Refactoring SPIN for Safety

Robert Palmer and Ganesh Gopalakrishnan

UUCS-06-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

February 14, 2006

Abstract

We show how to refactor SPIN for safety model checking resulting in a compact model checker occupying less than 200 lines of code without appreciable loss of performance while reusing much of SPIN's front-end facilities. In addition to being far easier to understand and being eminently suitable as a basis for extensions by the researcher and developer community, the resulting model checker is also eminently suitable for distributed model checking—a project that is underway. We also show that employing graphical means of visualizing the asynchronous product graph can be very valuable in debugging a model checker—a facility we implemented and extensively employed in both understanding the original SPIN and discovering three subtle flaws in it.

1 Introduction

Since SPIN is one of the best-known model checkers for reactive systems around, it would be highly desirable to have its code easily understandable and modifiable. Unfortunately, that is, in our experience, not the case. We are *not* attributing this to sloppy coding. Rather, we are attributing this situation to SPIN including decades of explicit state model checking know-how, and including full LTL model checking with various notions of fairness, the ability to do breadth- or depth first search, multiple state compression and representation techniques, partial order reductions, and an overall highly tuned implementation. So what does a researcher who needs only a subset of these capabilities (but including decent performance) do in order to build on top of SPIN? In this paper, we propose a refactored version of SPIN that achieves this goal for safety model checking.

Our work in this direction started while we were looking for a large benchmark example to drive our own implementation of a Safety only SPIN model checker forward. For this purpose, we were kindly given a large Promela model by Dr. Rangarajan of Honeywell under NDA—that of the DEOS scheduler [7]. Our SPIN verifier exhibited bugs—at that time implemented reusing much of the "official SPIN's" code (version 4.0.6)—exhibited three nasty bugs which we could not easily track down. Unfortunately, the methods we had at our disposal—a reverse-engineered description of the main DFS loop of SPIN (550 lines of compiler spliced and cryptic C code) presented as a UML activity diagram occupied a wall of five meters even when printed at 10pt font size. Having no way to share our Promela code with Dr. Holzmann, the author of SPIN, we waited for another public-domain example that recreated these bugs. This took us a long time and finally we found such an example (the Gaussean example in this paper)—in the mean time, we could (i) conclude that the bugs are not in "our code" but are present in SPIN's code, (ii) develop a graphical visualization facility to visualize and understand the original SPIN's execution and compare it with the execution of our own version of SPIN, (iii) develop the method to refactor SPIN that is described here, and (iv) produce a very simple and easy-to-follow version of SPIN, the core of which occupies only 200 lines of code (refactored as we discuss), runs with acceptable performance, and avoids the bugs described. These bugs are in the safety only subset of the original SPIN. All our examples and the refactored SPIN code are at our website [9].

We must emphasize that we do not know yet whether the present refactoring method will be suitable for handling full LTL–we suspect that it will not be amenable to the nested depth first search technique currently implemented in SPIN[4]. However, there are many situations where safety model checking is preferred over liveness checking—e.g., distributed model checking, where handling liveness has not been shown to scale extensively.

1.1 An Early Peek at the Details

1.1.1 State Generation

We determined that organizing the primary state generation step of the model checker as a function $f : state \rightarrow state\ list$, where f takes a state and returns a list of states that can be reached in a single *atomic* step of the verifier is conducive to a modular implementation. We justify this statement by noting that the outer loop to drive f is a mere 10 lines of C. In addition, optimizations, such as partial order reduction [6] or symmetry reduction [5], that can be viewed as as a function that take a state and return a state can be mapped over the state list returned by f. Further, if a queue is used instead of a stack the verifier performs breadth first search.[1].

The refactored SPIN implementation does have a dark side. First, although we have not yet had a problem in this regard, it is possible for the states that are waiting to be expanded to consume additional memory. This is true of any model checker that does not perform a strict depth first search. Second, the overall runtime of the safety only SPIN model checker we developed is significantly more – usually a factor of four or five. The first problem can be overcome using magnetic disk. The second problem we hope to overcome using explicit parallelism. Both are left for future work.

