Model Checking Concurrent Programs

By:
Ian Barland
Moshe Vardi
John Greiner

Model Checking Concurrent Programs

By:
Ian Barland
Moshe Vardi
John Greiner

Online:
< http://cnx.org/content/col10294/1.3/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: October 27, 2005
PDF generated: February 4, 2011
For copyright and attribution information for the modules contained in this collection, see p. 100.

Table of Contents

Chapter 1

Concurrent Programming and Verification: Outline[1]

1.1 Verification of Concurrent Programs

How can we know whether our programs have bugs? This is the pre-eminent question in software engineering today. Test suites are important, but no amount of tests can conclusively show a program to be bug-free. In concurrent programs in particular, bugs can be difficult to find and replicate through trial (and error).

Ideally we would **prove** that a program meets certain specs. How to meet this goal routinely has been a long-standing goal in computer science. Historically, there have been significant difficulties in making such proofs practical, but verification technology is significantly improving. In this module, we look at some of the issues involved — how to describe, reason about, and verify properties — specifically as applied to concurrency.

We will look at two related ways to understand concurrent programs. First, we write small programs in Promela, a language with a familiar C-like syntax, but suited for concurrent programs. We use Promela's partner tool, SPIN, to verify various properties our program's behavior. Second, we introduce state-based **transition systems** — an equivalent model for concurrent programs, but one which is more suitable for reasoning about their behavior.

Together, Promela and SPIN are designed to easily let us check for many common concurrency behaviors. Historically, SPIN allowed only a collection of special-purpose checks. More generally, however, we can use **temporal logic** to express our own properties for SPIN to check. We will look at this special kind of logic and see how to apply it to verification.

This module is meant to be taught within the context of a concurrent programming course, or an operating systems course covering the basics of concurrent programming. As such, this will not cover concurrent programming techniques. However, we will quickly review the necessary background.

Outline

- Basic issues (Chapter 2) — an optional review of the necessary background (with exercises (Chapter 3))
- Modeling (Chapter 4) — an overview of modeling and verifying concurrency, using Promela and transition systems (with exercises (Chapter 5))
- Temporal Logic (Chapter 6) — for specifying other concurrent behaviors (with exercises (Chapter 7))
- Other references:
  - The SPIN home page[2]
  - The SPIN book[3] . Several examples in our notes are inspired by this book.

[1] This content is available online at <http://cnx.org/content/m12311/1.12/>.
[2] http://www.spinroot.com
[3] http://www.spinroot.com/spin/Doc/Book_extras/index.html

- On-line manuals[4] for Promela and SPIN.
- Installing SPIN[5]
- jspin, SpinSpider: Tools for Teaching Concurrency with Spin[6]

[4] http://spinroot.com/spin/Man/index.html
[5] http://spinroot.com/spin/Man/README.html
[6] http://stwww.weizmann.ac.il/G-CS/BENARI/jspin/index.html

Chapter 2

Concurrent Processes: Basic Issues[1]

2.1 Basic Concepts

Concurrency is the execution of two or more independent, interacting programs over the same period of time; their execution can be interleaved or even simultaneous. Concurrency is used in many kinds of systems, both small and large.

Example 2.1
The typical home computer is full of examples. In separate windows, a user runs a web browser, a text editor, and a music player all at once. The operating system interacts with each of them.

Almost unnoticed, the clock and virus scanner are also running. The operating system waits for the user to ask more programs to start, while also handling underlying tasks such as resolving what information from the Internet goes to which program.

Example 2.2
Using a web-based airline reservation system, Joe and Sue each want to buy a ticket. Each has a separate computer, and thus a separate web browser. Their two web browsers plus the airline's web server together comprise the concurrent program.

As this example illustrates, the term **concurrent system** might be more appropriate. However, for verification purposes, we will view them as a single, distributed entity to be modeled.

A concurrent system may be implemented via **process**es and/or **thread**s. Although details can vary upon platform, the fundamental difference is that processes have separate address spaces, whereas threads share address spaces. We will follow the common theoretical practice and ignore this distinction, using shared variables when desired, and using the two terms interchangeably.

We view these threads as executing relatively independently. However, since they are acting together towards some goal, they must need to communicate and coordinate. The two most common communication techniques in processes and threads are through, respectively, **message passing** and **shared variables**. In order to communicate at the right times, they must **synchronize**, together arriving at agreed-upon control points. Often, one or more threads **block**, or stop and wait for some external event. In this module, we will use only shared variables, although the tools we use also allow message passing.

Even though we have multiple flows of control, that doesn't imply we need multiple processors. Concurrent programs may be executed on a single processor by interleaving their control flows. In order to understand what happens in a concurrent program over time, we must understand how the individual operations of the threads can interleave. This is independent of how many processesors we use, although it is convenient to think of only having one processor. To consider the possible interleavings, we need to know which actions are **atomic**, or indivisible. If an atomic operation begins, we know that it won't be interrupted

[1] This content is available online at <http://cnx.org/content/m12312/1.16/>.

by a context switch until it is done. We generally assume that each hardware machine instruction is atomic, but a programming language might guarantee that even more complicated operations are atomic as well.

Definition 2.1: State

A state captures all the current information in a program. This includes all local and global variables' values and each thread's current program counter. More generally, this includes all the memory and register contents.

See Also: trace, state-space

Definition 2.2: Trace

A trace is the sequence of operations (and their data) performed in a particular run of a concurrent program. Equivalently, it is the sequence of states during a run — *i.e.*, the collection of variable and program counter values.

See Also: state, ω-trace

Definition 2.3: Atomic

An atomic operation is indivisible. A context switch to another process cannot happen during this operation.

Example: Statement-level Atomicity

Let's examine the possible interleavings of the code fragments for two threads. First, let's assume that each assignment statement is atomic.

```
1  thread0:
2  {
3    x=x+1
4  }

5  thread1:
6  {
7    x=x*2
8  }
```

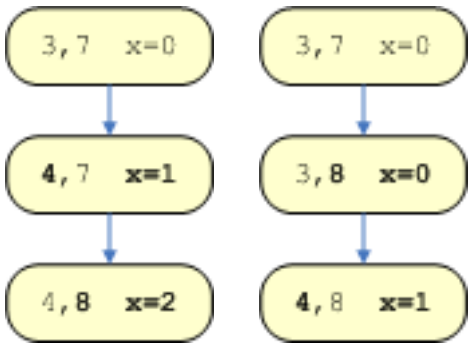


Figure 2.1: There are two possible traces, as either thread's statement could execute first. Bold type highlights anything modified or assigned to each step.

Here, each trace is a timeline, representing one possible interleaving of operations. From top to bottom, each state represents one point in time, with a current line number for each thread, plus

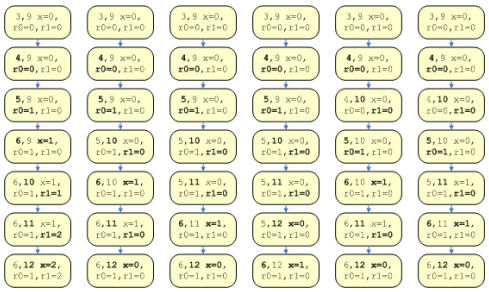
the value of each variable. The first state is the starting point, before execution begins. Here, we assume that `x` is initially zero. We will use traces and their timeline diagrams throughout this module to understand concurrent programs.

Example: Instruction-level Atomicity

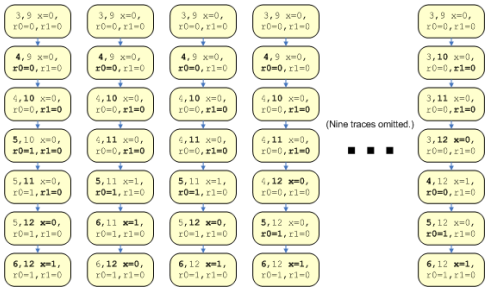
For contrast, let's change what is atomic. Now, assume that each machine instruction operation, such as loading, adding, or storing a register, is atomic. Again, we assume `x` is initially zero. Observe that with finer-grained atomicity, there are many more ways to interleave the threads, and (in this case) more possible result values for `x`.

```
1  thread0:
2  {
3    r0 =  x
4    r0 += 1
5    x  =  r0
6  }

7  thread1:
8  {
9    r1 =   x
10   r1 ≪= 1  /* Shifting left one position multiplies by 2. */
11   x  =   r1
12 }
```



(a)



(b)

Figure 2.2: With finer-grained atomicity, there are many more ways to interleave the threads.

Both the syntax and semantics of the concurrency primitives can vary among programming languages and concurrency libraries. We will use a particular language, Promela[2] , which implements the most commonly-used features of concurrent programs. In Promela, assignment statements are defined to be atomic. We'll later contrast atomic with non-atomic (Example 4.3: A Race Condition) assignment statements.

2.2 Problems with Concurrency

Programmers are comfortable with debugging single-threaded code, where most questions involve "does this sequence of instructions compute the correct answer?" Concurrency, however, introduces the added complications of communication and synchronization; a single task may be spread across several sequences of instructions running asynchronously, and multiple threads can interact in unintended ways. The following are five common categories of problems with concurrency.

Definition 2.4: Deadlock
(Informal) Deadlock is when two or more threads stop and wait for each other.

Definition 2.5: Livelock
(Informal) Livelock is when two or more threads continue to execute, but make no progress toward the ultimate goal.

Example 2.3: Deadlock vs. Livelock
As an analogy, consider two people walking in a hallway towards each other. The hallway is wide enough for two people to pass. Of interest is what happens when the two people meet. If on the same side of the hallway, a polite strategy is to step to the other side. A more belligerent strategy is to wait for the other person to move. Two belligerent people will suffer in deadlock, glaring face to face in front of each other. Two polite people could suffer from livelock if they repeatedly side-step simultaneously. (No conclusions on morality are to be inferred from the fact that one polite and one belligerent person don't have any problems.)

[2] http://spinroot.com/spin/Man/index.html

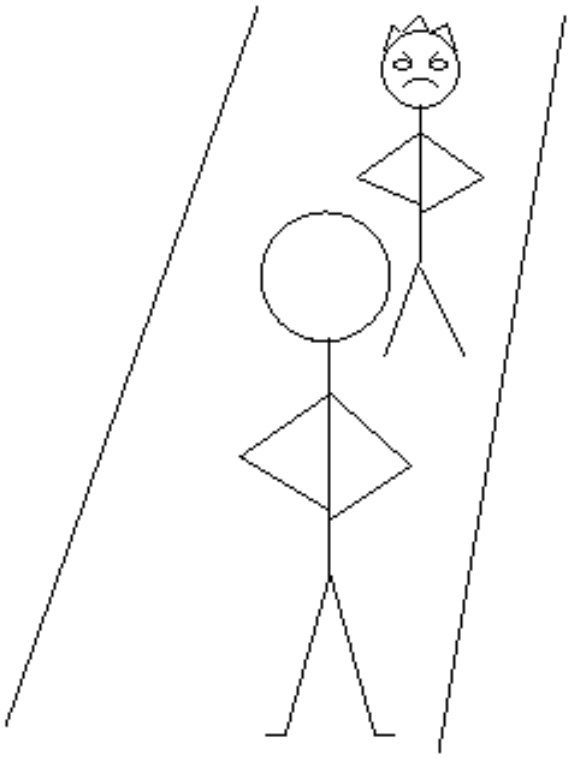


Figure 2.3: Two belligerent hallway-walkers can deadlock.

Definition 2.6: Fairness
(Informal) Fairness is the idea that each thread gets a turn to make progress.

There are more specific notions of fairness that describe when and how often threads are guaranteed to get a turn. For example, do we expect that each thread should be executed as often as any other, or is it acceptable if one runs a hundred steps for each step of any other thread? When considering fairness, one must also consider the system code implementing the threads. The implementation includes a scheduler that determines how to interleave the threads, and the scheduler might or might not provide any fairness guarantees.

Definition 2.7: Starvation
(Informal) Starvation is when some thread gets deferred forever.

Example 2.4: Fairness and Starvation
Assume we are modeling a store. Consumers buy products, reducing the shelf inventory. Meanwhile, employees restock the shelves, increasing the shelf inventory. More simply, let thread C be a loop that repeatedly decrements a counter, and thread P be a loop that repeatedly increments that same counter.

Unless our scheduler (which may be nature) otherwise gives some guarantees, this system does not ensure that the two threads execute fairly. In particular, it is possible for one thread to starve the other.

Example 2.5: The express checkout line, and the woes of Grover

A Sesame Street[3] skit has Grover in line at the grocery store, his basket heaped full of animal crackers and other essentials. A little old lady gets in line behind him, holding only a single candle, muttering how she is late for her grandson's very first birthday party. Grover muses that she has to wait a long time for his big basket to be rung up, whereas if she went first it's hardly any delay at all for him. So he invites her to get in front of him in line, and she gratefully accepts.

You might guess what happens next though: Next in line is the Swedish Chef, needing only a single meatball (Bork! Bork! Bork!), and then fish-flinging Lew Zealand[4] , who is buying just one little herring, followed by ... [insert Grover's resigned sigh here].

The scheduler, in this case, is the social protocol at the checkout line (which includes Grover's politeness). This input situation demonstrates how Grover gets starved. The alternate protocol of having an additional express checkout line (without allowing any butt-ins) is an attempt to minimize customers' wait-delay without introducing starvation.

Definition 2.8: Race Condition

A race condition is when some possible interleaving of threads results in an undesired computation result.

Example

Returning to our airline reservation system example (Example 2.2), suppose Joe and Sue each see that one seat is left. It is a race condition error if the system allows both of them to successfully reserve the one seat.

Typically, a race condition is an oversight by the programmer who did not realize the interleaving was possible. Race conditions are notoriously hard to debug and test for because they can well occur in highly unlikely situations.

More generally, these and other issues are categorized as being either **safety** or **liveness** issues.

- Safety properties are those that say "bad things never happen". Examples are the lack of deadlock, the lack of livelock, and the lack of race conditions. For a particular program, a safety property might be that inventory levels never become negative, or that customers can't cut ahead in line.
- Liveness properties are those that say "good things eventually happen". An example is fairness. For a particular program, a liveness property might be that each customer is served, or that after each program error, an error message is printed. Typically, liveness issues can't be detected by looking at a program at a particular moment, but instead require considering an entire (infinite) trace.

In this module, we will concentrate on the four issues of deadlock, livelock, race conditions, and fairness. We will see how to write and automatedly detect these problems (Chapter 4) in some concurrent programs, using the tools Promela and SPIN[5] . We will also learn how to formally specify more intricate properties using temporal logic (Chapter 6).

2.3 Avoiding and Detecting Problems with Concurrency

There are known programming techniques to avoid certain problems such as deadlock and some kinds of race conditions. Some of these techniques — notably locks and semaphores — are so common that they are often embedded into programming languages. While the details of such techniques are outside the scope of this module, it is important to recognize that although they greatly help writing concurrent problems, they don't mean that concurrency problems can't arise.

[3] http://www.sesameworkshop.org/sesamestreet/
[4] http://www.gusworld.com.au/games/mrm/lewzealand.jpg
[5] http://spinroot.com/spin/Man/index.html

Given concurrent code, how can we determine if it behaves as expected? Certainly we can test it on various inputs, but we can never try every possible input. Neither can we test every possible interleaving on every possible input. As always, testing can reveal errors, but it can't demonstrate the lack of errors.

Since concurrency problems can lead to programs that don't terminate, it's clear that detecting concurrency problems can't be any easier than the Halting Problem[6] , which is itself unsolvable. So we can't write a checker which takes a concurrent program as input and always determines whether it's free of (say) deadlock.

But just as we can study individual programs and conclude whether or not they will halt, for many real-world concurrent programs, we **can** determine whether or not they are susceptible to deadlock. For example, communications protocols generally have a finite number of options and are amenable to automated checking. Thus automated tools (such as SPIN) are valuable aid in catching and eliminating real-world bugs.

[6] http://en.wikipedia.org/wiki/Halting_problem

Chapter 3

Concurrent Processes: Basic Issues: Homework Exercises[1]

Exercise 3.1 *(Solution on p. 13.)*

Assume we have two concurrent threads/processes, each with simple straight-line code and sharing a global variable x. Assume that each individual statement is atomic.

```
 1  thread0:
 2  {
 3    x=0
 4    x=x+1
 5  }

 6  thread1:
 7  {
 8    x=0
 9    x=x+1
10    x=x+2
11  }
```

Without using SPIN, give one trace illustrating each of the possible final values of x.

Exercise 3.2 *(Solution on p. 13.)*

Answer the previous exercise, except with different atomicity. Assume that each variable lookup, variable assignment, and addition are atomic, as in a typical machine language. Specifically, use the following code:

```
1  thread0:
2  {
3    x  =  0
4    r0 =  x
5    r0 += 1
6    x  =  r0
7  }
```

[1] This content is available online at <http://cnx.org/content/m12938/1.2/>.

```
 8  thread1:
 9  {
10    x  =  0
11    r1 =  x
12    r1 += 1
13    x  =  r1
14    r1 =  x
15    r1 += 2
16    x  =  r1
17  }
```

Solutions to Exercises in Chapter 3

Solution to Exercise 3.1 (p. 11)
There are three possible final values of x: 1, 3, 4.

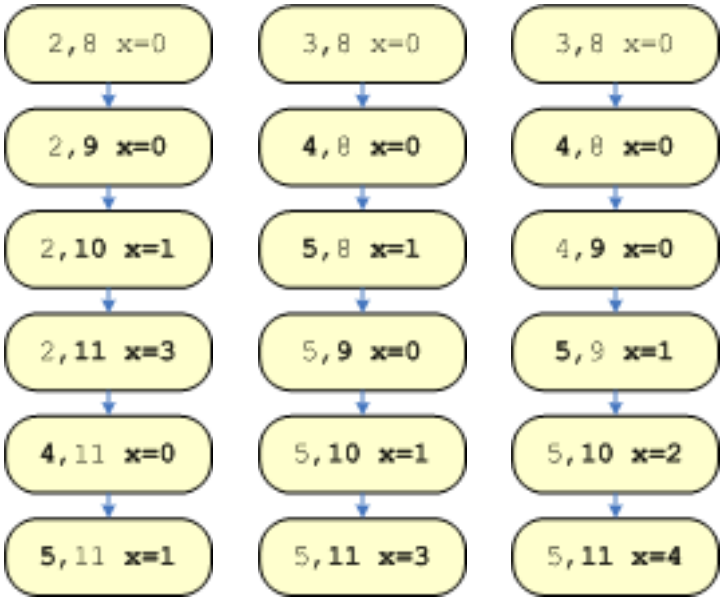


Figure 3.1: For 1 and 3, the only trace for each is given. For 4, there are many traces, with one sample given.

Solution to Exercise 3.2 (p. 11)
The values 1, 3, and 4 are possible as before. It should be clear that values greater than 4 are impossible, since at most a total of 4 can ever be added. Similarly, values less than 1 are impossible, since at least 1 is added after the last variable lookup. So, we're left with the question of whether 2 is a possible result. It is, and one possible trace is given.

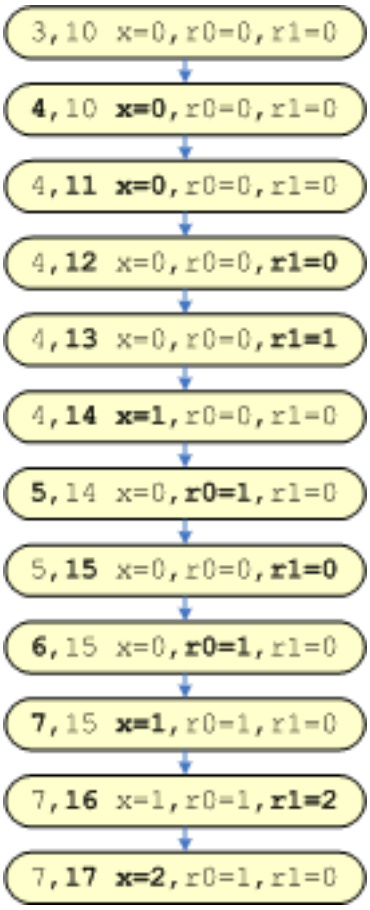


Figure 3.2: A trace resulting in the value 2.

Chapter 4

Modeling Concurrent Processes[1]

We will model concurrency in two ways. First, we will use Promela, a language with C-like syntax. It is not a fully featured programming language, and is not intended for general computation. Instead, Promela ("PROcess MEta-LAnguage") programs are intended to be simplifications or models of real-world systems, for use in verification. SPIN ("Simple Promela INterpreter") is the tool for executing and verifying programs written in Promela. Second, we will use a simple state-based transition system that will help in understanding the specification and verification of Promela programs.

Here, we introduce Promela, SPIN, and the state-based transition system through a series of examples. For the moment, we will use SPIN merely as an interpreter, to run of Promela programs. In the next section (Section 4.5: Verification), we will introduce the verification features of SPIN. (Reference manuals[2] and download/install instructions[3] are available via the SPIN homepage, spinroot.com[4] .)

4.1 Promela and SPIN Basics

We start with a series of examples illustrating race conditions.

Example 4.1: A Tiny First Program

```
1   /* A variable shared between all processes. */
2   show int bal = 0;
3
4   active proctype deposit()
5   {
6     bal++;
7   }
8
9   active proctype withdraw()
10  {
11    bal--;
12  }
```

We have two threads, one running `deposit` and one running `withdraw`. The `proctype` keyword specifies that the following is code for a thread/process, while the `active` keyword specifies that the

[1] This content is available online at <http://cnx.org/content/m12316/1.21/>.
[2] http://spinroot.com/spin/Man/index.html
[3] http://spinroot.com/spin/Man/README.html
[4] http://spinroot.com

thread is started immediately when we start the program. Variables declared outside the body of a `proctype` are shared. The keyword `show` before a variable declaration will direct SPIN to display the value as it changes.

Here, the two processes of `deposit` and `withdraw` can interleave arbitrarily. Regardless, with this very simple example, we will always get the same result balance.

To run the code, we use SPIN. We'll describe how to use the program `xspin`, which uses a graphical interface. More specifically, these instructions are for UNIX version 4.1.3. The PC and Mac versions are identical, except for how to start the program. For details, see the program's README[5] . Ask your local system administrator where the program is installed on your computer.

ASIDE: There is also a version based on the command-line, called `spin`. It is more difficult to use interactively, but is appropriate for use non-interactive use, such as with scripts. For its options, see the manual pages for `spin`[6] and the related `pan`[7] . `xspin` is just a graphical front-end to `spin`. The underlying `spin` commands and output are displayed at the bottom of the main `xspin` window. These can be ignored.

NOTE: To run either `xspin` or `spin` from Rice University's Owlnet, first type `setenv PATH /home/comp607/bin:$PATH`.

Within SPIN, you'll work with a Promela program. If you already have a Promela program saved, you can open it with the "File" menu's "Open" option. Alternatively, start SPIN with the Promela program's filename:

```
xspin filename.pml
```

(The conventional suffix for Promela programs is `.pml`.) Either of these loads the Promela code into an editor window, where it can be modified. To create a new program, you can type into this window, or you can copy and past it from another editor.

After loading or typing a program, it's always a good idea to check your typing. In the "Run" menu, use "Run Syntax Check". It will report and highlight the first syntax error, if any.

Before the first run, you need to specify some parameters, even if those are the defaults. From the "Run" menu, select the "Set Simulation Parameters" option. Change the "Display Mode" defaults, by making sure that the "MSC" and "Data Values" panels are each selected, as well as each feature under the "Data Values" panel. Use the "Start" button to start the run. This opens three new windows. From the "Simulation Output" window, you can choose to single-step the run or let it finish. For now, choose "Run". The body of this window displays each step taken. Throughout the run, the "Data Values" window displays the values of variables. The "Message Sequence Chart" displays changes to each process and variable. Each column represents a separate process or variable, and the vertical axis represents time. For now, the only changes to processes are when they stop.

Each process gets a unique process identifier number, starting with 0. These are displayed in the output of the various windows. Later (p. 45), we will see how to access these IDs from within the Promela program.

For additional runs, from the "Run" menu of the main window, you can select "(Re)Run Simulation". However, each run will be identical, because it makes the same choices for interleaving. To vary runs, or **traces**, change the "Seed Value" in the simulation options. Also, be sure to try the "Interactive" simulation style, where you make each choice in the interleaving.

