Pruning state spaces with extended beam search

M.T. Dashti, A.J. Wijs

REPORT SEN-R0610 NOVEMBER 2006

ISSN 1386-369X

Pruning state spaces with extended beam search

ABSTRACT
State space explosion is still the main problem in the area of model checking. This paper focuses on using beam search, a heuristic search algorithm, for pruning state spaces while generating. Original beam search is adapted to the state space generation setting and two new variants, motivated by some practical case studies, are devised. Remarkably, the resulting framework is shown to encompass *A** search and some partial order reduction algorithms. These beam searches have all been implemented in the µCRL toolset. Case studies and comparisons with SPIN are also presented.

1998 ACM Computing Classification System: D.2.4 [Formal methods, Model checking], F.2.2 [Sequencing and scheduling], I.2.8 [Problem solving, Control methods, and Search]
Keywords and Phrases: Beam search; heuristics; partial order reduction; scheduling; state space generation
Note: The second author carried out this work under the project SEN2 - TIPSy.

Pruning State Spaces with Extended Beam Search

M. Torabi Dashti and A.J. Wijs
CWI, AMSTERDAM
{dashti, wijs}@cwi.nl

ABSTRACT
State space explosion is still the main problem in the area of model checking. This paper focuses on using beam search, a heuristic search algorithm, for pruning state spaces while generating. Original beam search is adapted to the state space generation setting and two new variants, motivated by some practical case studies, are devised. Remarkably, the resulting framework is shown to encompass A^* search and some partial order reduction algorithms. These beam searches have all been implemented in the μCRL toolset. Case studies and comparisons with SPIN are also presented.

1998 ACM Computing Classification System: D.2.4 [Formal methods, Model checking], F.2.2 [Sequencing and scheduling], I.2.8 [Problem solving, Control methods, and Search]
Keywords and Phrases: Beam search, heuristics, partial order reduction, scheduling, state space generation.
Note: The second author carried out this work under the project SEN2 - TIPSy.

Contents

1. Introduction

State space explosion is still the main problem in the area of model checking. Model checking has proved to be very useful in lots of cases, but just as easily one can find system models which yield enormous state spaces, that is, if they can in practice be generated at all. Abstracting away from unnecessary details can sometimes simplify the model and shrink the state space, but one cannot do this at leisure as it should still be possible to verify the desired properties. Another possibility is to use better tools, faster computers with more memory, or move to a distributed setting. But even then, there are very real limits to what can be achieved.

Over the years a number of techniques have emerged to prune, while generating, parts of the state space that are not (or do not seem) promising given the task at hand. Some of these techniques, such as partial order reduction algorithms [5], guarantee that no essential information is lost after pruning. Alternatively, this paper focuses mainly on heuristic pruning methods which heavily reduce the generation time and memory consumption but generate only an approximation to the state space. The idea is that a user-supplied heuristic function guides the generation algorithm such that ideally only relevant parts of the state space are actually explored. This is, in fact, at odds with the core idea of model checking when studying qualitative properties of systems, i.e. to exhaustively search the complete state space to find any corner case bug. However, heuristic pruning techniques can very well target performance analysis problems as approximate answers are usually sufficient when using model checking in quantitative analyses of systems (see, e.g., [3]). A comparison between the two types of pruning can thus not fairly be made since different problems allow different techniques.

In this paper, we investigate how *beam search* can be integrated into the state space generation setting. Beam search (BS) is a heuristic method for combinatorial optimisation problems, which has extensively been studied in artificial intelligence and operations research [7, 10, 15, 25, 19, 22, 30]. BS is similar to breadth-first search as it progresses level by level, but it does not explore all the encountered states. At each level, all the states are evaluated using a heuristic cost (or priority) function but only a fixed number of them is selected for further examination. This aggressive pruning heavily decreases the generation time, but may in general miss essential parts of the state space for the problem at hand, since wrong decisions can be made while pruning. Therefore, BS has so far been mainly used in searching trees with a high density of goal nodes. Scheduling problems, for instance, have been perfect targets for using BS as their goal is to optimally schedule a certain number of jobs and resources, while near-optimal schedules, which densely populate the search space, are in practice good enough, see e.g. [7, 10, 25, 22, 30].

The idea of using BS in state space generation tightly relates to directed model checking (DMC) [8], where heuristics are used to guide the state space exploration when checking qualitative properties. However, we mainly aim for checking quantitative properties, where approximate results are often sufficient. This singles out our work from the existing DMC techniques.

Contributions We motivate and thoroughly discuss adapting the BS techniques to deal with arbitrary structures of state spaces. By this, we stretch the idea of DMC to the field of quantitative analysis. Next, we extend the classic BS in two directions. First, we propose *flexible* BS, which, broadly speaking, does not stick to a fixed number of states to be selected at each search level. This partially mitigates the problem of determining this exact fixed number in advance. Second, we introduce the notion of *synchronised* BS, which aims at separating the heuristic pruning phase from the underlying exploration strategy. Possible combinations of these variants create a spectrum of search algorithms that, as will be described, encompasses some known search techniques such as A^* search and partial order reduction algorithms.

These beam searches have all been implemented in the μCRL [12] state space generation toolset [2]. Experimental results on comparing this toolset with Spin in scheduling are also reported.

Structure of the paper Section 2 introduces some preliminary notions. The classic BS is described in section 3. Section 4 deals with the adaptation of two existing variants of BS to the state space generation setting. There we also propose our extensions to the BS algorithms. Related issues such as memory management and selecting heuristic functions are also discussed there. In section 5, we describe how our framework encompasses A^* search and a partial order reduction algorithm for security protocols. Section 6 presents comparisons between an implementation of these techniques in the μCRL toolset with a SPIN implementation of depth-first branch-and-bound. Section 7 presents our related work and section 8 concludes the paper.

2. Preliminaries

In this section the definition of a labelled transition system and some related notations are recalled. Then breadth-first state space generation is described.

2.1 Labelled transition systems

A Labelled Transition System (LTS) is a tuple (Σ, s_0, Act, Tr), where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, Act is a finite set of action labels and $Tr \subseteq \Sigma \times Act \times S$ is the transition relation. A transition $(s, a, s') \in Tr$, denoted $s \xrightarrow{a} s'$, indicates that the system can move from state s to s' by performing action a. In the rest of the paper, unless explicitly stated, we consider finite LTSs, i.e. LTSs with a finite set of states.

