worst case performance. The Appendix gives the PROMELA source for the two models that were used to trigger *best case* and *worst case* performance.

Data – Table 2 gives the results of the experiments performed with the best and worst case test models as input. Instead of given estimated ratios for the

time requirements, we give memory use in Megabytes, and seconds of runtime measured on an 180 MHz SGI workstation, with 64 Mbytes of memory. For the memory requirements of a stack search the amount of memory used for the state space is subtracted from the amount of memory that is reported (see Appendix).

Application	Standard Search				Stack Search			
	States Reached	Depth Max.	Memory Mbyte	Time Sec.	States Reached	Depth Max.	Memory Mbyte	Time Sec.
Best Case	2047	30	1.25	0.09	2047	30	0.14	0.10
Worst Case	124	20	1.25	0.06	23,694,800	20	0.15	218.89

Table 2. Best and Worst Case Performance of Stack Search Algorithm

Interpretation – The memory reduction achieved is not as large as might be expected, because of a small overhead that goes to unrelated data structures used in the model checking process. The time penalty for the best case test is negligible, but for the worst case test it is beyond the maximum predicted by the hypothesis. We will consider these first two tests to be inconclusive and defer judgement until we can determine how likely it is to observe best or worst case performance in average practical applications.

Additional Tests – Performing extra tests is complicated by the fact that it is somewhat rare to find a larger application that can be run to completion with the stack search algorithm, so severe is the runtime overhead. A runtime overhead of three orders of magnitude, for instance, turns one minute of runtime into one day. Five orders of magnitude overhead, turns one minute into three months. This restricts us to only relatively small models to gather more data. The test results for three realistic applications for which the stack search does complete within a reasonable amount of time are given in Table 3.

Application	Standard Search				Stack Search			
	States Reached	Depth Max.	Memory Mbyte	Time Sec.	States Reached	Depth Max.	Memory Mbyte	Time Sec.
Ring (Appendix)	466	18	1.25	0.10	16,469,100	18	0.14	251.32
URP model [4]	1,363	146	1.35	0.13	26,018,900	146	0.15	1080.58
DTP model [4]	16,459	526	3.30	0.63	79,308,200	549	0.20	1562.81

Table 3. Additional Tests of the Stack Search Algorithm

Interpretation – The hypothesis as it was formulated for these tests has failed. The additional experiments show an increase of runtime well beyond what is allowed by the hypothesis. A reduction in memory use of approximately one order of magnitude is paired with an increase of runtime by more than an order of magnitude in all tests, except the one test that was deliberately constructed to behave well.

To explain these results, we can observe that the stack search is really a special case of a caching strategy with a zero-size cache. With shrinking cache size the depth-first search incurs exponentially rising costs. It is known [4] that when state caching is used in combination with a partial order reduction strategy, the exponential effect sets in for smaller cache sizes, but does not disappear. To understand and measure these effects more precisely we could now add a parameter to our test implementation of the stack search method, that denies the presence of a state in the statespace only with a certain test-controlled probability. The above experiments, however, will suffice for the purposes of this paper.

5 Conclusion

Many papers on model checking algorithms contain a section with experimental data. We have considered how, in an ideal world, such a section would be written.

A well-known dictum says: *"A program without a specification cannot be proven correct; it can neither be right nor wrong."* We can paraphrase this as: *"A test without a hypothesis cannot succeed or fail."* For a test to be able to succeed, it must also be able to fail, and either result can be of interest. But to succeed or fail, a test needs to have a precisely stated purpose. The purpose of a well-designed test is not to confirm, but to *challenge* our presumptions.

It is not hard to see the value of a self-imposed obligation to render a formal proof of correctness of even a trivially correct (sic) algorithm. Imposing more rigor in conducting a demonstration of practical significance similarly has the benefit of protecting us from occasionally misleading ourselves.

The occurrence of bugs is of course not restricted to algorithms or implementations. They can also appear in formal proofs and in the experiments we perform to demonstrate practical significance. That's the bad news. The good news is: bugs don't like rigor.