ASIDE: Each `xspin` run opens several new windows, without closing the old ones. The easiest way to close the windows for a run is to use the "Cancel" button in the "Simulation Output" window.

ASIDE: Experiment with the other display modes in the simulation options to see what else is available.

[5] http://spinroot.com/spin/Man/README.html
[6] http://spinroot/spin/Man/Spin.html
[7] http://spinroot/spin/Man/Pan.html

NOTE: Running a basic simulation is one of the few things which is easier in `spin` than `xspin`, since you don't need to set any of the simulation parameters:

```
prompt>  spin filename.pml
```

Definition 4.1: State
A snapshot of the entire program. In Promela, this is an assignment to all the program's variables, both global and local, plus the program counter, or line number, of each process. In other languages, the state also includes information such as each process' stack contents.

Definition 4.2: Trace
A sequence — possibly infinite — of successive states, corresponding to one particular run of a program.
See Also: state

Exercise 4.1 ***(Solution on p. 50.)***
How many traces are possible for the previous example's code (Example 4.1: A Tiny First Program)? Does each end with the same value for `bal`?

All the traces of a Promela program start at the same **initial state**. Any variables not explicitly initialized by the code are set to particular known values: zero, for numbers; and false, for booleans. (Compare to a C program, where two traces of the same program might each start from a different state.) And we've just seen that different traces may end in different states, or they may sometimes end in the same state while having taken different paths to reach that end state. Finally, if we look closely at all the possible traces, we see that often, several different traces might happen to pass through the same intermediate state. By combining the traces at these common states, we can depict the entire **state space** — a map of all the conceivable executions of our program!

Definition 4.3: State space
A directed graph with program states as nodes. The edges represent program statements, since they transform one state to another. There is exactly one state, the **initial state**, which represents the program state before execution begins.
See Also: state, **end state**, **valid end state**

Definition 4.4: Valid end state
(Informal) Some states of our state space may be designated valid end states. These represent points where all of the threads have completed execution.
See Also: deadlock, **end state**

Example 4.2

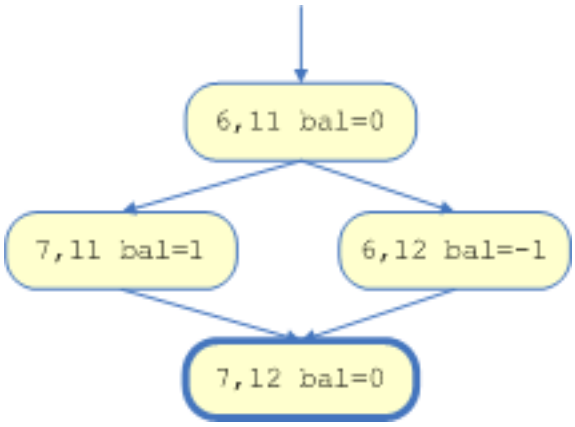


Figure 4.1: An example state space. The initial state is indicated by an incoming arrow. The valid end state is indicated by a thicker border.

The above is the state space of the Tiny First Program's code (Example 4.1: A Tiny First Program). The states are shown with the value of the only variable's value, plus the program point of each process. The edges are shown only as arrows, as the change in line numbers indicates which statement's and thread's execution each represents.

The possible traces are then all the different paths in the state space. State spaces are primarily helpful for understanding how to verify concurrent programs, and will be used mostly in the next section (Section 4.5: Verification).

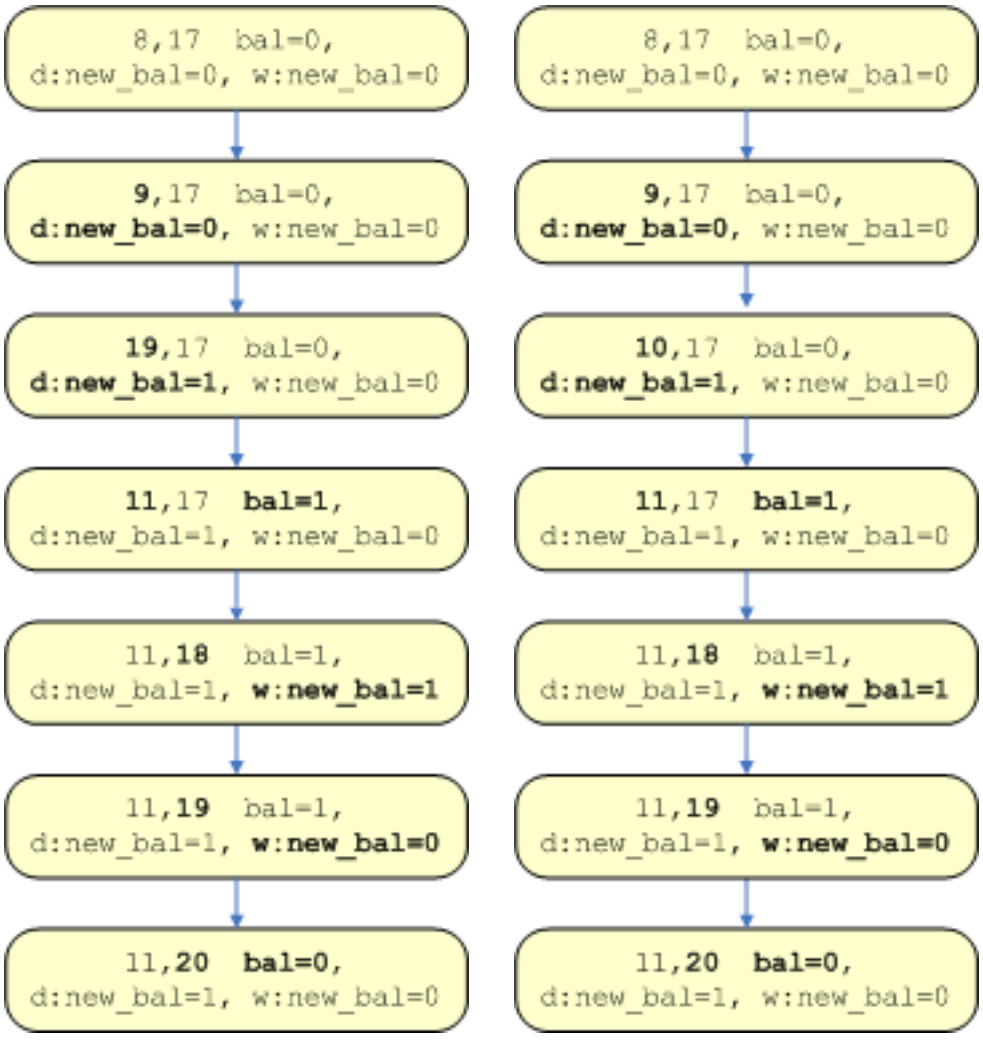
ASIDE: SPIN actually has two notions of states and state space. One form describes a single process' information, while the other describes the whole system. The latter is what we have defined and illustrated. The per-process form can be displayed in `xspin` by the "Run" menu's option to "View Spin Automaton for each Proctype". Also, the state numbering of this per-process form is reported by SPIN's output of traces. There is a close relation between the two forms, as the whole system is intuitively some sort of cross-product of the individual processes.

We now modify the previous example. The two assignment statements have each been broken up into three statements. These three simulate a typical sequence of machine-level instructions in a load/store style architecture. This is a way to model that level of atomicity within Promela.

Example 4.3: A Race Condition

```
 1  /* A variable shared between all processes. */
 2  show int bal = 0;
 3
 4  active proctype deposit()
 5  {
 6    show int new_bal;
 7
 8    new_bal = bal;
 9    new_bal++;
10    bal = new_bal;
11  }
12
```

```
13  active proctype withdraw()
14  {
15    show int new_bal;
16
17    new_bal = bal;
18    new_bal--;
19    bal = new_bal;
20  }
```



(a) (b)

Figure 4.2: Two possible traces for the above program resulting in the desired zero result.

Here, each process has its own local variable named `new_bal`. Note that we can also have SPIN `show` local variables in the "Message Sequence Chart" window. Run this code as described in the previous example.

Exercise 4.2 *(Solution on p. 50.)*

How many possible traces does this code allow? (Try it in SPIN!)

Exercise 4.3 *(Solution on p. 50.)*

More importantly, what is the result of the computation, *i.e.*, the value of `bal` at the end?

Exercise 4.4 *(Solution on p. 51.)*

Each of the previous two exercises could be very easily answered, if you were given the program's state space. Draw the state space.

Making the atomicity more fine-grained increases the possibility of concurrency problems. We will use this idea in many of the examples in this module, because we want to ensure that we can handle even the hard problems. The particular code sequence used in the previous example is also one of the shortest with a race condition.

Example 4.4: A Race Condition Variation

In Promela, initialization of a variable and assigning to a variable have subtly different semantics. This is illustrated by this variation on the previous example (Example 4.3: A Race Condition).

```
 1  /* A variable shared between all processes. */
 2  show int bal = 0;
 3
 4  active proctype deposit()
 5  {
 6    show int new_bal = bal;
 7
 8    new_bal++;
 9    bal = new_bal;
10  }
11
12  active proctype withdraw()
13  {
14    show int new_bal = bal;
15
16    new_bal--;
17    bal = new_bal;
18  }
```

In Promela, initializing variables is part of starting a process. Thus, all variables of all active processes are initialized (defaulting to 0) before any other statements are executed. Here, each copy of `new_bal` starts with the value 0, so `bal` always ends with either the value 1 or -1, but never 0.

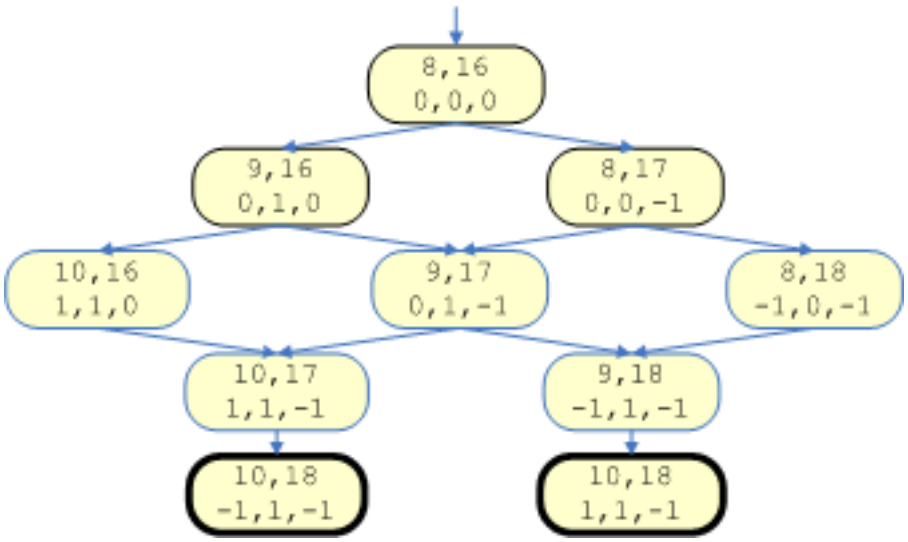


Figure 4.3: State space for above program with local variable initializations. For brevity, the variable names are elided. This leaves the information in the form of the `deposit` and `withdraw` line numbers and the values of `bal`, `deposit`'s `new_bal`, and `withdraw`'s `new_bal`, respectively.

Example 4.5: A Race Condition Generalized
The previous examples modeled two people simultaneously updating the same bank account, one making a deposit, and one a withdrawal. What if we had even more simultaneous updates? Let's focus only on deposits, so each is doing the same action. Rather than duplicating code with multiple `proctype`'s (say, `deposit1`, `deposit2`, and `deposit3`), Promela allows us to abstract that, as in the following:

```
1   /* Number of copies of the process to run. */
2   #define NUM_PROCS 3
3
4   /* A variable shared between all processes. */
5   show int z = 0;
6
7   active[NUM_PROCS] proctype increment()
8   {
9     show int new_z;
10
11    new_z = z;
12    new_z++;
13    z = new_z;
14  }
```

We've abstracted the variables' names, but otherwise, `increment` is the same as the previous `deposit`. The first substantial change is that we are starting three processes, rather than two, each with a copy of the same code. The bracket notation in the `proctype` declaration indicates how

many copies of that code are to be used. The number of copies must be given by a constant, and it is good practice to name that constant as shown.

Although vastly simplified, this is also the core of our web-based flight reservation system example (Example 2.2). The shared variable `z` would represent the number of booked seats.

Exercise 4.5 *(Solution on p. 52.)*

We can also ask questions like "What is the probability that we end with `z` having the value 3?" How would you solve this?

4.2 Guarded Statements and Blocking

Guarded statements are one of the most basic elements of the Promela language. As this section will introduce, they introduce the ability to synchronize processes. In the next section (Section 4.3: Control Flow in Promela), we see how they are also fundamental parts of the looping (`do`) and conditional (`if`) constructs. After that (Section 4.4: Process Interaction), we will return to the synchronization issues, exploring them in greater depth.

Definition 4.5: Guarded statement

A guarded statement, written in Promela as `guard -> body`, consists of two parts. The **guard** is a boolean expression. The body is a statement. Execution of the body will **block**, or wait, doing nothing, until the guard becomes true.

See Also: enabled

Example 4.6: Produce-One and Consume-One

```
1  bool ready = false;
2  int  val;
3
4  active proctype produce_one()
5  {
6    /* Compute some information somehow. */
7    val = 42;
8
9    ready = true;
10
11   /* Can do something else. */
12   printf("produce_one done.\n");
13 }
14
15 active proctype consume_one()
16 {
17   int v;
18
19   /* Wait until the information has been computed. */
20   ready ->
21
22   /* Now, we can use this. */
23   v = val;
24 }
```

This program leads to the following state space, illustrating two important traits about Promela's guards. First, the guarded statement of `consume_one` will **block** — sit and do nothing — until its

guard is satisfied. This is seen by the behavior of the first three states in the state space. Second, execution of the guard is itself a transition to a state. This allows code, such as `produce_one`'s `printf` to be interleaved between the guard and its body. (The otherwise unnecessary `printf` is there solely to make this point.) When such interruptions between the guard and body are undesired, they can be prevented, as shown later (Example 4.9: Locking to avoid race condition).

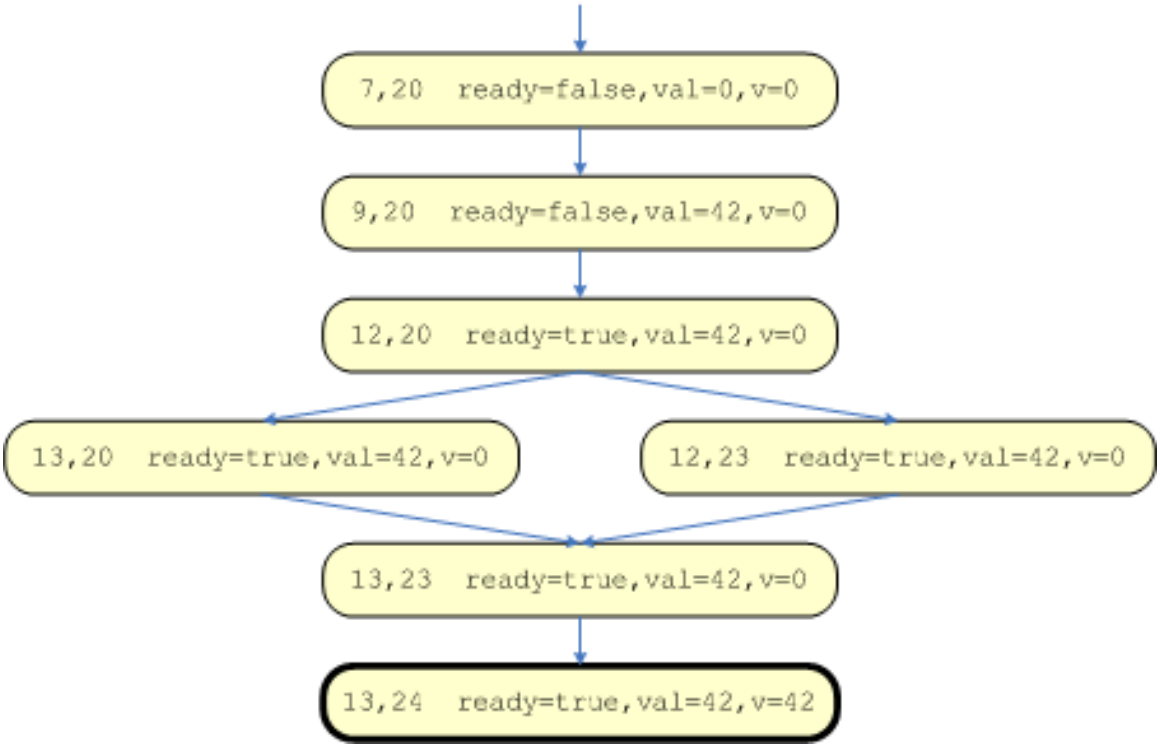


Figure 4.4: State space for the above program. The `consume_one` process cannot make progress until `produce_one` sets the `ready` variable.

Definition 4.6: Enabled
A guarded statement is enabled if its guard evaluates to true.
See Also: guarded statement, **blocked**

ASIDE: The previous definition of "enabled" is a sufficient simplification for our purposes. But more accurately, in Promela, `->` is just alternate syntax for `;`, and an expression is a valid statement by itself. Thus, we more properly define when each kind of Promela statement is enabled. In particular, boolean expressions are enabled when they evaluate to true, assignment statements are always enabled, and a compound statement

```
stmt_1 ;
... ;
stmt_n
```

is enabled if its first statement is. One simplification that this allows is that trivially true guards are unnecessary in Promela. The statement `true -> stmt` is equivalent to the simpler `stmt`.

Later (Section 4.4: Process Interaction) we will explore the possibility that a process stays blocked forever, or equivalently, never becomes enabled.

4.3 Control Flow in Promela

Of course, few interesting programs are made simply of straight-line code. We generally need to have more interesting control flow, including conditionals and iteration. These are illustrated in the remaining examples of this section.

Example 4.7: Random Walk 1
The following code illustrates use of the `do` statement, the primary iteration statement.

```
 1  #define  STARTING_DISTANCE  2
 2  show int dist = STARTING_DISTANCE;
 3
 4  active proctype stumble()
 5  {
 6    do
 7    :: true -> dist--;                        /* Approach. */
 8    :: true -> dist++;                        /* Retreat.  */
 9    :: dist <= 0 -> break;                     /* Success!  */
10    :: dist >= STARTING_DISTANCE*2 -> break;  /* Give up.  */
11    od;
12  }
```

This process consists solely of a loop from `do` to the matching `od`. The loop contains four guarded statements as alternatives, each prefaced with double colons (`::`). On each iteration, all guards are evaluated. From those evaluating to `true`, only one is chosen arbitrarily, and its body executed. (What if none are true? We'll see soon (Example 4.8: Random Walk 2).)

Executing a `break` terminates the loop. More generally, with nested loops, it terminates the most tightly enclosing loop.

A loop in the control flow leads to cycles in the state space.

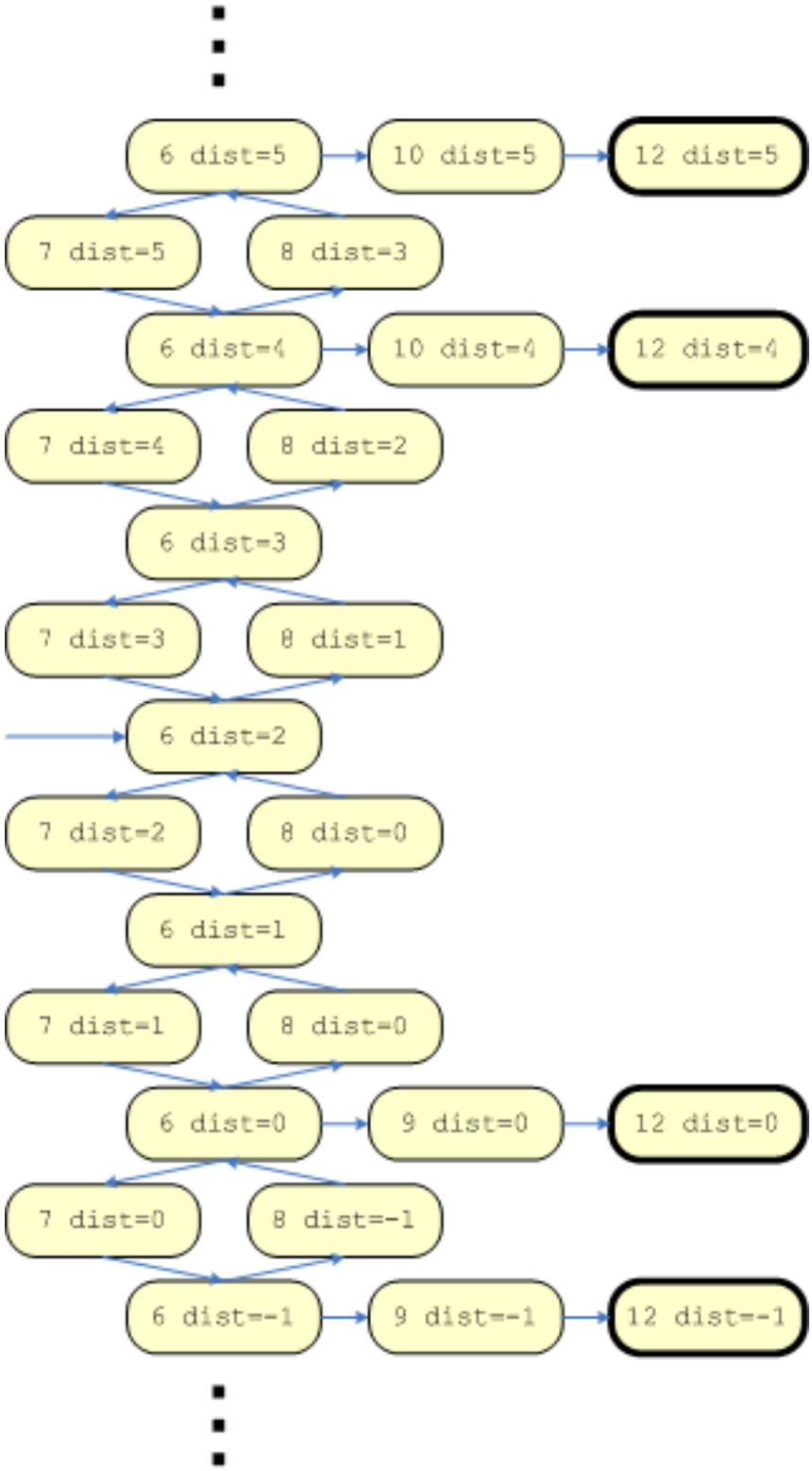


Figure 4.5: Random Walk 1 state space.

As usual, we have a state for each possible variable value. There are a finite number of states, since the `int` range is finite: $-2^{31} \ldots 2^{31} - 1$.

ASIDE: Since the large range of numbers is not needed for this example, it would have been better to use the `short` type, instead. With a smaller range, $-2^{15} \ldots 2^{15} - 1$, the state space would be significantly smaller, and SPIN could reason about the program more quickly.

Exercise 4.6
Run this code several times, observing the changing value of `dist`. As before, to see different behaviors, you'll need to change the "Seed Value" or use the "Interactive" simulation.

Exercise 4.7 ***(Solution on p. 52.)***
Is it equivalent to change the inequalities (<= and >=) to equalities (==) in this example? Why or why not?

Example 4.8: Random Walk 2
The following code is a variation on the previous, illustrating the `if` statement and the `else` keyword.

```
 1  #define  STARTING_DISTANCE  2
 2  show int dist = STARTING_DISTANCE;
 3
 4  active proctype stumble()
 5  {
 6    do
 7    :: (!(dist <= 0) &&
 8        !(dist >= STARTING_DISTANCE*2)) ->
 9       if
10       :: true -> dist--;                 /* Approach. */
11       :: true -> dist++;                 /* Retreat.  */
12       fi;
13    :: else -> break;
14    od;
15  }
```

First, consider the `do` loop. It has two guarded expressions, one with the guard `else`. On each iteration, if there are any enabled clauses, one is chosen arbitrarily, as before. But, if none are enabled the `else` clause, if any, is chosen. There may be only one `else` clause. An `else` clause is not the same as one with a `true` guard, as it can only be executed if no other clause is enabled.

