The Engineering of a Model Checker: the Gnu i-Protocol Case Study Revisited.

Gerard J. Holzmann

Bell Laboratories, Lucent Technologies,
600 Mountain Avenue, Murray Hill, NJ 07974, USA
gerard@research.bell-labs.com

Abstract. In a recent study a series of model checkers, among which Spin [5], SMV [9], and a newer system called XMC [10], were compared on performance. The measurements used for this comparison focused on a model of the i-protocol from GNU uucp version 1.04. Eight versions of this i-protocol model were obtained by varying window size, assumptions about the transmission channel, and the presence or absence of a patch for a known livelock error. The results as published in [1] show the XMC system to outperform the other model checking systems on most of the tests. It also contains a challenge to the builders of the other model checkers to match the results. This paper answers that challenge for the Spin model checker. We show that with either default Spin verification runs, or a reasonable choice of parameter settings, the version of Spin that was used for the tests in [1] (Spin 2.9.7) can outperform the results obtained with XMC in six out of eight tests. Inspired by the comparisons, and the description in of the optimizations used in XMC, we also extended Spin with some of the same optimizations, leading to a new Spin version 3.3.0. We show that with these changes Spin can outperform XMC on all eight tests.

Introduction

The details of the i-protocol, and the various versions of the model that were built to verify it, are given in [1] and need not be repeated here. We focus on the tests as reported in [1], specifically as they relate to Spin. The three main parameters that were modified to obtain the eight testcases for the i-protocol model are as follows:

1. A window size of either one or two is used. We distinguish these two cases here as `1--` and `2--`.
2. The possibility of message distortions is either included or excluded. We distinguish these two cases as `-f-` (*full*) and `-m-` (*mini*) respectively).
3. A patch to prevent the known livelock error is enabled or disabled. We distinguish these two cases as `--f` (*fixed*) and `--n` (*non-fixed*) respectively.

The eight testcases can then be identified as `1mf`, `1mn`, `2mn`, `2mf`, `1ff`, `1fn`, `2fn`, and `2ff`. Within these eight cases, we should distinguish two groups. In the first group we place all the *non-fixed* versions of the model: 1mn, 2mn, 1fn, and 2fn. All *fixed* versions are in the second group: 1mf, 2mf, 1ff, and 2ff.

The reason for the distinction is the way in which the tests in [1] were performed. For all non-fixed versions of the model the verification run was halted as soon as the livelock error had been reported. Because of unpredictable and uncontrollable variations in search order, the amount of work done up to that point is not a reliable indicator of the overall performance of the model checkers. The measurement can serve only as a soft heuristic. For completeness only, we have included these measurements also in the new tests that we have performed. As we shall see in this case, the results for the first group and the second group of models are not inconsistent, but this should be considered a coincidence.

Overview of Paper

For most of the measurements performed here we adopt the Promela model of the i-protocol as it was developed for the original study in [1], unmodified. We must observe, though, that when the runs of Spin are compared with runs of XMC, or other model checkers, we are not just comparing the *tools* but also the relative quality of the *models*.

The most striking observations in [1] are differences between the tools in the use of basic resources such as memory and runtime, but also differences in the number of reachable states that are searched to solve the model checking problem. In principle, for explicit state model checkers, for a given problem specification, the size of the reachable statespace *before* reduction should match. In all versions of Spin the reduction algorithms can be disabled to perform a reference measurement for this purpose, giving the size of the original state space before reduction. The model checker XMC does not support such an option, which makes it harder to verify that the models used are equivalent.

In what follows we will try to address these problems separately. We first review the measurements reported in [1] (reported below as S297), while adopting the model from [1] unchanged, and restrict ourselves to making some small corrections in the use of Spin alone. Two types of corrections are made separately: the removal of faulty parameter setting (reported as S297*) and the addition of a useful compile time directive (reported as S297+).

Next we consider what portion of the differences in the statespace sizes searched by the two tools (XMC and Spin) can be attributed to differences in reduction strategies. Several of the strategies that have been used in XMC can beneficially be added to Spin. We have done so, leading to a new release of Spin numbered Version 3.3.0. The new measurements show considerable gains in performance for Spin. Differences in statespace sizes searched by XMC and Spin remain, however, which leads to a closer consideration of the models that were used. We conclude the paper with a final set of measurements for an equivalent, but slightly rewritten, Promela model of the i-protocol. The results are illuminating.

