Chapter 8. Search Algorithms

> "If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe."
>
> —(Abraham Lincoln, 1809–1865)

In this chapter we will discuss the basic algorithms that SPIN uses to verify correctness properties of PROMELA models. The basic algorithms are fairly simple and can quickly be explained. But, if we are interested in applying a verifier to problems of practical size, the mere basics do not always suffice. There are many ways in which the memory use and the run−time requirements of the basic algorithms can be optimized. Perhaps SPIN's main strength lies in the range of options it offers to perform such optimizations, so that even very large problem sizes can be handled efficiently. To structure the discussion somewhat, we will discuss the main optimization methods that SPIN uses separately, in the next chapter. In this chapter we restrict ourselves to a discussion of the essential elements of the search method that SPIN employs.

We start with the definition of a depth−first search algorithm, which we can then extend to perform the types of functions we need for systems verification.

Depth–First Search

Consider a finite state automaton A = (S, s_0, L, T, F) as defined in Chapter 6 (p. 127). This automaton could, for instance, be the type of automaton that is generated by the PROMELA semantics engine from Chapter 7, capturing the joint behavior of a number of asynchronously executing processes. Every state in such an automaton then represents a global system state. For the discussion that follows, though, it is immaterial how the automaton was constructed or what it represents precisely.

The algorithm shown in Figure 8.1 performs a depth–first search to visit every state in set A.S that is reachable from the initial state A.s_0. The algorithm uses two data structures: a stack D and a state space V.

Figure 8.1 Basic Depth–First Search Algorithm

```
Stack D = {}
Statespace V = {}

Start()
{
      Add_Statespace(V, A.s0)
      Push_Stack(D, A.s0)
      Search()
}
Search()
{
       s = Top_Stack(D)
       for each (s, l, s') ∈ A.T
              if In_Statespace(V, s') == false
              {      Add_Statespace(V, s')
                     Push_Stack(D, s')
                     Search()
              }
       Pop_Stack(D)
 }
```

A state space is an unordered set of states. As a side effect of the execution of the algorithm in Figure 8.1, some of the contents of set A.S is reproduced in state space V, using the definition of initial state A.s_0 and of transition relation A.T. Not all elements of A.S will necessarily appear in set V, because not all these elements may effectively be reachable from the given initial state.

The algorithm uses just two routines to update the contents of the state space:

- Add_Statespace(V,s) adds state s as an element to state space V
- In_Statespace(V,s) returns true if s is an element of V, otherwise it returns false

A stack is an ordered set of states. If the symbols < and > indicate the ordering relation, we have for any stack D:

$$\forall s_1, s_2 \in D: (s_1 \neq s_2 \rightarrow s_1 < s_2 \vee s_1 > s_2)$$
$$\forall s_1, s_2, s_3 \in D: (s_1 < s_2 < s_3 \rightarrow s_1 < s_3)$$

Because of the ordering relation, a stack also has a unique top and bottom element. If the stack is non−empty, the top is the most recently added element and the bottom is the least recently added element.

The algorithm in Figure 8.1 uses three routines to access stack `D`:

- `Push_Stack(D,s)` adds state `s` as an element to stack `D`
- `Top_Stack(D)` returns the top element from `D`, if `D` is non−empty, and otherwise returns nil
- `Pop_Stack(D)` removes the top element from `D`, if `D` is non−empty, and otherwise returns nil

It is not hard to show that this algorithm secures that state space `V` will contain no duplicates and can grow no larger than `A.S`.

The algorithm deliberately stores only states in set `V`, and no transitions. In this case complete information about all possible transitions is of course already available in relation `A.T`, but we may not always have this information available in precomputed form. When SPIN executes algorithm 8.1, for instance, it constructs both state set `A.S` and transition relation `A.T` on−the−fly, as an interleaving product of smaller automata, each one of which represents an independent thread of control, as explained in more detail in Appendix A.

To modify the algorithm from Figure 8.1 for on−the−fly verification is straightforward. The modified algorithm starts from a global initial system state, which is now specified as a composite of component states: { $A_1.s_0$, $A_2.s_0$, ...,$A_n.s_0$ }. All other system states in `A.S` are unknown at this point. Instead of relying on a precomputed definition of `A.T`, the successor states of any given system state { $A_1.s_i$, $A_2.s_j$, ..., $A_n.s_k$ } are now computed on the fly from the transition relations of the individual components { $A_1.T_1$, $A_2.T_2$, ..., $A_n.T_n$ }, subject to the semantics rules from Chapter 7.

By avoiding the storage of transitions in the state space, we can gain a substantial savings in the memory requirements during verification. We will see shortly that to perform safety and liveness checks we only need the information that is collected in the two data−structures that are maintained by the algorithm from Figure 8.1: state space `V` and stack `D`.

The algorithm in Figure 8.1 has the following important property:

Property 8.1

The algorithm from Figure 8.1 always terminates within a finite number of steps.

Proof:

Before each new recursive call to routine `Search ()`, at least one state from `A.S` that is not yet contained in `V` must be added to `V`. Because set `A.S` is finite, this can only happen a finite number of times.□

Checking Safety Properties

The depth−first search algorithm systematically visits every reachable state, so it is relatively straightforward to extend the algorithm with an evaluation routine that can check arbitrary state or safety properties. The extension of the search algorithm that we will use is shown in Figure 8.2. It uses a generic routine for checking the state properties for any given state `s`, called `Safety (s)`.