Here, this means that the first clause of the `do`, *i.e.*, the entire `if` statement, is used whenever the distance is within the specified range. Otherwise, the loop is exited. Thus, unlike the previous example (Example 4.7: Random Walk 1), the loop is always exited once the distance is `0` or `STARTING_DISTANCE*2`.

Inside this loop, we have an `if` statement, which itself has two clauses. Like the `do`, it chooses arbitrarily among the enabled clauses, and may have an `else` clause.

Exercise 4.8
Run this code several times and observe its behavior.

Exercise 4.9 ***(Solution on p. 52.)***
What is the state space for this code?

Exercise 4.10 ***(Solution on p. 53.)***
Is it equivalent to change the <= and >= to == in this example? Why or why not?

Exercise 4.11 *(Solution on p. 53.)*

What is Promela's equivalent of the following C conditional statement?

```
if (condition)
   then_stmt;
else
   else_stmt;
```

Exercise 4.12 *(Solution on p. 54.)*

What is Promela's equivalent of the following C loop statement?

```
while (condition)
   body_stmt;
```

The previous examples illustrate what happens where there is a choice among guarded statements and at least one is enabled. What if none are enabled? In that case, that process is blocked, and it halts execution until it is no longer blocked.

Exercise 4.13 *(Solution on p. 54.)*

Create a Promela program illustrating synchronization and looping in the following way. One process counts down a global variable, until it reaches zero. Another process waits for the count to be zero to print a message. Run your program in "Interactive" mode and confirm that the second process is blocked, and thus does nothing, until the count reaches zero.

4.4 Process Interaction

We have so far seen very simple interactions between processes, using shared variables for communication and guarded statements for synchronization. This sections delves further into the possible interactions and introduces some additional Promela syntax for that.

While blocking is an essential tool in designing concurrent programs, since it allows process to synchronize, it also introduces the possibility of deadlock, when all processes are blocked. Since deadlock is a common problem in concurrent programs, we will later see how to verify whether a program can deadlock (Section 4.5.2: Deadlock and End States).

Exercise 4.14 *(Solution on p. 54.)*

Create a small Promela program that can deadlock. The program need not do anything computationally interesting. Run your program to see that it can get stuck.

One standard concurrent programming technique that uses guarded statements is **locking**. While details of its use are beyond the scope of this module, we'll provide a quick review. Locks are used when there is some resource or **critical section** of code that we want to control access to. For example, only one process at a time should be able to write data to a specific file, lest the file get intermingled inconsistently. To ensure this, the process must **acquire** the lock associated with writing this file before it can do the operation, and then must **release** it.

Example 4.9: Locking to avoid race condition

We can modify the Generalized Race Condition example (Example 4.5: A Race Condition Generalized) to eliminate its race condition. The lock controls access to the shared resource, *i.e.*, the shared variable `z`.

```
1   /* Number of copies of the process to run. */
2   #define NUM_PROCS 3
3
4   show int  z = 0;
5   show bool locked = false;
6
7   active[NUM_PROCS] proctype increment()
8   {
9     show int new_z;
10
11    atomic {
12      !locked ->        /* Lock available? */
13      locked = true;    /* Acquire lock.   */
14    }
15
16    /* Critical section of code. */
17    new_z = z;
18    new_z++;
19    z = new_z;
20
21    locked = false;     /* Release lock.   */
22    /* End critical section of code. */
23  }
```

As described before, when using a lock, we need to acquire the lock, execute the critical section, and release the lock. Releasing the lock is the final action of the critical section, as it allows another process to acquire the lock. To acquire the lock, clearly we need to check whether it is available. This guard is where each process potentially blocks, waiting for its turn to get the block.

The `atomic` keyword, as its name implies, makes a group of statements atomic. Without it, two processes could first each determine the lock was available, then each acquire the lock, defeating the purpose of the lock.

Exercise 4.15 *(Solution on p. 54.)*

What values of `z` are possible at the end of this program?

Exercise 4.16

Run this code several times and observe its behavior. Also, eliminate the `atomic` keyword and its associated braces, and repeat.

Exercise 4.17 *(Solution on p. 54.)*

Assume that the `atomic` keyword is removed, and there are only two processes, instead of three. Give a trace where both are in the critical section at the same time. What are the differences in the state spaces with and without the `atomic`?

Example 4.10: Tiny Server and Clients

We now have the tools to demonstrate how a tiny server and its clients can be modeled. The server process is a loop that repeatedly waits for and then processes requests. Clients send requests. For simplicity, in this example, the server won't communicate any results back to the clients, and the clients will only make one request each.

Servers generally allow multiple kinds of requests. Here, we'll use one more new piece of Promela syntax, `mtype`, to define some named constants, similar to an enumerated type.

```
/* Number of each type of client. */
#define NUM_CLIENTS 1

/* Define the named constants. */
mtype = {NONE, REQUEST1, REQUEST2};

/* Declare a shared variable.  */
show mtype request = NONE;

active proctype server()
{
  do
  :: request == REQUEST1 ->
     printf("Processing request type 1.\n");
     request = NONE;
  :: request == REQUEST2 ->
     printf("Processing request type 2.\n");
     request = NONE;
  od;
}

active[NUM_CLIENTS] proctype client1()
{
  atomic{
    request == NONE ->
    request = REQUEST1;
  }
}

active[NUM_CLIENTS] proctype client2()
{
  atomic{
    request == NONE ->
    request = REQUEST2;
  }
}
```

Function calls are another standard kind of control flow. Surprisingly, **Promela does not have function calls**! Every `proctype` instance is a separate process. Though if you really want, you could simulate a function call by creating a new process dynamically[8] , and blocking until that process returns.

4.5 Verification

So far, we have determined the possible behaviors of a program simply by running the program a bunch of times. For small programs, we can be very careful and make sure we exhibit all the possible traces, but the state space soon becomes unwieldy.

The real power of SPIN is as a tool for verification, our original goal. SPIN will **search the entire state space for us**, looking for (reachable) states which fail to have desired properties.

[8] http://www.spinroot.com/spin/Man/run.html

4.5.1 Assertions

The first verification technique we'll examine are **assertion**s, common to many programming languages. In Promela, the statement `assert(condition);` evaluates the condition. If the result is true, execution continues as usual. Otherwise, the entire program is aborted and an error message is printed.

When simulating a single run of the program, SPIN automatically checks these run-time assertions; this is the usage that most programmers should be familiar with from traditional programming languages. But additionally, we'll see that SPIN, in the course of searching the entire state space, verifies whether an assertion can **ever** fail! (Though of course it can only search finite, feasible state spaces; happily, "feasible" can often include hundreds of millions of states.)

Example 4.11
Consider our last race condition example (Example 4.5: A Race Condition Generalized). One of our original naïve expectations was that, within each process, the value of `z` at the end of the process is exactly one more than at the beginning. The previous examples have shown that to be wrong, but we had to run the program until we encountered a run when it failed. Here, the `assert` statement puts that expectation explicitly into the program, for SPIN to check.

```
 1  #define NUM_PROCS 3
 2
 3  show int z = 0;
 4
 5  active[NUM_PROCS] proctype increment()
 6  {
 7    show int new_z;
 8
 9    /* A saved copy of the old z, for the assertion. */
10    show int old_z;
11
12    old_z = z;
13    new_z = old_z + 1;
14    z = new_z;
15
16    assert(z == old_z+1);
17  }
```

It is often the case, as it is here, that to state the desired condition we need to add an extra variable — here, `old_z`. As always, it is important that when introducing such code for testing that you don't substantially change the code to be tested, lest you inadvertently introduce new bugs!

Exercise 4.18
Run this code several times, and observe when the assertion fails. This text indicates which assertion failed, and the line will be highlighted in the code window. To see which process' copy of `increment` failed and why, you have to look more closely at the steps shown.

```
spin: line  16 "pan_in", Error: assertion violated
spin: text of failed assertion: assert((z==(old_z+1)))
```

Certainly this run-time check is helpful, but what about the promise of checking whether an assertion **can** fail? We will need to use SPIN as a **verifier**, rather than just a **simulator**.

As with simulation, before our first verification, we need to specify some parameters, even if only using the defaults. From the "Run" menu, select the "Set Verification Parameters" options. For this example, make sure that the "Assertions" option is marked. Use the "Run" button to start the verification.

NOTE: Using `spin` by hand involves three steps: we must first create a verifier (a C program) from our promela source; then compile the verifier, and then run the verifier:

```
prompt>  spin -a filename.pml
prompt>  gcc pan.c -o filename
prompt>  filename
```

The `-a` flag tells SPIN to generate a verifier, which it always names `pan.c`.

As desired, the verification finds a problem, as reported in the "Verification Output" window. It also saves a description of the example run found where the assertion fails.

```
pan: assertion violated (z==(old_z+1)) (at depth 11)
pan: wrote pan_in.trail
```

The remainder of this window's output is not too important, as it describes which verification options were used and how much work was done in the verification.

ASIDE: `pan` is the standard SPIN default name for files related to Promela verification. With `xspin`, you'll see these filenames reported, but you don't need to use or remember them.

Now we can have SPIN show us the sample failed run it found. In the new "Suggested Action" window, select "Run Guided Simulation". This brings up the now-familiar simulation windows, which will show this particular run. This run is "guided" by a trace saved by the verification. (Alternatively, selecting "Setup Guided Simulation" allows you to change the display options first.)

NOTE: Running a guided simulation from the command line is similar to running a regular simulation, except you provide the `-t` flag to have SPIN use the pre-named trail file. It's often handy to use the `-p` flag to have SPIN print each line it executes:

```
prompt>  spin -t -p filename.pml
```

The trace found in our verification executed some of each thread. From our experience with this program, we know there are less complicated traces where the assertion fails. How can we find them?

Again, in the "Run" menu, select "Set Verification Parameters". Now, select "Set Advanced Options", then "Find Shortest Trail", and "Set". Then run the verification again.

```
pan: assertion violated (z==(old_z+1)) (at depth 11)
pan: wrote pan_in.trail
pan: reducing search depth to 10
pan: wrote pan_in.trail
pan: reducing search depth to 5
```

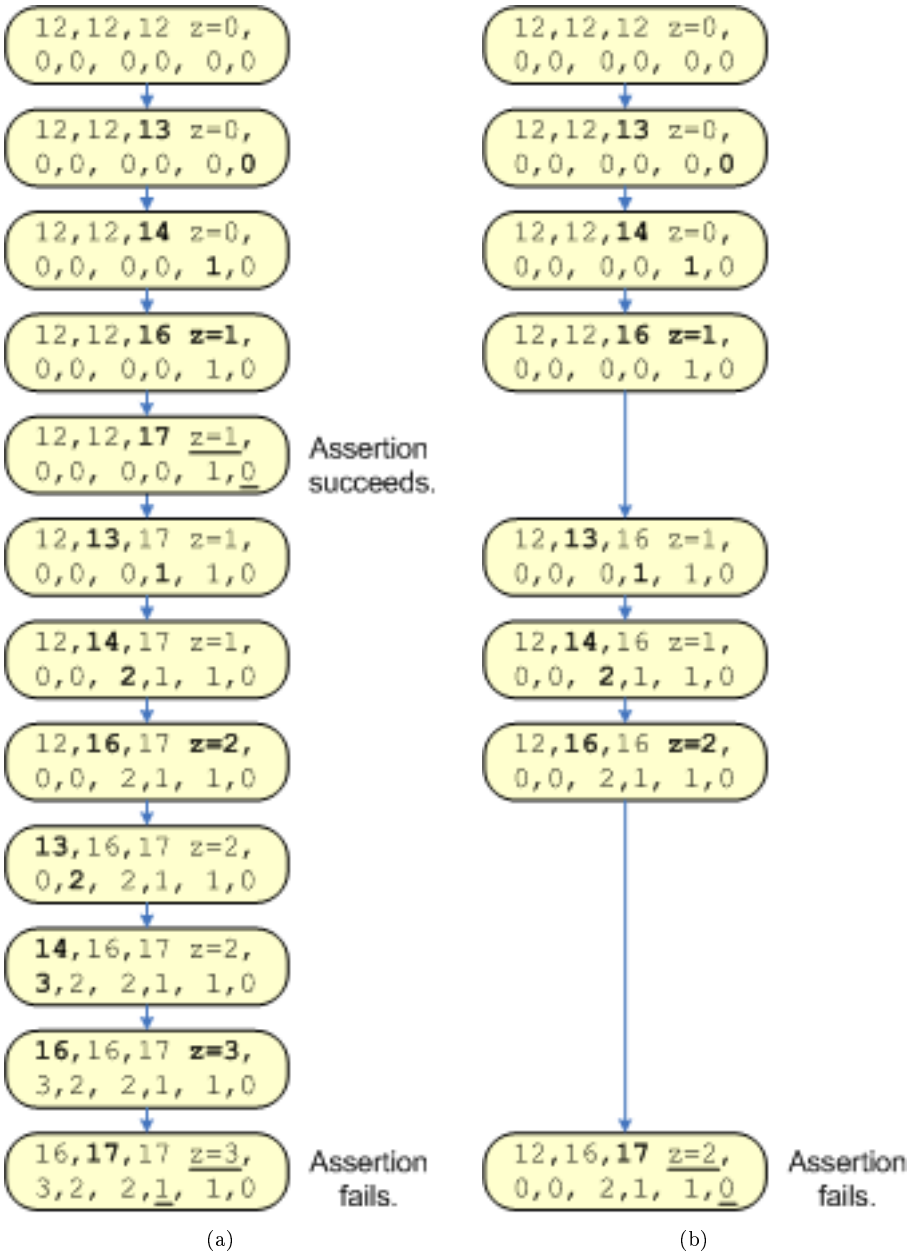


Figure 4.6: A nonminimal and a minimal trace failing the assertion in previous example (Example 4.11). For brevity, some variable names are omitted. The second line shows the `new_z` and `old_z` for each process, respectively. For the assertions, the relevant variables are underlined.

ASIDE: Experiment on your own with the other advanced options under "Error Trapping". Using breadth-first search is an alternate strategy to finding a shortest trace with an error.

Example 4.12
Previously (Example 4.11), we claimed that adding a lock fixed the race condition of the previous example. Let's now verify that claim. The following is the locking version of the code augmented with the same assertion code just used. Note that the assertion needs to be inside the critical section. (Otherwise, process B could execute its critical section, changing z, in between process A's critical section and its assertion.)

```
/* Number of copies of process to run. */
#define NUM_PROCS 3

show int  z = 0;
show bool locked = false;

active[NUM_PROCS] proctype increment()
{
  show int new_z;

  /* A saved copy of the old z, for the assertion. */
  show int old_z;

  atomic {
    !locked ->        /* Lock available? */
    locked = true;    /* Acquire lock.   */
  }

  /* Critical section of code. */
  old_z = z;
  new_z = old_z + 1;
  z = new_z;

  assert(z == old_z+1);

  locked = false;     /* Release lock.   */
  /* End critical section of code. */
}
```

Exercise 4.19
Run the verification as before. As expected, no errors are found. Happily, this time no error messages are reported before the version information in the "Verification Output" window, and SPIN doesn't suggest any guided simulation.

4.5.2 Deadlock and End States

4.5.2.1 Deadlock

A previous exercise (Exercise 4.14) asked you to write a small deadlocking program, and provided one solution. Running such a program in SPIN's simulation mode, some execution paths result in deadlock, *i.e.*, every single process waiting on other processes to do something first. How can we verify whether or not deadlock is possible? SPIN checks for deadlock automatically, as one of a hand-full of built-in checks, so we will see how that is reported.

Run SPIN's verifier on your tiny deadlocking program. The deadlock is reported by a somewhat obscure error message:

```
pan: invalid end state (at depth 0)
pan: wrote pan_in.trail
```

As usual, `xspin` allows you to easily run the guided simulation it found.

Deadlock occurs in a state when nothing further will happen—that is, there are no outgoing edges in the state space. We call this an **end state**. Most programs we've seen previously have a valid end state: all processes complete their last line. In contrast, in deadlock we are in an end state, yet at least one process hasn't completed. Hence "invalid". SPIN checks for end states by searching the entire state space (either by a depth- or breadth-first search).

4.5.2.2 Valid End States (optional)

We've mentioned that valid end states are those where every process has completed its last line, and that deadlock ends with at least one process which hasn't completed. However, that's not enough to conclude deadlock: sometimes it is desirable to have a trace reach an end-state with a process still (say) waiting for more input. Consider a server and its clients. By default, the server waits and does nothing. Only when a client makes a request, the server responds and does something. If all clients exit, it is expected, not an error, for the server to be blocked waiting for another request. In SPIN, we need to notate that it is valid for the server to end at that point, and not actually deadlock.

If we run SPIN's verifier on the code for the Tiny Server and Clients example (Example 4.10: Tiny Server and Clients), we get a spurious "invalid end state" error, since we **expect** the client to be alive and listening for for any more requests.

To let SPIN know that is acceptable and shouldn't actually be considered deadlock, we label the beginning of the server loop as being a valid end state: We give it a label whose first three letters are "`end`", as shown below. Run the verifier again on this, and there should be no error message.

```
 1  /* Number of each type of client. */
 2  #define NUM_CLIENTS 1
 3
 4  /* Define the named constants. */
 5  mtype = {NONE, REQ1, REQ2};
 6
 7  /* Declare a shared variable.  */
 8  show mtype request = NONE;
 9
10  active proctype server()
11  {
12   /* Waiting for a request is a valid place to end. */
13   endwait:
14    do
15    :: request == REQ1 ->
16       printf("Processing request type 1.\n");
17       request = NONE;
18    :: request == REQ2 ->
19       printf("Processing request type 2.\n");
20       request = NONE;
21    od;
```

```
22  }
23
24  active[NUM_CLIENTS] proctype client1()
25  {
26    atomic {
27      request == NONE ->
28      request = REQ1;
29    }
30  }
31
32  active[NUM_CLIENTS] proctype client2()
33  {
34    atomic {
35      request == NONE ->
36      request = REQ2;
37    }
38  }
```

More formally, what this does is to tag some of the states in the state space as being valid, if they happen to be an end-state.

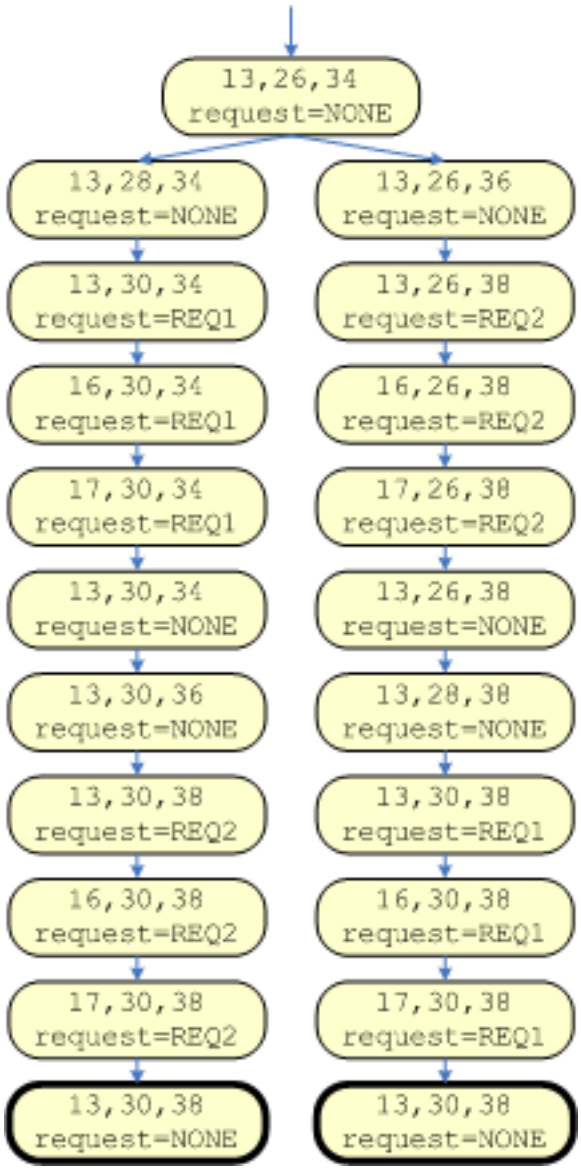


Figure 4.7: The state space for the above program. The `endwait` label makes the bottom two states be valid end states.

Putting the `end` label in front of the entire `do` statement may not seem as natural to you as:

```
do
::
 end1:
  request == REQ1 ->
  ...
```

```
::
 end2:
  request == REQ2 ->
  ...
od;
```

However, this is **not** acceptable! Try it, by entering the changes and running a syntax check. The problem is that `end1` and `end2` both represent the same program point, where the program waits for some guard to become true. It would be nonsensical to have (say) one of the two guards labeled an end-state without the other one being an end-state. To prevent surprising inconsistencies, Promela disallows labels in front of individual guards of a compound statement.

There are other syntactic restrictions of where labels can appear. The most commonly encountered is that they cannot appear at the end of a statement block. For example, instead of

```
{
  x = 1;
  y = 2;
 label:   /* Label not allowed here. */
}
```

you can introduce a dummy statement, and label it:

```
{
   x = 1;
   y = 2;
  label:
   skip;  /* A statement that does nothing. */
}
```

4.5.3 Mutual Exclusion — a morality play (optional)

It's worth showing several examples of correct and incorrect concurrent programs, and how SPIN can implement and attempt to verify them. We'll examine a sequence of programs, all dealing with mutual exclusion protocols. We hope to leave the gentle reader with an appreciation of the non-obvious nature of concurrent bugs (and hence the value of automated verification).

To get started, we revisit the fundamental race condition, phrased in terms of two processes simultaneously each being in their critical section.

```
     1  /* An INCORRECT attempt at Mutual exclusion.
 2   * Adapted from Ruys 2002: SPIN Beginner's Tutorial.
 3   */
 4
 5  bool flag = false;  /* Is some process in its crit section? */
 6  int  data = 0;
 7
 8  active[2] proctype proc()
 9  {
10    do
```

```
11    :: flag == false ->  /* Wait until nobody in crit section. */
12       flag = true;
13
14       /* Entering critical section. */
15       data++;
16       assert(data == 1);
17       data--;
18       /* Leaving critical section. */
19
20       flag = false;
21    od;
22  }
```

Exercise 4.20
Take a moment to run SPIN's verifier, and confirm that the `assert` statement indeed might be violated.

Here is another attempt which doesn't work. Can you see how it might go awry?

```
 1  /* An INCORRECT attempt at Mutual exclusion.
 2   * Version 2.
 3   * Adapted from Ruys 2002: SPIN Beginner's Tutorial.
 4   */
 5
 6  bool flagA = false;  /* A wants to enter its crit section. */
 7  bool flagB = false;  /* B, similarly. */
 8  int  data = 0;
 9
10  active proctype A()
11  {
12    do
13    :: flagA =  true;       /* Declare intent to enter crit section. */
14       flagB == false ->    /* Wait for B to leave their crit section, if in. */
15
16       /* Entering critical section. */
17       data++;
18       assert(data == 1);
19       data--;
20       /* Leaving critical section. */
21
22       flagA = false;
23    od;
24  }
25
26
27  active proctype B()
28  {
29    do
30    :: flagB =  true;       /* Declare intent to enter crit section. */
31       flagA == false ->    /* Wait for A to leave their crit section, if in. */
32
33       /* Entering critical section. */
34       data++;
```

```
35      assert(data == 1);
36      data--;
37      /* Leaving critical section. */
38
39      flagB = false;
40    od;
41  }
```

Exercise 4.21
Again, use SPIN to verify that the assertion can be violated. (Keep in mind that we can't use `atomic (Example 4.9: Locking to avoid race condition)` (or `d_step`) here; the entire point of a mutex algorithm is to be able to implement multi-statement atomicity using nothing but shared memory and the atomicity of simple-assignment.)