First Results

Tables 1 and 2 reproduce the results from [1] as they relate to XMC and Spin. The results show a disproportionate amount of memory used for the Spin runs, even for very small statespaces (e.g. for model 1mn). Dividing the reported 749 Mbytes of memory used for model 1mn by the 425 reported reachable states corresponds to an excessive amount of memory use of 1.76 Mbytes per state. Each state-vector in this model is no more than about 100 bytes long, so something is wrong.

The cause is not hard to find. The default stacksize used in the experiments in [1] was increased well beyond what was necessary to perform the runs. Using the default settings from Spin already produces more competitive numbers. In these tables we have included the results for the version of Spin that matches the on used in [1] with the stacksize set to a size that suffices to solve the problem, which is typically somewhat smaller than the default. These entries are labeled S297*, to distinguish them from the test results reported in [1]. Also added are entries labeled S297+ for the same runs with the loss-less COLLAPSE compression option [4] enabled, to reduce memory requirements some more for an additional runtime cost. The memory requirements can be reduced still further, at a higher run time penalty, by using the minimized automaton option [6], but we have not pursued this here.

The results for XMC and Spin are for different types of models, so a direct comparison of the results is not necessarily meaningful. We take a closer look at this issue towards the end of the paper, but meanwhile adopt the models precisely as they were used in the original study [1].

Table 1. Test Results First Group, from [1] for XMC and Spin Version 2.9.7 (S297)

Model	Tool	Reachable States	Memory (Mb)	Time (m:s)
1mn	XMC	341	5.0	0:01
	S297	425	749.0	0:10
	S297*	14K	1.8	0:01
	S297+	14K	1.5	0:01
2mn	XMC	1K	11.0	0:02
	S297	35K	751.0	0:12
	S297*	46K	7.6	0:03
	S297+	46K	5.1	0:04
1fn	XMC	961	9.0	0:01
	S297	5.2K	749.0	0:11
	S297*	5.2K	1.9	0:01
	S297+	5.2K	1.7	0:01
2fn	XMC	4K	35.0	0:05
	S297	17K	750.0	0:17
	S297*	68K	12.0	0:03
	S297+	68K	8.6	0:06

Table 2. Test Results Second Group, from [1] for XMC and Spin Version 2.9.7 (S297)

Model	Tool	Reachable States	Memory (Mb)	Time (m:s)
1mf	XMC	3K	78.0	0:17
	S297	322K	774.0	0:31
	S297*	322K	29.9	0:13
	S297+	322K	14.5	0:21
2mf	XMC	20K	475.0	1:49
	S297	1.9M	905.0	2:28
	S297*	1.9M	181.0	1:34
	S297+	1.9M	82.0	2:20
1ff	XMC	36K	1051.0	3:36
	S297	12.6M	1713.0	17:50
	S297*	12.6M	1088.0	17:38
	S297+	12.6M	497.0	31:06
2ff	XMC	315K	4708.0	47:15
	S297	-	-	-
	S297+	>28M	>1600.0	>89:00

All C programs were compiled with standard optimization enabled. All new measurements reported in this paper were made on an SGI IP27 Challenge MIPS R1000 computer with 1.6 Gigabytes of available memory. The performance of this machine matches the SGI IP25 Challenge MIPS R1000 used in [1], which makes a direct comparison of runtimes and memory use possible. All runtimes in the last column of Tables 1 and 2 are calculated as user time plus system time, as measured by the Unix® time command. The last test (for model 2ff) is clearly the most challenging one. It could not be completed with the model as given within the available 1.6 Gigabytes of memory on our machine.

The precise measurements from [1] could not be reproduced in three of the eight cases, curiously leading to larger problem sizes in the new tests. Nonetheless, the final results are substantially different from those reported in [1]. Spin version 2.9.7 outperforms XMC in six of the eight testcases. In each of these cases Spin used fewer resources, despite searching considerably larger statespaces. The difference is statespace size is most likely due to a more aggressive use of reductions, as outlined in [2], and as we shall investigate below. For models 1ff and 2ff, XMC is the winner in statespace size, memory use, and runtime requirements.