Figure 8.2 Extension of Figure 8.1 for Checking Safety Properties

```
Stack D = {}
Statespace V = {}

Start()
{
      Add_Statespace(V, A. s0)
      Push_Stack(D, A. s0)
      Search()
}

Search()
{
       s = Top_Stack(D)
*      if !Safety(s)
*      {      Print_Stack(D)
*      }
       for each (s, l, s')∈ A.T
              if In_Statespace(V, s') == false
              {      Add_Statespace(V, s')
                     Push_Stack(D, s')
                     Search()
              }
       Pop_Stack(D)
}
```

This routine could, for instance, flag the presence of a deadlock state by checking if state `s` has any successors, but it can also flag the violation of process assertions or system invariants that should hold at `s`. Since the algorithm visits all reachable states, it has the desirable property that it can reliably identify all possible deadlocks and assertion violations.

The only real issue to resolve is what precisely the algorithm should do when it finds that a state property is violated. It could, of course, merely print a message, saying, for instance:

```
dfs: line 322, assertion (a > b) can be violated, aborting.
```

There are two things wrong with this approach. First and foremost, this solution would leave it to the user to determine just how and why the assertion could be violated. Just knowing that a state property can be violated does not help us to understand how this could happen. Secondly, it is not necessary to abort a verification run after a single violation was found. The search for other violations can continue.

To solve the first problem, we would like our algorithm to provide the user with some more information about the sequence of steps that can lead to the property violation. Fortunately, all the information to do so is readily available. Our algorithm can produce a complete execution trace that demonstrates how the state property was violated. The trace can start in the initial system state, and end at the property violation itself. That information is contained in stack `D`. For this purpose, the algorithm in Figure 8.2 makes use of a new stack routine `Print_Stack(D)`:

> `Print_Stack(D)` prints out the elements of stack `D` in order, from the bottom element up to and including the top element.

When SPIN uses this algorithm, it prints out not just each state that is reached along the execution path that leads from the initial state to the state where a property violation was discovered, it also adds some details on the transitions from set `A.T` that generated each new state in the path. To allow SPIN to do so, all that is needed is to save an integer index into a lookup table of local process transitions with each element in stack `D`.

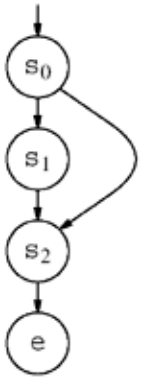
Note that the counterexamples that are produced with this algorithm are not necessarily the shortest possible counterexamples. This is unpleasant, but not fatal: it often suffices to look at just the last few steps in an execution sequence to understand the nature of a property violation.

Depth−Limited Search

We can adapt the depth−first search algorithm fairly easily into a depth−limited search that guarantees coverage up to a given depth bound. Such an algorithm is given in Figure 8.4. One change is the addition of an integer variable `depth` to maintain a running count of the size of stack `D`. Before growing the stack, the algorithm now checks the value of variable `Depth` against upper−bound `BOUND`. If the upper−bound is exceeded, routine `Search ()` does not descend the search tree any further, but returns to the previous expansion step.

This change by itself is not sufficient to guarantee that all safety violations that could occur within `BOUND` steps will always be found. Assume, for instance, an upper−bound value of three for the size of `D`. Now consider a state s_2 that is reachable from the initial state via two paths: one path of two steps and one path of one step. If s_2 has an error state `e` among its successors (i.e., a state exhibiting a property violation) that error state is reachable via either a path of three steps or via a path of two steps. The first path exceeds our presumed bound of three, but the second does not. If the depth−first search starts by traversing the first path, it will have added states s_0, s_1, and s_2 to state space `V` when it runs into the depth bound of three. It will then return, first to state s_1 and next to state s_0 to explore the remaining successor states. One such successor state of s_0 is s_2. This state, however, is at this point already in `V` and therefore not reconsidered. (It will not be added to the stack again.) The second path to the error state `e` will therefore not be explored completely, and the reachability of the error state within three steps will go unreported. The situation is illustrated in Figure 8.3.

Figure 8.3. Example for Depth−Limited Search



We can avoid this type of incompleteness by storing the value of variable `Depth` together with each state in state space `V`. The algorithm from Figure 8.4 uses this information in a slightly modified version of the two state space access routines, as follows:

- `Add_Statespace(V,s,d)` adds the pair (`s`, `d`) to state space `V`, where `s` is a state and `d` the value of `Depth`, that is, the current size of the stack
- `In_Statespace(V,s,d)` returns true if there exists a pair (`s'`, `d'`) in `V` such that `s' ≡ s` and `d' <− d`. Otherwise it returns false

Figure 8.4 Depth−Limited Search

```
Stack D = {}
Statespace V = {}
int Depth = 0
```

```
Start()
{
      Add_Statespace(V, A.s0, 0)
      Push_Stack(D, A.s0)
      Search()
}

Search()
{
*     if Depth >= BOUND
*     {      return
*     }
*     Depth++
      s = Top_Stack(D)
      if !Safety(s)
      {     Print_Stack(D)
      }
      for each (s, l, s') ϵ A. T
*           if In_Statespace(V, s', Depth) == false
*           {      Add_Statespace(V, s', Depth)
                   Push_Stack(D, s')
                   Search()
            }
      Pop_Stack(D)
*     Depth--
}
```

Compared to the simpler algorithm from Figure 8.2, this version of the depth–first search is clearly more expensive. In the worst case, if R is the number of reachable states that is explored, we may have to explore each state up to R times. This means that in this worst case there can be a quadratic increase in the run–time requirements of the algorithm, while the memory requirements increase only linearly with R to accommodate the depth field in V.