1.1.2 Visualization

We discovered a public-domain graph visualization and navigation tool called Guess [1] whose capacity as well as usability far exceeds that of tools such as Dot [2]. Figure 1 shows a sample output visualizing the state graph of DEOS generated up to depth 200. The statistics for this graph are shown in the table of figure 3. For viewing smaller graphs, Dot suffices. The flaws we discovered in SPIN version 4.0.6 are described through Dot-generated figures presented in this paper. The additional book-keeping to produce these graphs is negligible.

[1]We should also mention that all of our experiments for this paper are without partial order reduction we mention it here as to recognize the simplicity of adding it on to the f based organization.

1.2 Roadmap

Section 2.1 gives a high level overview of how the SPIN-generated verifier works. Against this backdrop, Section 2.2 describes the basic underpinnings of the refactored SPIN we have developed. Section 3 describes techniques used for debugging in more detail. Section 4 discusses performance issues and compares the refactored code with the original SPIN. Section 5 discusses future work and concludes. In the following sections, we will refer to the SPIN generated verifier and the SPIN compiler simply as "SPIN."

2 Refactoring SPIN

2.1 SPIN Basics

To generate the asynchronous product of a multiprocess model SPIN uses a standard recursive style depth first search technique and maintains the invariant that the states on the search stack are the sequence of states from the initial state to the state that is currently being explored.

SPIN represents the transition relations of each process in a Promela model as an array of transition lists, one for each control point. To generate the asynchronous product at a given state each process that is created in the model is given a "turn" where the current transition list is looked up for a process type based on the PC value for that process. This is the case unless SPIN is currently executing within an atomic block except when a synchronous transition is attempting to execute. It then becomes of interest to know what the behaviors of the single step, synchronous step, and atomic step are.

A single step of any given process corresponds with a single application of a transition to a state. The *D_STEP* coalesces all of the transitions internal to that `D_STEP` into a single application of the transition relation. Synchronous or rendezvous communications are viewed as a single *atomic* action of two processes. This is performed by applying the transition relation for the sending process and then only allowing processes with an enabled matching receive to apply the transition relation in the second half of the communication. The intermediate state is not saved in the hash table and other processes are not allowed to execute at such a state. If there are more than one enabled receiving process for any given sending process then it is possible to have multiple successor states.

Atomic blocks are sequences of transitions that are viewed as atomic to other processes in the model. An atomic block may contain a nondeterministic choice or a loop. Intermediate states in the atomic sequence are not saved in the hash table however they are put on the search stack. Other processes are not scheduled at states internal to an atomic block except for two cases. First there may be a rendezvous communication within the atomic block. In this case control is transfered to the receiving process if that process is also in an atomic block. In the second case some statement within the atomic block may not be executable. In this case the atomic sequence ends and other processes are scheduled.

In the event that an atomic block contains nondeterminism a successor representing each branch of the search tree will be produced. To implement this (and nondeterminism in synchronous channels) SPIN leverages the DFS stack. In particular all states that have enabled transitions that have not been explored are represented in the DFS stack and enabled transition $n+1$ at a given state will not be explored until all successor states resulting from executing enabled transition n have been explored.

For a detailed discussion of the SPIN system beyond our presentation please see [3]. As all of the above description, and several extensions and additional optimizations, are implemented as a single pseudo-recursive loop, we found that trying to interface with this code and/or extending or sub-setting it can be next to impossible.

2.2 Refactored SPIN based on $f : state \rightarrow state\ list$

The description in section 2.1 gives rise to an interesting approach toward organizing SPIN's basic depth first search. We will follow this approach in the discussion of our implementation. In what follows, we view f as the "interface" that the user's code needs to know. An experimenter need not know how f is realized.

