Accepted for publication July 2010 in Formal Aspects of Computing

Model Checking with Bounded Context Switching

Gerard J. Holzmann[1] and Mihai Florian[2]

[1]Jet Propulsion Laboratory,
California Institute of Technology, Pasadena, CA, USA
[2]Computer Science Department,
California Institute of Technology, Pasadena, CA, USA

Abstract. We discuss the implementation of a bounded context switching algorithm in the Spin model checker. The algorithm allows us to find counter-examples that are often simpler to understand, and that may be more likely to occur in practice. We discuss extensions of the algorithm that allow us to use this new algorithm in combination with most other search modes supported in Spin, including partial order reduction and bitstate hashing. We show that, other than often assumed, the enforcement of a bounded context switching discipline does not decrease but increases the complexity of the model checking procedure. We discuss the performance of the algorithm on a range of applications.

Keywords: Logic model checking, depth-first search, bounded context-switching, partial order reduction, bitstate hashing, software verification.

1. Introduction

It takes only one counter-example to a correctness property to disprove its validity for a given system. Model checkers can excel in finding such counter-examples in large search spaces, but the first counter-example they locate is not necessarily also the simplest. When depth-first search is used, counter-examples are often longer than necessary, and they can correspond to system executions that are more complex than necessary. Short error sequences can be found with a breadth-first discipline, but even the shortest possible counter-example is not necessarily also the simplest to understand, e.g., if it contains more context switches than strictly necessary to reproduce the error. A breadth-first search, furthermore, tends to increase the memory requirements for a search and cannot easily be used for the verification of anything other than basic safety properties.

In [QW04] a different type of search method was proposed, based on bounding the number of context switches (process preemptions) that can appear in an execution. The idea was first applied to static source code analysis, but later extended also for use in software testing, and in model checking, e.g., [MQ07b].

Correspondence and offprint requests to: Gerard J. Holzmann, Jet Propulsion Laboratory, 4800 Oak Grove Drive M/S 301-230, Pasadena, CA 91109, USA. e-mail: gerard@spinroot.com

```
Search(s)
  if s violates P
       report_error
   S = S ∪ s   // add s to statespace S
   for all successors s' of s
       if s' not in S
           Search(s')
```

Fig. 1. Standard depth-first search.

```
Search(s)
  if s violates P
       report_error
  S = S ∪ s   // add s to statespace S
  for all successors s' of s
       if s' not in S /\ CS(s') <= N
           Search(s')
```

Fig. 2. Bounded context switching, cf. [MQ07b].

As noted in [MQ07b], even relatively low bounds on the number of context switches suffice for a model checker to visit all the reachable states of a model at least once. But as we will show in this paper, with context bounding for the verification of even simple safety properties it no longer suffices to visit each reachable state just once. In the worst case, the number of required visits to each state can be as high as N*P, where P is the number of threads or processes in the system being analyzed and N is the bound that is imposed on the number of context switches. This is caused by the fact that a context-bounded search evaluates execution paths, and not just states. In a context-bounded search it can make a difference *how* a state is reached.

In this paper we report on the implementation of a bounded context switching algorithm in the Spin model checker [H04]. The algorithm can be invoked by compiling the model checking source (pan.c) with a new directive called -DBCS. The user can define a context switching bound with a new runtime parameter -Ln, where n is the bound. The default bound, when no explicit bound is defined with this parameter, is zero.

In Section 2 we discuss the basic implementation of this algorithm without the use of partial order reduction. In Section 3 we discuss several extensions of the basic algorithm. Section 3.1 discusses an extension to deal with cyclic executions; Section 3.2 describes an adaptation to support Spin's bitstate hashing method, and Section 3.3 discusses the integration of the new algorithm with Spin's partial order reduction method. Section 4 includes performance measurements. Section 5 discusses related work, and Section 6 concludes the paper.

2. The basic algorithm

The standard depth-first search, illustrated in Figure 1, explores all successor states of a given state s, and checks whether they were visited before. Every successor state that was not visited before is then also explored. This procedure repeats, recursively, until all reachable states are found. In Figure 1 we use S to denote the statespace: the set of all visited states, which is initially empty. The search starts with the call Search(s_0), where s_0 is the initial system state. If the set of reachable states is finite, the search will terminate. The algorithm in Figure 1 suffices for the verification of safety properties for finite state systems. The extension for the verification of liveness properties is well understood and not elaborated here [HPY96].

In bounded context switching, every context switch is counted towards a user-defined maximum. A high-level description of this algorithm is given in Figure 2. In this version of the depth-first search, successor state *s'* of state *s* must satisfy one additional condition, other than not having been visited before: the number of context switches *CS(s')* at *s'* is at most equal to bound *N*.

Although Figure 2 shows the essence of the algorithm, it leaves out important details. One such detail is that forced context switches (so-called non-preemptive context switches) are not counted towards the bound [MQ07b]. A context switch is considered forced if the currently executing process blocks and execution would come to a halt if no context switch was performed.

The algorithm in Figure 2 also has a more important flaw: it cannot guarantee a critical property:

- *P1*: If the algorithm in Figure 2 terminates with bound N without reporting an error, then no violations exist for any execution with N or fewer context switches.

The property fails for a simple reason. When a state is revisited with a number of context switches that is lower than the number seen on the earlier visits, then execution does *not* continue. It is of course possible that a counter-example can be constructed within bound N if the search were continued in this case.

The situation is similar to that of a depth-bounded search, as implemented in Spin with the -DREACH search option [H04]. Unless we store the depth at which a state is reached in the statespace S, and continue the search whenever a state is revisited via a shorter path than before, the search risks being incomplete. Only the *lowest* depth at which a state is reached is stored in the statespace jointly with each state. Clearly, this means that states can be revisited many times (up to the search bound imposed).

For a context-bounded search we must similarly record the lowest number of context switches seen when reaching each state, and again we will have to continue the search if a state is reached with a lower number than recorded before. Also here, then, both the space and especially the time requirements for the search can increase. Even with this increase, though, the complexity of the statefull search remains significantly better than a stateless search, with or without context bounding. In a stateless search, as for instance used in standard testing or random walks, the number of times that states can be revisited could become quadratic in the number of reachable states. (Because no states are recorded outside the search stack, the search could in the worst case end up revisiting all reachable states for each state visited.)

The bounded context switching algorithm that we implemented has the following properties.

1. At the initial system state, all successor states are explored the same as in a standard depth-first search.
2. At all other states, the model checker first tries to continue the same process that executed in the previous execution step, thus avoiding a context switch. We distinguish two main cases.
   (a) The selected process is not executable and blocks. The model checker will now expand its search by accepting successor states from all processes. The resulting context switch is considered forced (non-preemptive) and is not counted towards the bound.
   (b) The selected process does not block and the sub-tree of the successor state is explored. A preemptive context switch can be performed if the user-defined bound on the number of unforced context switches has not yet been reached. If other processes can contribute successor states, the number of context switches performed is now incremented for each such successor state generated in this way. We now distinguish four separate cases.
      i The successor state that is reached was not visited before (i.e., it does not appear in S). In this case, the successors of the new state are explored. We store the state jointly with the number of context switches that was needed to reach it.
      ii The successor state was reached before, with a number of context switches N that is smaller than the number M used in the current execution path, i.e. with $N < M$. In this case, the new path is not better than at least one previous path, and therefore we need not explore it again.
      iii The successor state was reached before, but with $N > M$. The current path is better, so we must explore it again. We store the new lower number M with the state, replacing the previous value N.
      iv The successor state was reached before with $N = M$. We distinguish two cases.
         A We reached the successor state via a transition that belongs to the same process as the transition that was executed on the earlier visit with the lowest number of context switches recorded so far. In this case we cannot discover any new paths and we need not explore the state again.
         B We reached the successor state via a transition that belongs to a different process than for the earlier visit. In this case we may discover new paths. Note that it would take one extra context switch for the earlier path to switch to the current process, which means that we have reached this point with one fewer required context switches.

```
1 Search (s,n,p)
2
3    Let A      be the set of transitions enabled in s
4    Let A'     be the subset of A of transitions with pid p
5    Let A''    be A - A'
6    Let a(s)   be the successor state of s after executing transition a
7    Let pid(a) be the pid associated with transition a
8
9    for the initial state, let A' be empty
10
11   if s violates P
12      report_error
13
14   S = S ∪ {(s,n,p)} // add extended state s to statespace S
15
16   for all a in A' do
17   {    if CONSTRAINT(a(s),n,pid(a))
18        {    Search(a(s),n,pid(a))
19   }    }
20
21   if A' ≡ ∅
22   {    u = 0   // do not count context switch
23   } else
24   {    u = 1   // count context switch
25   }
26
27   for all a in A'' do
28   {    if CONSTRAINT(a(s),n+u,pid(a))
29        {    Search (a(s),n+u,pid(a))
30   }    }
```

Fig. 3. Spin implementation of bounded context switching.

We can now more clearly see the reason why a bounded context switching algorithm can require us to visit the same state multiple times, if we reach the state via an execution path with a smaller number of context switches than on all previous visits. In the worst case, if our context bound is N and the number of executing processes is P, we may visit a state up to $N * P$ times. Note that with a BCS algorithm it is not only important which states we reach but also how we reach them. It is, of course, not necessary to store any state more than once, but a small amount of additional information is required. Minimally this additional information includes not only the lowest number of context switches that was needed to reach the state but also the id's of all processes that contributed the last step leading to the state (see iv.A and iv.B above).

The new algorithm for a model checking procedure based on bounded context switching, as implemented in Spin is illustrated in Figure 3. The definition of CONSTRAINT(s,b,id), used on lines 16 and 26 in Figure 3, is as follows:

$$(b \leq BOUND) \wedge (\forall x \forall y (s, x, y) \in S \Rightarrow (x > b \vee (x = b \wedge y \neq id)))$$

In the actual implementation we can reduce the memory requirements by not storing each triple of a state, a context bound, and a vector of pid numbers separately, but instead storing the state (the largest part of the data) just once in combination with the minimum context bound seen so far and the corresponding vector of pid numbers.

To see why the algorithm in Figure 3 satisfies property *P1*, consider an execution with N unforced context switches that leads to an error. The first step of the error sequence is necessarily explored by Figure 3, because it considers all process actions in this step. The algorithm in Figure 3 can only fail to explore the full error path if its search truncates on a previously visited state. For the successor state to be ignored, one of the two following cases must apply:

2b-ii: the number of context switches on the previous visit was smaller than the number on the current path, or

2b-iv-1: the number of context switches is the same in the earlier execution as in the new one, and the same process currently executing has produced at least one of the earlier visits.

In both cases, if the error path was not found on the earlier visit, it cannot be found on the new path either, since the new path either has a tighter or an identical constraint. A proof for a property equivalent to *P1* is given in Appendix A.

3. Extensions

In this section we discuss some extensions of the basic algorithm that we have made in the version that was implemented in the Spin model checker (version 5.2.3 and later). For simplicity, these extensions are not shown in the basic version of the algorithm discussed so far.

3.1. Dealing with cycles

Consider the case where A' (line 4 in Figure 3) is non-empty, but all successor states explored (line 16) match previously visited states that are on the depth-first search stack. i.e., the recursive call to *Search* on line 18 is never made. In the algorithm as shown in Figure 3, a context switch to a different process would count towards the bound (line 24).

In this case, though, the execution scenario explored, with the process selected under the rules for bounded context switching caught in an infinite loop, would be at odds with a basic fairness assumption that can be made about process scheduling.

In our implementation of bounded context switching we therefore do not count a context switch in this state (not shown in Figure 3). The same change is also needed for the integration with partial order reduction rules, to avoid cyclic deferral (known as the *ignoring problem* in partial order reduction theory), as discussed in Section 3.3.

3.2. Integration with bitstate hashing

Bitstate hashing was introduced in 1987 as a method to handle the large state space sizes that are often encountered in applications of logic model checking, using very high compression ratios on state storage [H87, H98]. The method trades a small probability of incompleteness of the search for a significant increase in problem coverage for large problem sizes. The method is used as a search option in most of the currently used logic model checking tools.

In brief, the bitstate hashing algorithm, comparable to a Bloom filter [B70], uses the fact that we can achieve high compression rates by treating the hash key as a bit-address in a large memory arena. If a state is reached, the bit at the address pointed to by the hash-key (taken modulo the bit-address range of the memory arena) is set to one. By checking for the state of the bit, the algorithm checks for previously visited states. If the number of bits is large compared to the number of states reached, the probability of a hash-collision will be small. In cases where hash-collisions do occur, the search risks becoming incomplete. The model checker retains its ability, though, to find counter-examples in the part of the state space that is searched. The probability of incompleteness depends on the ratio of the number of available bit-positions and the number of reachable states, and can be computed [H98]. The Spin model checker uses multiple independent hash-functions (and multiple bit-positions per state) to reduce the effect of hash collisions.