A state s' is called *reachable* from state s iff $s \to^* s'$, where $\to^*$ is the reflexive transitive closure of $\xrightarrow{a}$ for any $a \in Act$. When checking a *reachability* property, one searches for any s such that $s_0 \to^* s$ and $s \in \sigma$, where $\sigma \subseteq \Sigma$ is a set of goal states.

The set of enabled transitions in state s is defined as $en(s) = \{t \in Tr |\ \exists s' \in \Sigma,\ a \in Act.\ t = s \xrightarrow{a} s'\}$. For $T \subseteq Tr$, we define $nx(s,T) = \{s' \in \Sigma |\ \exists a \in Act.\ s \xrightarrow{a} s' \in T\}$.

2.2 Breadth-first state space generation

A state space generation (SSG) algorithm is normally provided with a specification as input and produces the state space that is described by that specification as output. A breadth-first SSG algorithm starts from the initial state of the specification, names it s_0, and runs algorithm 1. This algorithm is guaranteed to terminate when generating finite LTSs. Here, we confine to show the traversal strategy and abstract away from generating the output (file); memory management issues are discussed in a separate section.

Algorithm 1 Breadth-first state space generation

```
$Current := \{s_0\}$
$Expanded := \emptyset$
$Next := \emptyset$
while $Current \setminus Expanded \neq \emptyset$ do
  for all $s \in Current \setminus Expanded$ do
    $Next := Next \cup nx(s, explore(s))$
  end for
  $Expanded := Expanded \cup Current$
  $Current := Next$
  $Next := \emptyset$
end while
```

In algorithm 1, $explore : \Sigma \to \mathscr{P}(Tr)$, where $\mathscr{P}$ is the powerset constructor, is the function that provides the interface between the SSG algorithm and the underlying specification. For a state s, $explore(s)$ is the set of transitions which originate from s, i.e. $explore(s) = en(s)$ [1]. The sets *Current* and *Next* denote the set of states of the current and the next level, respectively. The set *Expanded* contains the set of states that have been expanded.

3. Beam search

Beam search (BS) [23, 17] is a heuristic search algorithm for combinatorial optimisation problems, which was originally used for speech recognition [15] and image understanding [19]. Later on, this technique has been applied to scheduling problems, for example in systems designed for complex job shop [2] environments [10,

[1]The function name *explore* is redundantly used here to highlight the relation between the SSG algorithm and the underlying specifi cation.

[2]The job shop problem is the classic scheduling problem in the literature. In its most basic form, we have a fi nite set M of resources, and a number of jobs $J_1,...,J_n$ which compete in using the resources in a specifi c order and for some time. The problem is to allocate the resources such that all the jobs are fi nished in minimal time.

23, 25, 22]. Since then new variants of BS, such as filtered BS [23, 24] and recovery BS [7, 27], have been introduced.

BS is similar to breadth-first search as it progresses level by level. At each level of the search *tree* (in section 4 we extend BS to handle cycles), it uses a heuristic evaluation function to estimate the promise of encountered nodes [3], while the *goal* is to find a path from the initial state (initial node of the tree) to a leaf node that possesses the minimal evaluation value among all the leafs. In each level, only the β most promising nodes are selected for further examination and the other nodes are permanently discarded. The *beam width* parameter β is fixed to a value before searching starts. Because of this aggressive pruning the generation time is a linear function of β and is thus heavily decreased. However, since wrong decisions can be made while pruning, BS is neither complete, i.e. is not guaranteed to find a solution when there is one, nor optimal, i.e. does not guarantee finding an optimal solution. To limit the possibility of wrong decisions one can increase the beam width, at the cost of increasing the required computational effort and memory usage.

Two types of evaluation functions have traditionally been used for BS [23]: *priority* evaluation functions and *total cost* evaluation functions, which lead to the *priority* and *detailed* BS variants, respectively. In priority BS at each node the evaluation function calculates a priority for each successor node and selects based on those priorities. At the root of the search tree, up to β most promising successors (i.e. those with the highest priorities) are selected, while in each subsequent level only one successor with the highest priority is selected per examined node. Below we quote the basic idea of traditional priority BS from [26]. In this algorithm (and similarly in traditional detailed BS), s_0 is the root of the search tree and all leafs are assumed to be located at the same level.

```
1. Set B = $\emptyset$, C = $\emptyset$
     • Branch $s_0$ generating children
     • Perform priority evaluation of each child node
     • Select min{$\beta$,number of children} best child nodes, add them to B
2. For each node in B:
     • Branch node generating children
     • Perform priority evaluation of each child node
     • Select best child node, add it to C
3. Set B = C; Set C = $\emptyset$
4. Stopping Condition: if all nodes in B are leaf, select node with lowest total cost and stop, otherwise go to step 2.
```

In detailed BS at each node the evaluation function calculates an estimate of the total cost of the best schedule that can be found continuing from the partial schedule represented by the node. At each level up to β most promising nodes (i.e. those with the lowest total cost values) are selected regardless of who their parent nodes are. If there are more than β nodes that receive the best evaluation value, a selection is made based on other criteria, e.g. the order of encountering the nodes (see section 4.4 for other possibilities). Clearly, when $\beta \to \infty$, detailed BS behaves as exhaustive breadth-first search. The following algorithm represents traditional detailed BS [26].

```
1. Set C = $\emptyset$, B = {$s_0$}
2. For each node in B:
     • Branch node generating children
     • Perform detailed evaluation of each child node
     • Select min{$\beta$,number of children} best child nodes, add them to C
3. Set B = $\emptyset$; Select min{$\beta, |C|$} best nodes in C, add them to B; Set C = $\emptyset$
4. Stopping Condition: if all nodes in B are leaf, select node with lowest total cost and stop, otherwise go to step 2.
```

[3] In this section we use the most common terminology when referring to BS, i.e. we reason about nodes and edges, as opposed to states and transitions. This emphasises that we adapt the BS techniques to a different setting.

In comparison, priority evaluation functions have a local view of the problem, since they only consider the next job to be scheduled, while total cost evaluation functions have a more global view, taking the complete schedule into account. In general, total-cost evaluation functions are computationally more expensive than priority evaluation functions, but often provide more accurate heuristics because of their global view [23].