By being substantially more clever though, interacting flags can be used to get two processes to share a critical section:

```
 1  /* A correct attempt at Mutual exclusion, huzzah!
 2   * Peterson, 1981.
 3   */
 4
 5  bool flagA = false;  /* A wants to enter its crit section? */
 6  bool flagB = false;  /* B, similarly. */
 7  pid  turn;           /* For politeness, offer other people a turn. */
 8  int  data = 0;
 9
10
11  active proctype A()
12  {
13    do
14    :: flagA = true;        /* Declare intent to enter crit section. */
15       turn  = 1 - _pid;    /* Offer the turn to the other. */
16
17       ((flagB == false) || (turn == _pid)) ->
18
19       /* Entering critical section. */
20       data++;
21       assert(data == 1);
22       data--;
23       /* Leaving critical section. */
24
25       flagA = false;
26    od;
27  }
28
29  active proctype B()
30  {
31    do
32    :: flagB = true;        /* Declare intent to enter crit section. */
33       turn  = 1 - _pid;    /* Offer the turn to the other. */
34
35       ((flagA == false) || (turn == _pid)) ->
36
```

```
37       /* Entering critical section. */
38       data++;
39       assert(data == 1);
40       data--;
41       /* Leaving critical section. */
42
43       flagB = false;
44     od;
45   }
```

Processes `A` and `B` are using their flags in symmetric ways, and we can certainly factor their common code to get two instances of the same proctype:

```
     1  /* A correct attempt at Mutual exclusion, huzzah!
 2   * Peterson, 1981.
 3   */
 4
 5  bool flag[2] = false;  /* Process #i wants to enter its crit. section? */
 6  pid  turn;             /* For politeness, offer other people a turn. */
 7  int  data = 0;
 8
 9  active[2] proctype P()
10  {
11    pid me, peer;
12
13    me   =     _pid;
14    peer = 1 - _pid;
15
16    do
17    :: flag[me] = true;   /* Declare intent to enter crit section. */
18       turn = peer;       /* Politely give others a chance, first. */
19
20       ((flag[peer] == false) || (turn == me)) ->
21
22       /* Entering critical section. */
23       data++;
24       assert(data == 1);
25       data--;
26       /* Leaving critical section. */
27
28       flag[me] = false;
29    od;
30  }
```

A paper proof of this algorithm requires careful thought; fortunately verifying this with SPIN is much easier. Note that if you wanted to tweak an innocuous line or two of Peterson's algorithm, without SPIN you'd have to reason extremely carefully to be sure you didn't lose mutual exclusion.

A different algorithm for mutual exclusion, the Bakery algorithm, can be thought of as having a roll of sequentially-numbered tickets; when a process wants access to the critical section (the bakery clerk), they take a number, and then look around to be sure they have the lowest ticket number of anybody, before striding up to the counter. The tricky bit is that "taking a number" means both copying the value of a global counter, and updating that counter, in the presence of other processes who might be doing so simultaneously. It seems like it's begging the question, to even do that much!

Promela code for a two-process implementaton of the Bakery algorithm is below; it can be generalized to n processes [9]

```
 1  /* A correct attempt at Mutual exclusion.
 2   * Lamport's "Bakery algorithm"
 3   * Procs which communicate only through ticket[].
 4   */
 5
 6  byte ticket[2];      /* 'byte' ranges from 0..255.  Initialized to 0. */
 7  int  data = 0;
 8
 9  active [2] proctype customer()
10  {
11    pid me   =   _pid;
12    pid peer = 1-_pid;             /* Special case for 2 procs only. */
13
14    skip;                          /* Doing some non-critical work... */
15
16    ticket[me] = 1;                /* Declare desire to enter crit section. */
17    ticket[me] = ticket[peer]+1;   /* In general: max(all-your-peers)+1.    */
18    ((ticket[peer] == 0) || (ticket[me] < ticket[peer])) ->
19
20    /* Entering critical section. */
21    data++;
22    assert(data == 1);
23    data--;
24    /* Leaving critical section. */
25
26    ticket[me] = 0;                /* Relinquish our ticket-number. */
27  }
28
29
30  /* The meaning of ticket[], for each process:
31   * ticket[me]==0 means I am not contending for crit section.
32   * ticket[me]==1 means I am in the process of taking a ticket
33   *               (I am looking at all other ticket holders)
34   * ticket[me] > 1 means I have received a ticket and will
35   *              enter the critical section when my turn comes up.
36   *
37   * A problem still present, when trying to generalize to > 2 procs:
38   *   Two procs may choose the same ticket-number;
39   *   when nobody has a smaller ticket than you, check them all for equal tix;
40   *   must break any ties reasonably (eg by smaller _pid).
41   *
42   * It's a bit sneaky, that 0 and 1 have special meanings
43   *  (and the fact that 1 is less than any other "valid" ticket-number
44   *  is important -- while anybody is calculating their ticket,
45   *  nobody else will enter the critical section.
46   *  This inefficiency can be avoided by being cleverer.)
```

[9] As the comments suggest, references to `ticket[peer]` must be replaced by looping to find the maximum ticket-number among all peers. Moreover, it becomes possible for two processes to grab the same ticket-number in this way. This situation has to be checked for, and some way of breaking ties included (*e.g.* smallest process ID number (`_pid`) wins).

```
47   */
```

SPIN can be used to verify this code. But it is worth taking the time to reason through the code [10] in your head, to compare your brain's horsepower with SPIN's steam-engine technology.

You might have noticed that this example (unlike earlier ones) only has each process perform their critical section once. No problem, we can put that in a `do` loop:

```
     1  /* An overflow bug sneaks into our Mutual exclusion, uh-oh!
 2   * Lamport's "Bakery algorithm"
 3   * Procs which communicate only through ticket[].
 4   */
 5
 6  byte ticket[2];
 7  int  data = 0;
 8
 9  active [2] proctype customer()
10  {
11    pid me   =   _pid;
12    pid peer = 1-_pid;
13
14    do
15    :: skip;                        /* Doing some non-critical work... */
16
17       ticket[me] = 1;              /* Declare desire to enter crit section. */
18       ticket[me] = ticket[peer]+1;
19       ((ticket[peer] == 0) || (ticket[me] < ticket[peer])) ->
20
21       /* Entering critical section. */
22       data++;
23       assert (data == 1);
24       data--;
25       /* Leaving critical section. */
26
27       ticket[me] = 0;              /* Relinquish our ticket-number. */
28    od;
29  }
```

SPIN verifies thi—wait a moment, the verification fails! And at depth 2058, no less. What's going on? A clue can be found in scrutinizing the output:

```
spin: line  17 "mutexG2.pml", Error: value (256->0 (8)) truncated in assignment
```

Our variable `ticket`, a `byte`, is overflowing!

Exercise 4.22 ***(Solution on p. 55.)***

1. Describe one example trace, where `ticket` can keep incrementing until overflow.
2. Does this overflow necessarily happen in every run?
3. What is the probability that this overflow will occur?
4. While changing the type of `ticket` from `byte` to `int` changes nothing conceptually, it **does** change SPIN's verification. Re-run the verification with this change. What is the difference?

[10] Note that the particular value `ticket == 1` is important — it corresponds to "I'm still calculating max of others", and doing so has priority over anybody entering the process.

The moral of the story is that concurrent protocols can difficult and subtle. The SPIN book[11] puts it well: "The number of incorrect mutual exclusion algorithms that have been dreamt up over the years, often supported by long and persuasive correctness arguments, is considerably larger than the number of correct ones. [One exercise (Exercise 7.4)], for instance, shows a version, converted into Promela, that was recommended by a major computer manufacturer in the not too distant past."

The following are some other ideas, each of these can form the kernel of a successful mutex algorithm.

- First-come, first-served. Within Promela/SPIN, we assume one statement is executed at a time, so processes must arrive at the critical section in some particular order.
- Pre-determined priority order, *e.g.*, by process ID.
- Least recently used. *I.e.*, processes defer to others that haven't recently executed this code.
- Random. *E.g.*, each process randomly chooses a number, with ties broken randomly also.

4.5.4 Issues in writing feasibly-verifiable programs (optional)

Example 4.13: Banking

Our first Promela examples were drastically simplified models of banking, with deposits and withdrawals. Now, let's consider a production-quality system of bank accounts. Its code base could easily entail millions of lines of code.

Before we start porting the entire system to Promela, let's consider what it is we want to use SPIN to verify. Ideally, we want to verify everything, including ensuring that each account balance is correct. But, think about that in conjunction with SPIN's approach using state spaces.

Even with only one bank account, if we model balances accurately, we need an infinite number of states — one for each possible balance. Similarly, the production system likely has no bound on the number of possible accounts. SPIN would have a wee bit of difficulty searching the entire state space in finite time. To make verification feasible, all SPIN models are required to be finite. (All data types, including `int`, have a finite range.)

How could we restrict the Promela program to guarantee finiteness? The most obvious and least restrictive option is to simply use the `int` type and ensure that any examples used should not cause overflow.

To make verification efficient, the state space should be relatively small. Even using `int`s as bank balances and account numbers, assuming only 1000 lines of code and ignoring any other program variables, we still have $2^{32}2^{32}1000$ states — over 18 sextillion. Even at a billion states per second, a verification could still take over half a millenium!

Let's abandon the goal having our Promela prototype track the particular balances accurately. Since our focus is on concurrency, we mainly want to ensure that when there are multiple simultaneous transactions, we don't lose or mix up any of the data. We might also want to verify the security protocols used in selecting appropriate accounts. But, we no longer need to even keep track of balances. Furthermore, it is highly unlikely that the production code might have errors that **only** occur when there are a large number of accounts, so we can comfortably use just a few.

Since we have not fully specified this example's original system, we will leave the details to the reader's imagination.

4.5.5 The lost Pathfinder (optional)

In 1997, the Mars Pathfinder[12] seemed to beam its data back to Earth just fine, except— sometimes it would appear to freeze up. The "fix"? Press Control-Alt-Delete from Earth, and re-boot its rover (incurring not only an expensive delay for Mission Control, but the full attention of late-night-TV comedians).

[11] http://spinroot.com/spin/Doc/Book_extras/index.html
[12] http://mpfwww.jpl.nasa.gov/default.html

The cause turned out to be a bug in the concurrency protocol, between a low-priority data-gathering mode, which was meant to yield to a high-priority data-sending mode. However, sometimes these priorities would become inverted, with the high-priority thread blocking on the (supposedly) low-priority one. (More detail.)[13] The bug [14] can be realized in Promela:

```
 1  /* From Spin primer/manual, Gerard Holzman. */
 2
 3  mtype = { free, busy };             /* states for the variable "mutex" */
 4  mtype = { idle, waiting, running }  /* states for the processes */
 5
 6  mtype highProc = idle;
 7  mtype  lowProc = idle;
 8  mtype mutex    = free;
 9
10  active proctype high_priority()
11  {
12   end:
13    do
14    :: highProc = waiting;
15       atomic {
16         mutex == free ->
17         mutex = busy
18       };
19       highProc = running;
20       /* work work work; produce data */
21       atomic {
22         highProc = idle;
23         mutex = free
24       }
25    od
26  }
27
28  active proctype low_priority() provided (highProc == idle)
29  {
30   end:
31    do
32    :: lowProc = waiting;
33       atomic {
34         mutex == free ->
35         mutex = busy
36       };
37       lowProc = running;
38       /* work work work; consume data and send it back to Earth. */
39       atomic {
40         lowProc = idle;
41         mutex = free
42       }
43    od
44  }
```

[13] http://www.time-rover.com/Priority.html

[14] This is of course not the full Sojourner Rover code, but it is the concurrency-protocol skeleton. The other thousands of lines of code would, of course, be omitted from a Promela prototype.

```
45
46
47  /* Note the "provided" clause as part of low_priority()'s declaration.
48   * This condition is monitered and confirmed before every single
49   * transition that low_priority() attempts.
50   * (This is a handier syntax than placing that guard condition
51   *  in front of every single statement within the process.)
52   */
```

Exercise 4.23
Can you detect the problem using SPIN?

4.5.6 Non-progress: livelock, starvation, and fairness

4.5.6.1 Livelock

As described previously (Example 2.3: Deadlock vs. Livelock), livelock is very similar to deadlock. But in livelock, computation doesn't get completely stuck — it simply doesn't do anything sufficiently interesting.

In previous examples, processes waited with a guarded statement. Here, as an example of **busy waiting**, we repeatedly check for the desired condition, and otherwise do something. Busy waiting for a condition that never comes true is one common situation that leads to livelock.

```
 1   show int x = 1;
 2
 3   active[2] proctype busywait()
 4   {
 5     do
 6     :: x != 0 ->
 7        printf( "%d waiting.\n", _pid );
 8     :: else ->
 9        break;
10     od;
11
12     printf( "Now this is more interesting.\n" );
13   }
```

This example also uses `_pid`, a read-only variable whose value is the process ID number (0-based) of the given process.

Now we want to verify this code. From the "Run" menu, select "Set Verification Parameters" and then turn on "Liveness" checking and "Non-Progress Cycles".

> NOTE: As usual, we use `-a` to have SPIN generate a verifier. Then, when compiling `pan.c`, we must include a flag to check for non-progress, `-DNP`. Finally, when running the resulting executable, we must also specify loop-checking via the flag `-l`.

```
prompt>  spin -a filename.pml
prompt>  gcc -DNP pan.c -o filename
prompt>  filename -l
```

(With all the varying flag names, you can begin to appreciate `xspin`'s interface!)

Verifying, SPIN now gives the following error message, declaring it is possible for the code to not make progress:

```
pan: non-progress cycle (at depth 6)
pan: wrote pan_in.trail
```

Now run the guided simulation. In the "Simulation Output" window, the start of the cycle is so labeled, and the end of the cycle is the last step displayed. Pictorially, the trace is as follows.

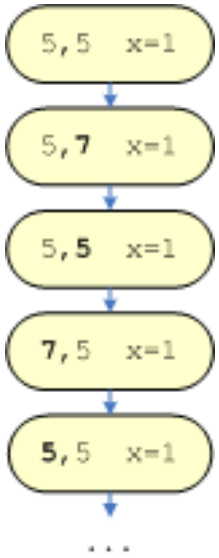


Figure 4.8: Trace of a non-progress cycle. Naturally, since it is a cycle, it repeats forever.

4.5.6.2 Starvation

Whereas livelock is when computation continues but no process makes actual progress, starvation is when computation continues, and some processes make progress but others don't.

Consider the following code. (Clearly, it does not compute anything interesting.)

```
 1  show int x = 0;
 2
 3  active proctype A()
 4  {
 5    do
 6    :: true -> x = 1 - x;
 7    od
 8  }
 9
10  active proctype B()
11  {
```

```
12    do
13    :: true -> x = 1 - x;
14    od
15  }
```

Our naïve expectation and desire is that `A` and `B` each get executed infinitely often (though not necessarily exactly alternating). The problem is that we could get unlucky — very [15] unlucky. It is possible for, say, `A` to get all the processor time, while `B` never even gets executed once, ever. More generally, maybe `B` gets its fair share of processor time for a while, but later priority is always given to `A`, and `B` never runs again.

This is simply another instance of a lack of progress. For example, if we consider only `B`'s loop to be progress, then there is a non-progress cycle.

Exercise 4.24 ***(Solution on p. 56.)***

Modify the code to test for this non-progress cycle, and then run SPIN's verifier.

Exercise 4.25 ***(Solution on p. 58.)***

How can we test for the possibility of `A` looping forever without executing `B`?

Exercise 4.26 ***(Solution on p. 58.)***

Of course, the faulty behavior we'd really like to have SPIN alert us to is "`A` **or** `B` is being starved", instead of having to choose one and name it.

Why does putting `progress` labels in both `A` **and** `B` **not** achieve what we want?

4.5.6.3 Fairness

Process scheduling software is usually written so as to avoid starvation. For example, a simple scheduler could simply pick processes in a round-robin fashion, executing some of each process 0, 1, 2, ..., 0, 1, 2, ... forever. However, this scheme inappropriately schedules time for blocked threads, doesn't allow the processes to be prioritized, and doesn't allow the set of running processes to change dynamically. Real schedulers manage not only the order, but also the frequency of context switches.

So far, we have considered verification with an arbitrary scheduler. Since we know nothing about the scheduler, we must consider all possible traces. (Remember that the scheduler might sometimes be nature, or even an adversary.) But, SPIN also allows you to specify some more restricted scheduling behaviors, thus reducing the state space. We will look at one of these mechanisms in SPIN — enforcing "weak fairness".

Definition 4.7: Weak Fairness

Each statement that becomes enabled and remains enabled thereafter will eventually be scheduled.

(Weak fairness is but one of many notions of fairness.)

Example 4.14

Let us return to our starvation example (p. 46), but this time we will have SPIN enforce weak fairness. To turn this feature on in `xspin`, select it from the verification options.

NOTE: The steps for creating and compiling the verifier are unchanged, but we execute the verifier with a flag `-f` for fairness, as well as `-l` for loop-checking.

```
prompt>  spin -a filename.pml
prompt>  gcc -DNP pan.c -o filename
prompt>  filename -f -l
```

[15] Insert our usual "nondeterministic, not random" caveat (Exercise 4.5) here.

```
 1  show int x = 0;
 2
 3  active proctype A()
 4  {
 5    do
 6    :: true -> x = 1 - x;
 7    od
 8  }
 9
10  active proctype B()
11  {
12    do
13    :: true -> x = 1 - x;
14    od
15  }
```

In this code, each process is always enabled, since each guard is just `true`. Thus, a weakly fair scheduler guarantees that at each point in the trace, each process will eventually be scheduled in the future. *I.e.*, it will repeatedly switch between the two processes as desired.

Problem

Now verify there are no non-progress cycles when weak fairness is guaranteed. *I.e.*, add the progress label to `B` again and verify that no errors are reported. Repeat with the progress label in `A`.

Example 4.15

Weak fairness is defined not in terms of processes, but of individual statements. This example illustrates how it applies to situations other than avoiding starvation, and in particular, when there is only one process.

```
show int x = 0;

active proctype loop()
{
  do
  :: true -> x = 1;
  :: x==1 -> x = 2;
  :: x==2 -> x = 3;
  od;
}
```

Without any kind of fairness enforced, it is possible to repeatedly execute only the first option.

Exercise 4.27 ***(Solution on p. 59.)***

Verify the previous statement. Where would you add progress labels?

Exercise 4.28

Now verify again with weak fairness enforced.

After `x` becomes 2, both the first and third options are enabled. Weak fairness is not sufficient to guarantee that the third will be ever be picked, as `x` could alternate between the values 1 and 2 forever. Since it wouldn't have the value 2 continuously, the weak fairness guarantee doesn't apply to the third option.

Exercise 4.29 ***(Solution on p. 59.)***

Verify this claim. Where are progress labels needed now?

ASIDE: **Strong fairness** states that if a statement is enabled infinitely often, it will eventually be executed. That would be sufficient to guarantee the third option is eventually chosen. SPIN does not have a built-in switch for enforcing strong fairness.

4.5.6.4 Some Unexpected Progress (optional)

Does the following code have a non-progress loop?

```
 1  int x = 0;
 2
 3  active proctype A()
 4  {
 5   progressA:
 6    /* Consider: A reaches this line, but never executes it. */
 7    x = 1;
 8  }
 9
10  active proctype B()
11  {
12    do
13    :: x == 0 -> skip;
14    :: x == 1 ->
15       progressB: skip;
16    od;
17  }
```

It does if `A` never runs, and `B` runs forever with `x` being zero. (Of course, this only happens if fairness is not being enforced.) But surprisingly, **SPIN does not report any non-progress cycles!** It views `A` as forever sitting in the state `progressA`, and thus feels that progress is always happening (even though that statement is never executed).

This mismatch isn't due to the fact that SPIN associates labels with the state just **before** executing the labeled statement; even if they were associated with the statement preceding the label, we could still construct a situation where SPIN views `A` as making progress even though it is doing nothing at all (just idling in a state tagged as a progress).

The fundamental mismatch is that to you and me, the idea of "making progress" corresponds to **making certain transitions**. But SPIN views labels only as relating to states. If there are transitions which don't correspond to progress, yet arrive at a labeled-progress-state (in particular: self-loops, possibly introduced for stutter-extension), then the state-based and transition-based approaches differ.

We'll see in the next section a way to still capture our transition-based notion with SPIN. Nevertheless, beware that sometimes a particular tool might have some surprising notions.

Solutions to Exercises in Chapter 4

Solution to Exercise 4.1 (p. 17)

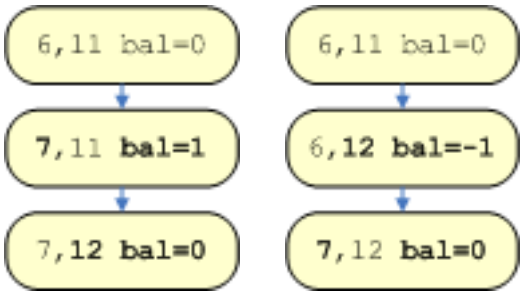


Figure 4.9: The two possible traces for the above program.

Did you see each one using SPIN? Each trace ends with `bal` equal to 0, as expected.

ASIDE: There are some minor differences between SPIN's behavior and the somewhat simplified conceptual behavior that we are describing; see the following list

- SPIN adds an extra transition and state at the end of each process. You can think of the transition as the process exiting, but it doesn't change any of the variables' values. We omit this, except for one homework exercise (Exercise 7.4).
- Marking variables with `show` introduces transitions and states to display their changing values. We have omitted these.
- Finally, as a minor presentational detail, SPIN doesn't report the start state of a trace like we do.

Solution to Exercise 4.2 (p. 20)
This code has 70 possible traces — **many** more than the previous version. Even with such a small program, it is too cumbersome to try to list them all. This is a fundamental reason why it is so hard for humans to debug a concurrent program, since it involves understanding all the possible ways threads can interact. Once we start using SPIN as a verifier, letting it enumerate these as a part of verification will be one of the benefits of automation. Later, we will verify this specific example (Example 4.4: A Race Condition Variation).
Solution to Exercise 4.3 (p. 20)
It can be 0, as expected, but it can also be 1 or -1! These new possibilities are the result of traces such as the following:

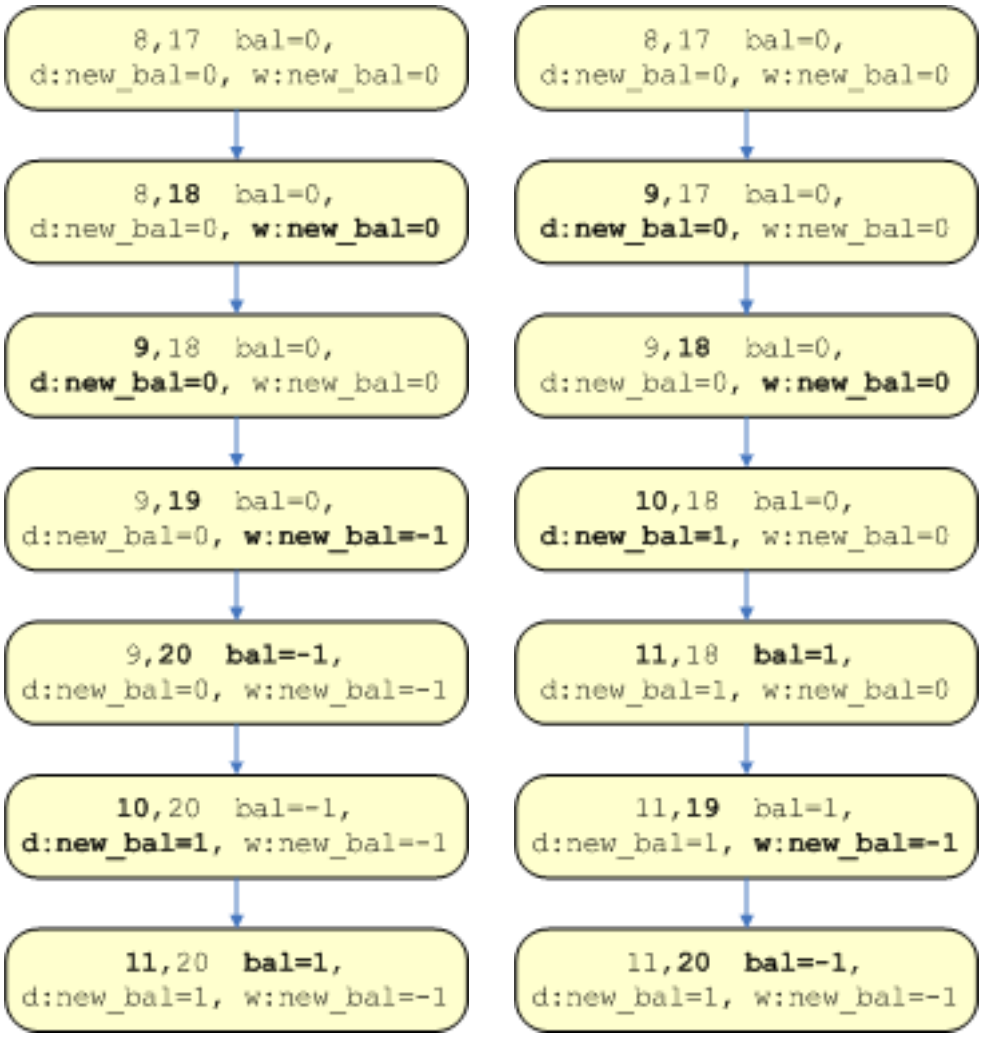


Figure 4.10: Two possible traces for the above program resulting in undesired non-zero results.