In bitstate mode, the smallest number of context switches seen so far along an execution path and the vector of pid numbers cannot be stored separately from the state, since only a minimal hash fingerprint is stored in the bitstate hash-array. To support the new search mode in bitstate explorations, it therefore needs to be modified.

In bitstate mode, when a new state is added to the statespace S, we store separate copies for both the current and all higher numbers of context switches, up to the bound. We will refer to this as the *replicated storage* method. Using this method, we can avoid unnecessary revisits of the state with a higher number of context switches than before: they will produce matches of the bitstate hashes. The vector of process id

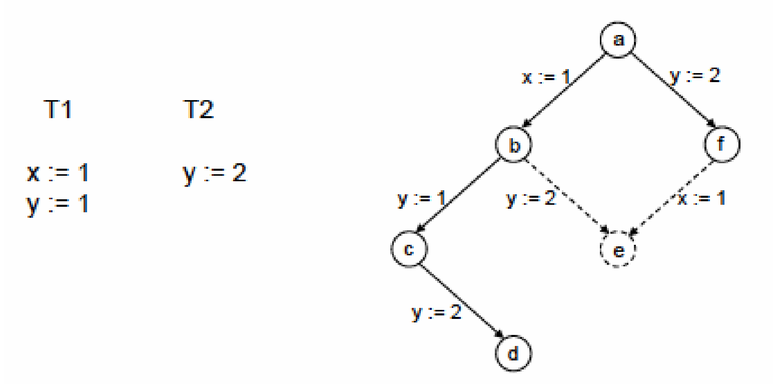

Fig. 4. Interaction of partial order reduction and bounded context switching, from [MQ07a].

numbers is not stored in this mode to avoid increasing the memory requirements. The latter can increase the probability of incompleteness, but tradeoffs like this are a standard part of bitstate searches (i.e., a bitstate search trades coverage for precision). The method to store multiple versions of a state, rather than just the one explicitly reached, is reminiscent of a partial order reduction method described in [MQ07a], where a partial-order happens-before graph can record multiple variants of a state, including those not yet encountered.

3.3. Integration with partial order reduction

Partial order reduction is a technique used in state space exploration that uses the fact that many executions sequences (paths in the reachability graph) are functionally equivalent. Not all interleavings of events in an execution, for instance, can produce different results. With the help of a partial order reduction method, the size of the reachability graph that needs to be explored can be reduced up to exponentially, while retaining the ability to prove or disprove all safety and liveness properties [P94, HP94, H04].

The Spin algorithm for partial order reduction is based on a computation, at compile time, of an independence relation among transitions. A *safe* transition in this context is provably independent of all other transitions in the system. This information is used at runtime (i.e., during the model checking process itself) to suppress transition interleavings that otherwise would be required to preserve completeness [HP94].

The integration of bounded context switching with partial order reduction was studied before, e.g. in [MQ07a]. The earlier method, though, was restricted to the verification of terminating executions only: the search algorithm itself does not terminate if this condition is not satisfied. In [MQ07a] the fact that all executions terminate is used by storing not state vectors but a *happens-before* relation for all states. This happens-before relation is the partial order of all actions (transitions) seen on all execution paths from initial state leading to the state being considered. This relation would grow without bound if non-terminating or cyclic paths are encountered. The limitation to terminating executions also means that the algorithm from [MQ07a] is limited to safety properties.

Because Spin handles both safety and liveness properties for both terminating and non-terminating executions, the integration of bounded context switching with partial order reduction requires a different solution.

A good example of a structure that would not be explored correctly with a naive combination of partial order reduction with bounded context switching is given in [MQ07a]. The example is reproduced here in Figure 4. In the example, variable x is local and variable y is global. The first assignment in thread $T1$ is a globally safe action, and would be selected by the partial order reduction algorithm in the initial system state (node a in the graph on the right). Once thread $T1$ is chosen, though, the BCS algorithm must also choose all further actions from thread $T1$, until it blocks. Only then will there be a forced context switch to thread $T2$, and the final assignment to y is performed. The alternative execution, with the same number of forced and unforced context switches, is the one where thread $T2$ executes first, terminates, and then thread

$T1$ executes both its statements. The complete sequence of transitions from node a to f to e and ending in a new node g is not shown in Figure 4.

The end-state (i.e., the final value of variable y) is different in these two executions, and therefore this version of the algorithm fails to explore all distinct sequences with the same context switching bound and would remain incomplete.

If we directly combine the BCS algorithm with Spin's partial order reduction rules, the same scenario would be possible. We therefore extend the algorithm as follows.

- First, consider a point in the execution where a partial order transition is available. Ignoring it would continue the BCS search unmodified and preserve all its properties, but it would not take advantage of partial order reduction.
- Next, consider the case where a partial order transition from a given process P is selected, and all transitions from other processes are ignored. If the execution of this transition does not require a context switch (i.e., process P was selected also in the previous step), then the extended algorithm is indistinguishable from the original and it will have the same properties. If, however, a context switch is required to make the partial order selection, we should count it as such. This means that the selection should not be allowed if the maximum number of context switches was already reached in the current execution path. This alone, though, does not suffice.
- The context switch that is performed in this case is a restricted one, since not all executable processes can contribute transitions. By partial order reduction theory we know that executing the partial order transition itself is safe and as far as the correctness properties of the system are concerned is invisible to all other processes. If after the partial order transition we switch back to the process that was executing under BCS rules before, the completeness of the search would again be preserved. We can allow for that additional context switch by giving a *"credit"* to the number of context switches that may be made in the remainder of the execution. The credit to be issued must be for *two* additional context switches. A first credit is needed to switch the current execution back to the originally executing process, after the (sequence of) partial order reduction transition(s) completes, and a second credit is needed to correct for the fact that the disappearance of the safe transition from a later point in the execution (where the BCS algorithm would normally have executed it) to the earlier point may also require an additional unforced context switch.
  - Consider the case where the execution that would be explored by the regular BCS algorithm consists of the transition sequence p_n,q_s,q_n, where p_n is a non-partial order transition from process p, q_s is a partial order transition and q_n a non-partial order transition from a different process q. Assume further that the last transition executed was a transition from process p.
  - The partial order reduction rules will select q_s to be executed before p_n at this point, where the BCS algorithm would have selected p_n. That is, using partial order reduction rules will change the transition sequence to: q_s,p_n,q_n. The original sequence p_n,q_s,q_n requires just one context switch (from p to q in the second step). The partial order sequence, however, requires two additional context switches, for a total of three: one from process p to q in the first step, another from q back to p after the first step, and a third from p to q after the second step, before the original execution context is restored. It is easy to see that the cost of the out-of-sequence execution of a partial order transition cannot be larger than two additional context switches.

  Issuing the two credits for a partial order transition is conservative since it allows the search engine not only to switch back to the original BCS processes that would have been selected without the out of order execution of the partial order transition, but it also allows other processes to be selected.
- The credit mechanism can be implemented by not counting context switches for partial order transitions before or after they are selected. The restriction that must be maintained, though, is that a partial order transition can only be considered in states where the context bound has not yet been reached (i.e., it must be possible to do a context switch, even if it is not counted).
- We also have to consider the issue of cyclic deferral (briefly also discussion in Section 3.1). If a partial order transition is selected and it leads to a successor state on the depth-first search stack, this can create an infinite deferral of non-partial order transitions that are enabled throughout the cycle. In partial order reduction theory this is a well-known issue called *the ignoring problem.* A search performed with partial order reduction but without bounded context switching takes this into account by forcing a full expansion

```
Search (s,n,p,safe)

  Let A      be the set of transitions enabled in s
  Let A'     be the subset of A of transitions with pid p
  Let A''    be A - A'
  Let A*     be a safe subset of A (the ample set), or A if no safe subset exists
  Let a(s)   be the successor state of s after executing transition a
  Let pid(a) be the pid associated with transition a

  for the initial state, let A' be empty and A* be A

  if s violates P
      report_error

  S = S ∪ {(s,n,p,safe)} // add extended state s to statespace S
  push(Q,s)              // push s onto stack Q

  if A* ≠ A ∧ ∃a, a∈A*: a(s) ∈  Q
  {     cycle = true
        A* =  ∅
  } else
  {     cycle = false
  }
  if A* ≠ A ∧ n < BOUND
  {    for all a in A* do
       {    if CONSTRAINT(a(s),n,pid(a),true)
            {    Search(a(s),n,pid(a),true)
       }    }
  } else
  {   for all a in A' do
      {    if CONSTRAINT(a(s),n,pid(a),false)
           {    Search(a(s),n,pid(a),false)
      }    }

      if A' ≡ ∅ ∨ safe ≡ true ∨ cycle ≡ true
      {   u = 0   // do not count context switch
      } else
      {   u = 1   // count context switch
      }

      for all a in A'' do
      {   if CONSTRAINT(a(s),n+u,pid(a),false)
          {    Search (a(s),n+u,pid(a),false)
  }   }   }
  pop(Q) // pop s from stack Q
```

Fig. 5. Combination of bounded context switching with partial order reduction.

of all successor states when a state is encountered that has no successors outside the depth-first search stack. We use the same rule in our implementation of the bounded context switching algorithm, and we treat it as a forced context switch, similar to the case where an executing thread blocks (i.e., the switch is not counted against the context bound).

The extended version of the algorithm is shown in Figure 5. This version of the algorithm can be shown to preserve all the properties of the simpler context bounded search algorithm, shown in Figure 3. A proof for this property can be found in Appendix B.

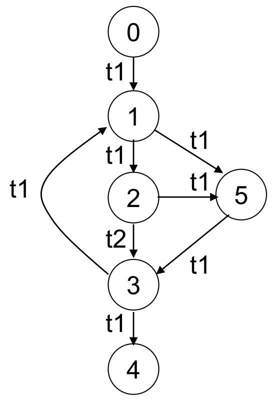

Fig. 6. Test case for bounded context switching algorithms.

4. Measurements

We begin by considering the test case shown in Figure 6. Assume that the single error state in this example is the final state numbered 4. The standard depth-first search will explore the state sequence: 0,1,2,3,4. There are two execution sequences that can be performed with no context switches: 0,1,5,3,4 and 0,1,2,5,3,4. The algorithm from Figure 2 cannot find either of these sequences. The algorithm truncates the search when it reaches the context bound at state 3 via the initial sequence 0,1,2,3, and then backtracks to state 2. From state 2, state 5 is explored, which then truncates when it reaches the previously visited state 3.

Our new algorithm from Figure 3 will find the longer sequence 0,1,2,5,3,4, because when it revisits state 3 it does so with a lower context bound than before. It will, however, not find the shorter equivalent 0,1,5,3,4. If we disable the context bound in CONSTRAINT on lines 16 and 26 of Figure 3, the algorithm will find both executions, but performs more work than needed, by exploring also all sequences that exceed the context bound. We can, however, find the shorter execution with Spin version 5.2.3 and later if we combine the context bounded search with a depth-limited search (using compilation directive -DREACH and runtime flag -i). We will return to this in Section 4.1.

Table 1 records the results of more detailed measurements with the Spin version 5.2.3 implementation of the BCS Algorithm, both with and without the use of partial order reduction. The models selected for this part of the study are of increasing complexity.

1. *nbq* is a model of a faulty algorithm for non-blocking queue operations,
2. *snoopy* is a model of a snooping cache algorithm, from the standard Spin distribution,
3. *dtp* is a model of a data transfer protocol,
4. *worst-case* is included in Appendix C and discussed model in more detail in Section 4.2,
5. *pftp* is a model of a file transfer protocol, also from the Spin distribution,
6. *cambridge* is a model of the Cambridge ring protocol,
7. *tpc* is a model of a simple call processing system,
8. *DEOS* is a model of an embedded real-time operating system, and
9. *mrf* is a model of the data management sub-system of a spacecraft.

Figure 7 includes the measurement results of four of the applications listed Table 1 over a slightly longer range of context switching bounds, to illustrate the effect of varying bounds visually. Each data point shows how many unique states are visited in the bounded searches, not counting repeat visits.

Consistent with observations first made in [MQ07b], only relatively small values for the context bound tend to be of interest. Relaxing the bound quickly allows the search to cover all reachable system states, although generally the *depth* of the search tree is smaller with bounded context switching than with a standard exhaustive depth-first search. For a bound of eight or more context switches, the search for each of the models shown in Figure 7 reaches all reachable states. For the DEOS model, more context switches

Table 1. Full state space sizes of nine sample verification models.