4. Adapting beam search for state space generation

4.1 Motivation

BS is typically applied on highly structured search trees, which contain all possible orderings of a given number of jobs, e.g. see [16, 27]. Such a search tree starts with n jobs to be scheduled, which means that the root of the tree has n outgoing transitions. Every node has exactly $n-k$ outgoing transitions, where k is the level in the tree where the node appears. State spaces, however, supposedly contain information on all possible behaviours of a system. Therefore, they may contain cycles and have more complex structures than the well-structured search trees usually subjected to BS. This necessitates modifying the BS techniques to deal with arbitrary structures of state spaces. Moreover, the BS algorithms search for a particular state in the search space, while in (and after) generating state spaces one might desire to study a property beyond simple reachability (see section 5 on partial order reduction as an instance of extended BS). We therefore extend BS to a state space *generation* setting, as opposed to its traditional setting that focuses only on *searching*. The notion of a particular "goal" (c.f. 3) is thus removed from the adapted BS (see section 4.6 for possible optimisations when restricting BS to verify reachability properties). This along with the necessary machinery for handling cycles raise memory management issues in BS, to which we will return in section 4.6.

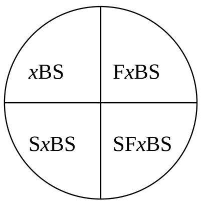

Figure 1: BS spectrum, $x \in \{\mathrm{D},\mathrm{P}\}$

Below, we first revisit priority and detailed BS for SSG. Next, we propose two variants of BS which have, in our case studies, proved essential for handling large state spaces. Flexible BS mitigates the problem of determining a sufficiently large beam width, while synchronised BS separates the pruning phase from the exploration strategy.

Figure 1 shows the spectrum of the variants that are described in the following sections. There, DBS and PBS correspond to the traditional detailed and priority beam searches, respectively, extended to deal with arbitrary state spaces (sections 4.2 and 4.3). The F and S prefixes refer to the flexible and synchronised variants respectively (sections 4.4 and 4.5).

4.2 Priority beam search for state space generation

Below we motivate and describe the changes that we have made to the traditional priority BS to deal with SSG.

PBS is shown in algorithm 2. There, the function $priority : Act \to \mathbb{Z}$ provides the priority of actions, as opposed to states [4]. We motivate this deviation by noting that *jobs* in the BS terminology correspond more naturally with *actions* in LTSs. Moreover, since PBS focuses on generating an approximate state space rather than looking for a particular goal, no total cost function is in general needed (and provided). The set *Buffer* temporarily keeps seemingly promising transitions. The function $prio_{min} : \mathscr{P}(Tr) \to \mathbb{Z}$ returns the lowest priority of the actions of a given set of transitions. We define $prio_{min}(\emptyset) = -\infty$. The function $getprio_{min} : \mathscr{P}(Tr) \to Tr$, given a set of transitions, returns one of the transitions having an action with the lowest priority. Note that the stopping condition of the traditional priority BS algorithm of section 3 is represented here by condition $Current \setminus Expanded \neq \emptyset$ of the **while** loop, which does not assume that all leafs occur in the same level (this, moreover, avoids cycles). The algorithm terminates when it has explored all the states in its beam.

In priority BS, originally, up to β children of the root are selected. The resulting beam of width β is then maintained by sprouting only one child per node in subsequent levels. In state spaces, however, the root has typically much less outgoing transitions than the average branching factor of the state space. Fixing the beam width at such an early stage is therefore not reasonable. Selecting all transitions at each levels until β or more transitions are found in a single level would be an option. However, if this number drastically exceeds β, it would not be clear which transitions should be pruned away. To mitigate this problem, instead of β, algorithm 2

[4]In general, *priority* can also depend on states: $priority : \Sigma \to Act \to \mathbb{Z}$. In this paper, we stick to fi xed priorities, which resembles the *dispatch* scheduling strategy in AI terminology [23], and leave the more general case to the future work.

is provided with the pair (α, l), where $\alpha, l \in \mathbb{N}$ and $\alpha^l = \beta$. The idea is that the algorithm uses the *priority* function to prune non-promising states from the very first level, but in two phases: before reaching nearly β states in a single level it consider the most promising α transitions for further expansion, but after that, it sticks to the original one child per node rule.

Algorithm 2 Priority BS for state space generation

```
$Current := \{s_0\}$
$Expanded := \emptyset$
$Next := \emptyset$
$Buffer := \emptyset$
$level := 0$
$limit := \alpha$
while $Current \setminus Expanded \neq \emptyset$ do
  for all $s \in Current \setminus Expanded$ do
    for all $s \xrightarrow{a} s' \in explore(s)$ do
      if $priority(a) > prio_{min}(Buffer)$ then
        if $|Buffer| = limit$ then
          $Buffer := Buffer \setminus \{getprio_{min}(Buffer)\}$
        end if
        $Buffer := Buffer \cup \{s \xrightarrow{a} s'\}$
      end if
    end for
    $Next := Next \cup nx(s, Buffer)$
    $Buffer := \emptyset$
  end for
  $Expanded := Expanded \cup Current$
  $Current := Next$
  $Next := \emptyset$
  $level := level + 1$
  if $level = l$ then
    $limit := 1$
  end if
end while
```

4.3 Detailed beam search for state space generation

The original idea of detailed BS does not need to change much to fit into the SSG setting except for when handling cycles. When exploring a cyclic LTS, to guarantee the termination of the algorithm, it is necessary to store the set of explored states (in the set *Expanded* in algorithms 1 and 2) to avoid exploring a state more than once. However, if a state is reached via a path with a lower cost, the state has to be re-examined. This is because the estimated total cost of each state depends on the cost to reach that state from the root. Thus a state (and subsequently its successors) may more competently qualify for further explorations if it is reached via a lower cost path. This is further detailed below.

We observe that the average running time of the traditional detailed BS algorithm of section 3 can be reduced if the order of exploration and evaluation is reversed. The merit of this reversal is that, since the number of nodes to be evaluated is a priori known in each level, evaluation of the states of a level containing no more than β states can altogether be avoided [5]. Besides that, to reduce the space complexity of the traditional detailed BS algorithm of section 3, while evaluating, only the β most promising states up to then can be kept in a set and the rest can be discarded (note that β^2 states are stored in the algorithm of section 3).

[5]Actually the evaluation of the heuristic estimation part, which is computationally the most expensive phase, is the part that is avoided. See algorithm 3 for details.