Implementing the interface required adding a data structure on which states could be captured. [2] A captured state is the state of the model and any other information needed to continue the model checking procedure from the represented state. In SPIN this includes the compression mask and the offsets within the state of various processes and channels. At the beginning of the model checking run we capture the initial state.

Single Steps

To execute a single step we restore the captured state, apply the SPIN provide transition

[2] In addition we provided our own hash table and stack.

relation, and capture the resultant state. The resultant state is returned as a successor. A `D_STEP` is handled in the same way. The synchronous communication is not as simple. Here we take the sending step into the rendezvous state. Every process that is not ready to receive on the corresponding synchronous channel is blocked by an internal mutex variable *boq*. Every process then gets a chance to receive. Any process that successfully receives causes a new state to be created. The new state is captured. We coalesce these new states into a list and return the list.

Atomic Steps

An atomic step may have multiple successor states. Thus our f function requires that all successors that are reachable through an atomic block be returned. In the case of a single non-atomic and non-synchronous statement there will be at most one successor state returned. There may be multiple states returned with a synchronous step, however all successors are at the same depth. With an atomic step multiple states at varying depths can be returned. The successor list of an atomic block does not have the same restrictions.

Finding the set of successors for an atomic step requires a secondary search. Our implementation uses a depth first search. We create a new stack and move only the process that currently has control until transitions are no longer within an atomic block (recall that control may be transfered via execution of a synchronous step). Executing the single step functions results in a new state or list of states. If the state was reached by a transition that is inside an atomic block it is pushed onto the secondary search stack. States that are encountered by executing transitions outside the atomic block are added to the list of atomic successors and are returned to the calling function. States that are internal to the atomic sequence are deleted.

Finding Deadlocks

Assertion checking is performed when the transition relation is applied. To find a deadlock one must take into account the possible presence of the *timeout* construct. If every process is allowed to generate all atomic successors for a given state and the resultant list is empty then a deadlock state could be present. The *timeout* construct acts as a system wide else and becomes enabled in this situation. Every process is then again allowed to attempt to generate a list of atomic successors. If no such list is available and the current state of each process is not a valid end state a deadlock exists and an error trail may be generated.

The remaining pieces of a safety only model checker are trivial once this functionality is implemented. To ease the implementation we use a separate hash table and stack but could have used those existing in SPIN.

3 Visualization

We applied our visualization techniques to every model where we have compared our f based model checker with SPIN. Of these, two examples were most enlightening. As mentioned in section 1 our f based model checker generates states that are not actually reachable in the DEOS model. In addition we found a similar problem while verifying a Gaussian elimination model from [8]. To help determine the cause of these additional states we developed a technique to examine the asynchronous product graph generated by SPIN and the f based model checker. This graph could then be visually inspected for anomalies and used to guide debugging efforts. We used two tools to render the graphs, Dot [2] and Guess [1].

We modified SPIN to generate atomic transition sequences and write them to a file in the requested format. Each sequence of transitions in the file is the sequence of transitions used to generate those states that are currently on the main search stack when a state is looked up in the hash table. In the Dot graphs a dashed edge represents a revisit and a solid edge represents a actual atomic step of SPIN. Each line of text next to the edge is a single step from the Promela source. The number in the vertex is the memory address of the state the vertex represents. Thus every edge represents an atomic step of SPIN. Our f based model checker records every single step taken between visits to the hash table. When a state is looked up in the hash table the list of single steps that was used to reach the current state is written to disk. Again dashed lines represent revisiting transitions.

The Guess visualization is slightly different. The viewer allows for users to zoom in and query nodes and edges for information. Large graphs such as the one in Figure 1 can be manipulated fairly easily. Here revisiting transitions are not displayed.

3.1 Error Trails and Guess

The primary motivation for this work has been the task of exhaustively verifying the DEOS scheduler model. Although this model is too large to represent in the hash table of a single machine our implementation seemed to behave differently than SPIN when performing a depth bounded verification. The problem was that our refactored SPIN would find additional states after a depth of 139 and generating about 4k states. Although not a show stopper in light of the errors discussed in section 3.2, our refactored version also found a false deadlock at a depth of 190 after generating about 12k states. Dot was able to create a postscript file with the image of the product graph but our 32 Itanium processor shared memory SGI machine with 16GB of memory was unable to display it.