Model	States	BCS+L0	BCS+L1	BCS+L2	BCS+L4	BCS+L8	BCS+L16	PO
nbq	616	56	187	504	616	616	616	–
	415	98	312	408	415	415	415	+
snoopy	61,619	24,118	53,739	61,209	61,619	61,619	61,619	–
	9,343	26,420	9,466	9,466	9,466	9,466	9,466	+
dtp	394,182	512	8,547	41,628	210,152	392,892	394,182	–
	55,647	512	64,600	55,856	55,647	55,647	55,647	+
worst-case	88,573	7,167	22,784	45,567	80,511	88,571	88,573	–
	88,753	7,167	18,687	38,144	75,008	88,552	88,573	+
pftp	439,894	143,405	376,557	438,495	439,894	439,894	439,894	–
	160,033	79,365	155,691	159,264	159,594	159,588	159,700	+
cambridge	532,532	190,503	506,892	530,411	532,532	532,532	532,532	–
	286,223	216,010	341,816	342,229	336,698	228,134	288,134	+
tpc	32,898,808	12,156,530	26,032,145	31,464,747	32,871,059	32,898,808	32,898,808	–
	4,717,906	12,946,643	15,221,436	15,259,211	15,076,229	13,611,964	13,600,187	+
DEOS	111,587,460	54	719,626	6,023,279	45,445,241	80,560,217	103,047,910	–
	22,452,390	89	549,414	3,703,497	14,324,934	17,011,889	21,409,723	+
mrf	398,236,000	6,042	576,964	12,303,586	138,032,220	388,580,190	398,236,000	–
	284,243,830	8,430	131,674,830	225,001,910	283,732,640	284,080,270	284,080,270	+

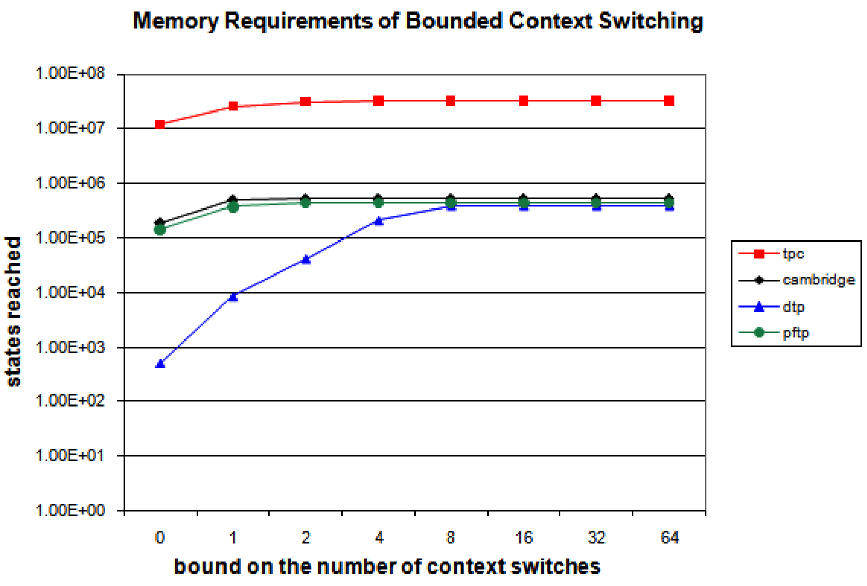

Fig. 7. Data from the first four rows in Table 1 extended to bounds from 0 to 64 (log-scales). The lines connecting the data points are not part of the data.

are needed to reach all states, but the bound needed to do so is also here surprisingly small (between 17 and 32).

As noted, the enforcement of larger bounds can impose significant performance penalties, slowing the rate at which new states are explored. This effect is illustrated in Figure 8, which gives the runtimes for each of the BCS runs shown in Figure 7. The overhead is caused by the number of times previously visited states must be revisited to explore all relevant execution scenarios. This means that especially for the larger bounds, a BCS algorithm cannot be considered to be in the class of search reduction techniques; quite the opposite is true. The objective for using a context bounded search algorithm in an explicit state model checker can only be the identification of more interesting execution scenarios, for instance to find simpler variants of known, but complex executions leading to safety or liveness violations.

Table 1 also shows the performance results of the BCS algorithm when the runs are performed with

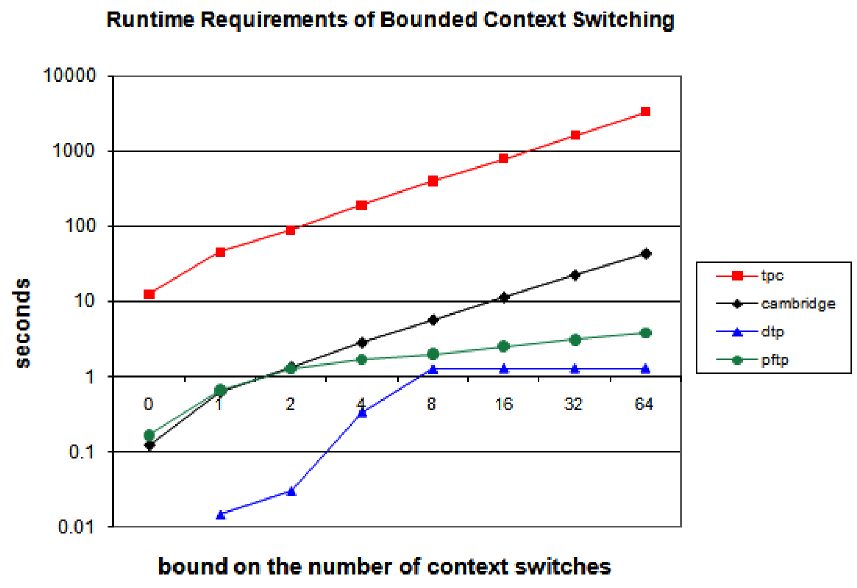

Fig. 8. Runtime requirements of BCS for bounds from 0 to 64 (log-scales).

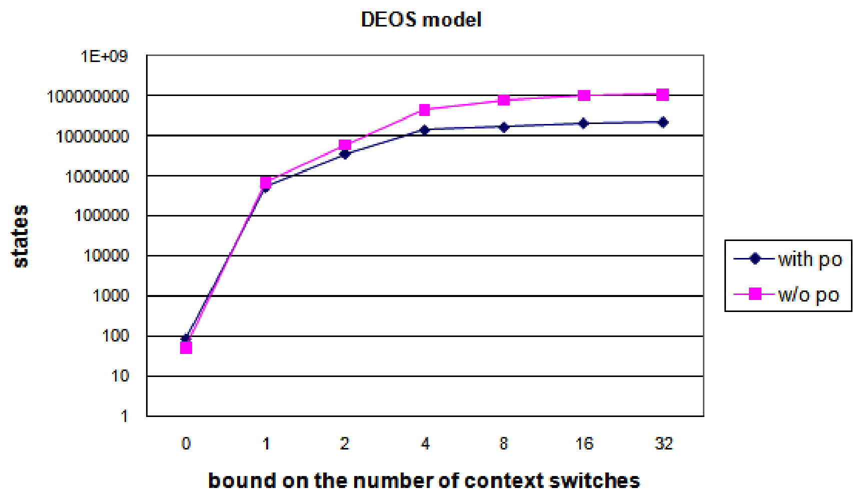

Fig. 9. Effect of partial order reduction on bounded context switching.

partial order reduction enabled. Figure 9 illustrates the difference between the two sets of runs for the large DEOS model.

The number of distinct states reached with a bound of zero is small both with and without partial order reduction enabled. The numbers gradually increase to match the size of a full search, which is again reached surprisingly quickly. Note that partial order reduction rules in combination with BCS can lead to different state space sizes than when the search is performed without BCS, even at larger bounds. The number of unique states reached with a bound of 32 and partial order reduction enabled, therefore, is not necessarily smaller than or equal to that of a full search without BCS.

The length of an execution is generally smaller when BCS rules are enforced than without those rules. It is not necessarily reduced further when partial order reduction is used as well. The maximum depth of the search tree for a standard search for the DEOS model, for instance, is 522,724 steps without partial order reduction and 177,911 steps with partial order reduction. With BCS enabled and a bound of 32, the maximum search depth drops to 68,420 steps without partial order reduction and to 69,760 with partial order reduction.

The time required to explore the DEOS model without partial order reduction and without context bounding is 376 seconds, corresponding to an effective exploration rate of 296,672 new states per second. The time required for the same run performed with bounded context switching enabled and a bound of 32 preemptions is 8,090 seconds, which corresponds to a much lower effective exploration rate of 13,797 new states per second. Both searches visit the same number of unique states, but the context bounded search revisits each state on average twenty times in this example, to explore all relevant paths, significantly affecting overall performance.

The time penalties are smaller in bitstate mode. Again without partial order reduction enabled, a verification of the DEOS model with a context bound of 32 takes 691 seconds in bitstate mode, and explores 82,149,957 unique states. The same bitstate run without context bounding takes only 346 seconds and visits a larger number of 111,587,020 unique states. We used a bitstate array of 2 GB for both these runs. Note that in bitstate mode we enter each state multiple times. With a bound of N, each state may be entered into the bitstate array up to N times, which reduces the capacity of the bitstate hash arena and hence reduces effective coverage.

4.1. Time and memory

The time that is required to perform a verification with Spin is closely correlated with the number of transitions that are explored in the depth-first search of the (partial order reduced) reachability graph, and the memory requirements are correlated with the number of states that are explored. As we have noted, the number of reachable states explored can be less than what a standard search (without the use of bounded context switching) would explore, but only for relatively small values of the bound. When at least five or six context switches are allowed, generally all reachable states will be explored. When bounded context switching is used, the number of transitions (which depends on the number of visits and revisits to reachable states) increases close to exponentially with increases of the bound. The maximum search depth reached can be expected to approximate that of a standard search more closely with each increase of the bound.

These effects can be seen clearly in Figure 10, where we performed measurements over a benchmark set of 171 different verification models. All models in this set had at least 200,000 reachable states. The key characteristics reported here were averaged over all models to give a clear indication of the dominant trends.

Based on these measurements, the conclusion seems justified that the use of bounded context switching is only advisable when the bound imposed is relatively small. For larger bounds, the time requirements of a model checking run can quickly become prohibitive. The relative improvement in the quality of error sequences generated is relatively minor in these cases.

4.2. Length of error sequences

We have done additional measurements with those applications from Table 1 that contain safety or liveness violations, to study how the quality of the error sequences that are generated by the bounded context switching algorithm compares with that of standard depth-first and breadth-first searches.

The *worst-case* model, shown in Appendix C, is a deliberate attempt to create a model that exhibits worst-case behavior when bounded context switching is used. The model is constructed in such a way that all active processes must take one step for an assertion violation to occur. We have used a version with ten processes, which means that at least nine unforced context switches are required for a counter-example to be possible.

The *mrf* model contains a liveness violation (the violation of a property expressed in linear-time temporal logic), which can only be found with the depth-first search. The remaining models contain safety violations that can be found with either a breadth-first or a depth-first search.

Table 2 lists the number of unique *states* that are explored by each algorithm up to the point where the first error is discovered and the error trail is written. Table 3 lists the number of *transitions* that are explored up to the same point in the execution. In all cases, finding the first counter-example takes only a fraction of a second. For the worst-case example, as expected, no error sequence is possible unless at least

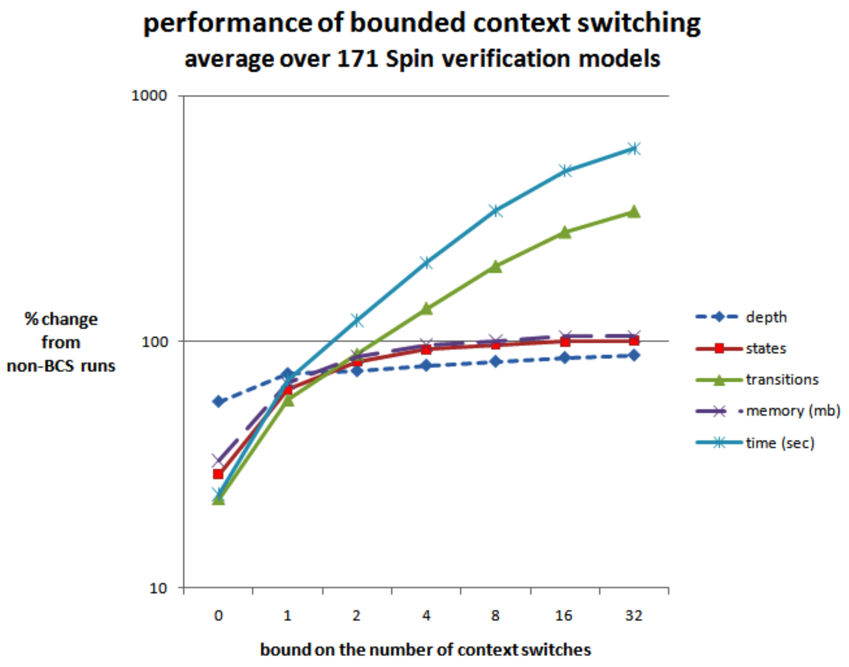

Fig. 10. Relative performance of bounded context switching, using partial order reduction.