Algorithm 3 shows detailed BS after the mentioned optimisations. The total cost evaluation function is called $f : \Sigma \to \mathbb{N}$. This function is decomposed into $f(s) = g(s) + h(s)$. The $g(s)$ function represents the cost taken to reach s from the root of the tree, which is defined as $g(s) = g(s') + cost(a)$ if $s' \xrightarrow{a} s$. The function $cost : Act \to \mathbb{N}$ assigns weights to actions that can, for instance, denote the time needed to perform different jobs in a scheduling problem. These weights are fixed before searching starts. Since the range of *cost* is non-negative numbers, if $s \to^* s'$ then $g(s') \geq g(s)$. Therefore, the values of g never decreases along a path. The $h(s)$ function is an estimation of the cost it would take to efficiently complete the schedule continuing from s. The f function is called *monotonic* iff $s \to^* s'$ then $f(s) \leq f(s')$. The function $getf_{max} : \mathscr{P}(\Sigma) \to \Sigma$, given a set of states, returns one of the states that has the highest f value. The sets *Current*, *Next* and *Expanded* contain pairs of states and corresponding g values, i.e. $\langle s, s.g \rangle$. Here $unify(X)$ and $update(X,Y)$ are defined as follows: $unify(X) = \{\langle s,g \rangle \in X \mid \forall g'. \langle s,g' \rangle \in X \Rightarrow g \leq g'\}$ and $update(X,Y) = \{\langle s,g \rangle \in X \mid \neg \exists g' \leq g.\ \langle s,g' \rangle \in Y\}$. Note that a state will be revisited only if it is reached via a path with a lower cost than the g cost assigned to it. This algorithm obviously behaves as breadth-first search when $\beta \to \infty$.

Algorithm 3 Detailed BS for state space generation

```
$s_0.g = 0$
$Current := \{\langle s_0, s_0.g \rangle\}$
$Next := \emptyset$
$Expanded := \emptyset$
while $Current \neq \emptyset$ do
  while $|Current| > \beta$ do
    $Current := Current \setminus \{getf_{max}(Current)\}$
  end while
  for all $s \in Current$ do
    for all $s \xrightarrow{a} s' \in explore(s)$ do
      $s'.g := s.g + cost(a)$
      $Next := Next \cup \{\langle s', s'.g \rangle\}$
    end for
  end for
  $Expanded := unify(Expanded \cup Current)$
  $Current := update(unify(Next), Expanded)$
  $Next := \emptyset$
end while
```

4.4 Flexible beam search

A major issue that still remains unaddressed in the BS adaptations of sections 4.2 and 4.3 is how equally competent candidates are pruned. Actions in LTSs can have several parameters. The same action can thus appear multiple times as an outgoing transition of a given state, each time having different parameter values, possibly leading to equally competent states. This potentially leads to situations where, during selection, a large number of transitions or states have equal evaluations (for some examples see [30]). In such cases, a selection has to be made among these equally competent candidates if they happen to be (one of) the most promising transitions or among the β-best states. These selections are beyond the influence of the evaluation (or priority) function and can undesirably make the algorithm non-deterministic. We, therefore, propose two variants of BS that we call *flexible detailed* and *flexible priority* beam searches, in which the beam width can change during state space generation.

In flexible detailed BS, at each level, up to β most promising states are selected plus any other state which is as competent as the worst member of these β states. This achieves closure on the worst (i.e. highest) total cost value being selected. Similarly, in flexible priority BS, in the first l levels (see section 4.2), at each state, up to α most promising outgoing transitions are selected plus any transition which has the same priority as the least competent member of these α transitions. At the $l+1^{\text{th}}$ level and onwards, at each state, all the transitions with the same priority as the most promising transition of that particular state are selected (i.e. $\alpha = 1$). Note that in FPBS, if the beam width is increased, it never returns to the intended β, while it can be readjusted to β in each level of FDBS.

4.5 Synchronised beam search

As is described in section 3, the classic BS algorithms were tailored for the breadth-first exploration strategy. Below, we explain a way to do BS on top of best-first [20] exploration algorithms. Broadly speaking, we separate the exploration strategy from the pruning phase, so that the exploration is guided with a (possibly different) heuristic function. This is particularly useful when checking reachability properties on-the-fly.

Below, we inductively describe *G-synchronised* xBS, where $G : \Sigma \rightarrow \mathbb{N}$ is the function that guides the exploration and $x \in \{\mathrm{D}, \mathrm{P}, \mathrm{FD}, \mathrm{FP}\}$ (denoting the four BS variants described previously). Let $\hat{S}_i$ denote the set of states to be explored at round i. We partition this set into equivalence classes $c_0, \cdots, c_n$, where $n \in \mathbb{N}$, such that $\hat{S}_i = c_0 \cup \cdots \cup c_n$ and $\forall s \in \hat{S}_i.\ s \in c_j \iff G(s) = j$. The pruning algorithm xBS is subsequently applied only on c_k where $c_k \neq \emptyset\ \wedge\ \forall j < k.\ c_j = \emptyset$. According to the pruning algorithm (which can possibly employ an evaluation function different [6] from G), some of the successors of c_k are selected, constituting the set $\hat{S}$. The next round starts with $\hat{S}_{i+1} = \hat{S} \cup \hat{S}_i \setminus c_k$. Modular implementation of synchronised BS variants can thus be conceived: a second algorithm takes care of distributing states over the equivalence classes, before pruning takes place (see appendix I for SDBS in pseudo-code). Using any constant function as G would clearly result in BS with breadth-first exploration strategy.

To mention a practically interesting candidate for G, we temporarily deviate from our general setting. Consider the problem of finding a path of minimal cost that leads to a particular action in a state space. This problem arises, for instance, in finding minimal time schedules (see, e.g., [30]). As another application, if for every action the *cost* function is set to 1, this problem corresponds to finding the minimal length trail when verifying a reachability property. Recall that the total cost function in DBS can be decomposed into $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the trace leading from the root to s. If $G(s) = g(s)$ in G-synchronised DBS, once a goal state (or a complete schedule) is found, searching can safely terminate. This is because in a goal state s, $f(s) = g(s)$ and since the algorithm always follows paths with minimal g (remember that g is monotonic), state s is reached before another state s' iff $g(s) \leq g(s')$. Note that here no state is re-explored, because states with minimal g are taken first and thus a state can be reached again only via paths with higher costs (c.f. section 4.3). Both g-synchronised DBS and g-synchronised PBS have been used in solving timed scheduling problems in [30], where minimal-time traces to a particular action label are searched for. The same pruning algorithm can be used to search for other kinds of traces, such as a shortest trace or a shortest minimal-time trace.