The algorithm from Figure 8.4 is implemented as an option in SPIN. To invoke it, the verifier is compiled with a special compiler directive -DREACH. If the verifier is compiled in this way, the run–time option -i can be used to iteratively search for the shortest error trail. This only works for safety properties though. An alternative algorithm to use in this case is a breadth–first search, which we will discuss shortly (see Figure 8.6).

The example from Figure 8.3 could be expressed in PROMELA as follows:

```
init { /* Figure 8.3 */

       byte x;
S0:    if
       :: x = 1; goto S1
       :: x = 2; goto S2
       fi;
S1:    x++;
S2:    x++;
E:     assert(false)
}
```

We can confirm that the default search with a depth−limit of three fails to find the assertion violation.

```
$ spin -a example.pml
$ cc -o pan pan.c
$ ./pan -m3
error: max search depth too small
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 12 byte, depth reached 2, errors: 0
       3 states, stored
       2 states, matched
       5 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.253 memory usage (Mbyte)

unreached in proctype :init:
        line 10, state 10, "-end-"
        (1 of 10 states)
```

If we switch to the depth−bounded search algorithm from Figure 8.4, and repeat the search with the same depth bound, the assertion violation is correctly reported:

```
$ spin -a example.pml
$ cc -DREACH -o pan pan.c
$ ./pan -m3
error: max search depth too small
pan: assertion violated 0 (at depth 2)
pan: wrote example.pml.trail
...
```

The search algorithm can also be extended to allow the verifier to iteratively home in on the shortest possible path to an error state, but adjusting the depth bound to the length of the last error path that was reported and allowing the search to continue after each new error, until the entire graph has been searched. In this case, we would replace the part of the code in Figure 8.4 that does the error check with the following fragment:

```
      if !Safety(s)
      {       Print_Stack(D)
*             if (iterative)
*                     BOUND = Depth
      }
```

Each new error path is now guaranteed to be shorter than all earlier paths, which means that the last reported path will also be the shortest. A run of the verifier that uses this option looks as follows:

```
$ ./pan -i      # iterative search for shortest error
pan: assertion violated 0 (at depth 3)
pan: wrote example.pml.trail
pan: reducing search depth to 3
pan: wrote example.pml.trail
pan: reducing search depth to 2
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 12 byte, depth reached 3, errors: 2
       4 states, stored
       2 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)
```

The first error path reported in this run is of length three. The new depth bound is then set to three and the search continues. Next, the shorter path of two steps is found and reported, which is also the shortest such path. Even though the worst case behavior of this algorithm is disconcerting, to say the least, this worst case behavior is rarely observed in practice. The typical use of the iterative search option is to find a shorter equivalent to an error path, only after the existence of an error path was demonstrated with a regular search. The length of that error path can then be used as an initial upper–bound for the iterative search.

Trade–Offs

The computational requirements for the depth–first search algorithm are linear in the number of reachable states in A.S. If A.S is itself computed from asynchronous components, which is the typical case in applications of SPIN, the size of this state set is in the worst case equal to the size of the Cartesian product of all component state sets A_i.S (cf. Appendix A). The size of this set can increase exponentially with the number of component systems. Although in practice the number of reachable states is no more than a small fraction of this upper bound, a small fraction of a potentially astronomically large number can still be very large.

One of the advantages of on–the–fly verification is that it allows us to trade memory requirements for run–time requirements when needed. One way to do this would be to randomly erase part of the state space when the algorithm runs out of memory, provided that the part erased contains no states that are still on the stack. Note carefully that the state space access routines (not the stack routines) serve only to prevent the multiple exploration of states. These routines do not affect the actual coverage of the search.

If we omit the call on routine Add_Statespace(), and replace the routine In_Statespace() with a new stack routine In_Stack(), we still have a correct algorithm that is guaranteed to terminate within a finite number of steps. The routine In_Stack(D,s) can be defined to return true if state s is currently contained in stack D, and false in all other cases.

The real purpose of the state space access routines is to improve the efficiency of the search by avoiding the repetition of work. If we completely eliminate the state space routines, as illustrated in Figure 8.5, the efficiency of the search could deteriorate dramatically. This change may cause each state to be revisited once from every other state in the state space, which means a worst–case increase in complexity from O(R) to $O(R^R)$ steps, where R is the total number of reachable states.

Figure 8.5 Stateless Search

```
Stack D = {}

Start()
{
*       Push_Stack(D, A.s0)
        Search()
}

Search()
{
       s = Top_Stack(D)
       if !Safety(s)
       {       Print_Stack(D)
               if (iterative)
                       BOUND = Depth
       }
       for each (s, l, s') ∈ A. T
*              if In_Stack(D, s') == false
*              {       Push_Stack(D, s')
                       Search()
               }
       Pop_Stack(D)
}
```

Intermediate solutions are possible, for example, by changing the state space V from an exhaustive set into a cache of randomly selected previously visited states, but it is hard to avoid cases of seriously degraded performance. SPIN therefore does not use caching strategies to reduce memory use. The optimization strategies implemented in SPIN are all meant to have a more predictable effect on performance. We discuss these strategies in Chapter 9.