Table 2. Number of unique states explored upto the first counter-example.

Model	BFS	DFS	BCS+L0	BCS+L1	BCS+L2	BCS+L4	BCS+L8	BCS+L16	PO
snoopy	5,346	3,591	1,354	1,352	1,501	1,015	1,236	1,236	–
	–	1,249	916	560	560	560	560	560	+
tpc	2,497	190	243	249	249	249	249	249	–
	–	93	243	198	198	198	198	198	+
nbq	385	437	56	41	41	41	41	41	–
	–	333	41	225	293	293	293	293	+
mrf	n/a	6,330	–	1,309	1,745	1,848	1,848	1,848	–
	n/a	5,140	–	1,381	1,587	1,602	1,602	1,602	+
worst-case	33,598	59,058	–	–	–	–	–	59,058	–
	–	49,224	–	–	–	–	–	49,224	+

nine context switches are allowed. For the other applications the number of states reached, or the length of the counter-example generated, does not change much for context-bounds larger than one.

For the purpose of the current study, the *quality* of a counter-example is correlated not with its length but with the number of context switches that it contains. Clearly, the shortest counter-example for a safety violation can always be generated with a breadth-first search, but that is not necessarily also the best counter-example. A depth-first, on the other hand, will tend to generate longer counter-examples. We are interested in knowing how much we can improve the quality of the counter-examples generated with a depth-first search when using the bounded context switching algorithms.

Table 4 lists the total number of context switches in each of the counter-examples that was generated with the different search algorithms. Note that the total number can exceed the bound because it will generally include both forced and unforced context switches. Additionally, when using partial order reduction there can also be context switches that are related to the enforcement of the partial order reduction rules.

The shortest counter-example for the *snoopy* protocol is 42 steps long, and contains 17 context switches. The best counter-example found with bounded-context switching and partial order reduction combined is 552 steps long and contains 228 context switches. This improves over the quality of the counter-example found with a standard depth-first search, with or without partial order reduction, though in this case not

Table 3. Number of transitions explored upto the first counter-example.

Model	BFS	DFS	BCS+L0	BCS+L1	BCS+L2	BCS+L4	BCS+L8	BCS+L16	PO
snoopy	6,576	5,507	1,524	1,522	1,698	1,130	1,403	1,402	–
	–	1,637	1,054	636	636	636	636	636	+
tpc	3,950	320	261	276	282	282	282	282	–
	–	99	261	244	244	244	244	244	+
nbq	565	752	57	42	42	42	42	42	–
	–	454	277	407	417	417	417	417	+
mrf	n/a	7,357	–	1,332	2,565	3,066	3,074	3,074	–
	n/a	5,587	–	1,490	2,027	2,056	2,056	2,056	+
worst-case	147,559	378,941	–	–	–	–	–	517,742	–
	–	291,801	–	–	–	–	–	266,645	+

Table 4. Total number of context switches in counter-example.

Model	BFS	DFS	BCS+L0	BCS+L1	BCS+L2	BCS+L4	BCS+L8	BCS+L16	PO/REACH
snoopy	17	1424	414	415	445	326	387	402	–/–
	–	523	335	228	228	228	228	228	+/–
	–	17	15	15	15	15	15	15	–/+
	–	17	15	17	17	17	17	17	+/+
tpc	19	70	67	67	67	67	67	67	–/–
	–	33	67	109	109	109	109	109	+/–
	–	20	21	16	16	16	16	16	–/+
	–	19	21	18	18	18	18	18	+/+
nbq	10	42	–	4	4	4	4	4	–/–
	–	32	4	6	6	6	6	6	+/–
	–	8	–	4	4	4	4	4	–/+
	–	10	4	6	6	6	6	6	+/+
mrf	–	310	–	6	6	6	6	6	–/–
	–	294	–	8	8	8	8	8	+/–
	–	4	–	4	4	4	4	4	–/+
	–	5	–	8	8	8	8	8	+/+
worst-case	10	10	–	–	–	–	–	10	–/–
	–	10	–	–	–	–	–	10	+/–
	–	10	–	–	–	–	–	10	–/+
	–	10	–	–	–	–	–	10	+/+

impressively so. The situation is different for the *nbq*, and *mrf* examples, and even for the *worst-case* model, where the bounded context switching algorithm succeeds in finding counter-examples with equal (*worst-case*) or far fewer context switches than both standard depth- and breadth-first search options. Figure 11, finally, shows the information from the first four models in Table 4 visually.

5. Related work

Bounded context switching algorithms were described earlier in [QW04, QR05, MQ07a, MQ07b, MQ08, LR08]. The first algorithms were designed for application in static source code analysis [QW04], and in software test systems, and they were often restricted to acyclic terminating executions. The formal model targeted in most of the earlier papers is that of pushdown automata.

The integration of a bounded context switching algorithm with partial order reduction methods was first discussed in [MQ07a], but was also restricted to terminating executions. The method discussed in [MQ07a] is based on the storage of a partial-order happens-before relation for visited states, which can only be used for strictly terminating (or otherwise depth-bounded) executions only.

In the earlier work, specifically [QW04, MQ07a, MQ07b], it was also demonstrated that relatively small

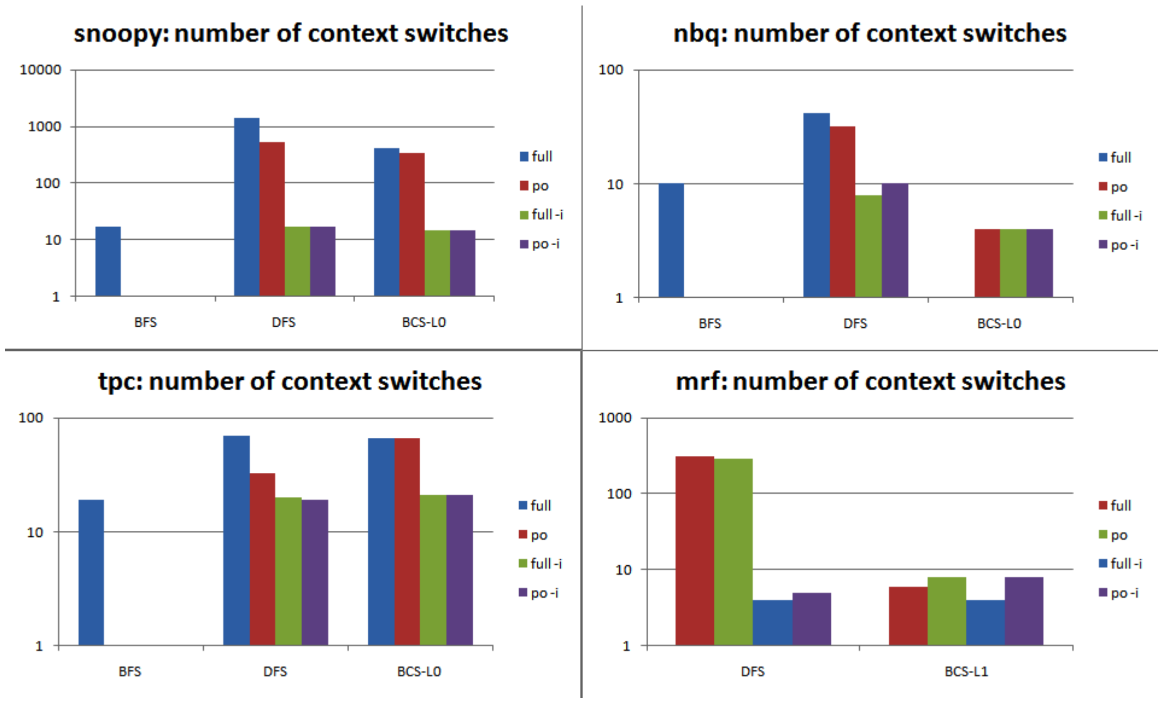

Fig. 11. Quality of counter-examples generated, cf. Fig 4. The y-axes are log-scales. The label *full* identifies searches done without partial order reduction. The label suffix *-i* identifies searches performed with Spin's iterative shortening algorithm, where the model checker is compiled with the additional directive -DREACH and the search is performed with parameter *-i*.

context bounds often sufficed to expose subtle concurrency related errors. In [QW04] a context bound of two was found sufficient, in [MQ07a] the largest context bound used was four. In this paper we have studied the effect of a context bounded search up to a bound of 64, as illustrated in Figures 8 and 9.

In [LR08] it is also suggested that the incremental change in the part of the state space that is covered decreases with each increase of the bound. In this paper we have measured this phenomenon more precisely and demonstrated an exponential decrease, as shown in Figures 10 and 9.

The method described in [LR08] is based on a conversion of a context-bounded concurrent program into a sequential one, which can then be analyzed with a sequential program analysis techniques that support the use of for symbolic constants and *assume* statements. Measurements reported in [LR08] were obtained with symbolic, instead of explicit state, model checkers and generally are based on different assumptions (e.g., about the distinction of forced vs unforced context switches, or how statespace sizes are calculated). This means that measurement results are not directly comparable, though qualitatively they are consistent with the results reported here.

A full implementation of a bounded context switching algorithm in an explicit state model checker, without a restriction to terminating executions, and with full support for partial order reduction and the verification of both safety *and* liveness properties was not studied before. As we have shown, by removing the assumption of terminating or depth-bounded executions, the problem becomes significantly more complex.

6. Conclusions

We have discussed the implementation of a bounded context switching discipline as an interesting additional search mode in an explicit state model checker. We have also described extensions of the basic algorithm that allow for an efficient integration of the new search mode with bitstate hashing and with partial order reduction. We have shown that bounded context switching cannot be considered a reduction method, as it often is, since it effectively increases, instead of reduces, the time requirements of the standard model checking algorithms.

Bounded context switching is not trivially compatible with partial order reduction methods. To enforce partial order reduction methods, the search algorithm can normally select safe transitions for execution, independent of the processes that execute those transitions. In a bounded context switching algorithm, on the other hand, we normally try to avoid context switches as long as possible. Partial order reduction, though, can be such a powerful search reduction technique that we are interested in finding workable compromises with bounded context switching algorithms. We have explored a method that aims to strike a balance between avoiding technically redundant states (under partial order reduction rules) and avoiding needless context switches.

We have proven that if our algorithm, used either with or without partial order reduction, fails to find a counter-example with a given context bound, then no such counter-example can exist.

We have shown that the bounded context switching algorithm can indeed generate counter-examples of a very high quality, if we measure the quality of a counter-example by the number of context switches that it contains. With partial order reduction the counter-examples are better than without, and with the addition of an iterative shortening method that is already part of the Spin model checker, the quality of the counter-examples is improved even further. The best traces found in this way tend to exceed the quality of counter-examples found with a breadth-first search. (The memory requirements for a breadth-first search, though, make that it is rarely a usable alternative for large problem sizes, and it is not an option at all in the verification of general liveness properties.)

Given some remaining uncertainty of which variant of which algorithm will produce the best possible result in any given case, it can be attractive to use the bounded context switching algorithm in combination with swarm-verification techniques. Searches for different context bounds can then be performed in parallel, and they can be used in combination with orthogonal alternative search modes, e.g., using reversed process or transition orderings, as explored in [HJG09]. As noted, the algorithm could further also be extended for use in combination with a breadth-first search discipline.

References

[B70] Bloom, B.H.: Spacetime tradeoffs in hash coding with allowable errors. *Comm. of the ACM*, Vol. 13, No. 7, pp. 422-426.

[HP94] Holzmann, G.J., and Peled, D.: An improvement in formal verification. *Proc. 7th Int. Conf. on Formal Description Techniques*, Bern, Switzerland, Oct. 1994, Chapman and Hall Publ., pp. 197-211.

[HPY96] Holzmann, G.J., Peled, D., and Yannakakis, M.,: On nested depth first search, *Proc. 2nd Spin Workshop*, American Mathematical Society, 1996, 23–32.

[H87] Holzmann, G.J.: On limits and possibilities of automated protocol analysis, *Proc. 6th Int. Conf. on Protocol Specification, Testing, and Verification*, INWG IFIP, Ed. H. Rudin and C. West, Zurich, Switzerland, June 1987.

[H98] Holzmann, G.J.: An analysis of bitstate hashing, *Formal Methods in System Design*, Vol. 13, No. 3, pp. 287-305, Kluwer, November 1998.

[H04] Holzmann, G.J.: *The Spin Model Checker: Primer and Reference Manual* Addison-Wesley, 2004.

[HJG09] Holzmann, G.J., Joshi, R., and Groce, A.: Swarm Verification Techniques, To appear in *IEEE Trans. on Software Engineering*, 2010.

[QW04] Qadeer, S., and Wu, D.: KISS: Keep it Simple and Sequential. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, (PLDI)*, Washington DC, 14–24 (June 2004).