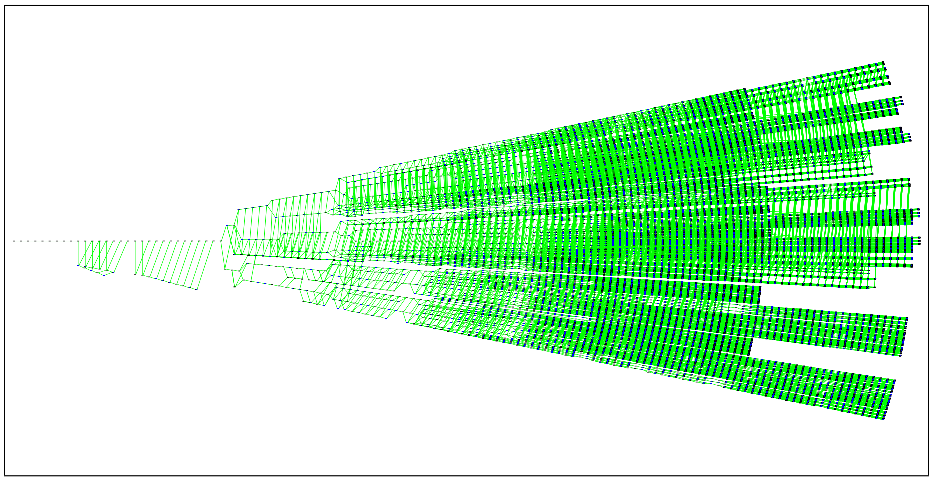

Figure 1: The asynchronous product of the DEOS model to a depth of 200 generated by SPIN and rendered with Guess.

To display this much larger graph, we investigated using the Guess graph viewer [1]. Figure 1 shows the product graph generated by SPIN. The image from refactored SPIN is similar. Both images are rendered in about 180 seconds on a laptop (in Java!).

Although the images are too dense to determine anomalies by manual inspection they provided a clue to the ultimate problem we were seeing. We concluded that there were additional states generated at multiple levels within refactored SPIN that should not have been created.

The error trail facility we have implemented creates an error trail when specifically requested. Each state retains two arrays that represent the process and transition id of the move made to generate the state. The earliest extra state was at a depth of 139. Our error trail serendipitously had a transition that could not be executed at a depth of 139. Closer examination of our SPIN provided transition relation showed that there were some `D_STEP` transitions that were not being properly guarded against execution while a synchronous transition was being executed. Here there was a check on the `boq` variable that was combined with other prepositions. When the combined preposition evaluated to false the execution passed the guard into the `D_STEP`.

3.2 Dot

Figure 2 shows part of the asynchronous product graph for the Gaussian Elimination model from [8] with two processes. Upon a close examination of the graph we see that SPIN has

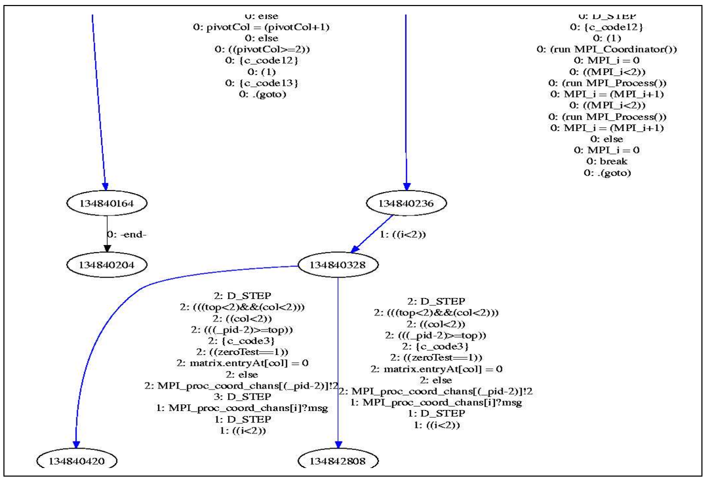

Figure 2: Part of the asynchronous product generated by SPIN and rendered with Dot for the Gaussian Elimination model.

two erroneous behaviors. First SPIN is not executing all enabled processes after two atomic steps at the vertex labeled 134840236 [3]. The sequence of steps shown by the edge into this vertex clearly shows three new processes being run.