Breadth–First Search

The implementation of a breadth–first search discipline looks very similar to that of the depth–first search. Instead of a search stack, though, we now use a standard queue. Successor states are added to the tail of the queue during the search, and they are removed from the head. A queue is just an ordered set of states. We use two new functions to access it.

- `Add_Queue(D,s)` adds state `s` to the tail of queue `D`,
- `Del_Queue(D)` deletes the element from the head of `D` and returns it.

This search procedure is illustrated in Figure 8.6.

Figure 8.6 Breadth–First Search Algorithm

```
Queue D = {}
Statespace V = {}

Start()
{
       Add_Statespace(V, A.s0)
       Add_Queue(D, A.s0)
       Search()
}

Search()
{
       s = Del_Queue(D)
       for each (s, l, s')∈ A. T
              if In_Statespace(V, s') == false
              {      Add_Statespace(V, s')
                     Add_Queue(D, s')
                     Search()
              }
}
```

Despite the similarities of depth–first and breadth–first searches, the two algorithms have quite different properties. For one, the depth–first search can easily be extended to detect cycles in graphs; the breadth–first search cannot. With the depth–first search algorithm, we also saw that it suffices to print out the contents of the stack to reconstruct an error path. If we want to do something similar with a breadth–first search, we have to store more information. One simple method is to store a link at state in state space `V` that points to one of the predecessors of each state. These links can then be followed to trace a path back from an error state to the initial system state when an error is encountered. It is then easy to use the breadth–first search algorithm for the detection of safety violations, for instance with the following extension that can be placed immediately after the point where a new state `s` is retrieved from queue `D`.

```
if !Safety(s)
{       Find_Path(s)
}
```

We have introduced a new procedure `Find_Path(s)` here that traces back and reproduces the required error path. SPIN has an implementation of this procedure that is enabled by compiling the verifier source text with the optional compiler directive `-DBFS`. For the example specification, for instance, this gives us the shortest error path immediately.

```
$ spin -a example.pml
$ cc -DBFS -o pan pan.c
$ ./pan
pan: assertion violated 0 (at depth 2)
pan: wrote example.pml.trail
...
```

The clear advantage of this method is that, if the memory requirements are manageable, it will guarantee that the shortest possible safety errors will be found first. The main disadvantages of the method are that it can sometimes substantially increase the memory requirements of the search, and that it cannot easily be extended beyond safety properties, unlike the depth−first search method.

Checking Liveness Properties

We will show how the basic algorithm from Figure 8.1 can be extended for the detection of liveness properties, as expressible in linear temporal logic. Liveness deals with infinite runs, ω–runs. Clearly, we can only have an infinite run in a finite system if the run is cyclic: it reaches at least some of the states in the system infinitely often. We are particularly interested in cases where the set of states that are reached infinitely often contains one or more accepting states, since these runs correspond to ω–accepting runs. We have seen in Chapter 6 how we can arrange things in such a way that accepting runs correspond precisely to the violation of linear temporal logic formulae.

An acceptance cycle in the reachability graph of automaton `A` exists if and only if two conditions are met. First, at least one accepting state is reachable from the initial state of the automaton $A.s_0$. Second, at least one of those accepting states is reachable from itself.

The algorithm that is used in SPIN to detect reachable accepting states that are also reachable from themselves is shown in Figure 8.7. The state space and stack structures now store pairs of elements: a state and a boolean value `toggle`, for reasons we will see shortly.

Figure 8.7 Nested Depth–First Search for Checking Liveness Properties

```
Stack D = {}
Statespace V = {}
State seed = nil
Boolean toggle = false

Start()
{
       Add_Statespace(V, A.s0, toggle)
       Push_Stack(D, A.s0, toggle)
       Search()
}

Search()
{
       (s,toggle) = Top_Stack(D)
       for each (s, l, s')∈ A. T
       {
             /* check if seed is reachable from itself */
             if s' == seed V On_Stack(D,s',false)
             {     PrintStack(D)
                   PopStack(D)
                   return
             }

             if In_Statespace(V, s', toggle) == false
             {      Add_Statespace(V, s', toggle)
                    Push_Stack(D, s', toggle)
                    Search()
             }
        }

        if s ∈ A. F && toggle == false
        {       seed = s /* reachable accepting state */
                toggle = true
                Push_Stack(D, s, toggle)
```

```
                Search()      /* start 2nd search */
                Pop_Stack(D)
                seed = nil
                toggle = false
    }

    Pop_Stack(D)
}
```

When the algorithm determines that an accepting state has been reached, and all successors of that state have also been explored, it starts a nested search to see if the state is reachable from itself. It does so by storing a copy of the accepting state in a global called `seed`. If this `seed` state can be reached again in the second search, the accepting state was shown to be reachable from itself, and an accepting ω–run can be reported.

The check for a revisit to the `seed` state contains one alternative condition that is equivalent to a match on the `seed` state itself. If a successor state `s'` appears on the stack of the first search (that lead us to the `seed` state), then we know immediately that there also exists a path from `s'` back to the `seed` state. That path is contained in stack `D`, starting at the state that is matched here and ending at the first visit to the `seed` state, from which the nested search was started.

Property 8.2

The algorithm from Figure 8.7 will detect at least one acceptance cycle if at least one such cycle exists.