[LR08] Lal, A., and Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis, *Proc. CAV 2008.*

[P94] Peled, D.: Combining partial order reduction with on-the-fly model checking, *Proc. CAV 2004*, Springer, LNCS 818, pp. 377-390.

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

[MQ07a] Musuvathi, M., and Qadeer, S.: Partial-order reduction for context-bounded state exploration, Microsoft Tech Report, MSR-TR-2007-12, February 2007, 19 pgs.

[MQ07b] Musuvathi, M., and Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs, Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, (PLDI), San Diego, June 2007.

[MQ08] Musuvathi, M., and Qadeer, S.: Fair stateless model checking, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Tucson, AZ, June 2008.

A. Proof of theorem 1

Let T be the set of transitions.

Definition 1. $G = (V, E, s_0)$ is the graph *constructed* by the DFS algorithm.

- s_0 is the initial state;
- $V = \{s \mid Search(s)$ was called by the DFS algorithm$\}$;
- $E = \{(s, a, q) \in V \times T \times V \mid s \xrightarrow{a} q\}$.

Definition 2. For a context switch bound N, $N \geq 0$, $G_N = (V_N, E_N, s_0)$ is the graph *constructed* by the BCS algorithm.

- s_0 is the initial state;
- $V_N = \{s \mid \exists k\ \exists id.\ Search(s, k, id)$ was called by the BCS algorithm$\}$;
- $E_N = \{(s, a, q) \in V_N \times T \times V_N \mid s \xrightarrow{a} q\}$.

Lemma 1. For a context switch bound N, $N \geq 0$, $G_N = (V_N, E_N, s_0)$ is a subgraph of $G = (V, N, s_0)$.

Proof. Because G was obtained by a DFS exploration, V contains all the states that are reachable from s_0.

V_N contains only states that are reachable from s_0, because whenever a new state s' is explored from a state s, s' is a successor of s. So $V_N \subseteq V$.

E_N is just the restriction of E to V_N, so $E_N \subseteq E$.

We have $V_N \subseteq$ and $E_N \subseteq E$, so G_N is a subgraph of G. $\Box$

Definition 3. For a transition a_1 leading to a state $s \in V$, and a transition a_2 enabled in s:
if $pid(a_1) \neq pid(a_2) \wedge (\exists a.\ pid(a) = pid(a_1) \wedge a \in enabled(s))$, then $switch(a_1, s, a_2) = 1$,
otherwise $switch(a_1, s, a_2) = 0$.

Definition 4. For a path $p = r_0 \xrightarrow{a_1} r_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} r_n$ from G, with $n > 0$, $cs(p)$ is the number of preemptive context switches along path p:

$cs(p) = \sum_{i=1}^{n-1} switch(a_i, r_i, a_{i+1})$, and $cs(\epsilon) = 0$.

Note that Lemma 1 implies that any path from G_N is also a path from G, so Definition 4 can also be used for paths from G_N.

Definition 5. For a context switch bound N, $N \geq 0$, a path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ from G_N is *explored* by the BCS algorithm if during the algorithm's execution there exists the following stack of calls of $Search$:
$Search(s_0, 0, \_))$, $Search(s_1, 0, pid(a_1)))$, ..., $Search(s_n, b_n, pid(a_n)))$

Note that any path explored by the BCS algorithm is also a path in G_N.

Definition 6. For a context switch bound N, $N \geq 0$,:
$CONSTRAINT(s, b, id) \equiv (b \leq N) \wedge (\forall x \forall y((s, x, y) \in S \Rightarrow (x > b \vee (x = b \wedge y \neq id))))$

Definition 7. For any path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ with $n > 0$, $last\_pid(p) = pid(a_n)$.

Lemma 2. For a context switch bound N, $N \geq 0$, any path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ explored by the BCS algorithm with the corresponding stack of $Search$ calls being: $Search(s_0, 0, \_))$, $Search(s_1, 0, pid(a_1)))$, ..., $Search(s_n, b_n, pid(a_n)))$, has the number of context switches $cs(p) = b_n$.

Proof. We prove the lemma by induction on the length of p.

The base cases:
If $p = \epsilon$, with the corresponding stack of $Search$ calls being: $Search(s_0, 0, \_)$, then we have $b_0 = 0$, $cs(\epsilon) = 0$, and so $cs(\epsilon) = b_0$.

If $p = s_0 \xrightarrow{a_1} s_1$, with the corresponding stack of $Search$ calls being: $Search(s_0, 0, \_)$, $Search(s_1, 0, pid(a_1))$, then we have $b_1 = 0$, $cs(s_0 \xrightarrow{a_1} s_1) = 0$, and so $cs(s_0 \xrightarrow{a_1} s_1) = b_1$.

The induction step:

We assume that any path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n$, $n > 0$, explored by the BCS algorithm with the corresponding stack of *Search* calls being: $Search(s_0, 0, \_))$, $Search(s_1, 0, pid(a_1)))$, ..., $Search(s_n, b_n, pid(a_n)))$, has $cs(p) = b_n$.

We prove that any path $p' = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} s_{n+1}$ explored by the BCS algorithm with the corresponding stack of *Search* calls being: $Search(s_0, 0, \_))$, $Search(s_1, 0, pid(a_1)))$, ..., $Search(s_n, b_n, pid(a_n)))$, $Search(s_{n+1}, b_{n+1}, pid(a_{n+1})))$, has $cs(p') = b_{n+1}$.

Let p be the prefix of path p' that does not include has last transition a_{n+1}. p has length n, and we can apply the induction hypothesis and we get that $cs(p) = b_n$.

From Definition 4 we have $cs(p') = cs(p) + switch(a_n, s_n, a_{n+1})$. Considering the induction hypothesis, we have to prove that $b_{n+1} = b_n + switch(a_n, s_n, a_{n+1})$.

If we consider the call of $Search(s_n, b_n, pid(a_n))$, then in this call, $CONSTRAINT(s_{n+1}, b_{n+1}, pid(a_{n+1}))$ is *true* because p' is a path explored by the BCS algorithm.

We have two cases:

a) $switch(a_n, s_n, a_{n+1}) = 0$, and this implies that either $pid(a_{n+1}) = pid(a_n)$, or the process $pid(a_n)$ has no enabled transitions in s_n.

If $pid(a_{n+1}) = pid(a_n)$, then $Search(s_n, b_n, pid(a_n))$, in line 18 of the BCS algorithm, calls $Search(s_{n+1}, b_n, pid(a_{n+1}))$. We get $b_{n+1} = b_n$. Because $switch(a_n, s_n, a_{n+1}) = 0$, we have $b_{n+1} = b_n + switch(a_n, s_n, a_{n+1})$.

If $pid(a_n)$ has no enabled transitions in s_n, then $Search(s_n, b_n, pid(a_n))$, in line 28 of the BCS algorithm ($u = 0$ in this case), calls $Search(s_{n+1}, b_n, pid(a_{n+1}))$. We get $b_{n+1} = b_n$. Because $switch(a_n, s_n, a_{n+1}) = 0$, we have $b_{n+1} = b_n + switch(a_n, s_n, a_{n+1})$.

b) $switch(a_n, s_n, a_{n+1}) = 1$, and this implies that $Search(s_n, b_n, pid(a_n))$, in line 28 of the BCS algorithm ($u = 1$ in this case), calls $Search(s_{n+1}, b_n + 1, pid(a_{n+1}))$. We get $b_{n+1} = b_n + 1$. Because $switch(a_n, s_n, a_{n+1}) = 1$, we have $b_{n+1} = b_n + switch(a_n, s_n, a_{n+1})$. □

Definition 8. For a path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n$ from G, for any i, $0 \leq i \leq n$, let $\pi_i = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_i} s_i)$ be the prefix of p containing the first $i + 1$ states and the first i transitions from p.

Lemma 3. For a path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ..., \xrightarrow{a_n} s_n$ from G, $n \geq 0$, for any i, j, $0 \leq i \leq j \leq n$, we have $cs(\pi_i) \leq cs(\pi_j)$.

Proof. Follows from the definition of the number of context switches along a path p from G. □

Theorem 1. For a context switch bound N, $N \geq 0$, for any path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n)$ from G with $n \geq 0$ and $cs(p) \leq N$ there exists a path $p_N = r_0 \xrightarrow{b_1} r_1 \xrightarrow{b_2} ... \xrightarrow{b_m} r_m$ with $r_0 = s_0$, such that p_N is explored by the BCS algorithm, and $cs(p_N) \leq cs(p)$.

Proof. From Lemma 3 we can deduce that for any prefix π_i with $i, 0 \leq i \leq n$, we have $cs(\pi_i) \leq cs(p)$.

The BCS algorithm starts by calling Search$(s_0, 0, _)$, so

(1) $\pi_0 = \epsilon$ is explored by the BCS algorithm. If $n = 0$ then we are done since $c(\pi_0) \leq c(p)$.

For $\pi_1 = s_0 \xrightarrow{a_1} s_1$, the BCS algorithms calls $Search(s_1, 0, pid(a_1))$ from Search$(s_0, 0, _)$, so

(2) $\pi_1 = s_0 \xrightarrow{a_1} s_1$ is explored by the BCS algorithm. If $n = 1$ then we are done since $c(\pi_1) \leq c(p)$.

Let k be the smallest integer for which π_k is not explored by the BCS algorithm. If no such k exists we are done since $p = \pi_n$ would have to be explored by the BCS algorithm.

If such a k exists, then from (1) and (2) we know that $k > 1$.

k is the smallest integer for which π_k is not explored by the BCS algorithms, so $\pi_{k-1} = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_{k-1}} s_{k-1}$ has to be explored by the BCS algorithm. This means that there exists the following call stack of *Search*: $Search(s_0, 0, \_)$, ..., $Search(s_{k-1}, b_{k-1}, pid(a_{k-1}))$.

Using Lemma 2 we can deduce that:

(3) $cs(\pi_{k-1}) = b_{k-1}$.

We know that $s_k \in successors(s_{k-1})$ since π_k is a path in G. The only reason why π_k would not be explored by the BCS algorithm is that $CONSTRAINT(s_k, b_k, pid(a_k))$ is false in line 17 or 27. Because $b_k = cs(\pi_k)$, we know that $b_k \leq cs(p)$.

$\neg CONSTRAINT(s_k, b_k, pid(a_k))$

$\equiv$ { definition of $CONSTRAINT$, De Morgan's laws, etc. }

$b_k > N \vee \exists x \exists y((s_k, x, y) \in S \wedge \neg(x > b_k \vee (x = b_k \wedge y \neq pid(a_k))))$

$\equiv$ { $b_k \leq N$, De Morgan's laws }
$\exists x \exists y((s_k, x, y) \in S \wedge x \leq b_k \wedge (x \neq b_k \vee y = pid(a_k)))$

Let x_0 be the smallest such x, and let y_0 be the corresponding y. We have:
(4) $(s_k, x_0, y_0) \in S \wedge x_0 \leq b_k \wedge (x_0 \neq b_k \vee y_0 = pid(a_k))$

From (4) we can deduce that there exists a path $\gamma_l^k = q_0 \xrightarrow{c_1} ... \xrightarrow{c_l} q_l$, with $q_0 = s_0$ and $q_l = s_k$, such that γ_l^k is explored by the BCS algorithm, and $cs(\gamma_l^k) = x_0$. So $cs(\gamma_l^k) \leq cs(\pi_k)$.

If $k = n$ we are done because we have found a path γ_l^k, from s_0 to s_n, such that γ_l^k is explored by the BCS algorithm, with $cs(\gamma_l^k) \leq cs(p)$.

If $k < n$, then we can construct a new path in G: $\gamma_k = \gamma_l^k(s_k \xrightarrow{a_{k+1}} s_{k+1} \xrightarrow{a_{k+2}} ... \xrightarrow{a_n} s_n)$.

We know that $cs(\gamma_l^k) \leq cs(\pi_k)$. We consider the two possible cases:
a) $cs(\gamma_l^k) < cs(\pi_k)$
In this case we can conclude, from the definition of cs, that $cs(\gamma_k) \leq cs(p)$.

b) $cs(\gamma_l^k) = cs(\pi_k)$
From (4) we must also have $last\_pid(\gamma_l^k) = last\_pid(\pi_k)$. From the definition of cs we can conclude that $cs(\gamma_k) = cs(p)$.

Combining cases (a) (b) we get: $cs(\gamma_k) \leq cs(p)$.

We have constructed a path γ_k in G with the following property: $cs(\gamma_k) \leq cs(p)$, and the largest suffix of γ_k not explored by the BCS algorithm has at most $(n - k)$ states. Remember that p had a suffix of $n - k - 1$ states unexplored by the BCS algorithm since π_k was not explored.