The difference in the statespace sizes explored, illustrated in Tables 1 and 2, does point to the potential importance of the optimization techniques that were employed in the design of XMC. In the next section we describe the implementation in Spin Version 3.3.0 of these and similar reduction techniques, and we repeat the experiments for Spin with these optimizations added.

Spin Version 3.3.0

In [2] several optimization techniques are mentioned that are likely in part responsible for the small statespace sizes that are constructed with the XMC model checker. Among these techniques are:

1. The merging of internal transitions into single internal steps to avoid redundant interleavings during the model checking process and to avoid the performance of extra work at run-time.
2. Dead variable elimination based on a standard control-flow analysis.
3. Aggressive optimization of the control structure, e.g., to eliminate dead branches.

We discuss each of these points in more detail below.

Statement Merging

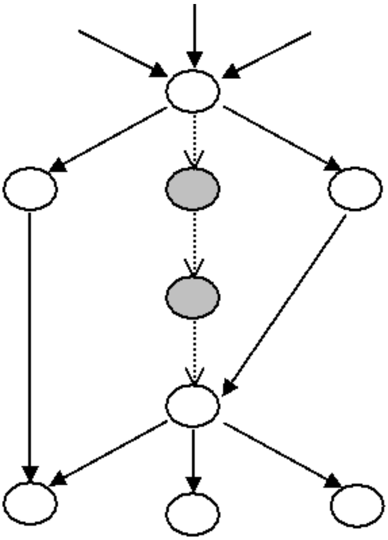
The principle of the statement merging technique is well established. Statement merging is a special case of partial order reduction. The partial order reduction method that is implemented in Spin version 2 was described in [3]. This method suppresses redundant interleavings of process actions wherever safely possible, but it does not attempt to perform compile-time optimizations when non-interleaved sequences of process actions can be merged into a single step. A proposal to add this capability was made earlier in [11], but not pursued at the time. In terms of Spin semantics, the change that has now been made in Spin version 3.3.0 amounts to the automatic assignment of **d_step** constructs to all pieces of code where this can safely be done under the partial order reduction rules from [3]. The implementation is conservative by not including selection or iteration statements into these merge sequences. The reason for this is as follows. A condition for the safety of the transformation we have applied is that no non-deterministic constructs be included in the merge sequences. It is, however, not possible to reliably determine at compile time whether the evaluation of the guards in a selection or iteration construct may produce non-determinism at run time.

Figure 1 illustrates the context in which statement merging can be applied. In the current implementation of Spin, we determine at compile time (statically) whether the conditions are satisfied for the creation of states with indegree and outdegree necessarily equal to one. This implementation is conservative, and might be improved upon to increase the amount of reduction obtained. Our implementation can, for instance, be extended by including also sub-sets of selection and iteration construct that *can* statically be determined to be deterministic. Such an extension can translate immediately into additional run time reductions.

Dead Variable Elimination

The second technique can similarly be done statically at compile-time in the model checker. Spin 3.3.0 recognizes globally dead, write-only variables and eliminates them from the state-vector. These types of variables can, for instance, appear in models that are parameterized to yield different interpretation of a single general specification, as we see in the current set of testcases. Eliminating these variables translates into memory savings by globally shrinking all state descriptors, without loss of information. It can reduce search spaces as well, by avoiding the storage of write-only values that provably can have no bearing on the correctness properties of a model. (An often seen case is the use of a write-only counter variable that, unless it is eliminated, can multiply the search space by the full range of the counter Cases such as this are discussed in more detail in [7].)

Fig. 1. Fragment of a graph reduced under standard partial order reduction rules. Transitions with both indegree and outdegree equal to one are merged with the new statement merging technique. Merging these statements avoids the creation at runtime of the shaded states.



Based on similar observations, Spin 3.3.0 performs a data-flow analysis at compile-time to also avoid the storage of values for *temporarily* dead variables. A variable is temporarily dead if it cannot be read until *after* the variable has been assigned a new value. Our implementation is again conservative by resetting temporarily dead variables to a fixed value of zero, instead of temporarily stripping these variables from the state descriptor itself. The motivation for the latter is efficiency: the model checker would be slowed down too by having to assemble and reassemble the state descriptor on the fly from a varying collection of elements at each step.