Proof

For an acceptance cycle to exist there must be at least one accepting state that is both reachable from the initial system state and reachable from itself. Consider the very first such state that is encountered in post–order during the depth–first search; call it z_a. This need not be the first accepting state encountered as such, since there may well be accepting states that are not reachable from themselves that were encountered earlier in the search.

After all states that are reachable from z_a have been explored with a `toggle` attribute false, `seed` is set to z_a. The value of `toggle` is now set to true, and a new search (the nested search) is initiated starting from `seed`. At this point there can only be states in the state space with a `toggle` attribute equal to true if they are either accepting states that were reached earlier in the first depth–first search, or if they are states that are reachable from those accepting states.

There is only one case in which the nested depth–first search starting at z_a can fail to generate a full path back to z_a. This case occurs if that path contains a state that is already present in the state space with a true `toggle` attribute. Call that earlier state z_e, and call the earlier accepting state from which it was reached z_n. Note that the nested depth–first search truncates when reaching state z_e for the second time.

Now, even though state z_a is reachable from an earlier accepting state z_n in the nested search, z_a could not have been reached from z_n in the first search, because in that case the nested search for z_a would have been initiated before the nested search for z_n. (Due to the post–order search discipline.) This necessarily means that the path that leads from z_e to z_a must intersect the depth–first search stack in the first search somewhere above state z_n. But if

that is the case, the earlier accepting state z_n is necessarily also reachable from itself, through the path that is on the stack. This contradicts the assumption that z_a is the first accepting state encountered that is reachable from itself. □

When, therefore, the first accepting state that is reachable from itself is generated the state space cannot contain any previously visited states with a true `toggle` attribute from which this state is reachable, and thus the self−loop is necessarily constructed.

The property we have proven above does not necessarily hold for any accepting state after the first one that is encountered that is reachable from itself. Therefore, this algorithm can only guarantee that if one or more acceptance cycles exist, at least one of them will be found. That property suffices to perform formal verification. After all, only one counterexample is needed to disprove a correctness claim.

As in Figure 8.1, we have defined the algorithm in Figure 8.7 for a single given automaton `A`. In the context of SPIN, the automaton `A` is not given but has to be computed. If, for instance, we want to verify an LTL property f for a given system automaton `S`, then we first convert ¬ f into a `never` claim (a Büchi automaton) `B`. The desired automaton `A` is then computed as the synchronous product of `S` and `B`. System automaton `S` may have to be computed as the asynchronous product of `n` components {S_1, S_2, ... S_n} as well. The definitions of synchronous and asynchronous products can be found in Appendix A.

Adding Fairness

LTL is rich enough to express many fairness constraints directly, for example, in properties of the form ($\Box$ `trigger` $\longrightarrow \Diamond$ `response`). Specific types of fairness can also be predefined and built into the search algorithm. Recall that the asynchronous product of finite automata that is the ultimate subject of LTL model checking is built as an interleaving of transitions from smaller automata, $A = A_1 \times A_2 \ldots A_k$ (cf. Appendix A). Each of the automata $A_1 \times A_2 \ldots A_k$ contributes transitions to the runs of A. Component automaton A_i is said to be `enabled` at state `s` of the global automaton `A` if `s` has at least one valid outgoing transition from A_i. We can now define two standard notions of fairness.

Definition 8.1 (Strong Fairness)

> An ω–run σ satisfies the strong fairness requirement if it contains infinitely many transitions from every component automaton that is enabled infinitely often in σ.

Definition 8.2 (Weak Fairness)

> An ω–run σ satisfies the weak fairness requirement if it contains infinitely many transitions from every component automaton that is enabled infinitely long in σ.

The two definitions differ just in the use of the terms infinitely often and infinitely long, yet the computational overhead that is required to check these two requirements is vastly different. As we shall see shortly, the check for weak fairness increases the run–time expense of a verification run by a factor that is linear in the number of component automata (i.e., the number of running processes in a SPIN model). To check strong fairness within a system like SPIN, however, would increase the run time of a basic verification by a factor that is quadratic in the number of component automata, which for all practical purposes puts it beyond our reach. Not surprisingly, therefore, SPIN only includes support for weak fairness, and not for strong fairness.

SPIN's implementation of the weak fairness requirement is based on Choueka's flag construction method (see the Bibliographic Notes at the end of this chapter). Although the details of the implementation in SPIN are complex, it is not hard to describe the intuition behind the algorithm.

The depth–first search algorithm from Figure 8.1 explores the global reachability graph for an automaton `A`. Assume again that `A` itself is computed as the product of `k` component automata $A_1, \ldots, A_k$. We will now create (`k + 2`) copies of the global reachability graph that is computed by the algorithm from Figure 8.1. We preserve the acceptance labels from all accepting states only in the first copy of the state graph, that for convenience we will call the 0–th copy. We remove the accepting labels from all states in the remaining (`k + 1`) copies. Next, we make some changes in the transition relation to connect all copies of the state graph, without really removing or adding any behavior.