We now rename γ_k to γ_{k_1}. We can reapply the same reasoning we used for p to this new path γ_{k_1} and we end up with a new path γ_{k_2} that has a smaller largest suffix that is unexplored by the BCS algorithm and also $cs(\gamma_{k_2}) \leq cs(\gamma_{k_1})$.

Note that if we keep applying the same reasoning, the length of the largest suffix that is unexplored by the BCS algorithm keeps decreasing. At some point this length will become 0. We end up with a chain of paths: p, γ_{k_1}, γ_{k_2}, ..., γ_{k_m} that has $cs(p) \geq cs(\gamma_{k_1}) \geq cs(\gamma_{k_2}) \geq ... \geq cs(\gamma_{k_m})$, $k_1 < k_2 < ... < k_m$ and $k_m = n$.

We were able to create a path $p_N = \gamma_{k_m}$ from s_0 to s_n that is explored by the BCS algorithm and has $cs(p_N) \leq cs(p)$. $\square$

B. Proof of theorem 2

Definition 9. In the BCSPO algorithm, lines 26, 31, and 42, for a context switch bound N, $N \geq 0$, $CONSTRAINT(s, k, id, safe)$ is defined as:
$(k \leq N) \wedge (\forall x \forall y \forall z.\ (s, x, y, z) \in S \Rightarrow (x > k \vee (x = k \wedge y \neq id)) \vee (x = k \wedge y = id \wedge safe \wedge \neg z))$.

Definition 10. For a state s, a non negative integer k, a process identifier id, and a boolean $safe$, $search(s, k, id, safe) \equiv true$ if and only if $Search(s, k, id, safe)$ was called by the BCSPO algorithm.

Definition 11. For a context switch bound N, $N \geq 0$, $G_N^{PO} = (V_N^{PO}, E_N^{PO}, s_0)$ is the graph *constructed* by the BCSPO algorithm.

- s_0 is the initial state;
- $V_N^{PO} = \{s \mid \exists k\ \exists id\ \exists safe.\ search(s, k, id, safe)\}$;
- $E_N^{PO} = \{(s, a, q) \in V_N^{PO} \times T \times V_N^{PO} \mid s \xrightarrow{a} q\}$.

Lemma 4. For a context switch bound N, $N \geq 0$, $G_N^{PO} = (V_N^{PO}, E_N^{PO}, s_0)$ is a subgraph of $G = (V, N, s_0)$.

Proof. V contains all the states that are reachable from s_0. V_N^{PO} contains only states that are reachable from s_0, because whenever a new state s' is explored from a state s, s' is a successor of s. So $V_N^{PO} \subseteq V$. E_N^{PO} is just the restriction of E to V_N^{PO}, so $E_N^{PO} \subseteq E$.
We have $V_N^{PO} \subseteq$ and $E_N^{PO} \subseteq E$, so G_N^{PO} is a subgraph of G. □

Definition 12. For a context switch bound N, $N \geq 0$, and a state $s \in V_N^{PO}$: $cspo(s)$ is the minimum number of counted context switches used by the BCSPO algorithm to reach state s.
$cspo(s) = \min\{k \mid \exists id\ \exists safe.\ search(s, k, id, safe)\}$.

Definition 13. For a state $s \in V$, a non negative integer k, a process identifier id, and a boolean $safe$: $cycle(s, k, id, safe) \equiv true$ if and only if $Search(s, k, id, safe)$ was called by the BCSPO algorithm, $ample(s) \neq enabled(s)$, and in the call of $Search(s, k, id, safe)$ there exists a transition $a \in ample(s)$ such that $a(s)$ is on the stack Q.

Definition 14. For a context switch bound N, $N \geq 0$, for a state $s \neq s_0$ from G_N^{PO} and a transition a enabled in s:
if $(\forall id\ \forall safe.\ search(s, cspo(s), id, safe) \Rightarrow (\neg cycle(s, cspo(s), id, safe) \wedge pid(a) \neq id \wedge \neg safe \wedge$
$(\exists a' \in enabled(s).\ pid(a') = id))$, then $switchpo(s, a) = 1$, otherwise $switchpo(s, a) = 0$. Also, for any action a enabled in s_0, $switchpo(s_0, a) = 0$.

Lemma 5. For a context switch bound N, $N \geq 0$, for a state s, a non negative integer k, $k \leq N$, a process identifier id, and a boolean $safe$, if the BCSPO algorithm, in lines 26-28, 31-33, or 42-44, calls: $ifCONSTRAINT(s, k, id, safe)\{Search(s, k, id, safe)\}$, then $s \in V_N^{PO}$ and the following properties hold:

- $cspo(s) \leq k$;
- if $cspo(s) = k$, then $(\exists sf.\ search(s, cspo(s), id, sf))$;
- if $cspo(s) = k \wedge safe$, then $search(s, cspo(s), id, true)$.

Proof. Let's consider the case when $CONSTRAINT(s, k, id, safe)$ is $true$.
In this case, the BCSPO algorithm calls $Search(s, k, id, safe)$ and, according to Definition 11, $s \in V_N^{PO}$.
From Definition 10 we have $search(s, k, id, safe)$, and using Definition 12 we get that $cspo(s) \leq k$.
We have $search(s, k, id, safe)$, so if $cspo(s) = k$, we also have $search(s, cspo(s), id, safe)$. Moreover, if $safe$ is $true$, we also have $search(s, cspo(s), id, true)$.
Let's consider the case when $CONSTRAINT(s, k, id, safe)$ is $false$.
$\neg CONSTRAINT(s, k, id, safe)$
$\equiv$ {Definition 9, De Morgan's Laws}
$\neg(k \leq N) \vee \neg(\forall x \forall y \forall z.\ (s, x, y, z) \in S \Rightarrow (x > k \vee (x = k \wedge y \neq id)) \vee (x = k \wedge y = id \wedge safe \wedge \neg z))$
$\equiv$ {$k \leq N$, De Morgan's Laws}
$\exists x \exists y \exists z.\ \neg((s, x, y, z) \in S \Rightarrow (x > k \vee (x = k \wedge y \neq id)) \vee (x = k \wedge y = id \wedge safe \wedge \neg z)))$
$\equiv$ {properties of $\Rightarrow$, De Morgan's Laws}
$\exists x \exists y \exists z.\ (s, x, y, z) \in S \wedge\ (x \leq k \wedge (x \neq k \vee y = id)) \wedge (x \neq k \vee y \neq id \vee \neg safe \vee z)))$

Let x^0, y^0, z^0 be some values of x, y, z that make the above formula $true$.

Line 15 of the BCSPO algorithm is the only one that adds elements to S, and because $(s, x, y, z) \in S$, we know that $Search(s, x, y, z)$ was previously called by the BCSPO algorithm. So we have $search(s, x, y, z)$.

From Definition 12: $cspo(s) \leq x$, and $x \leq k$, so $cspo(s) \leq k$.

If $cspo(s) = k$, then $cspo(s) = x$, and $x = k$. In this case $(y = id) \wedge (y \neq id \vee \neg safe \vee z)$, which is the same as $(y = id) \wedge (\neg safe \vee z)$.

We have $(y = id)$ so we also have $search(s, cspo(s), id, z)$.

If $safe$ is $true$, then z must be $true$, so we have $search(s, cspo(s), id, true)$. □

Lemma 6. For any path $r_0 \xrightarrow{b1} r_1 ... \xrightarrow{b_n} r_n$ from G_N, with $n > 0$, if for all i, $0 \leq i < n$, $ample(r_i) \neq enabled(r_i)$ and $b_{i+1} \in ample(r_i)$, then for all i, $0 \leq i < n$, $cspo(r_{i+1}) \leq cspo(r_i)$.

Proof. For any i, $0 \leq i < n$, $r_i \in V_N^{PO}$, so the BCSPO algorithm called $Search(r_i, cspo(r_i), id, safe)$.

$b_{i+1} \in ample(r_i)$, and $ample(r_i) \subset enabled(r_i)$, so $b_{i+1} \in enabled(r_i)$.

We have the following cases:

a) If $cycle(r_i, cspo(r_i), id, safe)$ is $true$, and $pid(b_{i+1}) = id$, then the BCSPO algorithm, in lines 31-33, calls: $if\ CONSTRAINT(r_{i+1}, cspo(r_i), pid(b_{i+1}), false)\ \{Search(r_{i+1}, cspo(r_i), id, false)\}$.

b) If $cycle(r_i, cspo(r_i), id, safe)$ is $true$, and $pid(b_{i+1}) \neq id$, then the BCSPO algorithm, in lines 42-44, calls: $if\ CONSTRAINT(r_{i+1}, cspo(r_i), pid(b_{i+1}), false)\ \{Search(r_{i+1}, cspo(r_i), pid(b_{i+1}), false)\}$.

c) If $cycle(r_i, cspo(r_i), id, safe)$ is $false$, and $cspo(r_i) < N$, then, because $b_{i+1} \in ample(r_i)$ the BCSPO algorithm, in lines 26-28, calls:
$if\ CONSTRAINT(r_{i+1}, cspo(r_i), pid(b_{i+1}), true)\ \{Search(r_{i+1}, cspo(r_i), pid(b_{i+1}), true)\}$.

In cases a),b), and c) we have $cspo(r_i) \leq N$ because $r_i \in V_N^{PO}$. By applying Lemma 5 we get that $cspo(r_{i+1}) \leq cspo(r_i)$.

There is one more case left:

d) If $cycle(r_i, cspo(r_i), id, safe)$ is $false$, and $cspo(r_i) = N$, then because $r_{i+1} \in V_N^{PO}$ we have $cspo(r_{i+1}) \leq N$, so $cspo(r_{i+1}) \leq cspo(r_i)$. □

Lemma 7. For any cycle, $r_0 \xrightarrow{b1} r_1 ... \xrightarrow{b_n} r_n$ from G_N, with $n > 0$ and $r_0 = r_n$, if for all i, $0 \leq i < n$, $ample(r_i) \neq enabled(r_i)$ and $b_{i+1} \in ample(r_i)$, then for all i, $0 \leq i \leq n$, $cspo(r_i) = cspo(r_0)$.

Proof. Using Lemma 6 we get that $cspo(r_n) \leq cspo(r_{n-1}) \leq ... \leq cspo(r_0)$.

Because $r_n = r_0$, we have $cspo(r_n) = cspo(r_0)$, so for all i, $0 \leq i \leq n$, $cspo(r_i) = cspo(r_0)$. □

Lemma 8. For any cycle, $r_0 \xrightarrow{b1} r_1 ... \xrightarrow{b_n} r_n$ from G_N, with $n > 0$ and $r_0 = r_n$, if for all i, $0 \leq i < n$, $ample(r_i) \neq enabled(r_i)$ and $b_{i+1} \in ample(r_i)$,then $(\forall i.\ 0 \leq i \leq n.\ cspo(r_i) = N) \vee$ $(\exists k.\ 0 \leq k < n.\ \exists id \exists safe.\ cycle(r_k, cspo(r_k), id, safe))$.

Proof. If $cspo(r_0) = N$, then we are done, because using Lemma 7 we have $\forall i.\ 0 \leq i \leq n.\ cspo(r_i) = N$.

If $cspo(r_0) = c$, $c < N$, then $\forall i.\ 0 \leq i \leq n.\ cspo(r_i) = c$.

Without loss of generality we can assume that r_0 is the first state for which $Search(r_0, c, id_0, safe_0)$ was called for some id_0, and some $safe_0$.

We have two cases, in the call of $Search(r_0, c, id_0, safe_0)$:

a) $cycle(r_0, c, id_0, safe_0)$ was $true$, then we are done, because for $k = 0$, $\exists id \exists safe.\ cycle(r_0, c, id, safe)$.

b) $cycle(r_0, c, id_0, safe_0)$ was $false$, then the BCSPO algorithm, in lines 26-28, called:
$if\ CONSTRAINT(r_1, c, pid(b_1), true)\ \{Search(r_1, c, pid(b_1), true)\}$. $CONSTRAINT(r_1, c, pid(b_1), true)$ must be $true$, otherwise r_0 would not have been the first state reached with the context bound c (r_1 would have been reached before r_0). We now have that $Search(r_1, c, pid(b_1), true)$ was called by the BCSPO algorithm with r_0 on the stack Q.

If for any i, $0 < i < n$, $\exists id_i \exists safe_i.\ Search(r_i, c, id_i, safe_i)$ was called by the BCSPO algorithm with r_0 on the stack Q, and $\neg cycle(r_i, c, id_i, safe_i)$, then we prove that: $\exists id_{i+1} \exists safe_{i+1}.\ Search(r_{i+1}, c, id_{i+1}, safe_{i+1})$ was called by the BCSPO algorithm with r_0 on the stack Q or $cycle(r_{i+1}, c, id_{i+1}, safe_{i+1})$.