> *"Bugs are by far the largest and most successful class of entity, with nearly a million known species. In this respect, they outnumber all the other known creatures four to one.*
> Prof. Snopes' *"Encyclopedia of Animal Life"*, as quoted in [7], p. 31.

6 Appendix

Test Setup – The experiments reported in this paper were performed with SPIN version 3.2.0 (`http://netlib.bell-labs.com/netlib/spin/`). The results should be similar for most other versions of SPIN. A small discrepancy in the numbers of reached states is possible for some older versions of SPIN, due to the recent revision of the partial order reduction strategy described in [6].

Stack Search Implementation – The search algorithm from the `pan.c` model checkers generated by SPIN on a `spin -a model` command was modified to imitate a stack search algorithm by editing the `pan.c` file with the following script.

/bin/ed pan.c
/match outside stack/
$s/.*/tmp->tagged = (S_A)?V_A : (depth+1);\ Lstate = tmp;\ return\ 0;/$
w pan.c
q

The change is designed to be conservative, being slightly more efficient than a true implementation of a stack search algorithm, which would have to recreate and store every previously visited state on each new visit. In the version of the algorithm tested, the previously created state is preserved, but returned to the stack when revisited, by resetting its `tagged` field to a non-zero value.

Best Case Test – The following PROMELA program attempts to trigger best-case performance in the stack search, by minimizing the number of revisits.

$\#define\ N\ 10$
$byte\ a;\ chan\ q = [N]\ of\ \{\ byte,\ byte\ \};$
$active\ [2]\ proctype\ T()\ \{\ do\ ::\ atomic\ \{\ a < N\ ->\ q!a, \_pid;\ a++\ \}\ od\ \}$

Worst Case Test – The matching attempt to trigger worst case performance, by maximizing the number of revisits.

$\#define\ N\ 4$
$byte\ a;$
$active\ [N]\ proctype\ T()\ \{\ do\ ::\ a = (a+1)\%N\ ::\ break\ od\ \}$

Ring Protocol – The following is the PROMELA source for the ring protocol, that was used for the measurements reported in Table 3.

$\#define\ N\ 6$
$chan\ ring[N] = [1]\ of\ \{\ byte\ \};$
$active\ [N]\ proctype\ node()\ \{\ ring[(\_pid+1)\%N]!1;\ ring[\_pid]?1\ \}$

References

1. Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. A note on reliable full-duplex transmission over half-duplex lines, *Comm. of the ACM*, Vol. 12, No. 5, 260-265.
2. Dolev, D., Klawe, M., and Rodeh, M., An O(n log n) unidirectional distributed algorithm for extrema finding in a circle, *Journal of Algorithms*, Vol 3., 1982, pp. 245-260.
3. Holzmann, G.J., Algorithms for automated protocol verification, *AT&T Technical Journal*, Vol. 69, No. 2, Feb. 1990, pp. 32-44.
4. Godefroid, P., Holzmann, G.J., and Pirottin, D., State Space Caching Revisited, *Formal Methods in System Design.* Vol. 7, No. 3, Kluwer Academic Publ., 1995, pp. 1-15.
5. Holzmann, G.J., and Peled, D. An improvement in formal verification, *Proc. Formal Description Techniques, FORTE94*, Chapman & Hall, pp. 197-211, October 1994.
6. Holzmann, G.J., Peled, D., and Yannakakis, M., On Nested Depth First Search, *The Spin Verification System*, pp. 23-32. American Mathematical Society, June 1996.
7. Van der Linden, P., *Expert C programming*, Prentice Hall, 1994.
8. Popper, K.R., *Logic of scientific discovery.* Basic Books. original 1934, revised edition 1959.
9. Popper, K.R., *Conjectures and refutations: the growth of scientific knowledge.* Basic Books, 1962.
10. West, C.H., and Zafiropulo, P. Automated validation of a communications protocol: the CCITT X.21 recommendation, *IBM J. Res. Develop.*, Vol. 22, No. 1, pp. 60-71.