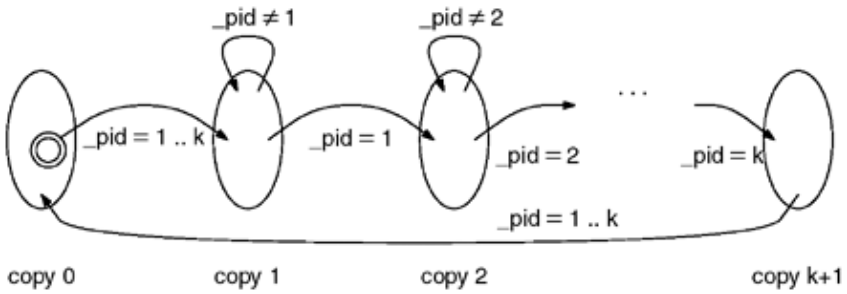
We change the destination states for all outgoing transitions of accepting states in the 0–th copy of the state space, so that they point to the same states in the next copy of the state space, with copy number one.

In the `i`–th copy of the state graph, with $1 \leq$ `i` $\leq$ `k`, we change the destination state of each transition that was contributed by component automaton A_i (i.e., the `i`–th process) to the same state in the (`i` + 1)–th copy of the state graph. For the last copy of the state space, numbered (`k` + 1), we change all transitions such

that their destination state is now in the 0−th copy of the state graph.

The unfolding effect is illustrated in Figure 8.8, which is based on a similar figure in Bosnacki [2001].

Figure 8.8. (k+2) Times Unfolded State Space for Weak Fairness



These changes do not add or remove behavior, but it should be clear that any accepting ω−run in the (k + 2) times unfolded state space now necessarily includes transitions from all k component automata. Note particularly that there can be no accepting cycles that are contained within the 0−th copy of the state graph, since all transitions that emerge from accepting states lead out of that graph. This means that we can use the nested depth−first search procedure from Figure 8.7 on the unfolded state graph to detect all fair accepting runs in the original graph.

We have to make one final adjustment to this procedure to account for the fact that, according to Definition 8.2, a component automaton that has no enabled transitions in a given state need not participate in an infinite run that traverses that state. To account for this we can add a null transition from every state s in the i−th copy of the state graph, $1 \leq i \leq k$, to the same state s in the (i + 1)−th copy whenever automaton component i has no enabled transitions in s. Without that null transition a complete cycle through all (k + 2) copies would of course not always be possible.

The algorithm thus modified can enforce weak fairness, but not strong fairness. Consider, for instance, the case where a component automaton is intermittently blocked and enabled. With the procedure we have outlined we cannot detect whether this automaton should or should not be included in an accepting ω−run.

Drawing the unfolded state space for even small examples can create rather complex graphs. A trivial example is shown in Figure 8.9, where we have drawn a two−state state space, which we assume to be generated by the joint execution of two asynchronous processes. The process numbered one contributes one single transition, from state s_1 to state s_2, and the process numbered two contributes one single transition back from state s_2 to state s_1. State s_2 is assumed to be accepting. Clearly, in this case there is just one behavior, and this one behavior corresponds to a fair acceptance cycle. With k equal to two, the unfolded state space, shown in Figure 8.10, contains four copies of the state space, each with two states. Only five of the eight states in the unfolded state space are reachable, though. The acceptance cycle is indicated with the bold arrows in Figure 8.10. Note also that as a side effect of the state space unfolding the length of the counterexample that will be generated by SPIN can be longer than strictly necessary. The path contains four steps in this case. The shortest possible counterexample would contain just two steps.

Figure 8.9. A Two−State Global State Space Example

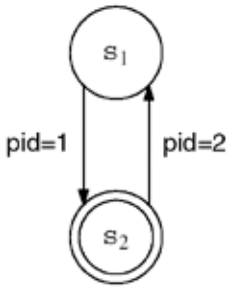
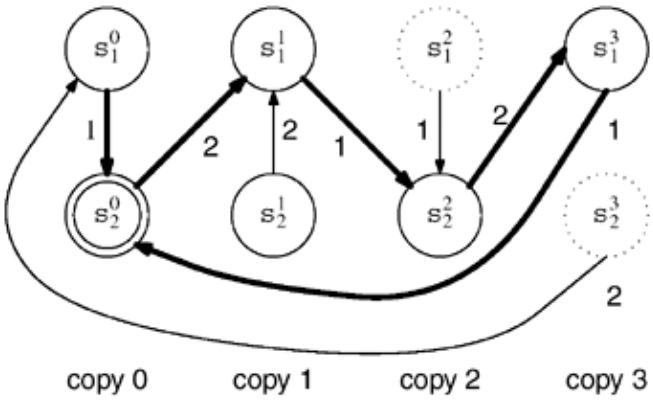


Figure 8.10. Unfolded State Space for Example in Figure 8.9



As another, only slightly more realistic, example, consider the following PROMELA model with two processes:

```
active proctype A()
{       bit i;

accept:
        do
        :: i = 1-i
        od
}

active proctype B()
{       bit i;

        do
        :: i = 1-i
        od
}
```

The verifier is generated and compiled as usual.

```
$ spin -a fairness.pml
$ cc -o pan pan.c
```

The default search for acceptance cycles then quickly succeeds.

```
$ ./pan -a
pan: acceptance cycle (at depth 2)
pan: wrote fairness.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance cycles       + (fairness disabled)
        invalid end states      +

State-vector 16 byte, depth reached 3, errors: 1
       3 states, stored (5 visited)
       5 states, matched
      10 transitions (= visited+matched)
       0 atomic steps
...
```