In the call of $Search(r_i, c, id_i, safe_i)$ with $\neg cycle(r_i, c, id_i, safe_i)$, the BCSPO algorithm, in lines 26-28, calls: $if\ CONSTRAINT(r_{i+1}, c, pid(b_{i+1}), true)\ \{Search(r_{i+1}, c, pid(b_{i+1}), true)\}$.

If $CONSTRAINT(r_{i+1}, c, pid(b_{i+1}), true)$ is $true$, then we are done, because r_0 is on the stack Q, and the BCSPO algorithm calls $Search(r_{i+1}, c, pid(b_{i+1}), true)$.

If $CONSTRAINT(r_{i+1}, c, pid(b_{i+1}), true)$ is $false$, then we must have had a previous call

$Search(r_{i+1}, c, id_{i+1}, safe_{i+1})$ for some id_{i+1} and some $safe_{i+1}$. This call could not happen before the call for r_0 because r_0 was first reached with $cspo(r_0) = c$. Because the exploration is a depth first search, r_{i+1} was discovered on a path that contains r_0, but not r_i. In this case r_0 was on the stack Q of the $Search(r_{i+1}, c, id_{i+1}, safe_{i+1})$ call.

Repeatedly applying the above observation up to $n-1$, we get that either a cycle was found for some $i < n-1$, and in this case we are done, or $\exists id_{n-1} \exists safe_{n-1}$. $Search(r_{n-1}, c, id_{n-1}, safe_{n-1})$ was called by the BCSPO algorithm with r_0 on the stack Q. Because $b_n \in ample(r_{n-1})$, and because $b_n(r_{n-1}) = r_n$, and $r_n = r_0$,we must have $cycle(r_{n-1}, c, id_{n-1}, safe_{n-1})$. $\square$

Definition 15. For a path $p = s_0 \xrightarrow{a_1} ... \xrightarrow{a_n} s_n$, $n > 0$, from G, $ls(p) = s_n$, the last state of p. Also, $ls(\epsilon) = s_0$.

Definition 16. For a path $p = s_0 \xrightarrow{a_1} ... \xrightarrow{a_n} s_n$, $n > 0$, from G, $ft(p) = a_1$, the first transition of p.

Lemma 9. For a context switch bound N, $N \geq 0$, and a path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n$ from G with $n \geq 0$ and $cs(p) \leq N$, we can construct a sequence of paths $\pi_0, \pi_1, ..., \pi_m$, $n \leq m$, such that $\pi_0 = p$, for all i, $0 \leq i \leq m$, $\pi_i = \eta_i \theta_i$, and the following properties hold:

- η_i is a path from G_N^{PO} with $|\eta_i| = i$;
- for all i, $0 \leq i < m$, $|\theta_{i+1}| \leq |\theta_i|$, and $|\theta_m| = 0$;
- forall i, $0 \leq i < m-1$, $cspo(ls(\eta_{i+1})) + switchpo(ls(\eta_{i+1}), ft(\theta_{i+1})) + cs(\theta_{i+1}) \leq cspo(ls(\eta_i)) + switchpo(ls(\eta_i), ft(\theta_i)) + cs(\theta_i)$;
- if $m > 0$, then $cspo(ls(\eta_m)) \leq cspo(ls(\eta_{m-1})) + switchpo(ls(\eta_{m-1}), ft(\theta_{m-1})) + cs(\theta_{m-1})$

Proof. We construct $\pi_0 = \eta_0 \theta_0$ with $\eta_0 = \epsilon$, and $\theta_0 = p$. So $\pi_0 = p$.

$|\eta_0| = 0$ and η_0 is a path from G_N^{PO}. Also, $|\theta_0| = n$.

If $n = 0$, we also have that $m = 0$, and $\theta_0 = \epsilon$. The other properties trivially hold because $m = 0$.

We construct $\pi_1 = \eta_1 \theta_1$ with $\eta_1 = s_0 \xrightarrow{a_1} s_1$ and $\theta_1 = s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n$.

We have $|\eta_1| = 1$. s_0 is the initial state, so the BCSPO algorithm calls $Search(s_0, 0,, false)$. In this call, line 10 ensures that $A* = A$, and $A' = \emptyset$, so on line 40 we have $u = 0$. The algorithm, in lines 42-44, calls: $ifCONSTRAINT(s_1, 0, pid(a_1), false)$ $\{Search(s_1, 0, pid(a_1), false)\}$. According to Lemma 5, $cspo(s_1) \leq 0$, so $cspo(s_1) = 0$, and $s_1 \in V_N^{PO}$. We have that η_1 is a path from G_N^{PO}.

θ_1 is θ_0 with the first transition removed so, $|\theta_1| = |\theta| - 1$, and we have $|\theta_1| \leq |\theta_0|$.

If $n = 1$, then $m = 1$, and $|\theta_m| = 0$. If $m = 1$, then:
$cspo(ls(\eta_m)) \leq cspo(ls(\eta_{m-1})) + switchpo(ls(\eta_{m-1}), ft(\theta_{m-1})) + cs(\theta_{m-1})$
$\equiv$ {the way η_0, θ_0, η_1, θ_1 were constructed}
$cspo(s_1) \leq cspo(s_0) + switchpo(s_0, a_1) + cs(s_0 \xrightarrow{a_1} s_1)$
$\equiv \{cspo(s_1) = 0$, $cspo(s_0) = 0$, $switchpo(s_0, a_1) = 0$, $cs(s_0 \xrightarrow{a_1} s_1) = 0\}$
true

If $n > 1$, then $m > 1$, and
$cspo(ls(\eta_1)) + switchpo(ls(\eta_1), ft(\theta_1)) + cs(\theta_1) \leq cspo(ls(\eta_0)) + switchpo(ls(\eta_0), ft(\theta_0)) + cs(\theta_0)$
$\equiv$ {the way η_0, θ_0, η_1, θ_1 were constructed}
$cspo(s_1) + switchpo(s_1, a_2) + cs(\theta_1) \leq cspo(s_0) + switchpo(s_0, a_1) + cs(\theta_0)$
$\equiv$ {definition of cs, etc.}
$0 + switchpo(s_1, a_2) + cs(\theta_1) \leq 0 + 0 + switch(a_1, s_1, a_2) + cs(\theta_1)$
$\equiv$
$switchpo(s_1, a_2) \leq switch(a_1, s_1, a_2)$
$\equiv$ {see below why $switch(a_1, s_1, a_2) = 0 \Rightarrow switchpo(s_1, a_2) = 0$}
true

If $switch(a_1, s_1, a_2) = 0$, then either $pid(a_1)$ doesn't have any transitions enabled in s_1 or $pid(a_2) = pid(a_1)$. Because a_1 was explored from s_0, s_1 was reached with $cspo(s_1) = 0$ from the process $pid(a_1)$. From Definition 14 we get that $switchpo(s_1, a_2) = 0$.

We now show how to construct π_{i+1} assuming that we have constructed $\pi_0, ..., \pi_i$.

Let $\eta_i = r_0 \xrightarrow{b_1} r_1 \xrightarrow{b_2} ... \xrightarrow{b_i} r_i$, and $\theta_i = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} ... \xrightarrow{c_j} q_j$, with $q_0 = r_i$.

We have the following cases:

case 1. There are safe transitions in q_0, but the BCSPO algorithm reached q_0 by calling

$Search(q_0, cspo(q_0), id, safe)$, and in this call a cycle was detected. Formally:
$ample(q_0) \neq enabled(q_0) \wedge (\exists id \exists safe.\ cycle(q_0, cspo(q_0), id, safe))$.

In this case, the BCSPO algorithm, in lines 31-33, or in lines 42-44 (with $u = 0$), calls:
$if\ CONSTRAINT(q_1, cspo(q_0), pid(c_1), false)\ \{Search(q_1, cspo(q_0), pid(c_1), false)\}$.

case 2. There are safe transition in q_0, the BCSPO algorithm never reached q_0 with $cspo(q_0)$ and closed a cycle, but in every call of $Search(q_0, cspo(q_0), id, safe)$ the BCSPO algorithm, in lines 26-28, explored c_1 by calling:
$if\ CONSTRAINT(q_1, cspo(q_0), pid(c_1), true)\ \{Search(q_1, cspo(q_0), pid(c_1), true)\}$. Formally:
$ample(q_0) \neq enabled(q_0) \wedge \neg(\exists id \exists safe.\ cycle(q_0, cspo(q_0), id, safe)) \wedge cspo(q_0) < N \wedge c_1 \in ample(q_0)$

case 3. There are safe transition in q_0, the BCSPO algorithm never reached q_0 with $cspo(q_0)$ and closed a cycle, and $cspo(q_0) = N$. Also, the BCSPO algorithm reached q_0 with $cspo(q_0)$ such that executing c_1 does not require a switch, either because it is continuing with the same process, q_0 was reached via a safe transition, or the process with which q_0 was reached blocks. Formally:
$ample(q_0) \neq enabled(q_0) \wedge \neg(\exists id \exists safe.\ cycle(q_0, cspo(q_0), id, safe)) \wedge cspo(q_0) = N \wedge$
$(\exists id \exists safe.\ search(q_0, cspo(q_0), id, safe) \wedge (pid(c_1) = id \vee safe \vee \neg(\exists a.\ pid(a) = id)))$

In this case, the BCSPO algorithm, in lines 31-33, or in lines 42-44 (with $u = 0$), calls:
$if\ CONSTRAINT(q_1, cspo(q_0), pid(c_1), false)\ \{Search(q_1, cspo(q_0), pid(c_1), false)\}$.

case 4. There are no safe transition in q_0, but the BCSPO algorithm reached q_0 with $cspo(q_0)$ such that executing c_1 does not require a switch, either because it is continuing with the same process, q_0 was reached via a safe transition, or the process with which q_0 was reached blocks. Formally:
$ample(q_0) = enabled(q_0) \wedge$
$(\exists id \exists safe.\ search(q_0, cspo(q_0), id, safe) \wedge (pid(c_1) = id \vee safe \vee \neg(\exists a.\ pid(a) = id)))$

In this case, the BCSPO algorithm, in lines 31-33, or in lines 42-44 (with $u = 0$), calls:
$if\ CONSTRAINT(q_1, cspo(q_0), pid(c_1), false)\ \{Search(q_1, cspo(q_0), pid(c_1), false)\}$.

In cases 1, 2, 3, and 4, we pick: $\eta_{i+1} = r_0 \xrightarrow{b_1} ... \xrightarrow{b_i} r_i \xrightarrow{c_1} q_1$, and $\theta_{i+1} = q_1 \xrightarrow{c_2} ... \xrightarrow{c_j} q_j$.

The construction ensures that $|\eta_{i+1}| = i + 1$, and $|\theta_{i+1}| = |\theta_i| - 1$. So $|\theta_{i+1}| \leq |\theta_i|$. If $|\theta_{i+1}| = 0$, we pick $m = i + 1$ and we stop the construction. If not, we have $i < m - 1$, and we continue with the construction.

In all the four cases the BCSPO algorithm called:
$if\ CONSTRAINT(q_1, cspo(q_0), pid(c_1), safe)\ \{Search(q_1, cspo(q_0), pid(c_1), safe)\}$,
for $safe$ either $true$ (case 2) or $false$ (cases 1, 3, and 4).

Using Lemma 5 we get that $q_1 \in V_N^{PO}$, so η_{i+1} is a path in G_N^{PO}. Also,
$cspo(q_1) \leq cspo(q_0) \wedge (cspo(q_1) = cspo(q_0) \Rightarrow (\exists sf.\ search(q_1, cspo(q_0), pid(c_1), sf)))$.

If $m = i + 1$, we are done because $cspo(q_1) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$.

If $i < m - 1$, then we have two sub cases:

a) $cspo(q_1) < cspo(q_0)$

Here we have $cspo(q_1) + switchpo(q_1, c_2) + cs(\theta_{i+1}) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$, no matter what the value of $switchpo(q_1, c_2)$ is, because $cs(\theta_{i+1}) \leq cs(\theta_i)$.

b) $cspo(q_1) = cspo(q_0)$ and $(\exists sf.\ search(q_1, cspo(q_0), pid(c_1), sf)))$.

q_1 was reached with $cspo(q_1)$ from $pid(c_1)$, so: $switch(c_1, q_1, c_2) = 0 \Rightarrow switchpo(q_1, c_2) = 0$.