After three atomic steps there is a rendezvous communication where another process has executed in the middle shown on the edge from 134840328 to 134840420 (this would be acceptable as a concurrently enabled process should be able to execute except the implementation of SPIN tries to forbid this). The cause is a missing `IfNotBlocked` macro on a `D_STEP` transition that guards against executing in the middle of a rendezvous communication. Figure 4 shows the corresponding part of the graph generated by our f based model checker [4].

4 Performance of the Refactored SPIN

Figure 3 shows the performance of refactored SPIN as compared to unmodified SPIN. The models shown are as follows:

1. The DEOS model to a depth of 200 steps discussed in [7].
2. The Gaussian Elimination model from [8].

[3]The label on the vertices is the actual memory location for that state when represented in the hash table.
[4]Here the vertex labels are the index in the open addressed hash table.

Model	Implementation	States	Time	Permanent	Peak
DEOS	SPIN	33,315	0.570	20.928	20.928
DEOS	Refactored	33,315	4.030	19.638	20.117
Gauss	SPIN	184	0.010	1.573	1.573
Gauss	Refactored	228	0.030	1.067	1.170

Figure 3: Comparison of performance between implementations.

Each model was compiled without optimization as the DEOS model when compiled −03 causes FCC to segfault.

The fields of the table in Figure 3 are as follows. Implementation indicates whether we are running SPIN or the refactored version of SPIN. States is the number of unique states generated. Time is the user time reported by the Unix time command. Permanent is the amount of memory consumed as reported by the model checker that is never released–this is memory used to store the states generated by the model checker. Peak is the largest amount of memory that is held at a given time by the model checker.

Each of these models shows a significant slow down in the f based model checker. This overhead is primarily due to the necessity of capturing states. This is to be expected as we introduce multiple memory copies in order to capture each state. As a trade off the ability to better understand and reason about the implementation is of significant value and we believe it justifies the additional time.

It is possible that some significant additional resources could be necessary to store those states waiting for expansion (although the results shown in figure 3 do not demonstrate this). The use of a strict depth first search in SPIN reduces the number of states waiting to be expanded to the minimum. We have not yet encountered any difficulty due to this possibility, however, should some difficulty arise it is clearly possible for states waiting to be expanded to be pushed out to magnetic disk.

5 Conclusions

We have developed an extensible base model checker within the framework of the widely used and modified SPIN model checker. Our refactored SPIN is less than 200 lines of modularized C code that supports the view that a model checker is primarily a function that takes a state and returns a list of successor states. To use the refactored SPIN in unmodified SPIN we require the introduction of exactly six lines of code.

We have developed a graphical visualization technique for use in understanding what SPIN and any extension of the SPIN system is really doing. In the process we have discovered three subtle defects in the SPIN verifier. These defects could not have been discovered so easily without the graphical representation of the asynchronous product graph.

Our efforts have uncovered three subtle defects in the SPIN system. The combination of trying to implement a model checker for Promela semantics and fully understand the behavior of the SPIN model checker using a graphical representation helped in making these discoveries.

We have compared performance with that of unmodified SPIN and find the results slower than unmodified SPIN but acceptable for our application. We plan to expose the interfaces for SPIN's state compression and alternative state set representations. Current plans include the application of our f based model checker as a basis for scalable safety only parallel model checking.

References

[1] E. Adar. Guess: The graph exploration system. http://www.hpl.hp.com/ research/idl/projects/graphs.