4.6 Memory management

Memory management is a challenging issue in SSG. Although BS reduces memory usage due to cutting away parts of the state space, still explored states need to be accessed to guarantee the termination of the exploration in case of cyclic LTSs. Keeping the whole set of visited states in the memory is usually susceptible to early state space explosion. This can be counter-measured by taking into account specific characteristics of the problem at hand and the properties that are to be checked. Below we discuss some possible optimisations when applying BS:

- When aiming at a reachability property (such as reachability of a goal state, checking invariants and hunting deadlock states), checking the property can easily be embedded in the exploration algorithm, so that once a state satisfying the desired property is reached the search terminates and the witness trace is reported (see, e.g., section 4.5). This however cannot be extended to arbitrary properties.

- If there are no cycles in the state space, there is in principle no need to check whether a state has already been visited. Therefore, only the states from the current level need to be kept and the rest can be removed

[6]Using different functions for guiding exploration and pruning in principle allows dealing with multi-priced optimisation problems, c.f. [1].

from memory [7], i.e. flushed to high latency media such as disks. Prominent examples of systems with acyclic LTSs are large classes of scheduling problems, which has been a traditional target of BS, and most security protocols (see section 5).

- If in DBS variants each state has a unique g cost associated to it, e.g. denoting a notion of progress, since g is monotonic, there cannot be any transition from states with higher g to the states with lower g values: $g(s) < g(s') \Rightarrow s \not\to^* s'$. Consequently, states with costs strictly lower than the cost of the states to be processed can be removed from memory. This resembles sweep-line state exploration [4].

- In G-synchronised BS variants with monotonic G function, bit-state hashing [13] can be used to reduce the memory usage. This technique is however inherently incomplete, i.e. may miss exploring parts of the state space, and in particular when used in BS there is the possibility of ignoring a previously visited state when it is reached via a path with a lower cost. We observe that in G-synchronised BS variants with monotonic G, if a visited state is reached again, it will be reached via a more costly path, eliminating the mentioned problem. Note that the approach still remains an approximation to BS and can therefore be seen as a trade-off between memory usage and having a tight grip on pruning.

4.7 Heuristic functions and selecting the beam width

Effectiveness of BS hinges on selecting good heuristic functions. Heuristic functions, as George Polya put in 1945, are meant “to discover the solution to the present problem” [18], and thus heavily depend on the problem being solved. Heuristics constitute a whole separate body of research and, here, we only refer to a few case studies on using heuristics in pruning state spaces: Some heuristic functions for pruning search spaces of typical job-shop scheduling problems are presented in [23, 28]. Similarly, [16] and [30] present detailed discussions on pruning heuristics when dealing with the scheduling problem of a wafer stepper machine and a chemical analyser, respectively.

Selecting the beam width β is another challenge in using BS. The beam width intuitively calibrates the time/memory usage of the algorithm on one hand and the accuracy of the results on the other hand. Therefore, in practice the time/memory limits of a particular experiment determine β. However, to reduce the possibility of surprises, such as getting astonishingly better results with $\beta = 101$ than with $\beta = 100$, we recommend using flexible BS variants. This, however, comes at the price of losing a tight grip on the memory consumption (see also section 4.4). For more discussions on selecting β and its relation to the quality of answer we refer to [23].

5. CONNECTIONS TO OTHER SEARCH ALGORITHMS

Having described the BS spectrum of figure 1, we observe that some existing search techniques fit neatly in there. We here briefly discuss some of these connections to other search algorithms. This can particularly be interesting in practice where one looks for umbrella theories and tools to cover as many existing techniques as possible to ease their use and interoperability.

First, we observe that the basic AI search algorithm A^* [20] can be seen as an instance of SFDBS, the bottom right corner of the spectrum of figure 1.

Lemma 1. *Given a monotonic total cost function f, f-synchronised flexible detailed beam search, with $\beta > 0$, behaves as A^*.*

Proof. See appendix I. □

Second, uniform cost search [20] can be seen as an instance of SDBS, the bottom left corner of the spectrum. (The claim is fairly obvious, hence we omit the proof.)

Lemma 2. *When $\beta \to \infty$, g-synchronised detailed BS behaves as uniform cost search.*

Third, we note that the partial order reduction (POR) algorithm of [6] for security protocols can be seen as an instance of FPBS. The main principle of POR is to exploit the commutativity of concurrently executed

[7] In this case, some states may be revisited, hence undesirably increasing the search time.

transitions in order to generate only a sufficient fraction of the state space by exploring a subset of enabled transitions $ample(s) \subseteq en(s)$ at each state s. This resembles priority BS since at each state, based on the suitability of the enabled transitions, some of the successors are pruned away while generating. However, in contrast to priority BS, no essential information is lost in POR as the *ample* set is selected such that a certain class of desired properties is preserved. We refer to [5] for a general introduction to POR. Here we observe that the POR algorithm of [6] partly behaves as FPBS and partly as PBS. Due to space constraints, we refer to appendix II for a translation from this algorithm to our pruning framework.

6. Experimental results

In this section we report our experimental results [8] on solving the *Cannibals and missionaries* problem [14]. The problem can informally be described as follows [9]: C missionaries and C cannibals stand on the left bank of a river that they wish to cross. There is a boat available which can ferry up to B people across. The goal is to find a schedule for ferrying all the cannibals and all the missionaries safely across, i.e. never on a shore on in the boat the cannibals outnumber the missionaries. We use a μCRL implementation of BS and a Spin implementation of the depth-first branch-and-bound algorithm to solve this problem. Below, we describe the used techniques and discuss the results, shown in table 1. These experiments have been performed on a single machine with a 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running SUSE Linux 9.2.

In μCRL we first applied the minimal cost search (that is an implementation of uniform cost search of lemma 2, detailed in [30]), denoted MCS in table 1. Second, we used g-SFDBS with $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s)\rangle.(2 \times C))$ as the heuristic part of DBS, where $C(s)$ and $M(s)$ are the numbers of cannibals and missionaries on the left bank in state s, respectively, and $\langle C(s) \neq M(s)\rangle$ is a Boolean expression returning 1 if $C(s) \neq M(s)$, and 0 otherwise. The intuition behind this heuristic is that, first, we want to minimise $C(s)$ and $M(s)$, hence the first part of the function. Second, we support having an equal number of cannibals and missionaries on the left bank as an easy way to avoid deadlock states where $C(s) > M(s)$. The second part of the function puts an extra penalty on states where these numbers are not equal. Our experiments showed that in practice there so many unsuccessful termination states in the model that some deadlock avoidance in the heuristic function is unavoidable. In these experiments, BS proved to be applicable using a fairly stable (flexible) beam width of 20, which partially shows the suitability of the heuristic used.