We have:
$cspo(q_1) + switchpo(q_1, c_2) + cs(\theta_{i+1}) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$
$\equiv$ {definition of cs, etc.}
$switchpo(q_1, c_2) + cs(\theta_{i+1}) \leq switchpo(q_0, c_1) + switch(c_1, q_1, c_2) + cs(\theta_{i+1})$
$\equiv \{switch(c_1, q_1, c_2) = 0 \Rightarrow switchpo(q_1, c_2) = 0\}$
$true$

case 5. There are safe transition in q_0, the BCSPO algorithm never reached q_0 with $cspo(q_0)$ and closed a cycle, and $cspo(q_0) = N$. Also, every time q_0 was reached with $cspo(q_0)$, executing c_1 requires a switch. Formally:
$ample(q_0) \neq enabled(q_0) \wedge \neg(\exists id \exists safe.\ cycle(q_0, cspo(q_0), id, safe)) \wedge cspo(q_0) = N \wedge$
$\neg(\exists id \exists safe.\ search(q_0, cspo(q_0), id, safe) \wedge (pid(c_1 = id) \vee safe \vee \neg(\exists a.\ pid(a) = id)))$

From Definition 14 we get that $switchpo(q_0, c_1) = 1$. We also know that $cspo(q_0) = N$.

From the induction hypothesis we have:
$cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i) \leq cs(p)$, so: $N + 1 + cs(\theta_i) \leq cs(p)$.

But $cs(p) \leq N$, $cs(\theta_i) \geq 0$, and we get a contradiction. So this case is not possible.

case 6. There are no safe transition in q_0, and the BCSPO algorithm never reached q_0 with $cspo(q_0)$

such that executing c_1 does not require a switch. Formally:
$ample(q_0) = enabled(q_0) \wedge$
$\neg(\exists id \exists safe.\ search(q_0, cspo(q_0), id, safe) \wedge (pid(c_1) = id \vee safe \vee \neg(\exists a.\ pid(a) = id)))$

So in this case $switchpo(q_0, c_1) = 1$.

Just like in cases 1-4 we pick: $\eta_{i+1} = r_0 \xrightarrow{b_1} ... \xrightarrow{b_i} r_i \xrightarrow{c_1} q_1$, and $\theta i + 1 = q_1 \xrightarrow{c_2} ... \xrightarrow{c_j} q_j$.

The construction ensures that $|\eta_{i+1}| = i + 1$, and $|\theta_{i+1}| = |\theta_i| - 1$. So $|\theta_{i+1}| \leq |\theta_i|$. If $|\theta_{i+1}| = 0$, we pick $m = i + 1$ and we stop the construction. If not, we have $i < m - 1$, and we continue with the construction.

The BCSPO algorithm, in lines 42-44 (with $u = 1$), calls:
$if\ CONSTRAINT(q_1, cspo(q_0) + 1, pid(c_1), false)\ \{Search(q_1, cspo(q_0) + 1, pid(c_1), false)\}$.

From the induction hypothesis we have that $cspo(q_0) + switchpo(q_0, c_1) \leq cs(p)$. We know $cs(p) \leq N$, and $switchpo(q_0, c_1) = 1$, so we must have $cspo(q_0) + 1 \leq N$.

Using Lemma 5 we get that $q_1 \in V_N^{PO}$, so η_{i+1} is a path in G_N^{PO}. Also,
$cspo(q_1) \leq cspo(q_0) + 1 \wedge (cspo(q_1) = cspo(q_0) + 1 \Rightarrow (\exists sf.\ search(q_1, cspo(q_0) + 1, pid(c_1), sf)))$.

If $m = i+1$, we are done. $cspo(q_1) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$ because $cspo(q_1) \leq cspo(q_0) + 1$, and $switchpo(q_0, c_1) = 1$.

If $i < m - 1$, then we have two sub cases:

a) $cspo(q_1) < cspo(q_0) + 1$

Here we have $cspo(q_1) + switchpo(q_1, c_2) + cs(\theta_{i+1}) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$, no matter what the value of $switchpo(q_1, c_2)$ is, because $cs(\theta_{i+1}) \leq cs(\theta_i)$.

b) $cspo(q_1) = cspo(q_0) + 1$ and $(\exists sf.\ search(q_1, cspo(q_0), pid(c_1), sf)))$.

Because q_1 was reached with $cspo(q_1)$ from $pid(c_1)$, we have: $switch(c_1, q_1, c_2) = 0 \Rightarrow switchpo(q_1, c_2)$.

We have:
$cspo(q_1) + switchpo(q_1, c_2) + cs(\theta_{i+1}) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$
$\equiv$ {definition of cs, etc.}
$cspo(q_0) + 1 + switchpo(q_1, c_2) + cs(\theta_{i+1}) \leq cspo(q_0) + 1 + switch(c_1, q_1, c_2) + cs(\theta_{i+1})$
$\equiv \{switch(c_1, q_1, c_2) = 0 \Rightarrow switchpo(q_1, c_2)\}$
$true$

case 7. There are safe transition in q_0, the BCSPO algorithm never reached q_0 with $cspo(q_0)$ and closed a cycle, and in every call of $Search(q_0, cspo(q_0), id, safe)$ c_1 was not explored. But there exists another transition c_k which appears in θ_i after c_1, and c_k was explored by calling, in lines 26-28:
$if\ CONSTRAINT(c_k(q_0), cspo(q_0), pid(c_k), true)\ \{Search(c_k(q_0), cspo(q_0), pid(c_k), true)\}$. Also c_k is the first transition from θ_i that is in $ample(q_0)$. Formally:
$ample(q_0) \neq enabled(q_0) \wedge \neg(\exists id \exists safe.\ cycle(q_0, cspo(q_0), id, safe)) \wedge cspo(q_0) < N \wedge$
$c_1 \notin ample(q_0) \wedge (\exists k.\ 1 < k \leq j.\ c_k \in ample(q_0) \wedge \neg(\exists l.\ 1 < l < k.\ c_l \in ample(q_0)))$

From the conditions satisfied by the ample set, we know that c_k is independent with all the previous transitions in θ_i.

In this case we pick: $\eta_{i+1} = r_0 \xrightarrow{b_1} ... \xrightarrow{b_i} r_i \xrightarrow{c_k} c_k(q_0)$, and
$\theta_{i+1} = c_k(q_0) \xrightarrow{c_2} c_k(q_1) ... \xrightarrow{c_{k-1}} c_k(q_{k-1}) \xrightarrow{c_{k+1}} q_{k+1} ... \xrightarrow{c_j} q_j$.

The construction ensures that $|\eta_{i+1}| = i + 1$, and $|\theta_{i+1}| = |\theta_i| - 1$. So $|\theta_{i+1}| \leq |\theta_i|$.

Because $k > 1$ we must have $|\theta_i| > 1$, so $|\theta_{i+1}| > 0$. So we have $i < m - 1$, and we have to continue with the construction.

Using Lemma 5 we get that $c_k(q_0) \in V_N^{PO}$, so η_{i+1} is a path in G_N^{PO}. Also:
$cspo(c_k(q_0)) \leq cspo(q_0) \wedge (cspo(c_k(q_0)) = cspo(q_0) \Rightarrow search(c_k(q_0), cspo(q_0), pid(c_k), true)))$.

We have two sub cases:

a) $cspo(c_k(q_0)) < cspo(q_0)$, and here:
$cspo(c_k(q_0)) + switchpo(c_k(q_0), c_1) + cs(\theta_{i+1}) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$ because $cs(\theta_{i+1}) \leq cs(\theta_i)$.

b) $cspo(c_k(q_0)) = cspo(q_0)$, and $search(c_k(q_0), cspo(q_0), pid(c_k), true)$, so $switchpo(c_k(q_0), c_1) = 0$.

We get $cspo(c_k(q_0)) + switchpo(c_k(q_0), c_1) + cs(\theta_{i+1}) \leq cspo(q_0) + switchpo(q_0, c_1) + cs(\theta_i)$ because $cs(\theta_{i+1}) \leq cs(\theta_i)$.

case 8. There are safe transition in q_0, the BCSPO algorithm never reached q_0 with $cspo(q_0)$ and closed a cycle, and in every call of $Search(q_0, cspo(q_0), id, safe)$, c_1 was not explored, and none of the transitions in $ample(q_0)$ appears in θ_i. Formally:
$ample(q_0) \neq enabled(q_0) \wedge \neg(\exists id \exists safe.\ cycle(q_0, cspo(q_0), id, safe)) \wedge cspo(q_0) < N \wedge$
$c_1 \notin ample(q_0) \wedge \neg(\exists k.\ 1 < k \leq j.\ c_k \in ample(q_0))$

Because $\mathit{ample}(q_0) \neq \mathit{enabled}(q_0)$, from the conditions imposed on the ample set, we know there exists $d \in \mathit{ample}(q_0)$.

In this case we pick: $\eta_{i+1} = r_0 \xrightarrow{b_1} ... \xrightarrow{b_i} r_i \xrightarrow{d} d(q_0)$, and $\theta_{i+1} = d(q_0) \xrightarrow{c_2} d(q_1) \xrightarrow{c_3} ... \xrightarrow{c_j} d(q_j)$.

The construction ensures that $|\eta_{i+1}| = i + 1$, and $|\theta_{i+1}| = |\theta_i|$. So $|\theta_{i+1}| \leq |\theta_i|$.

Because $|\theta_i| > 0$ (otherwise we would have already stopped with the construction), we also have $|\theta_{i+1}| > 0$. So $i < m - 1$, and we must continue with the construction.

Using Lemma 5 we get that $d(q_0) \in V_N^{PO}$, so η_{i+1} is a path in G_N^{PO}. Also:
$\mathit{cspo}(d(q_0)) \leq \mathit{cspo}(q_0) \wedge (\mathit{cspo}(d(q_0)) = \mathit{cspo}(q_0) \Rightarrow \mathit{search}(d(q_0), \mathit{cspo}(q_0), \mathit{pid}(d), \mathit{true})))$.

We have two sub cases:

a) $\mathit{cspo}(d(q_0)) < \mathit{cspo}(q_0)$, and here
$\mathit{cspo}(d(q_0)) + \mathit{switchpo}(d(q_0), c_1) + \mathit{cs}(\theta_{i+1}) \leq \mathit{cspo}(q_0) + \mathit{switchpo}(q_0, c_1) + \mathit{cs}(\theta_i)$ because $\mathit{cs}(\theta_{i+1}) = \mathit{cs}(\theta_i)$.

b) $\mathit{cspo}(d(q_0)) = \mathit{cspo}(q_0)$, and in this case we have $\mathit{search}(d(q_0), \mathit{cspo}(q_0), \mathit{pid}(d), \mathit{true})$, so $\mathit{switchpo}(d(q_0), c_1) = 0$. We get
$\mathit{cspo}(d(q_0)) + \mathit{switchpo}(d(q_0), c_1) + \mathit{cs}(\theta_{i+1}) \leq \mathit{cspo}(q_0) + \mathit{switchpo}(q_0, c_1) + \mathit{cs}(\theta_i)$ because $\mathit{cs}(\theta_{i+1}) = \mathit{cs}(\theta_i)$.

What is left to prove is that the construction eventually terminates. The only situation in which the construction would not terminate is that in which we apply case 8 infinitely often without applying any other cases in between.

If this were the case, then we would get in η_i an increasing suffix formed just with safe transitions. Because the number of states is finite, this suffix must contain a cycle formed with only safe transitions. By applying Lemma 8 we get that either any state r in the cycle has $\mathit{cspo}(r) = N$, or at least one state r was reached with $\mathit{cspo}(r)$ and a cycle was detected. But this would imply that when r was reached we either had to apply case 3, or we had to apply case 1. We reach a contradiction because for r we are not allowed to apply case 8. So case 8 can be applied only a finite number of times without applying other cases.

Each of the cases, except case 8, decreases the size of θ_i, and so the construction eventually terminates. □

Theorem 2. For a context switch bound N, $N \geq 0$, and for any path $p = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n$ from G with $\mathit{cs}(p) \leq N$, there exists a path $p_N = r_0 \xrightarrow{b_1} r_1 \xrightarrow{b_2} ... \xrightarrow{b_m} r_m$ from G_N^{PO}, the graph constructed by the BCSPO algorithm, with $r_0 = s_0$, $\mathit{cspo}(\mathit{ls}(p_N)) \leq \mathit{cs}(p)$, and p_N stuttering equivalent to p.

Proof. While applying Lemma 9, each of the 8 cases, ensures that for all i, $0 \leq i < m$, π_{i+1} is stuttering equivalent to π_i. Cases 1-4, and 6 don't reorder transitions, so they can't change the order of visible transitions. Case 5, is impossible. Case 7, takes an invisible transition from θ_i and moves it, without changing the order of visible transitions. And case 8, adds an invisible transition without changing the order of visible transitions.

Because for all i, $0 \leq i < m$, π_{i+1} is stuttering equivalent to π_i, we have that π_m is stuttering equivalent to π_0.

Lemma 9 also ensures that $\mathit{cspo}(\pi_m) \leq \mathit{cs}(\pi_0)$. But $\pi_0 = p$, and π_m is a path from G_N^{PO}, so we can pick $p_N = \pi_m$. □

C. Spin model

A Spin model to trigger worst-case behavior with bounded context switching.

```
#define N       10

int count;

active [N] proctype p()
{
        count++; count--
}

never {
        do
        :: assert(count != N)
        od
}
```