In particular, after one process has updated and printed the global variable, the other process can update it **based upon the original un-updated value**.

Solution to Exercise 4.4 (p. 20)

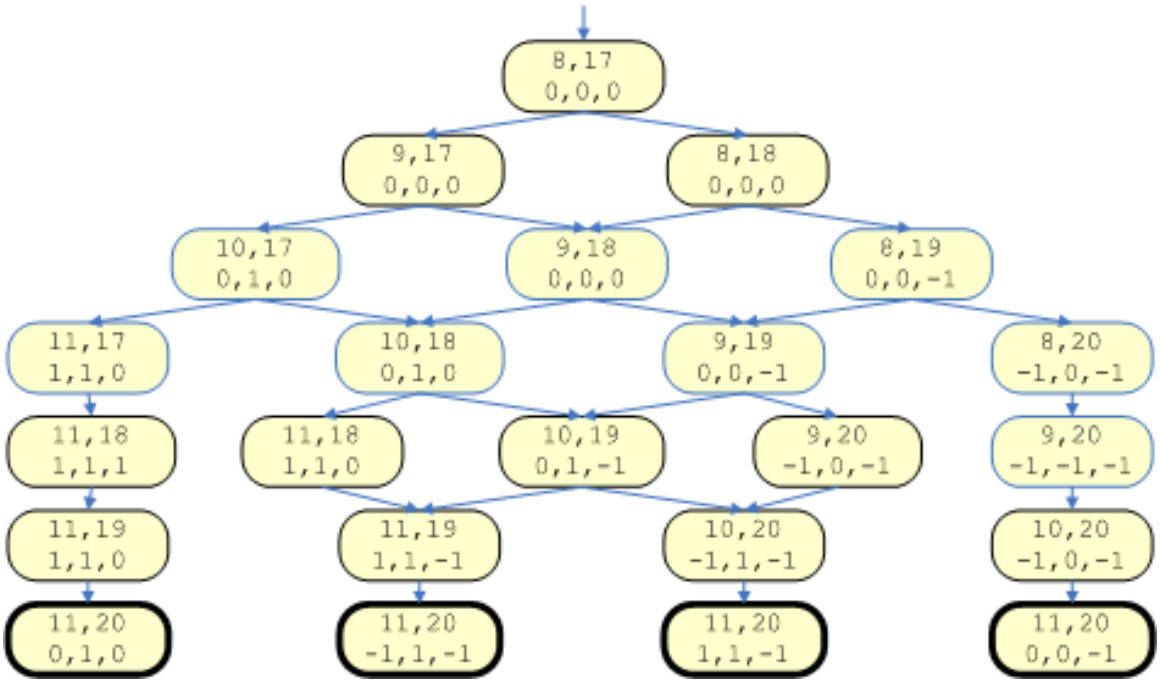


Figure 4.11: State space for above program. For brevity, the variable names are elided. This leaves the information in the form of the `deposit` and `withdraw` line numbers and the values of `bal`, `deposit`'s `new_bal`, and `withdraw`'s `new_bal`, respectively.

Whew! See how much larger this is than the state space of simpler version of this program (Example 4.2).

Solution to Exercise 4.5 (p. 22)

Divide the number of traces meeting this condition by the total number of conceivable traces. [16]

However, this assumes that each possible trace is equally likely — a dangerous assumption! In real life, with a particular operating system and hardware, some traces might be much more common than others. For example, it might be very unlikely (but possible) that two context switches would occur only a small number of instructions apart. So, a more realistic answer would be based upon a statistical analysis of repeated runs.

But an even more preferable answer is to say that this is a trick question! Rather than thinking of our concurrent system as being probabilistic, it is often better to stop short and merely call it **non-deterministic**. For example, if processes are distributed across the internet, then which trace occurs might depend on network traffic, which in turn might depend on such vagaries as the day's headlines, or how many internet routers are experimenting with a new algorithm, or whether somebody just stumbled over an ethernet cord somewhere. In practice, we can't come up with any convincing distribution on how jobs get scheduled. Moreover, sometimes we want to cover adversarial schedules (such as guarantees of how our web server protocol acts during denial-of-service attack).

Solution to Exercise 4.7 (p. 26)

No. For example, the first clause could be chosen until `dist` was negative. Unlike in the original code given, the successful termination clause could not be chosen until `dist` is incremented to zero again. Try it in SPIN, and look for such a trace.

[16] If you're truly interested, the code is simple and straightforward enough that it's not too hard to count these traces: $\frac{3!}{\frac{9!}{3!3!3!}} = \frac{6}{1680} = \frac{1}{280}$.

Solution to Exercise 4.9 (p. 26)

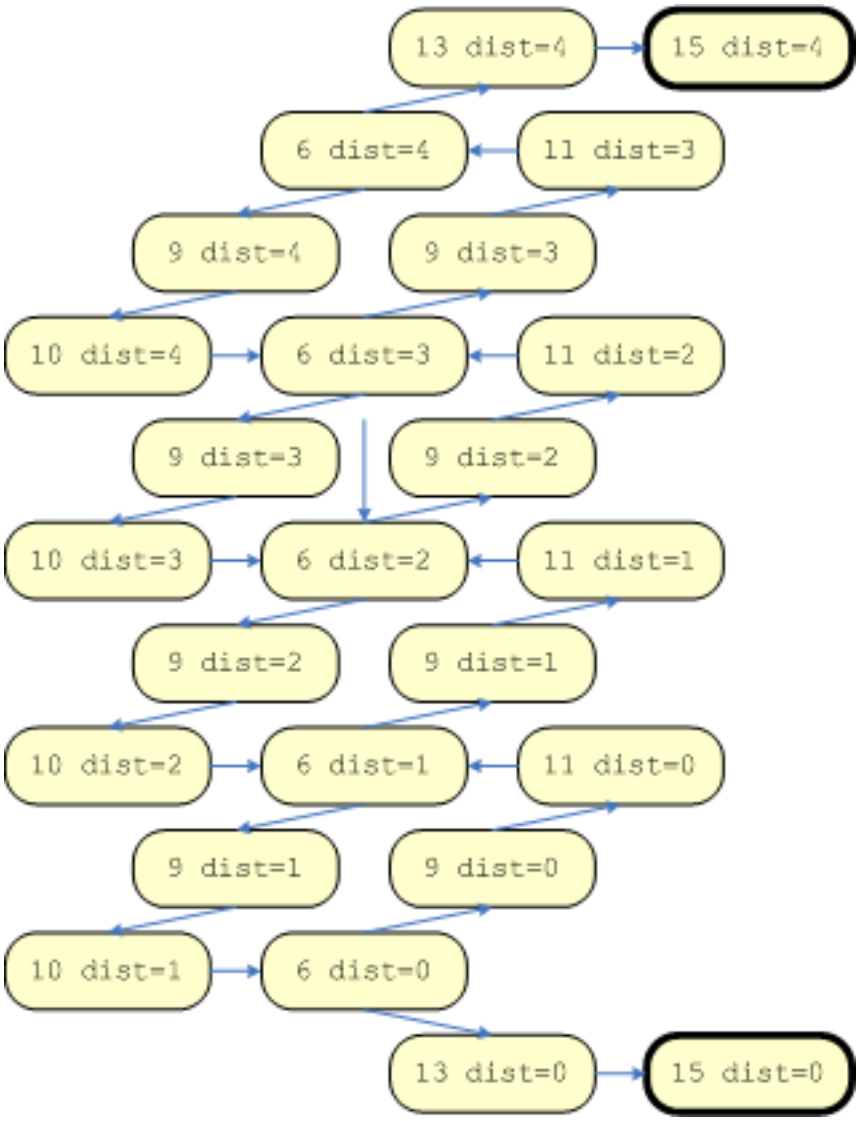


Figure 4.12: Random Walk 2 state space.

As desired, the program will always exit if `dist` reaches either of its limit values.

Solution to Exercise 4.10 (p. 26)

Yes. As mentioned, because of the `else` clause, the loop will terminate once the distance is out of range. Since the distance is only ever changed by 1 on an iteration, it is sufficient to check just the boundary conditions of the range.

While equivalent, it is probably a not good idea to make this change, since its correctness is dependent on other factors.

Solution to Exercise 4.11 (p. 27)

```
if
:: condition -> then_stmt;
:: else -> else_stmt;
fi;
```

Solution to Exercise 4.12 (p. 27)

```
do
:: condition -> body_stmt;
:: else -> break;
do;
```

Solution to Exercise 4.13 (p. 27)

```
show int count = 5;

active proctype count_down()
{
  do
  :: count > 0 -> count--;
  :: else -> break;
  od;
}

active proctype wait()
{
  count == 0 ->
  printf("I'm done waiting!\n");
}
```

Solution to Exercise 4.14 (p. 27)
There are many possible programs, but the following is a simple example.

```
show int x = 1;

active[2] proctype wait()
{
  x == 0 ->
  printf("How on earth did x become 0?!?\n");
}
```

Solution to Exercise 4.15 (p. 28)
As desired, only the value 3, since there are 3 processes.
Solution to Exercise 4.17 (p. 28)
The following figure shows one such trace. Notice that `z` is effectively incremented only once, instead of twice.

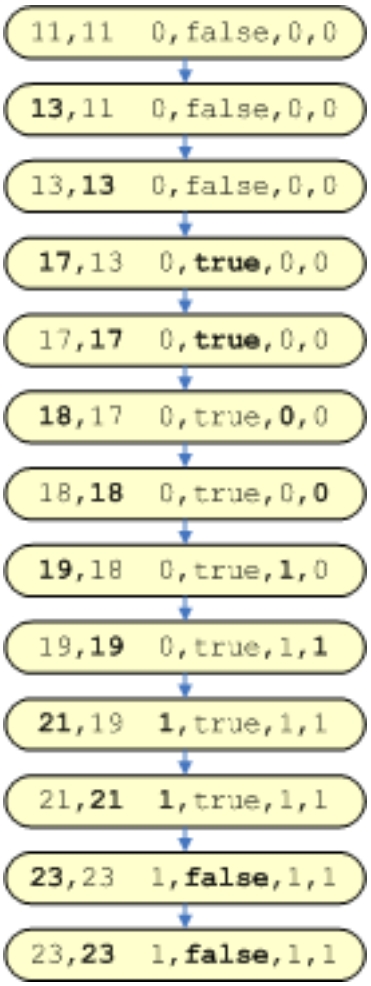


Figure 4.13: An undesired possible trace for `increment()`, if `atomic` keyword were removed. For brevity, states show the line number for each process, the `z` and `locked` values, and each process' `new_z` value, respectively.

The critical difference in state spaces is that the `atomic` keyword prohibits processes interleaving between the guard and its body. This reduces the size of the state space. In this case, as in many others, it also reduces the number of different end states and, thus, possible result values.

Solution to Exercise 4.22 (p. 42)

1. Call the two processes P1 and P2. P1 might get the first `ticket`=1 and enter its critical section. While there, P2 takes `ticket`=2, blocking until P1 is done. Then P2 enters its critical section, but before P2 finishes P1 again requests its critical section, taking `ticket`=3. This process repeats—each process requesting the critical section (and bumping up the `ticket` number) before its peer finishes.
2. No. If there is a moment when no processes are wanting the critical section, then the `ticket` number (intentionally) drops back to zero.
3. This is a trick question (as we've mentioned in a previous problem's solution (Exercise 4.5)): For example, road construction may fundamentally shift the rate of clients in the bakery, or scheduling might even become adaptively adversarial (perhaps: a grudge-holding customer trying to continually thwart another customer, by continually taking numbers just to ask for a glass of water).

4. SPIN runs out of state space before it detects the cycle. ¶ This justifies our initial use of `byte`: there is no conceptual difference in the verification errors between using `byte` and `int`, but by keeping the state space from being extraneously large, we get better insight to what the difficulty was. (Imagine if we'd used `int` first — we might have seen the state-space error, presumed that other parts of our program made verification infeasible, crossed our fingers and given up.)

Solution to Exercise 4.24 (p. 47)
Since we are currently considering only `B`'s loop to be progress, we only add a progress label in `B`.

```
 1   show int x = 0;
 2
 3   active proctype A()
 4   {
 5     do
 6     :: true -> x = 1 - x;
 7     od
 8   }
 9
10   active proctype B()
11   {
12    progress_B:
13     do
14     :: true -> x = 1 - x;
15     od
16   }
```

Now we want to verify this code First, run a syntax check. Partly, this is just another reminder of a good habit. But, it also forces `xspin` to re-scan your code and recognize the progress label. After doing this, re-run the verifier, and indeed it will find a non-progress cycle The following are two of the possible traces it could find, one minimal, and one not.

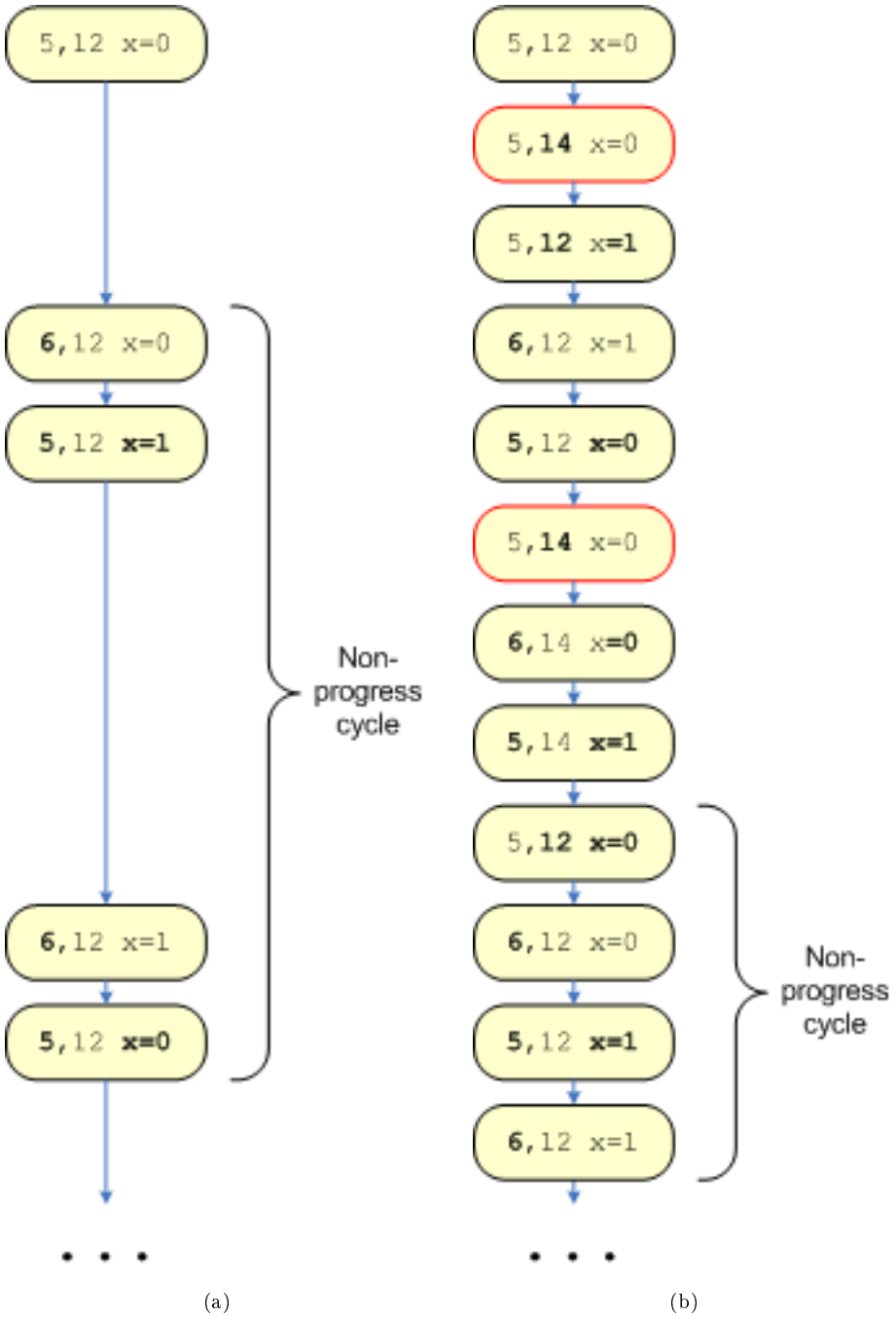


Figure 4.14: A minimal trace that only executes `A` and a nonminimal trace that executes `B` a little first. Progress states are depicted with red borders.

The next figure shows the entire state space, which has a bit of a surprise. Some of the states are duplicated, differing **only** in their status as progress states or not. To see why that is necessary, contrast the following. **Entering** line 14, as in the first transition of the above nonminimal trace, entails progress. This

corresponds to each of the red progress states. However, say that `B` execute for one step, long enough to pass the guard, but not yet execute the assignment. Now, if only `A` executes, `B` never makes further progress. This corresponds to the black non-progress states with `B`'s line counter at 14.

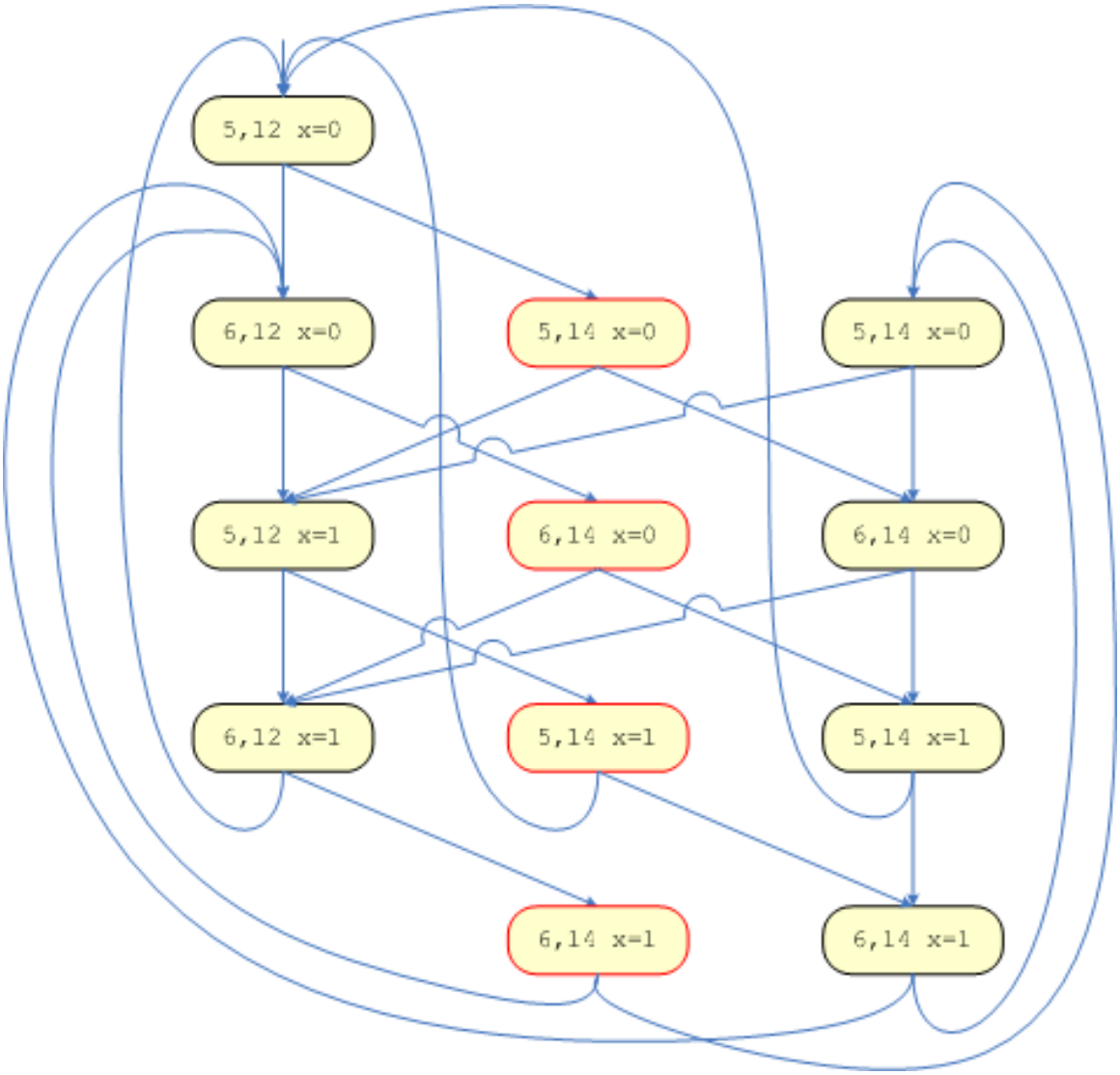


Figure 4.15: State space for above program.

Solution to Exercise 4.25 (p. 47)
Remove the progress label from `B`, and add one to the corresponding place within `A`.

Solution to Exercise 4.26 (p. 47)
Putting in two progress labels, and getting a green light from SPIN's verifier, would only mean "It is verified that in every trace, either `A` or `B` is making progress." But we want to confirm "... `A` **and** `B` are making progress."

In this toy example, where `A` and `B` are identical (up to differing in progress labels), we can argue by symmetry that lack of non-progress-for-`A` is enough to imply lack of non-progress-for-`B`. But in general, SPIN

makes us run two separate verifications (though we will see some possible code contortions later).

Solution to Exercise 4.27 (p. 48)

We want a progress label within the second option. Although that is not allowed before the guard, it is sufficient for this example to add it after the guard.

```
active proctype loop()
{
  do
  :: true -> x = 1;
  :: x==1 ->
     progress_2: x = 2;
  :: x==2 ->
     progress_3: x = 3;
  od;
}
```

The `progress_3` label is unnecessary since it would only be encountered after the second option is executed. However, it is arguably a good idea not to reason through such arguments, and instead let the tool formally do the reasoning.

After the first execution of the first option, the second one is continously enabled, since `x==1`. With weak fairness enforced, we are thus guaranteed that `x` eventually becomes 2.

Solution to Exercise 4.29 (p. 48)

Only the previous `progress_3` label should be used.

Chapter 5

Modeling Concurrent Processes: Homework Exercises[1]

The Dining Philosophers Problem is a standard example of concurrent programming. The idea is that a group of n philosophers sit at a single round table for dinner. There are n forks, one placed between each plate. To successfully eat each bite, a philosopher needs both of the adjacent forks. Thus, as one consequence, two adjacent philosophers cannot eat at the same time, since they cannot both have the fork inbetween them at the same time. The question is what strategies can the philosophers have such that, each philosopher eventually eats. Typically, but not necessarily, we also require that each philosopher has the same strategy.

Exercise 5.1 ***(Solution on p. 64.)***

To keep the problem small, let's assume we have three philosophers. As a slight simplication of some of the following problems, we'll ignore maxims about code reuse, and choose to use separate procedures for each philsopher's strategy.

1. Here is one attempted solution. Repeatedly, each philosopher tries to pick up the left fork, then tries to pick up the right fork, and then eats and drops the forks. A problem with this solution is that it has a race condition – two philosophers can have the same fork at the same time.
   a. Run this code in SPIN. Provide some output from a sample run.
   b. Add appropriate `assert` statements to the code to test for a race condition. If necessary, you may add other code to keep track of information to use in these assertions.
   c. Use SPIN to find a shortest trace illustrating a race condition.
2. Here is another attempted solution. It uses the same strategy, except that each fork has a lock. A problem with this solution is that it can deadlock.
   a. Use SPIN to show that the previous race conditions cannot happen.
   b. Use SPIN to find a shortest trace illustrating deadlock.
   c. Recode this version so it uses multiple copies of only a single `proctype`.
3. Informally, deadlock is often viewed in the more restrictive sense of deadlocking on the acquisition of locks. This is equivalent to considering the case where the only guard conditions are those testing boolean locks. In this sense, deadlock can only happen where there are at least two locks involved. The following problems are instances of two general strategies for avoiding this: using fewer locks or being careful with locks. In the following, code reuse will not matter – you may write code using either a single or multiple `proctype`s. ¶ One attempt is to only have a lock for only every other fork. The idea is that each philosopher only needs to grab one lock.