In our Spin experiments we followed the technique of [21]. The idea is that the LTL formula that is checked is modified during verification to reflect the best solution found so far. This can effectively implement a branch-and-bound mechanism, denoted DFS BnB Prop in table 1, in Spin. In these experiments the LTL property is $\diamond(q \geq U)$, where q denotes the total cost of a path and U denotes an upper bound on this cost that is set by the user. In the standard DFS, U is fixed during the search, while in the BnB search a C-code in the model updates U to the best (lowest) upper bound found in the search. The search through a trace stops only when either the property does not hold or a deadlock state is found. In the latter case, we have slightly adjusted the technique of [21] to deal with unsuccessful terminations. Hence, the current best cost is updated only on successful terminations. Contrary to the results of [21], here the DFS BnB technique does not prune much, compared to standard DFS. This can be due to the differences in the nature of the problems that studied. We believe that in our case there are many more possible schedules with very long traces, in comparison with TSP analysed in [21].

As a final note, the numbers of states of the μCRL experiments in table 1 are not meant to be compared with the corresponding numbers in the Spin experiments, since the tools seem to count states in completely different ways. The numbers can however be used to reason about different techniques of each particular tool.

7. Related work

In the related literature on BS, the emphasis is typically put on how BS is useful for obtaining a solution to a specific problem and no general modelling framework is presented, for instance [16, 26]. Reusing their implementation of BS on other case studies is therefore not straightforward. We provide a general framework,

[8] See `http://www.cwi.nl/~wijs/TIPSy` for a complete report.

[9] See, e.g., `http://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem`.

Table 1: Experimental results. n.s.: No solution exists [14]; o.o.t.: out of time (set to 12 hours); o.o.m.: out of memory (set to 900 MB)

Problem	Result	μCRL MCS		μCRL g-SFDBS				SPIN DFS		SPIN DFS BnB Prop.	
(C,B)	T	# States	Time	T	β	# States	Time	# States	Time	# States	Time
(3,2)	18	147	00:03.80	18	3	142	00:03.73	28,535	00:00.32	26,062	00:00.29
(10,3)	n.s.	396	00:03.94	n.s.	10	396	00:03.90	76,432	00:00.42	76,432	00:00.41
(10,4)	44	1,378	00:04.38	46	10	1,129	00:04.42	253,815	00:01.60	251,400	00:01.53
(20,4)	104	2,537	00:05.32	106	10	2,191	00:05.38	445,801	00:02.66	408,053	00:02.34
(50,10)	142	25,868	00:11.22	148	10	8,035	00:08.47	3,703,900	00:27.33	3,472,070	00:23.69
(50,20)	116	90,355	00:20.15	120	15	17,361	00:11.45	12,647,000	02:05.25	12,060,300	01:49.59
(100,10)	292	49,141	00:19.65	296	10	16,274	00:14.46	14,709,600	02:49.32	13,849,300	02:23.34
(100,30)	222	366,608	01:05.79	228	15	61,380	00:32.06	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(300,10)	892	143,549	01:01.94	896	10	49,514	00:48.47	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(300,30)	680	1,008,436	04:10.72	684	15	205,556	02:30.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,50)	1,076	4,365,536	21:40.52	1,080	20	685,293	10:33.28	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,100)	1,036	17,248,979	77:16.36	1,040	20	1,170,242	16:47.10	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(1000,50)	2,160	8,551,996	70:00.14	2,168	20	1,397,100	37:02.03	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(1000,250)	o.o.t.	o.o.t.	o.o.t.	2,032	20	5,317,561	240:22.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.

based on an expressive specification language, instead of case-based tools. The expressive language allows describing complex systems and various problem restrictions, e.g. see [30].

The depth-first branch-and-bound technique used for scheduling in SPIN [21] can at best be compared with *g*-SFDBS, since both searches avoid exhaustively searching the state space. These however achieve their goals very differently, one bounding in the depth of the state space, the other in the breadth. Moreover, the branch-and-bound guarantees finding optimal solutions, while *g*-SFDBS uses heuristics and does therefore not make this guarantee. See section 6 for more details.

Concerning scheduling using UPPAAL, quite some research has been done, leading to the UPPAAL CORA tool. In several papers by Behrmann et al. (see [1] and references therein), linearly priced timed automata are introduced as an extension of timed automata with prices on both transitions and locations. They deal with reachability analysis using the standard branch-and-bound algorithm. A number of basic exploration techniques can be used for branching, and bounding is done based on heuristics.

Our work is also related to the body of research on DMC. Using A^* [9] and using genetic algorithms to guide the search [11] are among notable works in this field. DMC uses heuristics to guide the search in finding a counter-example to a functional property (belonging to LTL in [9, 11]) with a minimal exploration of the state space. In contrast, we generate an approximation to the state space on which an arbitrary property can be checked afterwards (the result would of course not be exact, hence being useful mainly in quantitative analyses). Nonetheless, there are strong similarities as well: As is described in section 4.5, A^* search can be seen as an instantiation of BS. Genetic algorithms have been used in [11] to guide the search toward a goal state when hunting deadlocks and checking assertions. This approach is similar to ours as the algorithm is in general not guaranteed to explore the whole state space.

8. CONCLUSIONS

In this paper, we extended and made available an existing search technique to be used for system verification and our case studies indeed showed the usefulness and flexibility of the method. We observed that BS can be tuned to encompass some other (heuristic) search algorithms, thus providing a generic method, paving the path towards a more compact, yet flexible, state space generation framework.

Future work BS can in principle deal with infinite state spaces given that the heuristic function does not cut away all finite solutions. The termination condition then needs to adapt as well. This however has yet to be investigated. We are currently working on a distributed implementation of the BS variants proposed in this paper (see [29] for its theoretical setting) .

Bibliography

[1] G. Behrmann, K.G. Larsen, and J.I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

[2] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 250–254, 2001.

[3] E. Brinksma, H. Hermanns, and J.P. Katoen, editors. *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science*, volume 2090 of *LNCS*. Springer, 2001.

[4] S. Christensen, L.M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *TACAS'01*, pages 450–464. Springer, 2001.