[2] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing 2001*, pages 483–484, 2001.

[3] G. J. Holzmann. *The SPIN Model Checker Primer and Reference Manual*. Addison-Wesley, 2004.

[4] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

[5] C. N. Ip and D. L. Dill. Verifying systems with repplicated components in Murφ. In *CAV '96:Computer Aided Verification*, pages 147–158, 1996.

[6] R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, May 2002.

[7] J. Penix, W. Visser, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger. Verifying time partitioning in the DEOS scheduling kernel. *Formal Methods in System Design*, 2004.

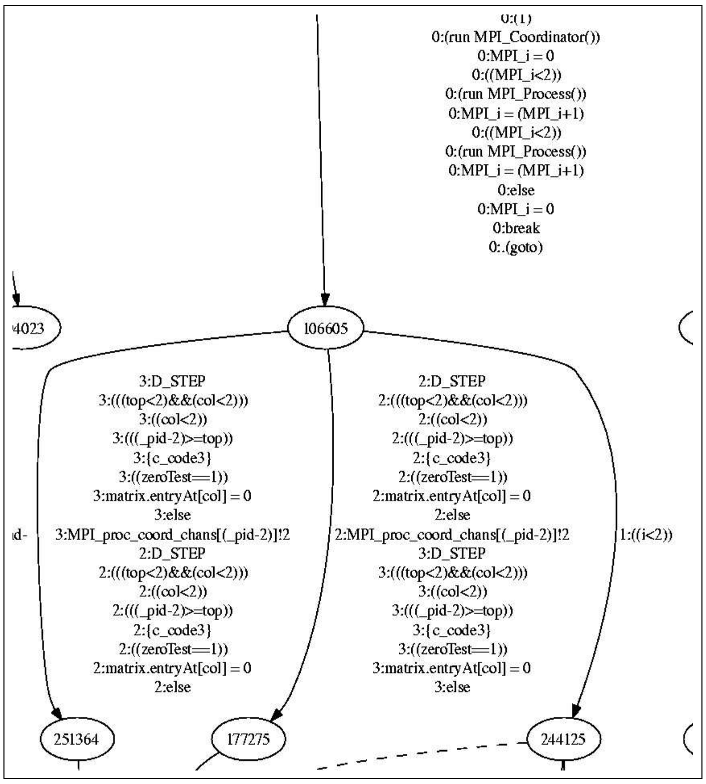

Figure 4: Part of the asynchronous product generated by our f based model checker and rendered with Dot for the Gaussian Elimination model.

[8] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. Technical Report UM-CS-2005-15, Department of Computer Science, University of Massachusetts, 2005.

[9] http://www.cs.utah.edu/formal_verification.

A Additional Graphs

Figure 4 shows the portion of the asynchronous product graph that corresponds to the portion shown in figure 2. The step into a `D_STEP` in the middle of a synchronous communication is present however the number of enabled processes at state 106605 [5] is correct (whereas in Figure 2 at state 134840236 only one process is enabled).

[5]The label in the vertex represents the hash table index where this state is stored.

B Commands used with SPIN

For generating the models we turned off all optimizations. Our aim here is to fully understand what the SPIN verifier is doing at the most fundamental level. The command we used was:

```
> spin -o1 -o2 -o3 -a model
```

To compile our models again we disable the optimizations. The command used for SPIN was:

```
> gcc -g -o pan pan.c -DNOREDUCE
```

For our f based model checker we splice in a separate file using the C preprocessor. This makes the intrusion into the body of SPIN exactly six (6) lines of code. The command used was:

```
> gcc -g -o pan pan.c -DNOREDUCE -DEXTENSION -DINCREMENTAL_HASH
```

As mentioned above we supplied a separate hash table and stack. The hashing algorithm consumes a comparable amount of time as compared to the hashing algorithm in SPIN. Thus we have the additional flag `-DINCREMENTAL_HASH`.