This cycle, however, is restricted to just executions of the process of type `B`, which is not very interesting (and violates a reasonable assumption of finite progress that we could make for both processes in this system).

```
$ spin -t -p fairness.pml
  1: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  2: proc 0 (A) line  6 "fairness.pml" (state 1) [i=(1-i)]
  <<<<<START OF CYCLE>>>>>
  3: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  4: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
spin: trail ends after 4 steps
#processes: 2
  4: proc 1 (B) line 12 "fairness.pml" (state 2)
  4: proc 0 (A) line 5  "fairness.pml" (state 2)
  2 processes created
```

The problem is resolved if we enforce weak fairness. We repeat the search as follows:

```
$ ./pan -a -f
pan: acceptance cycle (at depth 4)
pan: wrote fairness.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim           - (none specified)
        assertion violations  +
        acceptance cycles     + (fairness enabled)
        invalid end states    +
```

```
State-vector 16 byte, depth reached 11, errors: 1
       3 states, stored (15 visited)
      10 states, matched
      25 transitions (= visited+matched)
       0 atomic steps
...
```

Note that a little more work was done in the search, reflected by an increase in the number of states that were visited and the number of transitions that were executed. The number of states stored, however, does not increase. The new cycle that is found now looks as follows:

```
$ spin -t -p fairness.pml
  1: proc 1 (B) line 13  "fairness.pml" (state 1) [i=(1-i)]
  2: proc 1 (B) line 13  "fairness.pml" (state 1) [i=(1-i)]
  3: proc 0 (A) line  6  "fairness.pml" (state 1) [i=(1-i)]
  4: proc 1 (B) line 13  "fairness.pml" (state 1) [i=(1-i)]
  <<<<<START OF CYCLE>>>>>
  5: proc 1 (B) line 13   "fairness.pml" (state 1) [i=(1-i)]
  6: proc 1 (B) line 13   "fairness.pml" (state 1) [i=(1-i)]
  7: proc 0 (A) line  6   "fairness.pml" (state 1) [i=(1-i)]
  8: proc 1 (B) line 13   "fairness.pml" (state 1) [i=(1-i)]
  9: proc 1 (B) line 13   "fairness.pml" (state 1) [i=(1-i)]
 10: proc 1 (B) line 13   "fairness.pml" (state 1) [i=(1-i)]
 11: proc 0 (A) line  6   "fairness.pml" (state 1) [i=(1-i)]
 12: proc 1 (B) line 13   "fairness.pml" (state 1) [i=(1-i)]
spin: trail ends after 12 steps
#processes: 2
 12:   proc 1 (B) line 12 "fairness.pml" (state 2)
 12:   proc 0 (A) line 5  "fairness.pml" (state 2)
2 processes created
```

This time the trace includes actions from both processes. The trace is of course not the shortest possible one. At a risk of increasing the complexity of the search, we can try to find a shorter variant by proceeding as follows:

```
$ cc -DREACH -o pan pan.c
$ ./pan -a -f -i
pan: acceptance cycle (at depth 4)
pan: wrote fairness.pml.trail
pan: reducing search depth to 12
pan: wrote fairness.pml.trail
pan: reducing search depth to 11
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim            - (none specified)
        assertion violations   +
        acceptance cycles      + (fairness enabled)
        invalid end states     +
```

```
State-vector 16 byte, depth reached 11, errors: 2
       4 states, stored (19 visited)
      16 states, matched
      35 transitions (= visited+matched)
       0 atomic steps
...
```

Although the verifier did find a shorter variant of the cycle in this case, in general this is not guaranteed to be the case. Note that the `-DREACH` variant of the search is designed for the detection of safety errors, using a single depth-first search. It can lose its essential properties in a nested search. This can mean, for instance, that an interative search for cycles without the `-DREACH` option could in principle find a shorter variant of a cycle than a search performed with the option, though not in this case. So, for liveness properties, it can be worth trying both approaches to see which one produces the shortest trail. When the `-DREACH` option was used, the verifier had to do a little more work to build the state space, and it could run out of memory sooner than the variant without the option.

The shorter acceptance cycle that was found looks as follows:

```
$ spin -t -p fairness.pml
  1: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  2: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  3: proc 0 (A) line  6 "fairness.pml" (state 1) [i=(1-i)]
  4: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  5: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  6: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  7: proc 0 (A) line  6 "fairness.pml" (state 1) [i=(1-i)]
  <<<<<START OF CYCLE>>>>>
  8: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
  9: proc 0 (A) line  6 "fairness.pml" (state 1) [i=(1-i)]
 10: proc 1 (B) line 13 "fairness.pml" (state 1) [i=(1-i)]
 11: proc 0 (A) line  6 "fairness.pml" (state 1) [i=(1-i)]
 12: proc 0 (A) line  6 "fairness.pml" (state 1) [i=(1-i)]
spin: trail ends after 12 steps
#processes: 2
12:   proc 1 (B) line 12 "fairness.pml" (state 2)
12:   proc 0 (A) line  5 "fairness.pml" (state 2)
2 processes created
```

The SPIN Implementation

(Can be skipped on a first reading.) SPIN's implementation of the weak fairness algorithm differs on minor points from the description we have given. The modifications are meant to reduce the memory and run−time requirements of the search.