[5] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[6] E.M. Clarke, S. Jha, and W. Marrero. Partial order reductions for security protocol verification. In *TACAS'00*, volume 1785 of *LNCS*, 2000.

[7] F. Della Croce and V. T'kindt. A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *J. of the Operational Research Society*, 53:1275–1280, 2002.

[8] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Int. J. Softw. Tools Technol. Transf.*, 5(2):247–267, 2004.

[9] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2-3):247–267, 2004.

[10] M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, CMU, 1983.

[11] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *TACAS'02*, volume 2280 of *LNCS*, pages 266–280. Springer, 2002.

[12] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[13] G.J. Holzmann. An analysis of bitstate hashing. *Form. Methods Syst. Des.*, 13(3):289–307, 1998.

[14] R. Lim. Cannibals and missionaries. In *APL '92*, pages 135–142, New York, NY, USA, 1992. ACM Press.

[15] B.T. Lowerre. *The HARPY speech recognition system*. PhD thesis, CMU, 1976.

[16] S. Oechsner and O. Rose. Scheduling cluster tools using filtered beam search and recipe comparison. In *Proc. 2005 Winter Simulation Conference*, pages 2203–2210. IEEE, 2005.

[17] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, 1995.

[18] G. Polya. *How to solve it*. Princeton University press, 2nd edition, 1945.

[19] S. Rubin. *The ARGOS Image Understanding System*. PhD thesis, CMU, 1978.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[21] T.C. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *SPIN'03*, volume 2648 of *LNCS*, pages 1–17, 2003.

[22] I. Sabuncuoglu and M. Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118:390–412, 1999.

[23] P. Si Ow and E.T. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:35–62, 1988.

[24] P. Si Ow and E.T. Morton. The single machine early/tardy problem. *Management Science*, 35:177–191, 1989.

[25] P. Si Ow and S.F. Smith. Viewing scheduling as an opportunistic problem-solving process. *Annals of Operations Research*, 12:85–108, 1988.

[26] J. Valente and R. Alves. Beam search algorithms for the single machine total weighted tardiness scheduling problem with sequence-dependent setups. Working Paper 186, Faculdade de Economia do Porto, 2005.

[27] J. Valente and R. Alves. Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. *Comput. Ind. Eng.*, 48(2):363–375, 2005.

[28] J. Valente and R. Alves. Improved heuristics for the early/tardy scheduling problem with no idle time. *Computers & OR*, 32:557–569, 2005.

[29] A.J. Wijs and B. Lisser. Distributed Enhanced Beam Search for Quantitative Model Checking. To appear in Proc. MoChArtIV, `http://www.cwi.nl/~wijs/dbsearch.pdf`, 2006.

[30] A.J. Wijs, J.C. van de Pol, and E. Bortnik. Solving Scheduling Problems by Untimed Model Checking. In *Proc. FMICS'05*, pages 54–61. ACM Press, 2005. Extended version as CWI technical report SEN-R0608, `http://db.cwi.nl/rapporten/abstract.php?abstractnr=2034`.

Appendix I
Beam search versus A^* search

In this section we establish an equality between an instance of BS and A^* search. For a description of A^* search we refer to [9, 20]. We refer to algorithm 4 for synchronised detailed beam search in pseudo-code.

Lemma 1. *Given a monotonic total cost function f, f-synchronised flexible detailed beam search, with $\beta > 0$, behaves as A^*.*

Proof. In this proof, f-synchronised flexible detailed beam search, with $\beta > 0$, is called F^* search, to save some space.

First, to observe that A^* and F^* are very similar in spirit, assume that f is increasing (as a special case of being monotonic), i.e. $s \to s' \Rightarrow f(s) < f(s')$. The major difference between A^* and F^* is that in F^* all the states of *Current* set with the minimal f are collected in c_i, for some i, and all expanded in one go (i.e. nothing is pruned, since the search is flexible and the members of c_i all have the same f value), while in A^* they are expanded one by one. However, if f is increasing, all the children of these states (with minimal f) will have an f value higher than their parents. Therefore, A^* will do exactly what F^* does, i.e. it will first explore the members of c_i before considering any other state. Below, we describe the case where f is monotonic, but not necessarily increasing.

We present an inductive proof. Clearly at the root of the search graph these algorithms behave the same. Now assume that up to n^{th} round they are equivalent. Below we prove that they exhibit the same behaviour in the $n+1^{th}$ round.

Let c^0 be the class of states that have the minimal f in *Current* . The set of states expanded by A^* before considering the elements of $Current \setminus c^0$ can be characterised as $\overline{c} = \{s'|\ \exists s \in c^0.\ s \to^* s' \ \wedge\ f(s') = f(s)\}$. The set of states expanded by F^*, before incorporating the result back into *Current*, is clearly c^0. As described earlier, if f is strictly increasing, then $\overline{c} = c^0$ and thus A^* and F^* behave similarly. Let $c^1 = \{s'|\ \exists s \in c^0, a \in Act.\ s \xrightarrow{a} s' \ \wedge\ f(s') = f(s)\}$. Note that $c^1 \neq \emptyset$ only if f is monotonic but not strictly increasing. Observe that in F^*, $c^1 \subseteq unify(Next \cup Current)$, c.f. algorithm 4. This is because

I Clearly, $c^1 \subseteq Next$.

II *Current* is rewritten with $Current \setminus c^0$ before this unification step. Therefore, when *unify*ing, $\forall s \in c^1, s' \in Current.\ s = s' \implies s.g \leq s'.g$.