Automata Pruning

The third technique will eliminate dead parts of the control structure. Parts of the model can become unreachable, as with globally dead variables, for specific parameter settings that are used to specialize the behavior of a general model specification. The savings will be in avoiding the evaluation of guard conditions at run time in cases where a static evaluation at compile time can predict that the conditions are necessarily false for the given parameter settings. Especially for larger models than considered here (e.g., [8]), this option can prove to be beneficial. For the i-protocol, however, this reduction method does not appear to contribute significant extra savings.

Stack Cycling

In addition to the three optimizations inspired by [2], we have also implemented a new stack cycling technique that can help avoid the memory expense of model checking runs that require an unusually large search depth. This method exploits the observation that the typical access pattern of the search stack is very different from that of the regular state space. The stack is only accessed in a strictly predictable order: frames high on the stack, close to the root of the search tree, will not be accessed until the depth-first search eventually returns to that level, while frames low on the stack will be accessed frequently. We can build an effective caching mechanism that swaps large portions of the stack to disk, without significant run-time overhead.

In the implementation of this discipline in Spin 3.3.0, the default stack size, predefined by Spin, always suffices to perform a search, no matter how deep the search stack grows. The predefined allocated stack serves as a 'window' into the real stack, which can grow dynamically. In cases where the real stack grows larger than the allocated stack, the bottom half of the stack is swapped to disk, and the search continues with the freed up slots now available to grow the stack further. Because the stack is accessed via pointers, there is no additional overhead beyond the writing of the bottom half of the current stack window to disk in a single write operation, and the reassignment of one single stack pointer. When the search retreats, and stack frames are needed that are currently on disk, the reverse happens. Half the stack window is read back in from disk in a single read operation, and the stack pointer is adjusted. There is remarkably little overhead involved, and the process is fully transparent to the model checking process itself.

New Measurements

The first three optimizations are part of the new default search strategy in Spin 3.3.0. Stack cycling is kept as a compile-time option, since the need for it is still relatively rare. We have used it on just one of the tests for the i-protocol, shown in Table 3. For the most challenging model, `2ff`, this option can shave an extra 160

Mbytes from the memory requirements, reducing it to 606 Mbytes, in return for an increase of runtime of approximately 5 percent.

Table 3. Test Results first group, comparing XMC and Spin Version 3.3.0 (S330)

Model	Tool	Reachable States	Memory (MB)	Time (m:s)
1mn	XMC	341	5.0	0:01
	S330	3K	1.6	0:01
	S330+	3K	1.5	0:01
2mn	XMC	1K	11.0	0:02
	S330	15K	3.0	0:01
	S330+	15K	2.2	0:01
1fn	XMC	961	9.0	0:01
	S330	851	1.3	0:01
	S330+	851	1.3	0:01
2fn	XMC	4K	35.0	0:05
	S330	20K	3.8	0:02
	S330+	20K	2.8	0:01

Table 3 lists all test results, comparing the performance of XMC as reported in [1] with the results for Spin 3.3.0. For the test results marked S330, we only used parameter settings that were sufficient to solve the problem. On the results marked S330+ we added the optional COLLAPSE compression mode [4] to reduce the memory requirements without affecting coverage, but while sacrificing some speed.

Curiously, the numbers of states searched by Spin, with one exception (1fn) remains larger than reported for XMC, yet the resource requirements are considerably smaller in all cases. We return to this issue in the last section of this paper.