A first difference is that the SPIN implementation does not actually store (k + 2) full copies of each reachable state. Doing so would dramatically increase the memory requirements for the weak fairness option. It suffices to store just one copy of each state plus (k + 2) bits of overhead. The additional bits record in which copy of the state graph each state was visited: the i−th bit is set when the state is encountered in the i−th copy of the state graph. Creating one extra copy of the state graph now requires just one extra bit per state. If the reachable state space contains R states, each of B bits, the memory requirements for the algorithm can thus be reduced from (R x B) x (k + 2) to (R x B) + (k + 2) bits.

Another small difference is that the nested depth−first search for cycles is not initiated from an acceptance state in the 0−th copy of the state graph, but from the last copy of the state graph. Note that whenever this last copy is reached we can be certain of two things:

- A reachable accepting state exists.
- A (weakly) fair execution is possible starting from that accepting state.

Each state in the last copy of the state graph now serves as the seed state for a run of the nested depth−first search algorithm, in an effort to find a cycle. As before, all transitions from the last copy of the state graph move the system unconditionally back to the 0−th copy of the state graph, and therefore the only way to revisit the seed state is to pass an accepting state and close the cycle with a fair sequence of transitions.

Complexity Revisited

In the worst case, the algorithm from Figure 8.7 for checking liveness properties uses twice as much run time as the algorithm from Figure 8.2, which sufficed only for checking safety properties. Clearly, in the algorithm from Figure 8.7 each state can now appear in the state space twice: once with the a true `toggle` and once with a false `toggle` attribute. The algorithm can, however, be implemented with almost no memory overhead. As we noted earlier in the discussion of weak fairness, each state needs only be stored once, and not twice, with the right bookkeeping information. The bookkeeping information in this case requires just two bits per state. The first bit is set to one when the state is visited with a `toggle` value false; the second bit is set to one if the state is visited with a `toggle` value true (i.e., in the nested part of the search). Clearly, the combination (0,0) for these two bits will never be seen in practice, but each of the three remaining combinations can appear and the two bits together suffice to accurately identify all reached states separately in each of the two (virtual) state graphs.

The algorithm from Figure 8.2 incurs a computational expense that is linear in the number of reachable states for a given system model, that is, there is a largely fixed amount of work (computation time and memory space) associated with each reachable system state. Adding cycle detection increases the run−time expenses by a factor of maximally two, but does not impact the memory requirements noticeably. Adding a property automaton (e.g., a `never` claim generated from an LTL requirement) of `N` states increases the expense of a straight reachability by another factor of maximally `N`. The size `N` of the property automaton itself, though, can increase exponentially with the number of temporal operators used in an LTL formula. This exponential effect, though, is rarely, if ever, seen in practice.

Adding the weak fairness constraint causes an unfolding of the reachable state space by a factor of (`k` + 2), where `k` is the number of active processes in the system. In the implementation, the memory cost of the unfolding is reduced significantly by storing each copy of a reachable state not (`k` + 2) times but once, and annotating it with (`k` + 2) bits to record in which copies of the state graph the state has been encountered. If we use the nested depth−first search cycle detection method, the memory overhead per reachable state then remains limited to 2(`k` + 2) bits per reached state. In the worst case, though, the run−time requirements can increase by a factor of 2(`k` + 2), although in practice it is rare to see an increase greater than two.

Clearly, automated verification can be done most efficiently for pure safety properties: basic assertions, system invariants, absence of deadlock, etc. Next in efficiency is the verification of liveness properties: proving the absence of non−progress cycles or the presence of acceptance cycles. Next in complexity comes the verification of LTL properties. The more complex the temporal formula, the more states there can be in the corresponding property automaton, and the greater the computational expense of the verification can be. This is a worst−case assessment only, though. In many cases, there is a tight coupling between transitions in a property automaton that is generated from an LTL formula and the reachable states of a system, which means that often the computational expense is only modestly affected by the size of the property automaton or the size of the underlying LTL formula.

Bibliographic Notes

The best known alternative method to detect acceptance cycles in a finite graph is based on the construction of all the maximal strongly connected components in the graph. If at least one component contains at least one accepting state, then an accepting ω−run can be constructed. The strongly connected components in a graph can be constructed with time and space that is linear in the size of the graph with a depth−first search, Tarjan [1972]. Tarjan's procedure requires the use of two integers to annotate each state (the depth−first number and the low−link number), while the nested depth−first search procedure from Figure 8.7 requires the addition of just two bits of information. Tarjan's procedure, though, can identify all accepting ω−runs in a graph. The nested depth−first search procedure can always identify at least one such run, but not necessarily all. The nested depth−first search procedure is compatible with all lossless and lossy memory compression algorithms that we will explore in the next chapter, while Tarjan's procedure is not.

State space caching methods were described in Holzmann, Godefroid, and Pirottin [1992], and in Godefroid, Holzmann, and Pirottin [1995].

The nested depth−first search procedure was first described in Courcoubetis, Vardi, Wolper, and Yannakakis [1990], and its application to SPIN is described in Godefroid and Holzmann [1993]. A similar procedure for the detection of non−progress cycles can also be found in Holzmann [1991], and is discussed in more detail in Holzmann [2000]. A modification of the nested depth−first search procedure to secure compatibility with partial order reduction methods is described in Holzmann, Peled, and Yannakakis [1996].

Choueka's flag construction method was first described in Choueka [1974]. Its potential use for the enforcement of fairness with a nested depth−first search was mentioned in Courcoubetis et al. [1990]. An alternative, detailed description of SPIN's implementation of the weak fairness algorithm can be found in Chapters 3 and 7 of Bosnacki [2001].