[1] This content is available online at <http://cnx.org/content/m12939/1.2/>.

   a. Modify the code to do this with an even number of philosophers and forks, say four of each.
   b. Does this code have the race condition or deadlock? Use SPIN to determine this. If no, show SPIN's successful output. If yes, find a shortest trace illustrating the problem.

4. A well-known solution is to number the forks somehow and ensure that locks are always obtained in numerical order (instead of always left-then-right). Note: Maintain the provided lock-based code's lock convention of first-acquired last-released.
   a. Modify the code to do this.
   b. Use SPIN to show that this does not have race conditions or deadlock.
   c. Use SPIN to determine whether using a first-acquired first-released protocol has race conditions or deadlock.
5. Assume that philosophers always try to pick up both forks, eat, and then drop both forks, all philsophers use the same strategy, but we have not chosen a particular strategy yet. If we assume weak fairness is enforced and deadlock is avoided, does each philosopher eat repeatedly? What if we assume strong fairness?

Exercise 5.2 ***(Solution on p. 64.)***

In the following, consider an arbitrary number of philosophers. Three philosophers is only a special case.

1. Assume we number the forks, and each philosopher picks up forks in numerical order. Because the table is round, this implies that if most philosophers pick up the left fork first, that one will instead pick up the right fork first. This is a well-known strategy for avoiding deadlock. Is that strategy weakly fair? Strongly fair?
2. Consider the strategy of acquiring forks in numerical order and releasing them in the opposite order. Compare its behavior in SPIN's simulation mode with SPIN's built-in weak fairness enforcement turned off vs. on.

Exercise 5.3 ***(Solution on p. 64.)***

The following is an example of the distinction between deadlock and livelock. Consider two people walking in a hallway towards each other. The hallway is wide enough for two people to pass. Of interest is what happens when the two people meet in the hall. When meeting on the same (left/right) side of the corridor, a polite strategy is to step to the other side of the hallway. A more belligerent strategy is to wait for the other person to move. With two polite people there is the possibility of livelock, while with two belligerent people there is the possibility of deadlock. (As an aside, note that one polite and one belligerent person together in a hall don't have any problems.)

1. Model the livelock problem in Promela. Use SPIN to demonstrate that your Promela program in fact models the problem.
2. Model the deadlock problem in Promela. Use SPIN to demonstrate that your Promela program in fact models the problem.

Exercise 5.4 ***(Solution on p. 65.)***

The process scheduler in an operating system is a typical example of where we are concerned with fairness. If we have multiple processes running on a single processor, we break time into finitely-long intervals during which we execute one of the processes. We'd like to ensure that each process gets a "fair" share of the CPU time.

Your model should have an "OS" process (the scheduler) and multiple "user" processes, using this framework. The user processes are non-terminating — they always want to get the CPU's next time slice. *I.e.*, the user processes will be continually enabled from the perspective of your own scheduler, although not from that of SPIN. But, we aren't modeling the user computation,

those processes just print a message. Your scheduler, which is also non-terminating, somehow picks among the processes which will execute.

1. a. Write a scheduler which does not exhibit weak fairness. *I.e.*, it is possible for one process not to make progress. However, do not write a trivial scheduler that simply always picks one process and never the other.
   b. Use SPIN to verify that your scheduler doesn't allow deadlock.
   c. Use SPIN to demonstrate that your scheduler does not exhibit weak fairness.
2. a. Write a scheduler which does exhibit weak fairness, *i.e.*, that each process is guaranteed to make progress. Do not use SPIN's built-in weak fairness enforcement, rather your scheduling algorithm must somehow enforce that condition.
   b. Consider if we added a progress label to the user `proctype`, inside its loop. Why is this insufficient for SPIN to verify the weak fairness?
   c. Describe how we could modify the program to verify the weak fairness. You do not need to implement this. Use only the features of Promela/Spin we've covered so far. (Soon, we'll introduce new features that allow a simpler and better approach.)

Solutions to Exercises in Chapter 5

Solution to Exercise 5.1 (p. 61)

1. You should add assertions to each philosopher, immediately before or after the `printf`. They assert that the two forks are being held by that philosopher.
2. The key to recoding is to use `_pid`, so that each process (philosopher) knows its number. Since process numbers start with 0, not 1, you can either use `_pid+1`, or renumber the philosophers 0...2. Then, replace the `type="inline">mtype` with a simple numerical encoding. To compute the appropriate `right_fork` value, use the modulo operator (`%`).
3. If any fork lacks a lock, race conditions are possible.
4. Using a FIFO acquisition and release protocol does not change correctness. With left-then-right acquisition, the problem can still deadlock, while with a numerical acquisition, it doesn't. After all, the deadlock is a result of the acquisition strategy, not the release strategy.
5. Weak fairness is conditional on processes being continuously enabled. But here, when one philosopher grabs a fork, the adjacent philosopher sharing that fork is no longer enabled. So with three philosophers, one philosopher eating repeatedly is weakly fair. Similarly, with four philosophers, two of them sitting opposite who are eating repeatedly is weakly fair. ¶ Strong fairness is conditional on processes being enabled infinitely often. It also does not imply that each philosopher gets to eat. For example, consider four philosophers 1...4 and four forks A...D. After starting with 2 grabs A, 2 grabs B, they can cycle through the following actions: 4 grabs C, 2 drops B, 4 grabs D, 2 drops A, 4 drops D, 2 grabs A, 4 drops C, 2 grabs B. The other two philosophers are never enabled, and thus it is strongly fair for them not to eat.

Solution to Exercise 5.2 (p. 62)

1. That particular strategy is neither weakly nor strongly fair, in general. (It is weakly fair when there are only 2 or 3 philosophers.) For example, it allows for one philosopher to repeatedly grabs and drop forks, while the others do nothing.
2.

Solution to Exercise 5.3 (p. 62)
A key to modeling the problem is to abstract away the unnecessary details. In particular, there is no need to model the people walking down the hall, as nothing interesting happens then. Similarly, the hallway can be modeled as being only wide enough for two people to pass. By these restrictions, we reduce the data to only four possibilities: each of the two people can be on either of two sides of the hall. Keeping the state space small allows SPIN to solve more problems, solve them faster, and report traces that are more succinct and, thus, easier to understand.

The following is one of the many ways to code this.

```
/* Code assumes these two values add to two. */
#define NUM_BELLIGERENT 2
#define NUM_POLITE      0

/* Indicates position of two people.
 * Values: 0 or 1, for two sides of hall.
 *         2 means unitilized.
 */
int position[2] = 2;

active [NUM_BELLIGERENT] proctype belligerent()
{
  /* Non-deterministically set up starting position. */
```

```
  if
  :: true -> position[_pid] = 0;
  :: true -> position[_pid] = 1;
  fi;

  /* Wait for other person to initialize. */
  position[1 - _pid] != 2;

  do
  :: position[1 - _pid] == position[_pid] ->
     /* Wait for other person to move. */
     skip;
  :: else ->
     /* Success. */
     break;
  od;
}

active [NUM_POLITE] proctype polite()
{
  /* Non-deterministically set up starting position. */
  if
  :: true -> position[_pid] = 0;
  :: true -> position[_pid] = 1;
  fi;

  /* Wait for other person to initialize. */
  position[1 - _pid] != 2;

  do
  :: position[1 - _pid] == position[_pid] ->
     /* Move to other side. */
     position[_pid] = 1 - position[_pid];
  :: else ->
     /* Success. */
     break;
  od;
}
```

Solution to Exercise 5.4 (p. 62)

1. The simplest such scheduler is one that non-deterministically chooses which process is scheduled next.
2. The simplest such scheduler just alternates between the two processes. ¶ Adding a progress label to, say, only process A allows SPIN to verify that A makes progress. But that says nothing about B making progress. Adding a progress label to both A and B allows SPIN to verify that **at least one** of A and B makes progress, both not necessarily both. ¶ With what we've seen, here's two possible ways to use SPIN to convince ourselves that the code is weakly fair.
   - Add a progress label to only process A. SPIN verifies that A must gets turns. Then add a progress label to only process B. SPIN verifies that B must get turns. Thus, both A and B must get turns. ¶ Unfortunately, this requires two separate verifications. Also, editing the code inbetween the verifications is error-prone, as it is easy to forget to delete the first progress label.

- Add code, probably to the scheduler, that keeps track of the scheduling choices. Add logic and assertions to check if this behavior is correct. ¶ It is easy to check, for example, that the processes are indeed strictly alternated. Or, more flexibly, that after 100 scheduling choices, both processes were scheduled at least once. But, it is tricky to capture the idea that each will eventually get scheduled. ¶ Adding substantial code for verification is always undesirable because we must separately reason that this additional code is correct and doesn't interfere with the original behavior.

Chapter 6

Using Temporal Logic to Specify Properties[1]

So far, we've seen some of the built-in checks performed by SPIN: assertions, deadlock, non-progress, and one notion of fairness. But what about other properties we'd like to check for? We might be interested in properties which the implementors of our particular tool weren't interested in, or hadn't thought of. Fortunately, SPIN does include a general property-checking mechanism: If we can state our property in a particular formal **temporal logic**, then SPIN will allow us to check the property.

First, we'll introduce the syntax and semantics of one particular temporal logic, **LTL**. Then (Section 6.6: Using Temporal Logic in SPIN), we'll return we'll return to SPIN and how it uses temporal logic for verification.

First, we reflect that some properties — safety properties — can be phrased in terms of "some particular state is unreachable". For example, a state where two processes are sending information to the printer simultaneously never occurs. Failed assertions can be viewed as a transition to a fail-state, and we require that this state is unreachable. Deadlock can be viewed as a state with no legal outgoing transition, and not all processes sitting in an end-state.

Similarly we reflect that other properties — liveness properties — are about entire traces, not just some state in a trace. For instance, non-progress cycles is a cycle which does not contain any states labeled as progress states.

We want a richer language to have a way of talking about traces, but that is precise and not subject to English's vagaries. We'll work towards using logic — a temporal logic specifically designed to be conducive to expressing concepts which arise in concurrent programming. But before we can even do that, we need a clear model of concurrent programs, and more formal definitions of some ideas we've already seen.

6.1 Concurrent Programs as State Space Automata

The notions we've seen of states (Definition: "State", p. 17), traces (Definition: "Trace", p. 17), and state spaces (Definition: "State space", p. 17) aren't specific to Promela, but are useful in any formalization of concurrent systems. In fact, we are ready to give one precise definition of a "concurrent system" in terms states and traces. It is similar to many conventional automata definitions[2] , but includes a way to relate the current state to propositions[3], which we can later construct formulas out of.

Definition 6.1: State Space Automaton
A State Space Automaton A is a tuple $< S, s_0, T, \text{Prop}, P >$ where:

[1] This content is available online at <http://cnx.org/content/m12317/1.13/>.
[2] http://en.wikipedia.org/wiki/Automata_theory
[3] "Propositional Logic: propositions" <http://cnx.org/content/m10715/latest/>

- S is a set of states. As before (Definition: "State", p. 17), a **state** is an assignment to all program variables (both local an global), including the program counter (line number) for each process.
- $s_0 \in S$ is the **initial state** or **start state**.
- $T \subseteq S \times S$ is the **transition relation**. Equivalently, this is the edge relation of the state space (Definition: "State space", p. 17).
- Prop, a set of propositions.
- $P : S \to 2^{\text{Prop}}$ is a mapping which, given a state, indicates all the properties that are true in that state.

It is the set Prop which will allow us to talk about our program variables in our logic formulas. The elements of Prop are simple propositions involving program variables. For example,

- philosopher_2_has_fork
- (ctr > 3)
- (in[].size() == max_buffer_size) (that is, the buffer `in[]` is full)

Each of these might be true in some states and false in other states; the function P gives a truth assignment to the propositions (a different truth assignment for each state).

Definition 6.2: Trace

Given a state space automaton, a **trace** σ (sometimes called an ω-**trace**) is a (possibly infinite) sequence of states σ_0, σ_1, σ_2, ... which respects the automaton's transtion relation T.

Thus a trace is a path through the state space (as we have already seen (Definition: "State space", p. 17)). Now that we have logic propositions associated with each state, we can see that during a trace, those propositions will change their truth value over time. We'll illustrate this with diagrams like the following.

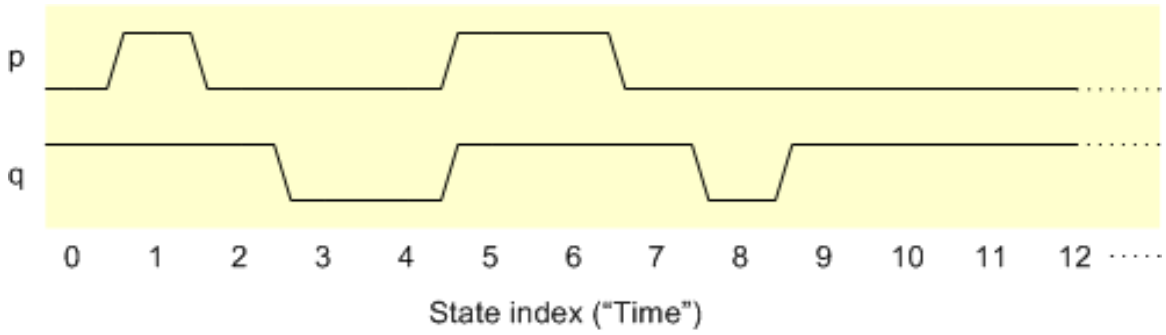


Figure 6.1: A timing diagram for a trace σ, and propositions p and q. Each propositional variable's value during the trace is depicted as a line. When the line is high, the proposition is true; when low, false.

In this example, p and q are propositions from Prop. Since in the trace σ, we have p being true at (for example) index 2, we write $\sigma_2 \vDash p$. In this diagram, it is also the case that $\sigma_0 \vDash (p \vee q)$. [4]

[4] Technically, these diagrams are a bit too suggestive: a trace is only a sequence of discrete states, with no concept of time existing in between states. And the numbering of the x-axis is only an index to the sequence of states — it's not meant to imply that the system's states necessarily correspond to periodic samplings of our program. Yes, it would be more accurate to replace these continuous timelines with a textual list of true and false (to represent the automaton's mapping P). However, the diagrams better communicate information to us humans, especially those familiar with such timelines as used by hardware circuit designers. We just need to be careful not to read in between the times.

Although a bit unintuitive at first, it will be convenient to convert all finite traces into infinite ones. To do the conversion, we simply envision that once the automaton reaches its ostensibly last (nth) state, it languishes furiously in that state over and over:

Definition 6.3: Stutter-Extend
The finite trace $\sigma_0,\ldots,\sigma_n$ can be stutter-extended to the infinite trace $\sigma_0,\ldots,\sigma_n,\sigma_n,\sigma_n,\ldots$.

To ensure this still satisfies the definition of a trace (where successive states must obey the automaton's transition relation), it is often assumed that every state has a transition to itself. Note that this convention precludes assuming that something must change between one state and the next, which is plausible since we are modeling asynchronous systems. This will be reflected in our formal logic below, which will have no built-in primitive for "the-next-state".

6.2 State Formulas

Now that we have a formal model of what a trace is, we can start to make formal statements about what happens in a trace. The simplest statements are **state formulas**, *i.e.*, propositional formulas built out of Prop. For a given trace σ and state formula ψ, we can immediately decide whether a particular state σ_i satisifes ψ (notationally, "$\sigma_i \vDash \psi$").

NOTE: Deciding whether some particular state of a trace satisfies a formula also depends implicitly on the automaton Σ. Technically we should write $\Sigma,\sigma_i \vDash \psi$, though in practice Σ is clear from context.

Example 6.1
For example, given a trace σ, we might ask about what is happening[5] at σ_{957}, and whether $\sigma_{957} \vDash \neg$Proc1_is_printing, or whether $\sigma_{957} \vDash$ ((ctr $>$ 3) $\rightarrow$ philosopher_2_has_fork). The answer would be found by taking the automaton's function P, and seeing what values the truth assignment $P(\sigma_{957})$ assigns to our formula's individual propositions such as (ctr $>$ 3).

Example 6.2
Consider the trace σ we saw previously (Figure 6.1) and is repeated here.

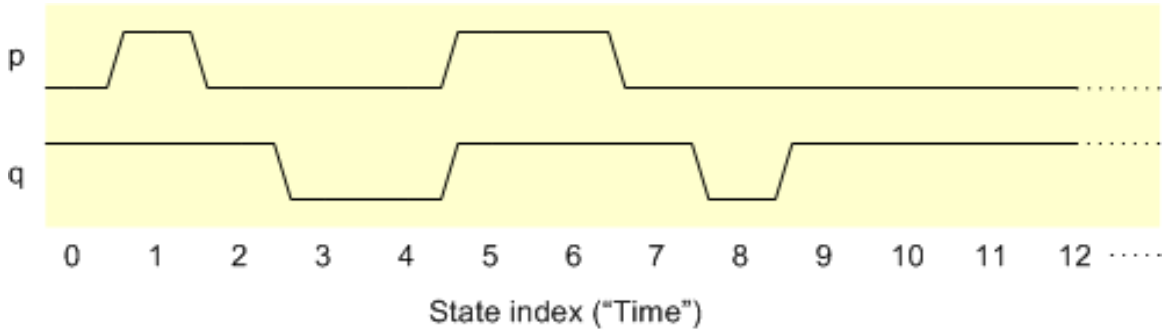


Figure 6.2: A timing diagram for a trace σ, and propositions p and q.

For which i does $\sigma_i \vDash (p \wedge q)$? For which i does $\sigma_i \vDash (p \vee q)$? How about $\sigma_i \vDash (p \rightarrow q)$?

However it doesn't make sense *a priori* to ask whether an **entire trace** satisfies some particular state formula; unlike regular propositional logic, the truth of state formulas changes over time, as the trace σ progresses.

[5] http://babylon5.cybersite.com.au/lurk/synops/006.html

6.3 Introducing Temporal Connectives

So, how **do** we talk about formulas holding (or not) over time? In addition to making formulas out of $\wedge$,$\vee$,$\neg$ and propositions from the set Prop, we'll allow the use of **temporal connectives**.

- $\Box$ — **always**, or **henceforth**: We say that $\Box\phi$ is true at a moment i, iff ϕ is true from moment i onwards.
- [U+25C7] — **eventually**: We say that [U+25C7]ϕ is true at moment i, iff ϕ will eventually be true at moment i or later.
- U — **strong until**: We say that $(\phi U \psi)$ is true at moment i, iff ψ eventually becomes true, and until then ϕ is true.
- W — **weak until**: Like Strong Until, but without the requirement that ψ eventually becomes true.

As a mnemonic for the symbols "$\Box$" and "[U+25C7]", we can imagine a square block of wood sitting on a table. Orienting it like $\Box$, it will always sit there; orienting it like [U+25C7], it will eventually teeter.

More formally, we define these connectives as follows.

Definition 6.4: Always
$\sigma_i \vDash \Box\phi$ iff $\forall j \geq i . \sigma_j \vDash \phi$
Example

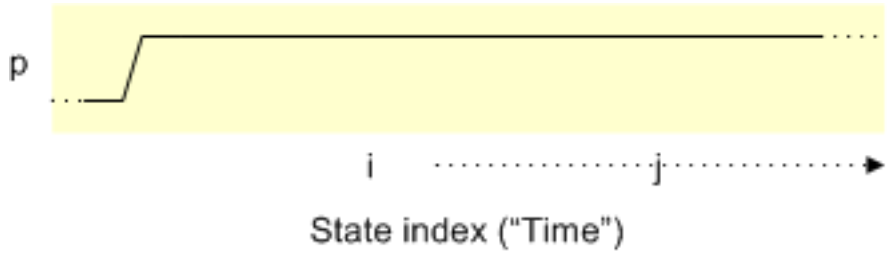


Figure 6.3: A generic trace σ satisfying $\sigma_i \vDash \Box p$. p is true in state i and all later states.

Exercise 6.1 ***(Solution on p. 83.)***
Decide whether each is true or false for following trace σ (the same one seen earlier (Figure 6.1)).

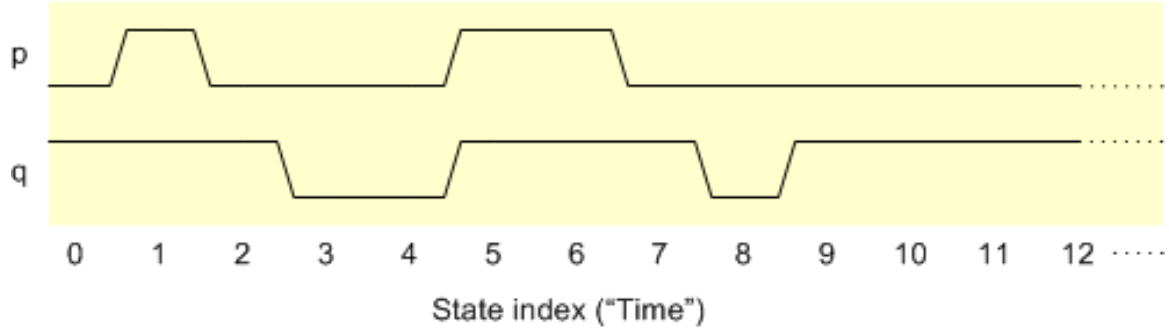


Figure 6.4: A timing diagram for a trace σ, and propositions p and q

1. $\sigma_9 \vDash \Box q$
2. $\sigma_9 \vDash \Box \neg p$
3. $\sigma_0 \vDash \Box q$
4. $\sigma_0 \vDash \Box (p \rightarrow q)$

Definition 6.5: Eventually
$\sigma_i \vDash$ [U+25C7]ϕ iff $\exists k \geq i . \sigma_k \vDash \phi$
Example

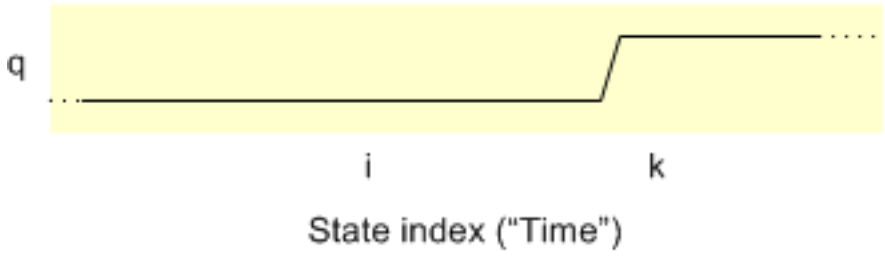


Figure 6.5: A generic trace σ satisfying $\sigma_i \vDash$ [U+25C7]q. q is true in some state k after state i.

Exercise 6.2 ***(Solution on p. 83.)***
Decide whether each is true or false for the trace σ seen earlier (Figure 6.1).

1. $\sigma_9 \vDash$ [U+25C7]p
2. $\sigma_9 \vDash$ [U+25C7]$\neg p$
3. $\sigma_0 \vDash$ [U+25C7]q
4. $\sigma_0 \vDash$ [U+25C7]$\neg q$
5. $\sigma_9 \vDash$ [U+25C7] $(q \rightarrow p)$
6. $\sigma_0 \vDash$ [U+25C7]$\neg (p \rightarrow q)$

Before continuing on with our two other connectives (strong- and weak-until), let's pull back a bit and look at our notation.

NOTE: The nit-picky — er, the **careful** reader will have noticed that although we write $\sigma_i \vDash \ldots$, really, the truth value of the formula depends not on just a single state, but rather that state and the entire suffix of the trace after that. So writing $< \sigma, i > \vDash \ldots$ would be more accurate. Moreover, we are using not just the trace but also the timing diagrams, which are just the graph of an automaton A's mapping function ("P"). Thus the correct formulation really needs to be written $< A, \sigma, i > \vDash \ldots$ (and it is required that σ be a legal trace with respect to the transitions of A). However, now that we realize our abuse of notation, we'll revert and just write $\sigma_i \vDash \ldots$.