Table 4. Test Results second group, comparing XMC and Spin Version 3.3.0 (S330)

Model	Tool	Reachable States	Memory (MB)	Time (m:s)
1mf	XMC	3K	78.0	0:17
	S330	148K	14.0	0:05
	S330+	148K	6.8	0:09
2mf	XMC	20K	475	1:49
	S330	965K	87.0	0:39
	S330+	965K	37.7	1:01
1ff	XMC	36K	1051.0	3:36
	S330	2.5M	214.0	2:15
	S330+	2.5M	90.7	3:12
2ff	XMC	315K	4708.0	47:15
	S330+	21.9M	606.0	62:00

Based on the measurements for SMV as reported in [1], for the given model in Promela and the given model in SMV, the Spin verification appears to outperform

SMV on these tests in seven out of eight cases. According to [1], in four cases SMV ran out of memory, in three cases it used substantially more memory, and in only one case (1ff) it used less. Unfortunately, in the absence of a more focused series of experiments comparing SMV and Spin directly, and in the absence of a more careful examination of differences and similarities between the SMV model and the Spin model used in these tests, no conclusions can be drawn from these observations.

The Effect of Each Optimization

To get a better indication of the effect that is contributed by the optimizations separately, we measured the performance of the model checker for one of the testcases with the individual optimizations enabled or disabled. We used the 2mf model for these measurements. Automata pruning and the elimination of globally dead variables were not a factor in the measurements for this specific set of models, so we have omitted these from Table 5.

For reference, the first row in the table matches the result for 2mf in Table 4, and the last matches the result for 2mf in Table 2 for Spin version 2.9.7. As an additional checkpoint, we also include the results for version 3.3.0 with all new optimizations turned off, in the fourth row of Table 5. It turns out to use slightly less memory and slightly more runtime than version 2.9.7, due to small and otherwise unrelated, changes in the code. Statement merging appears to contribute almost all the benefits that are observed in this experiment. The dataflow analysis appears to carry a small penalty in this case, unless it is combined with statement merging.

Table 5. Test results for Spin Versions 3.3.0 and 2.9.7 on model version 2mf.

Version	States	Memory (Mb)	Time (m:s)
All optimizations enabled	965K	87	0:58
Statement merging only	970K	100	0:59
Dataflow analysis only	1.9M	179	2:17
No optimizations (Version 3.3.0)	1.9M	173	2:12
No optimizations (Version 2.9.7)	1.9M	181	1:59

The Accuracy of the Model

If we review the results tabulated in [1], a few things stand out. The first we mentioned earlier in this paper: the large divergence in the sizes of the statespaces searched by each of the tools, despite the fact that all tools attempt to solve the same problem. Part of this divergence can be attributed to differences in reduction techniques employed, but not all. A second item is the difference in the sizes of the state-descriptors that are used in the *explicit-state* model checkers. We can obtain this number nominally by dividing overall memory use reported by the number of states that was stored. For the Murφ tool, for instance, the size of a state descriptor comes out at roughly 60 bytes. For Spin, however, the number is roughly 128: twice as large. A comparison of the two models used for these tests, the one for the Murφ tool and the one built for the Spin tool, shows important differences. The Spin model, as constructed for these experiments, uses more processes, more types of data, fewer cases of atomicity, etc. etc. It is not hard to improve upon the Spin model, without affecting its functionality though. The author spent 1 hour of editing time to make mostly straightforward changes to the Spin version of the i-protocol model, following the recommendations documented in [7]. Sink and source processes, as defined in [7], were removed. The revised Spin model is a better match to the competing models. The results of the measurements with this revised model, for both Spin version 2.9.7 and Spin version 3.3.0, are tabulated in Tables 6 and 7, again compared against the original XMC measurements from [1].

Table 6. Final Results, first group, revised model, XMC and Spin Versions 2.9.7 and 3.3.0

Model	Tool	Reachable States	Memory (MB)	Time (m:s)
1mn	XMC	341	5.0	0:01
	2.9.7	86	1.8	0:01
	3.3.0	77	1.8	0:01
2mn	XMC	1K	11.0	0:02
	2.9.7	111	1.8	0:01
	3.3.0	101	1.8	0:01
1fn	XMC	961	9.0	0:01
	2.9.7	86	1.8	0:01
	3.3.0	77	1.8	0:01
2fn	XMC	4K	35	0:05
	2.9.7	111	1.8	0:01
	3.3.0	101	1.8	0:01

In these final measurements, all Spin tests were run with compression enabled, with a search stack limit set to a level that accommodated the most complex version of the model (2ff). The numbers for XMC are again taken from [1], measured on hardware with identical performance. As before, all runs in the first group stop when the livelock error is detected (and all runs successfully detect this error). All results for the second group of models, shown in Table 7, are for exhaustive verification runs, that stop when the full state space of the model has been explored, and the absence of liveness and safety errors was proven.