The set $c^1 \setminus c^0$, if non-empty, will thus be selected and expanded in the next round of F^*. Note that in both F^* (due to *update*) and A^*, $c^1 \cap c^0$ is not re-explored. Let $c^j = \{s'|\ \exists s \in c^{j-1}, a \in Act.\ s \xrightarrow{a} s' \ \wedge\ f(s') = f(s)\}$

Algorithm 4 Synchronised detailed BS for state space generation

```
$s_0.g = 0$
$Current := \{\langle s_0, s_0.g\rangle\}$
$Next := \emptyset$
$Expanded := \emptyset$
while $Current \neq \emptyset$ do
  $i := -1$
  $found := \mathsf{F}$
  while $found = \mathsf{F}$ do
    $i := i+1$
    $c_i := \{\langle s, s.g\rangle \in Current \mid G(s) = i\}$
    if $c_i \neq \emptyset$ then
      $found := \mathsf{T}$
      $Current := Current \setminus c_i$
    end if
  end while
  while $|c_i| > \beta$ do
    $c_i := c_i \setminus \{get\, f_{max}(c_i)\}$
  end while
  for all $s \in c_i$ do
    for all $s \xrightarrow{a} s' \in explore(s)$ do
      $s'.g := s.g + cost(a)$
      $Next := Next \cup \{\langle s', s'.g\rangle\}$
    end for
  end for
  $Expanded := unify(Expanded \cup c_i)$
  $Current := update(unify(Next \cup Current), Expanded)$
  $Next := \emptyset$
end while
```

for $j > 0$. The argument above can be repeated to show that while $c^j \setminus (\cup_{k=0,\cdots,j-1} c^k) \neq \emptyset$, this set will be the the set that is expanded by F^* before the elements of $Current \setminus c^0$ are considered. Noting that $\overline{c} = \cup_{k\geq 0} c^k$ completes the proof. □

Appendix II
Partial order reduction as BS

In this section, we describe an implementation of the POR algorithm of [6] based on the priority BS framework. This is a POR algorithm tailored for verifying reachability properties of security protocols, such as authentication and key distribution protocols [6].

We model a security protocol as an asynchronous composition of a finite number of acyclic deterministic named processes. These processes model roles of honest participants of the protocol. We consider the Dolev-Yao (DY) model (see [6]) as the attacker, which is also modelled as a process.

Processes communicate by sending and receiving messages. A message is a pair $m = (p, c)$, where p is the identity of the intended receiver process and c is the content of the message. We let Msg be the set of all messages that can be communicated. (For a formalisation of Msg see, e.g., [6].) To model the DY intruder, which has complete control over communication media, we assume it plays the role of the communication media. All messages are thus channelled through the intruder. To send or receive a message m, a participant p performs the actions $\mathbf{send}(p,m)$ or $\mathbf{recv}(p,m)$, respectively. Even though process p sends the message m with the intention that it should be received by process q, i.e. $m = (q, c)$, it is in fact the intruder that receives the message from p, and it is from the intruder that q can receive m. The communication between participants of a protocol, via the DY intruder, is thus asynchronous and a participant has no guarantee about the origin of the messages it receives. Apart from **send** and **recv**, all other actions of processes are assumed *internal*, i.e. not communicating with the intruder. These are symbolic actions which typically denote security claims of protocol participants and no semantics is associated with them. An internal action is called *invisible* if it does not appear in the property being verified. Else, it is called *visible*.

In [6] it is observed that the knowledge of the DY intruder is non-decreasing and with more knowledge more states are reachable for the intruder. Intuitively it means that when verifying reachability properties **send** actions can be prioritised over other actions. This is the heart of the POR algorithm proposed in [6]. The set of transitions to explore at each state s, i.e. $ample(s)$, is chosen in [6] as the following:

- If $en(s)$ contains (a transition with) an invisible internal action, then $ample(s)$ is a singleton containing an arbitrary invisible action picked from $en(s)$.
- Suppose $en(s)$ does not contain an invisible internal action, but does contain a **send** action. In this case $ample(s)$ is an arbitrary **send** action picked from $en(s)$.
- If $en(s)$ does not contain an invisible action or a **send** action, $ample(s) = en(s)$.

To specify the requirements of security protocols, in [6], a first order logic where quantifiers range over protocol participants, augmented with past time modal operator is considered. Their POR algorithm is shown to preserve formula of this logic.

Although Clarke et al. embed this reduction into a depth-first search algorithm, one can equally modify the breadth-first SSG of algorithm 1 so that at each state s only the members of $ample(s)$ are explored. Below we describe how this algorithm can be implemented in the priority beam search framework of section 4.2. We consider a protocol P comprising a finite set of honest processes, denoted P. The set of actions available to these processes is denoted Act. As honest processes communicate only via the intruder, Act can be divided into the following disjoint sets [1]:

A_0: The set of visible internal actions $\mathbf{v}(p,m)$, for some $p \in P$ and $m \in Msg$.

A_1: The set of invisible internal actions $\mathbf{i}(p,m)$, for some $p \in P$ and $m \in Msg$.

A_2: The set of $\mathbf{send}(p,m)$ actions, for some $p \in P$ and $m \in Msg$.

A_3: The set of $\mathbf{recv}(p,m)$ actions, for some $p \in P$ and $m \in Msg$.

We define $\hat{A}_i : \mathscr{P}(Tr) \to \mathscr{P}(Tr)$, for $i \in \{0,1,2,3\}$, such that $\hat{A}_i(T) = \{s \xrightarrow{a} s' \in T \mid a \in \mathrm{A}_i\}$. Four priority levels are assigned to these classes of actions: p_0 to A0, p_1 to A1, etc. Now, at each state s, if $\hat{A}_1(en(s)) \neq \emptyset$, then $ample(s) = t$, where t is an arbitrary member of $\hat{A}_1(en(s))$. Else, if $\hat{A}_2(en(s)) \neq \emptyset$, then $ample(s) = t$, where t is an arbitrary member of $\hat{A}_2(en(s))$. If both these conditions fail to hold, then $ample(s) = en(s)$. The priority BS algorithm of section 4.2 is adapted to algorithm 5 to implement this POR algorithm.

Algorithm 5 Priority BS adapted for the POR algorithm of [6]

```
$Current := \{s_0\}$
$Next := \emptyset$
while $Current \neq \emptyset$ do
  for all $s \in Current$ do
    $T_1 = \{s \xrightarrow{a} s' \in explore(s) \mid priority(a) = p_1\}$
    if $T_1 \neq \emptyset$ then
      Select any $t \in T_1$
      $Next := Next \cup nx(s,\{t\})$              // PBS with $\beta = 1$
    else if $T_1 = \emptyset$ then
      $T_2 = \{s \xrightarrow{a} s' \in explore(s) \mid priority(a) = p_2\}$
      if $T_2 \neq \emptyset$ then
        Select $t \in T_2$
        $Next := Next \cup nx(s,\{t\})$            // PBS with $\beta = 1$
      else if $T_1 \cup T_2 = \emptyset$ then
        $Next := Next \cup nx(s, explore(s))$     // FPBS with $\beta > 0$
      end if
    end if
  end for
  $Current := Next$
  $Next := \emptyset$
end while
```

[1] These action labels do not need to be fixed and in our implementation are provided to the algorithm as an input. However, for the sake of simplicity, we fix these labels here.