In practice, we are often considering an entire trace, and wondering whether some property holds always or eventually. It's extremely natural to extend our notation from particular indices to the trace itself:

NOTE: **Trace σ models formula ϕ** ("$\sigma \vDash \phi$") iff $\sigma_0 \vDash \phi$.

For example, from the above exercises, since $\sigma_0 \vDash (p \rightarrow q)$, we say simply $\sigma \vDash (p \rightarrow q)$.

Definition 6.6: Strong Until
$\sigma_i \vDash (\phi U \psi)$ iff $\exists k \geq i . (\sigma_k \vDash \psi \wedge \forall j . (i \leq j < k \rightarrow \sigma_j \vDash \phi))$.

Example

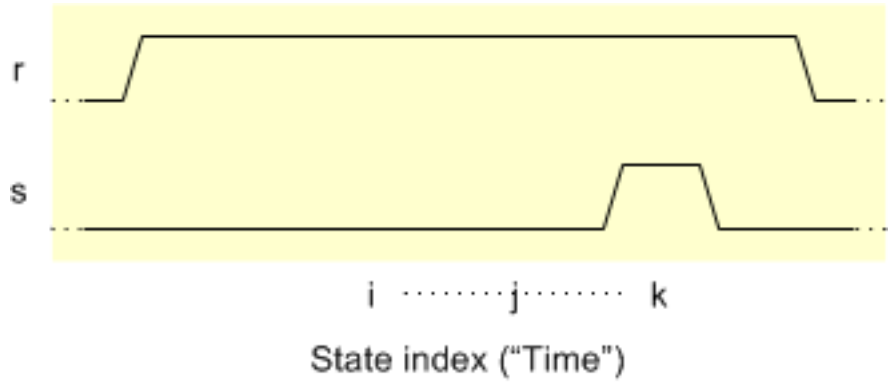


Figure 6.6: A generic trace σ satisfying $\sigma_i \vDash (rUs)$. r is true in states i through $(k-1)$.

Exercise 6.3 ***(Solution on p. 83.)***

Decide whether each is true or false for following trace σ: (This is the same trace seen earlier (Figure 6.1).)

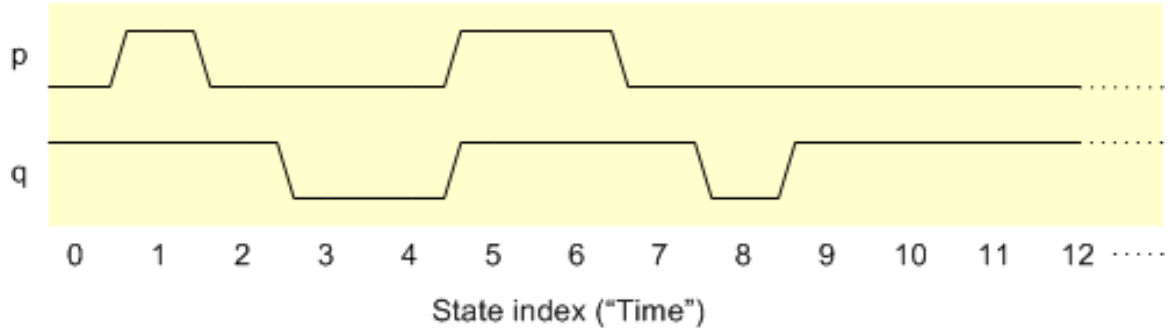


Figure 6.7: A timing diagram for a trace σ, and propositions p and q.

1. $\sigma_0 \vDash (qUp)$
2. $\sigma_2 \vDash (qUp)$
3. $\sigma_5 \vDash (qUp)$
4. $\sigma_9 \vDash (qUp)$

Exercise 6.4 ***(Solution on p. 83.)***

Give the formal semantics of Weak Until (in the same way we have given semantics for $\Box$, [U+25C7], and Strong Until).

Exercise 6.5 ***(Solution on p. 83.)***

Weak Until could be omitted from our core set of LTL connectives, since it can be defined in terms of the remaining connectives. Provide such a definition. That is, give an LTL formula equivalent to $(\phi W \psi)$.

Any logic that includes temporal connectives is a **temporal logic**. More specifically, this is **Linear Temporal Logic** (**LTL**). The "linear" part of this name means that we are looking only at individual traces, rather than making formulas that discuss all possible traces of a program. After all, a single program can result in many possible traces. An LTL formula does not allow us to say, for example, that a property holds in all possible executions of the program. However, SPIN overcomes some limitations by essentially testing an LTL formula against all possible traces. Will will return to SPIN in the next section (Section 6.6: Using Temporal Logic in SPIN).

In summary, the syntax used in SPIN is given by the following grammar.

SPIN's Grammar for LTL formulas

ltl	::=		atom	
		|	(ltl binop ltl)	
		|	unop ltl	
		|	(ltl)	
unop	::=		[]	"Always"
		|	<>	"Eventually"
		|	`!`	"Not"
binop	::=		`U`	"Strong Until"
		|	`W`	"Weak Until"
		|	`&&`	"And"
		|	`||`	"Or"
atom	::=		`true`	
		|	`false`	
		|	name	`#define`'d identifiers

Table 6.1

ASIDE: Some presentations of LTL use a box or "G" (Globally) for Always, and use a diamond or "F" for Eventually. Confusingly, some use "U" for Weak Until. Also, some include additional connectives like "X" (Next), which cannot be defined in terms of those given.

6.4 Do we really need a whole new logic?

We **could** use regular ol' first-order logic as our formal language for specifying temporal properties. We'd do this by adding an explicit time index to each proposition in Prop, turning it into a unary relation. For example, instead of considering Philosopher_B_has_fork in state 17, we'd instead write Philosopher_B_has_fork(17), and so on.

This approach, though, quickly becomes cumbersome. Consider the everyday concept "Someday, X will happen". We have to go through some contortions to shoehorn this concept into first-order logic: "There exists a day d, such that d is today or later, and X will happen on d". Quite a mouthful for what is about the simplest temporal concept possible!

The awkwardness stems from the fact that in English, "eventually", "always", and "until" are primitives. When we design a formal language to capture with these concepts, shouldn't we too include them as primitives? Having our formal representations reflect our natural conceptions of problems is the first law[6] of programming!

This is why we have decided to incur the overhead of defining a new language, LTL. Still, you have noticed that the semantics we've given are phrased in terms of "$\forall i \ldots$". From a language designer's perspective, we might say that LTL is a high-level language, and we give an interpreter for it written in first-order logic (over the domain of trace-indices).

6.5 Translating between English and LTL: Examples

Let's look at how to turn a natural-English specification into temporal logic, and vice versa.

Example 6.3
How might we express the common client/server relationship "if there is a request (r), then it will be serviced (s)"?

Hmm, that seems obvious: $(r \rightarrow s)$ should do it, no? Alas, no. The problem is, the truth of this formula — which has no temporal connectives at all — only depends upon the trace's initial state, σ_0 — in this case, whether $\sigma_0 \vDash (\neg r \vee s)$. This certainly doesn't match our intent, since we want the request and service to be able to happen at points in the future. Let's try again...

Problem
Does the formula $\Box\,(r \rightarrow s)$ capture the English notion "if r is true, then s will be true"?

Solution
That's only a bit better: it does now allow the implication $(r \rightarrow s)$ to hold in any state σ_i. But our intent is to allow the servicing s to happen **after** the request r. The formula as given only expresses "Whenever r is true, then s is also true at that moment." If r and $\neg s$ both hold at the same time (which wouldn't be surprising), then $\Box\,(r \rightarrow s)$ fails for that trace.

After more thought, we can get something which **does** arguably match our intent: $\Box\,(r \rightarrow$ `[U+25C7]`$s)$. "Whenever a request is made, a servicing will eventually happen."

However, even this may not be what everybody wants. Some considerations:

- Even if r is raised (and lowered) several times, a single moment of s can satisfy the entire trace. This matches an intuition of r indicating a single ring of a telephone (and s being 'pick up the phone'), but may not match the intuition of r being "leave a voicemail" (and s being "respond to a voicemail").

[6] http://www.owlnet.rice.edu/~comp210/01fall/Lectures/lect01.shtml

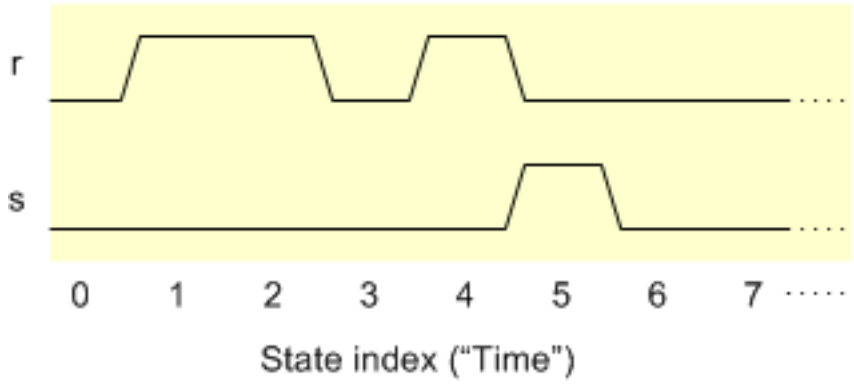


Figure 6.8: A trace showing multiple requests followed by a single servicing.

- A trace in which s is always true (regardless of r) certainly satisfies ϕ. Did you expect the English statement to encompass this?
- Any trace in which requests are serviced **instantaneously** will satisfy ϕ. While this might be intended, it might also be a bug of leftover servicings from previous requests.

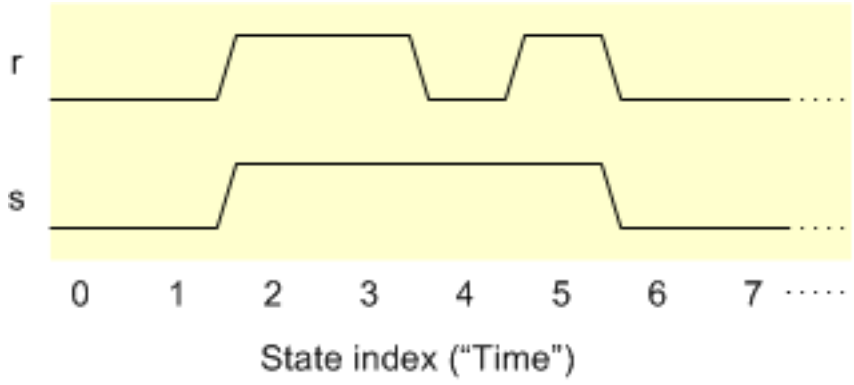


Figure 6.9: A trace showing multiple requests and a single servicing. The servicing arguably corresponds to only the first request and might be an undesired behavior.

- A trace in which a request is never made still satisfies this formula. While that is probably what is intended, it may not have occurred to you from the English statement alone.

One advantage to having a formal language is that because the semantics are precisely defined, it helps us think about corner cases such as the above (and, gives an unambiguous meaning to the result, good for writing specs).

Example 6.4
As a more complicated variation, we generally want to match requests and their servicing. For example, what if there are two requests, r_1 and r_2, arriving in that order? In a particular context, do we expect that there are also exactly two servicings, s_1 and s_2? If so, do we also expect them in that order?

Problem
As a partial solution, how can we express "whenever r_1, s_1 will happen before s_2"?

Solution

If we are also requiring that s_2 indeed actually happen, then we can use the following: $\square((r_1 \wedge \texttt{[U+25C7]}\, s_2) \rightarrow (\neg s_2 U s_1))$.

Note that since LTL does not have quantifiers (like $\forall$ and $\exists$) we cannot use variable indexing in our formulas. In other words, we can't talk about r_i and s_j, but instead, must always refer to specific variables like r_1 and s_2.

From these examples and the ones following, we can see that English is typically too ambiguous. To get the appropriate logic formula, we need to go back to the original problem and determine what was truly meant. Formalizing your specifications into temporal logic gives you the opportunity to closely examine what those specifications really should be.

Given a formula, try to understand it by translating it into English. Providing sample timelines that satisfy the formula is also quite helpful.

Exercise 6.6 *(Solution on p. 83.)*

Describe the meaning of $\square\,\texttt{[U+25C7]}\, p$ in words and a timeline.

Exercise 6.7 *(Solution on p. 84.)*

Describe the meaning of $\square(q \rightarrow \square\neg p)$ in words and a timeline.

Example 6.5

A railroad wants to make a new controller for single-track railroad crossings. Naturally, they don't want any accidents with cars at the crossing, so they want to verify their controller. Their propositions Prop include train_is_approaching, train_is_crossing, light_is_flashing, and gate_is_down.

Problem 1

Brainstorming: Using natural English, what are some properties we'd like to have true? Feel free to use words like "always", "eventually", "while", and "until". You may add new propositions to Prop, if you think it's appropriate.

For each, feel free to demonstrate that the one property is not sufficient: give a trace which satisfies the property but is either unacceptable or unrealistic. (For instance, "the gate is always down" is safe but unacceptable; "a train is always crossing" is unrealistic since there aren't infinite trains.)

Solution

We can think of lots of examples, such as these.

- Whenever a train passing, the gate is down.
- If a train is approaching or passing, then the light is flashing.
- If the gate is up and the light is not flashing, then no train is passing or approaching.
- If a train is approaching, the gate will be down before the train passes.
- If a train has finished passing, then later the gate will be up.
- The gate will be up infinitely many times.
- If a train is approaching, then it will be passing, and later it will be done passing with no train approaching. (Thus, trains aren't infinitely long, and there are gaps between the trains.)

To formalize such statements, we would start with the primitive propositions involved. These could be

- a ("a train is approaching the crossing")
- p ("a train is passing the crossing")
- l ("the light is flashing")
- g ("the gate is down")

This choice forces us to not refer to individual trains and, thus we must simplify some of our properties, *e.g.*, "If a train is approaching, the gate will be down before the next train passes." (Think about the consequences of not making this change.)

Some of these English descriptions are ambiguous, however. *E.g.*, can a single train be approaching and passing simultaneously? When writing formal specifications, we'll be forced to think about what we mean to say, and provide an unambiguous answer, one way or the other. Continuing this example, we'll say that once a train is passing, it is no longer considered to be approaching.

Problem 2

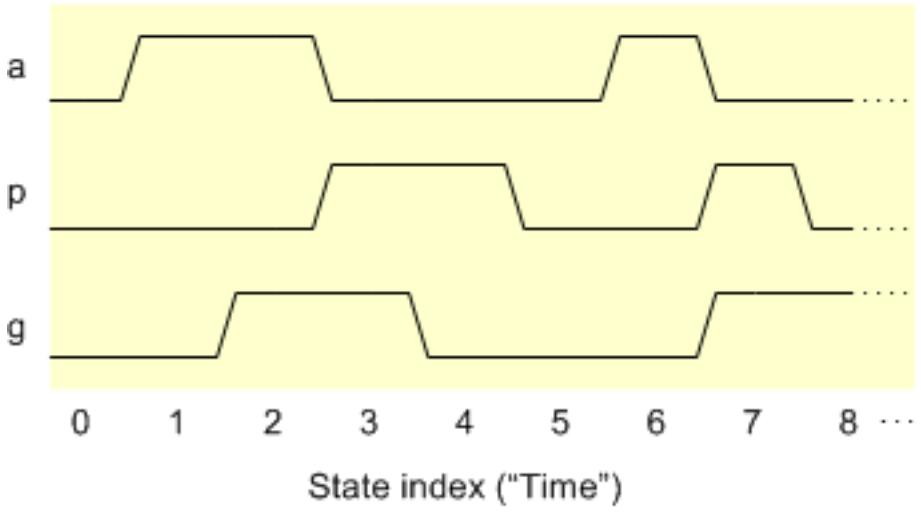
Encode some of the properties from the previous exercise (Example 6.5, Problem 1) in temporal logic. It is often helpful to first turn the statement into a timeline (essentially a trace).

1. "Whenever a train passing, the gate is down."
2. "If a train is approaching or passing, then the light is flashing."
3. "If the gate is up and the light is not flashing, then no train is passing or approaching."
4. "If a train is approaching, the gate will be down before the next train passes."
5. "If a train has finished passing, then later the gate will be up."
6. "The gate will be up infinitely many times."
7. "If a train is approaching, then it will be passing, and later it will be done passing with no train approaching."

Solution

Keep in mind that for any concept, there are many different (but equivalent) formulas for expressing it.

1. $\Box(p \rightarrow g)$
2. $\Box((a \vee p) \rightarrow l)$
3. $\Box((\neg g \wedge \neg l) \rightarrow (\neg p \vee \neg a))$

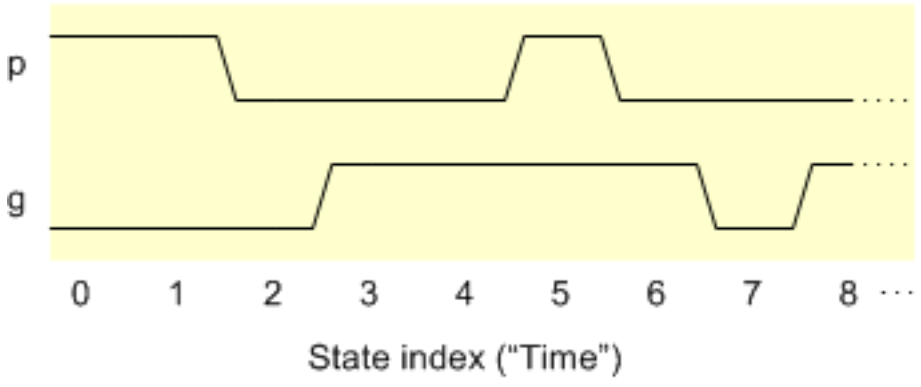


4.

Figure 6.10: A trace satisfying this property.

Observe in the trace shown that in one instance, the gate goes down at the same time the train passes. In the LTL version we're using, we can't express the idea "... **strictly** before the next train passes." Instead, we'll allow this simultaneity as a possibility. $\Box(a \rightarrow (\neg p W g))$ *I.e.*, if a train is approaching, then a train is not passing until the gate is down. However,

this still allows that the gate could be back up by the time a train passes, and a train might never pass.

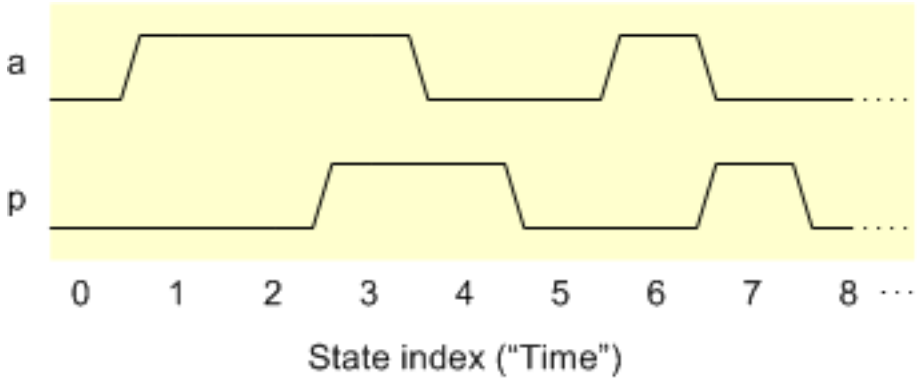


5.

Figure 6.11: A trace satisfying this property.

$\square((pU\neg p) \rightarrow$ [U+25C7] $(\neg p \rightarrow$ [U+25C7] $g))$ Since our logic cannot talk about the past, we have to shift our perspective to start while the train is still passing. We've effectively reworded the statement to say that if a train is passing and will finish passing, then sometime after it is no longer passing, the gate will be up. However, the status of the gate at other times is not commented on. We're only saying there is one moment after the train passes that the gate is up.

6. $\square$ [U+25C7] $\neg g$



7.

Figure 6.12: A trace satisfying this property.

$\square(a \rightarrow$ [U+25C7] $(p \wedge$ [U+25C7] $(\neg p \wedge \neg a)))$

This last condition is not something that a controller would enforce. Instead, we would expect the train scheduling software to enforce this. The controller would be allowed to assume this, and we'd want to verify a property ψ only for traces that also satisfy our scheduling assumptions ϕ. Of course, we can also do this by incorporating assumptions into the formulas: $(\phi \rightarrow \psi)$.

Exercise 6.8 *(Solution on p. 84.)*

We've previously seen that SPIN knows some built-in concepts such as deadlock (Section 4.5.2: Deadlock and End States), livelock (Section 4.5.6.1: Livelock), weak- and strong-fairness (Section 4.5.6.3: Fairness). We'll see in the next section (Section 6.6: Using Temporal Logic in SPIN) that SPIN can also be used to verify **any** LTL formula. But first, let's characterize some of those built-in concepts as LTL formulas using the context of a print server.

We will use the following propositions:

- b : This is a bad state. (It corresponds to Promela `assert(!`b`)`).
- e : This is a valid end state. (It corresponds to all Promela processes being at an `end` label).
- p_1, p_2: Progress is being made (either of two types of progress); they correspond to a Promela process being at a certain `progress` label.[7]
- r: There is a pending request.
- s: A request is being serviced.

Give a temporal formula expressing each of the following:

1. No error occurs, *i.e.*, we never reach a bad state.
2. No deadlock occurs. (We'll take deadlock here to mean that all five of our variables stop changing; without having a different proposition for each state, this is close enough.)
3. Progress is always made. That is, this trace doesn't go forever without making progress.
4. Weak Fairness: if a request is made and held, it will get serviced. *I.e.*, it's not the case that a request stays true forever but is never serviced.
5. Strong Fairness: if a request is made infinitely often, then it will be serviced.

Exercise 6.9 *(Solution on p. 85.)*

For our simple producer/consumer model, what is the basic property we'd like to guarantee? Roughly, "each product is consumed exactly once". That is, Produce and Consume are called alternatingly, starting with Produce. (In particular, as stated this precludes modeling "produce twice and queue the results".)

Expressing this actually requires several clauses. Give colloquial English translations of the following LTL formulas. We can make the generic model a bit concrete by interpreting startProduce and finishProduce as starting and finishing the production of a fine, hand-crafted ale, with startConsume, finishConsume corresponding to enjoying such an ale. (The more health-concious reader might prefer to instead interpret these as start/finish the training for a marathon, and start/finish running a marathon, respectively.)

1. $\Box$(finishProduce $\rightarrow$ `[U+25C7]`startConsume)
2. $\Box$(finishProduce $\rightarrow$ ($\neg$startProduce$U$finishConsume))
3. ($\neg$startConsume$W$finishProduce)
4. $\Box$(finishConsume $\rightarrow$ ($\neg$startConsume$W$finishProduce))

Exercise 6.10 *(Solution on p. 85.)*

In the previous exercise (Exercise 6.9), the formulas entailed that production of an item could not even begin until the previous object had been **entirely** consumed. This requirement might make sense for the interpretation of marathon training/running, but it doesn't seem essential to the production of hand-crafted ale, even if we do want to preserve the policy of never queuing up products.

Suppose we have a computer network port whose buffer can store only one web page's data at a time. The producer could correspond to some server sending a web page to that computer, and the consumer might be a browser which processes the data being received.

[7] Promela does includes a handy syntax for testing whether a particular process is at a particular label: "`philosopher[3]@eating`" is true when the fourth (0-indexed) `philosopher` process is at the label `eating`.

1. Do we want to allow the producer to be producing the next product before the consumer finishes the current one?
2. What change would you make to the formulas if you **did** want to allow this produce-while-consuming behavior (as in the ale interpretation)?

Note that we want to include the possibility that nothing is ever produced — we don't **always** want to enforce weak fairness. Consider an operating system's code for a print server — it is truly important that not owning a printer doesn't mean the OS always crashes!

Through this section, we've seen some of the common patterns that can arise in the formulas. Following upon the idea of Design Patterns[8] as describing common programming idioms, "Specification Patterns" describe common logic specification idioms. For a list of LTL specification patterns, see Property Pattern Mappings for LTL[9] .