In the first group of tests, Spin explores a smaller state space in all 4 cases, by a comfortable margin. But, as noted before, the tests in this first group are not necessarily indicative of overall tool performance. The tests in the second group are a better indicator for this (since the verifications are run to completion in all cases).

Table 7. Final Results, second group, revised model, XMC and Spin Versions 2.9.7 and 3.3.0

Model	Tool	Reachable States	Memory (MB)	Time (m:s)
1mf	XMC	3K	78.0	0:17
	2.9.7	17K	2.1	0:07
	3.3.0	13K	1.9	0:05
2mf	XMC	20K	475.0	1:49
	2.9.7	75K	3.4	0:03
	3.3.0	55K	2.9	0:02
1ff	XMC	36K	1051.0	3:36
	2.9.7	35K	2.5	0:01
	3.3.0	25K	2.3	0:01
2ff	XMC	315K	4708.0	47:15
	2.9.7	208K	6.5	0:08
	3.3.0	161K	5.2	0:06

For the second group of tests Spin explores more states than XMC in 2 out of 4 cases, and fewer states than XMC in the other 2 of the 4. The numbers are close enough, though to give some reassurance that the models are comparable. In all cases in this group Spin uses fewer resources, by a very comfortable margin.

Comparing the results from Tables 6 and 7 to those presented in Table 3 and 4 emphasizes the importance of model construction. The effect on complexity of a well-designed [7] or a poorly designed model can easily overpower the effect of the search algorithm used, or the choice of model checking tool. With these observations we do not intend to underestimate the significance of what was achieved in [1]. Testing and comparing model checkers in a scientifically meaningful way is hard. It is perhaps an art form that none of us have quite mastered yet.

Conclusion

The suggestions for optimization from the report on the XMC model checker [2] are sufficiently general that they can readily be included in other model checking systems. We have implemented three of the optimization techniques in the Spin model checker, and confirmed that these optimizations can reduce its resource requirements. The remaining discrepancies in the numbers of states searched during the model checking process between XMC and Spin Version 3.3.0 are only partly due to further opportunities for reduction that may exist in the Spin model checker. The data in Tables 6 and 7, however, shows that the remaining differences can be attributed to differences in the way that the models were constructed.

In the measurements from Tables 6 and 7, Spin outperforms XMC on memory use and time consumption by one to three orders of magnitude. In the implementation of most model checking algorithms there is a delicate trade-off to be made between time and space. It is almost always the case that reducing the memory requirements further can be achieved only at the expense of the time requirements, and vice versa. All memory requirement figures reported here for Spin can be reduced further, in some cases significantly, by using more aggressive memory compression options (e.g., as the minimized automaton representation detailed in [6]). We have resisted the temptation to do so here.

Figure 2 summarizes the results we have obtained for the original Spin models from [1] by plotting the relative memory use for the three most significant series of measurements for all eight test cases. The vertical axis shows memory use in Megabytes. The horizontal axis lists the various testcases and versions of the tool. We have omitted the result from Table 6 in this Figure. If plotted, it would show a very considerable advantage for the Spin verifications in all tests.

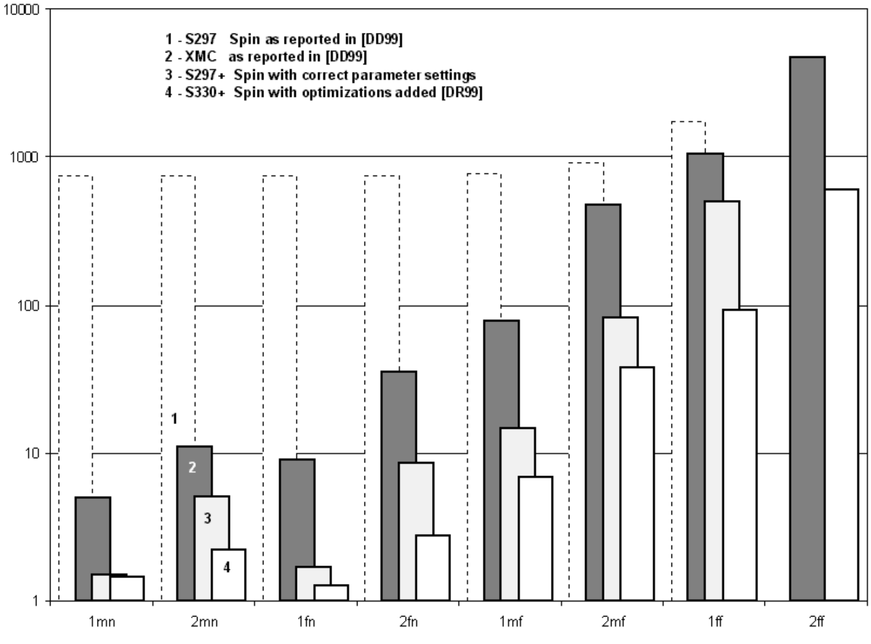


Fig. 2. Graphical View of Relative Performance of XMC, and Spin Versions 2.9.7 and 3.3.0.

The reference performance is the one given for XMC in [1], shown here in dark grey, and labeled with the number 2. Dotted, and labeled by 1, is the performance of Spin 2.9.7 as reported in [1]. In light grey, labeled 3, we give the corrected measurements performed here, for the same tests with Spin 2.9.7, as listed in Tables 1 and 2. Next to that, in white, and labeled 4, we give the performance on the same problem when Spin 3.3.0 is used.

The Promela source for the i-protocol as it is tested here and in [1] is available online from: http://www.cs.sunysb.edu/~lmc/papers/tacas99/proti.spin.1fn.
The revised Spin model for `2ff`, represented in Table 6, is available from the current author. Also available from the website shown above is the C source for both version 1.04 and version 1.06.1 of the GNU uucp i-protocol. The source is approx. 1500 lines of C. It should be feasible to produce a version on which we can directly perform Spin model checking without first manually constructing a Promela model. We have earlier applied such a method in the verification of a commercial telephone call processing system. The method we have developed is detailed in [8]. At the time of writing we are extending this method to tackle C implementations of general protocols, such as the one discussed here.

Acknowledgements

The author thanks Scott Smolka, Xiaoqun Du, Yifei Dong for their gracious patience in answering the many questions about details of the tests performed and the models used in [1].

References

[1] Y. Dong, X. Du, et al., Fighting livelock in the i-protocol: a comparative study of verification tools. *Proc. TACAS99*, Amsterdam, The Netherlands, March 1999.

[2] Y. Dong and C.R. Ramakrishnan. An optimizing compiler for efficient model checking. Unpublished manuscript, available from: http://www.cs.sunysb.edu/~lmc/papers/compiler/.

[3] G.J. Holzmann, D. Peled, An improvement in formal verification, *Proc. Conf. on Formal Description Techniques*, Proc. FORTE94, Berne, Switzerland, 1994.

[4] G.J. Holzmann, State Compression in Spin, *Proc. Third Spin Workshop*, April 1997, Twente University, The Netherlands.

[5] --, The model checker Spin, *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.

[6] -- and A. Puri, A Minimized Automaton Representation of Reachable States, *Software Tools for Technology Transfer*, Springer Verlag, Vol. 3, No. 1, 1999.

[7] --, Designing executable abstractions, *Proc. Workshop on Formal Methods in Software Practice*, Clearwater Beach, Fl., March 1998, ACM Press.

[8] -- and M.H. Smith, A practical method for the verification of event driven software. *Proc. ICSE99, Int. Conf. on Software Engineering*, Los Angeles, CA, May 1999, pp. 597-607.

[9] K. McMillan, *Symbolic model checking.* Kluwer Academic, 1993.

[10] Y.S. Ramakrishna, C.R. Ramakrishnan, et al., Efficient model checking using tabled resolution. *Proc. CAV97*, *LNCS* 1254, pp. 143-154, Springer Verlag.

[11] H. van der Schoot and H. Ural, An improvement on partial order model checking with ample sets. *Computer Science Technical Report*, TR-96-11, Univ. of Ottawa, Canada, Sept. 1996.