In addition, there are also algebraic equivalences for LTL, just as there are for propositional logic[10]. We have already seen a DeMorgan-like example; there are also less obvious equivalences, such as distributing $\Box$`[U+25C7]` over $\vee$: $\Box$`[U+25C7]` $(\phi \vee \psi) \equiv (\Box$`[U+25C7]`$\phi \vee \Box$`[U+25C7]`$\psi)$ and the redundancy of $\Box$-over-$\vee$: $(\Box\phi \vee \Box\psi) \equiv \Box(\Box\phi \vee \Box\psi)$ It can be fun to dabble in rewriting LTL formulas, but (alas) we won't visit that topic further here.

6.6 Using Temporal Logic in SPIN

So, we have a Promela program and an LTL formula to verify. How do we use the tool? In SPIN, in the "Run" menu, select "LTL Property Manager". This opens a new window. The top half contains several subparts for entering an LTL formula, while the bottom half is for using it.

In the "Formula" textbox, you can type an LTL formula. If you prefer, you can enter the operators using the buttons below, rather than typing. Also, you can use the "Load" button to restore a previously-saved LTL formula from a file. For a formula, the next checkboxes allow you to either verify that the formula always holds or never holds. The "Notes" textbox is simply descriptive text. If you plan on saving the formula, it is helpful to remind yourself what this formula means and is used for. The "Symbol Definitions" textbox is where the LTL variables (typically `p`, `q`, *etc.*) are related to the Promela code. This is done through `#define`s which are temporarily added to your Promela code.

Example 6.6

Let's consider another critical section example. The following code simply puts the critical section into a loop. Also included is our standard code to check that the mutual exclusion property is satisfied, although the assertion is commented out.

```
 1  /* Number of copies of the process to run. */
 2  #define NUM_PROCS 3
 3
 4  show bool locked = false;
 5  show int  num_crit = 0;
 6
 7  active[NUM_PROCS] proctype loop()
 8  {
 9    do
10    :: true ->
11
12       atomic {
```

[8] http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
[9] http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml
[10] "Propositional Logic: equivalences" <http://cnx.org/content/m10717/latest/>

```
13          !locked ->
14          locked = true;
15        }
16
17        /* Critical section of code. */
18        /* Something interesting would go here. */
19
20        num_crit++;
21        /* assert(num_crit == 1); */
22        num_crit--;
23
24        locked = false;
25        /* End critical section of code. */
26     od;
27   }
```

Using temporal logic, we can accomplish the same check without the assertion. There are several similar ways of doing this.

One way is to verify that `num_crit` always has the value 0 or 1. To do this, we'll make the following definition in xspin's "Symbol Definitions" textbox:

```
#define p ((num_crit == 0) || (num_crit == 1))
```

We then want to verify that p always holds, so our formula is `[]p` and mark that we always want this formula to hold.

To verify with this formula is a two-step process. First, using the "Generate" button generates what is known as a **never claim**, shown in the window. This is more Promela code that is temporarily added to your program. This creates a special process that checks the LTL formula and is guaranteed to be executed after every normal step of your code. You don't need to understand the generated never claim, although SPIN presents it for advanced users. The never claim uses a couple syntactic features of Promela not covered in this module. Next, using the "Run Verification" button opens a small version of the verification options window that we're already familiar with. Running the verification puts any output in the LTL window.

NOTE: To verify temporal formulas from the command-line, you need to do three things: First, generate a `never` claim using `spin -f formula`. For example,

```
prompt>  spin -f '[] (p ->  <> (q || !r))'
```

Be sure to quote the formula so the shell doesn't try to interpret special characters. Next, take the output (which is Promela code for a `never` claim), and paste it at the end of your Promela source file. Also add `#defines` to the top of your file, defining each propositional variable used in your LTL formula. Finally, verify the resulting code.

Example 6.7

Generate and verify the code from the previous example (Example 6.6).

The automatically-generated `never` claim is shown below. It uses labels and `gotos` which we haven't discussed, but which will be familiar to many programmers. (This code represents a small

automaton, in this case with two states, which gets incorporated into the larger state space automaton that we've described.) The use of the constant `1` here is equivalent to the more mnemonic `true`.

```
never {    /* !([]p) */
T0_init:
        if
        :: (! ((p))) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all
        skip
}
```

Running the verification, it does not report any errors. However, note that verifying an LTL property automatically turns off deadlock checking:

```
Full statespace search for:
        ...
        invalid end states      - (disabled by never claim)
```

However, it still reports a warning that the process never exits, in case we care:

```
unreached in proctype loop
        line 27, "pan.___", state 11, "-end-"
        (1 of 11 states)
```

Exercise 6.11 *(Solution on p. 85.)*

We want to accomplish the same goal as the previous example (Example 6.6), but with a formula that should **never** hold. What's an appropriate formula?

Exercise 6.12 *(Solution on p. 85.)*

Now let's verify something that can't be done easily with just assertions. We also want to make sure that `num_crit` continually alternates between the values of 0 and 1, rather than maintaining the same value. How can we do this?

Exercise 6.13 *(Solution on p. 85.)*

How could we use LTL to verify whether no process is starved? Of course, this verification should fail, generating a trail where at least one process is in fact starved.

Solutions to Exercises in Chapter 6

Solution to Exercise 6.1 (p. 70)

1. True; from σ_9 onwards, q stays true.
2. True. Even stronger, $\sigma_7 \vDash \Box\neg p$.
3. False, since σ_3 [U+22AD] q.
4. True, since at each individual index i (from 0 on up), $\sigma_i \vDash (p \rightarrow q)$.

Solution to Exercise 6.2 (p. 71)

1. False; from σ_9 onwards, p stays false.
2. True. This is a much weaker statement than $\sigma_9 \vDash \Box\neg p$, which we already saw was true.
3. Extremely true, since q is already true in σ_0 (and elsewhere, too).
4. True, since $\neg q$ is true in σ_3 (and elsewhere too).
5. False.
6. False. We already mentioned that $\sigma_0 \vDash \Box(p \rightarrow q)$; if you think about it, this means that $\sigma_0 \vDash \neg$ [U+25C7] $\neg(p \rightarrow q)$. (While we won't spend time discussing algebraic equivalences[11] for LTL formulas, this is certainly reminiscent of DeMorgan's Law[12].)

Solution to Exercise 6.3 (p. 72)

1. True. q is true in σ_0, and it doesn't take long before p becomes true in σ_1.
2. False. Starting at σ_2, we see that p doesn't become true until σ_5, yet q doesn't stay true that long.
3. Very true, since p is **already** true in σ_5. (In the definition, take k=5; then there are not any j to even need to worry about.)
4. False. Since p is false from σ_9 onward, we can't even find a $(k \geq 9)$ such that $\sigma_k \vDash p$. However, it is true that $\sigma_9 \vDash (qWp)$.

Solution to Exercise 6.4 (p. 72)
$\sigma_i \vDash (\phi W \psi)$ iff $(\forall j \geq i . \sigma_j \vDash \phi \vee (\exists k \geq i . \sigma_k \vDash \psi \wedge \forall i \leq j < k . \sigma_j \vDash \phi))$

Solution to Exercise 6.5 (p. 73)
Here are three equivalent definitions. The first follows directly from the above exercise (Exercise 6.4).

- $(\phi W \psi) \equiv (\Box\phi \vee (\phi U \psi))$
- $(\phi W \psi) \equiv ($ [U+25C7] $\neg\phi \rightarrow (\phi U \psi))$
- $(\phi W \psi) \equiv (\phi U (\psi \vee \Box\phi))$

Solution to Exercise 6.6 (p. 76)
Literally, we can simply say "always, eventually p". While that phrase is fairly common among logicians, it's not very natural or meaningful to most people. A clearer, but more long-winded, equivalent is "at any point in time, p will happen in the future".

More concise, though, is "p happens infinitely often". To understand that, consider a timeline. In words, at any time t_0, p will happen sometime in the future; call that time t_1. And at moment t_1+1, p will happen sometime in the future; call that t_2. Repeat forever.

[11] "Propositional Logic: equivalences" <http://cnx.org/content/m10717/latest/>
[12] "Reference: propositional equivalences" <http://cnx.org/content/m10540/latest/>

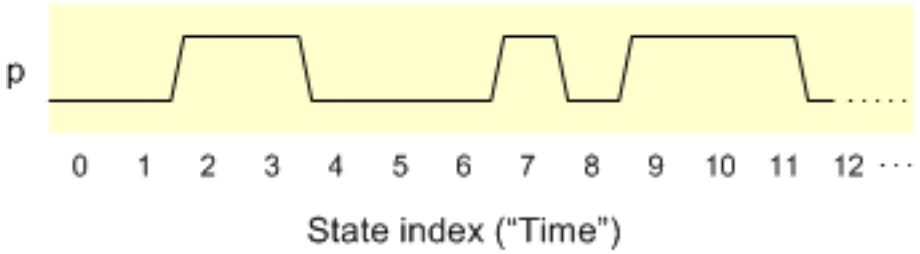


Figure 6.13: The beginning of a trace where p happens infinitely often.

Solution to Exercise 6.7 (p. 76)
Again, a literal translation is "always, if q then always not p". Correct, but a bit more idiomatic-English would be "whenever q, then forever not p" or "Once q, p is forever false".

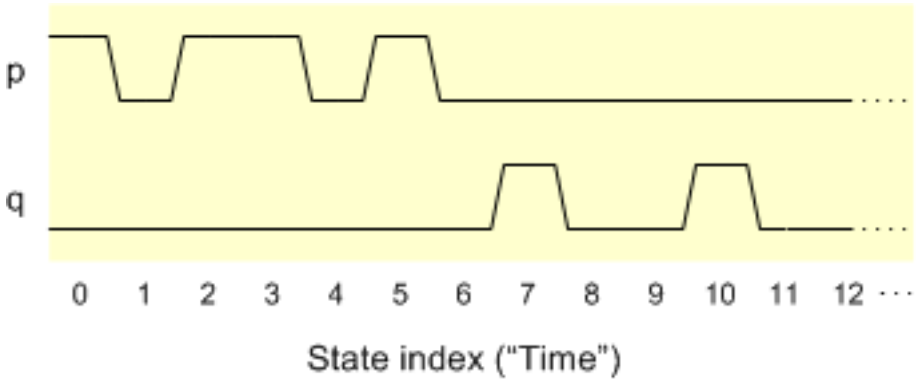


Figure 6.14: The beginning of a trace satisfying the property.

Solution to Exercise 6.8 (p. 79)
For each of these problems, there are many equivalent formulas, some of which are easier to verify as equivalent than others.

1. $\Box\neg b$
2. $\neg$[U+25C7] $((\Box b \vee \Box\neg b) \wedge (\Box p \vee \Box\neg p) \wedge (\Box r \vee \Box\neg r) \wedge (\Box s \vee \Box\neg s) \wedge \Box\neg e)$. Remember that if we're in an end-state, then it doesn't count as deadlock.
3. $\Box$[U+25C7] $(p_1 \vee p_2) \equiv (\Box$[U+25C7]$p_1 \vee \Box$[U+25C7]$p_2)$. In fact, SPIN provides a built-in variable `np_`, which corresponds to "all processes are in a non-progress state". When SPIN does its built-in check for non-progress cycles, it actually just verifying `!<>[]np_`. You might recall (Section 4.5.6.4: Some Unexpected Progress (optional)) that SPIN's idea of progress is state-based, not transition-based: SPIN (counterintuitively) considers it progress when a process does nothing but squat in a `progress` state. (This can only happen if weak fairness is not being enforced, or a guard is being labeled as progress.) How might we capture "progress" but still preclude those traces which just sit idly at a progress label? We could add a second progress label immediately following p_1, call it q_1. Similarly, we can introduce q_2. Then we could verify that $((\Box$[U+25C7]$p_1 \wedge \Box$[U+25C7]$q_1) \vee (\Box$[U+25C7]$p_1 \wedge \Box$[U+25C7]$q_1))$. (This assumes that all labels are private to each process.)

4. "It's always not the case that r is always on, and yet s doesn't happen an infinite number of times": $\neg\Box(\Box r \wedge \neg\Box\texttt{[U+25C7]}s)$. One equivalent formula is "if r stays on forever, then it's being serviced infinitely often.": $\Box(\Box r \rightarrow \Box\texttt{[U+25C7]}s)$.
5. $(\Box\texttt{[U+25C7]}r \rightarrow \Box\texttt{[U+25C7]}s)$

Solution to Exercise 6.9 (p. 79)

1. If an ale is produced, then an ale will be consumed (possibly immediately). Note that we don't talk about **which** ale is consumed; in the following clauses we will try to enforce that it really is the same ale that was recently produced.
2. If an ale is produced, then no ale-production is started until after an ale has been consumed.
3. No ale is consumed before (an initial) ale is produced. Note that a trace in which no ale is ever produced still satisfies this formula.
4. If an ale is consumed, nothing else will be consumed until an(other) ale has been produced.

Solution to Exercise 6.10 (p. 79)

1. No. That would mean the buffer is being overwritten with a new web-page's data, even though the browser is still trying to process the current page in the buffer.
2. We would replace $\Box(\text{finishProduce} \rightarrow (\neg\text{startProduce}\,U\,\text{finishConsume}))$ with $\Box(\text{finishProduce} \rightarrow (\neg\text{finishProduce}\,U\,\text{finishConsume}))$.

Solution to Exercise 6.11 (p. 82)
Using the same definition of `p`, we can simply use `<>!p`. *I.e.*, we don't want it to ever happen that `p` is false.
Solution to Exercise 6.12 (p. 82)
Use the following symbol definitions:

```
#define p (num_crit == 0)
#define q (num_crit == 1)
```

and verify that

```
([]<>p) && ([]<>q)
```

always holds.
Solution to Exercise 6.13 (p. 82)
In essence, we want to verify

```
([]<>(_pid == 0)) && ([]<>(_pid == 1)) && ([]<>(_pid == 2))
```

The problem is that `_pid` is a local variable to each process, but, unfortunately, `never` claims are their own separate process, and can't access those variables. One solution is to create an equivalent global variable:

```
pid running;
```

Inside the critical section code add the line

```
running = _pid;
```

Use the following symbol definitions:

```
#define p (running == 0)
#define q (running == 1)
#define r (running == 2)
```

and verify whether

```
([]<>p) && ([]<>q) && ([]<>r)
```

always holds.

> ASIDE: The never claim's process will numbered higher than any started by an `active proctype`. So, in this case, its PID is 3.

As expected, the verification fails, and a trail file is created. In the author's sample verification, this trail has only processes 0 and 1 entering the critical section, although there are many possible counter-examples.

It turns out, SPIN has a special syntax to express whether a certain process is at a certain label. So instead of defining a new variable, we could alternately go to the critical section and add a new label (say "`crit:`") and then define:

```
#define p (loop[0]@crit)   /* 0th 'loop' process is at crit. */
#define q (loop[1]@crit)   /* 1st 'loop' process is at crit. */
#define r (loop[2]@crit)   /* 2nd 'loop' process is at crit. */
```

Chapter 7

Using Temporal Logic to Specify Properties: Homework Exercises[1]

Exercise 7.1 *(Solution on p. 91.)*
Give an English translation of the following LTL formulae. Try to give a natural wording for each, not just a transliteration of the logical operators.

1. ([U+25C7]$r \rightarrow (pUr)$)
2. $\Box(q \rightarrow \Box\neg p)$

Exercise 7.2 *(Solution on p. 91.)*
In the following, give an LTL formula that formalizes the given English wording. If the English is subject to any ambiguity, as it frequently is, describe how you are disambiguating it, and why.

1. "p is true."
2. "p becomes true before r."
3. - "p will happen at most once."
   - "p will happen at most twice."
4. "The light always blinks." Use the following proposition: p = the light is on.
5. "The lights of a traffic signal always light in the following sequence: green, yellow, red, and back to green, *etc.*, with exactly one light on at any time." ¶ Use the following propositions: g = the green light is on, y = the yellow light is on, and r = the red light is on.

Exercise 7.3 *(Solution on p. 91.)*
Recall the Dining Philosophers Problem from the previous homework (Exercise 5.1). Using temporal logic, formally specify the following desired properties of solutions to the D.P. Problem. Use the following logic variables, where $0 \leq i < N$:

- l_i: Philosopher i has his/her left fork.
- r_i: Philosopher i has his/her left fork.

For each question, your answer should cover exactly the given condition – nothing more or less. You may assume $N = 3$.

1. No fork is ever claimed to be held by two philosophers simultaneously.
2. Philosopher i gets to eat (at least once).
3. Each philosopher gets to eat infinitely often.

[1] This content is available online at <http://cnx.org/content/m12940/1.3/>.

4. The philosophers don't deadlock. (The main difficulty is to conceptualize and restate "deadlock" within this specific model in terms of the available logic variables.) ¶ You may assume philosophers pick up two forks in some order, eat, and drop both forks.
5. The philosophers don't deadlock. (The main difficulty is to conceptualize and restate "deadlock" within this specific model in terms of the available logic variables.) ¶ You may not assume philosophers pick up two forks in some order, eat, and drop both forks. For example, one might pick up a single fork and then drop it. Or, the philosophers might be lazy and never pick up a fork.
6. Describe a D.P. Problem run in which philosophers don't deadlock, but it is not the case that each philosopher gets to eat infinitely often.

Exercise 7.4 ***(Solution on p. 91.)***

They following algorithm is an attempt to implement mutual exclusion for two processes. Here, each process is willing to defer to the other. (It also introduces Promela's `goto`, which lets you branch to a label; this is an alternate way of implementing loops and other control flow.)

Verify whether the algorithm is correct or not.

- Verify using SPIN's built-in checks, without using temporal logic.
- Verify using temporal logic.
- If it is incorrect, find a shortest possible trace when it fails.

```
1  int x, y, z;
2
3  active[2] proctype user()
4  {
5    int me = _pid+1;                              /* me is 1 or 2. */
6
7   L1:
8    x = me;
9    if
10   :: (y != 0 && y != me) -> goto L1   /* Try again. */
11   :: (y == 0 || y == me)                        /* Continue...  */
12   fi;
13
14   z = me;
15   if
16   :: (x != me) -> goto L1                       /* Try again. */
17   :: (x == me)                                  /* Continue... */
18   fi;
19
20   y = me;
21   if
22   :: (z != me) -> goto L1                       /* Try again. */
23   :: (z == me)                                  /* Continue... */
24   fi;
25
26   /* Entering critical section. */
27   /* ... */
28
29   /* Leaving critical section. */
30 }
```

Exercise 7.5 ***(Solution on p. 93.)***

The following is pseudocode for an attempt to implement critical sections for `n` processes. It is based on the idea that processes take numbers, and the one with the lowest number proceeds next. This algorithm has one small flaw. Your task is to find and fix the flaw. In particular, show the following steps of this process.

1. Implement this algorithm in Promela, and show the resulting code. Include any code added for verification purposes, although that is counts towards the next problem's score.
2. Show the use(s) of SPIN to verify that there is a flaw and to determine what the flaw is. Briefly describe the flaw.
3. Fix the flaw in the code, and show either the fix or the resulting code. Again, include any code added for the next problem's verification.
4. Show the use(s) of SPIN to verify your final implementation.

Hints: First, do not radically change the algorithm. There is a straightforward solution that only changes/adds a line or two. Second, do not overly serialize the code. Since the entry protocol's `for` loop is already serialized, this really means that each each process should be able to calculate `max` concurrently.

- Shared variable declarations.

```
/* Is Pi choosing a number? */
boolean       choosing[n];

/* Pi's number, or 0 if Pi has none. */
unsigned int  number[n];
```

- Global initialization. Occurs once, before any process attempts to enter its critical section.

```
/* No process has a number. */
for all j ∈ {0,...,n-1}
   number[j] = 0;
```

- Critical section entry protocol, for process `i`. *I.e.*, each process has the following code immediately before its critical section. ¶

```
/* Choose Pi's number. */
choosing[i] = true;
number[i]   = max(number[0],number[1],...,number[n-1]) + 1;
choosing[i] = false;

/* For all other processes, ... */
for all j ∈ {0,...,i-1, i+1,...,n-1} in some serial order
    /* Wait if the other process is currently choosing. */
    while (choosing[j]) /* nothing */ ;

    /* Wait if the other process has a number and comes ahead of us. */
    if ((number[j] > 0) &&
        (number[j] < number[i]))
       while (number[j] > 0) /* nothing */ ;
```

Note that, because of the `if`'s test, it is equivalent to allow `j` to get the value `i` in this loop, as well. Although less intuitive, that simplifies the loop.

- Critical section exit protocol, for process `i`. *I.e.*, each process has the following code immediately after its critical section.

```
/* Clear Pi's number. */
number[i] = 0;
```

Solutions to Exercises in Chapter 7

Solution to Exercise 7.1 (p. 87)

1. "p is true before r."
2. "p is false after q."

Solution to Exercise 7.2 (p. 87)

1. This looks so simple and obvious, right? Unfortunately, it is ambiguous. The simple answer, p, says it's true **right now**. But, the likelier intended meaning is that it's always true, $\Box p$.
2. This can be reworded as "p becomes true while r is still false." $(\neg r W (p \wedge \neg r))$
3. The version of LTL we use cannot capture the notion of something being true for exactly one state. Instead, we must instead think in terms of something being true for "a while". Using that idea, we'll reword the original English into more explicit, long-winded forms. ¶ "p will happen at most once" becomes "p is false for a while, then it may become true for a while, then it may become false forever." It LTL, that can be written as $(\neg p W (p W \Box \neg p))$. ¶ Repeating that pattern, "p will happen at most twice" becomes $(\neg p W (p W (\neg p W (p W \Box \neg p))))$.
4. Here are three progressively simpler solutions which are equivalent.
   - $\Box ((p \rightarrow \texttt{[U+25C7]} \neg p) \wedge (\neg p \rightarrow \texttt{[U+25C7]} p))$
   - $(\Box (p U \neg p) \wedge \Box (\neg p U p))$
   - $(\Box \texttt{[U+25C7]} p \wedge \Box \texttt{[U+25C7]} \neg p)$
5. There are many ways to write this, but here's one. It states that whenever the green light is on, no other light is on, and it will stay on until the yellow one is on. Note that this implies the red light won't come on before the yellow one. What happens when the other lights are on is entirely parallel. Finally, at least one light is on. ¶ $\Box ((g \rightarrow ((\neg y \wedge \neg r) \wedge (g U y))) \wedge (y \rightarrow ((\neg r \wedge \neg g) \wedge (y U r))) \wedge (r \rightarrow ((\neg g \wedge \neg y) \wedge (r U y))) \wedge (g \vee y \vee r))$

Solution to Exercise 7.3 (p. 87)

1. $\Box (\neg (l_0 \wedge r_2) \wedge \neg (l_1 \wedge r_0) \wedge \neg (l_2 \wedge r_1))$
2. $\texttt{[U+25C7]} (l_i \wedge r_i)$
3. $(\Box \texttt{[U+25C7]} (l_0 \wedge r_0) \wedge \Box \texttt{[U+25C7]} (l_1 \wedge r_1) \wedge \Box \texttt{[U+25C7]} (l_2 \wedge r_2))$
4. Here are two solutions.
   - $\Box (\neg (l_0 \wedge l_1 \wedge l_2) \wedge \neg (r_0 \wedge r_1 \wedge r_2))$
   - $\Box \texttt{[U+25C7]} ((l_0 \wedge r_0) \vee (l_1 \wedge r_1) \vee (l_2 \wedge r_2))$
5. This simply says that it never gets stuck in one particular fork configuration. There would be many `if` statements, one per configuration, and this is abbreviated. ¶ $\Box (((l_0 \wedge l_1 \wedge l_2) \rightarrow \texttt{[U+25C7]} \neg (l_0 \wedge l_1 \wedge l_2)) \wedge ((\neg l_0 \wedge l_1 \wedge l_2 \wedge \neg r_2) \rightarrow \texttt{[U+25C7]} \neg (\neg l_0 \wedge l_1 \wedge l_2 \wedge \neg r_2)) \wedge \ldots)$
6. There are many possibilities. One is where philosopher 0 repeatedly eats, grabbing the forks so quickly that neither other philosopher has a chance to grab one that is shared with him.

Solution to Exercise 7.4 (p. 88)

As usual, we must check that no more than one process is in the critical section at a time. We can our usual code

```
in_crit++;
assert(in_crit == 1);
in_crit--;
```

and declaring `in_crit` global. This code passes verification, **but** warns us that this code isn't necessarily even executed!

For a critical section protocol to be valid, it must also guarantee that each process eventually enters the critical section. Since the `goto`s create a control flow loop, we can check this by looking for non-progress cycles, labeling the critical section as progress.

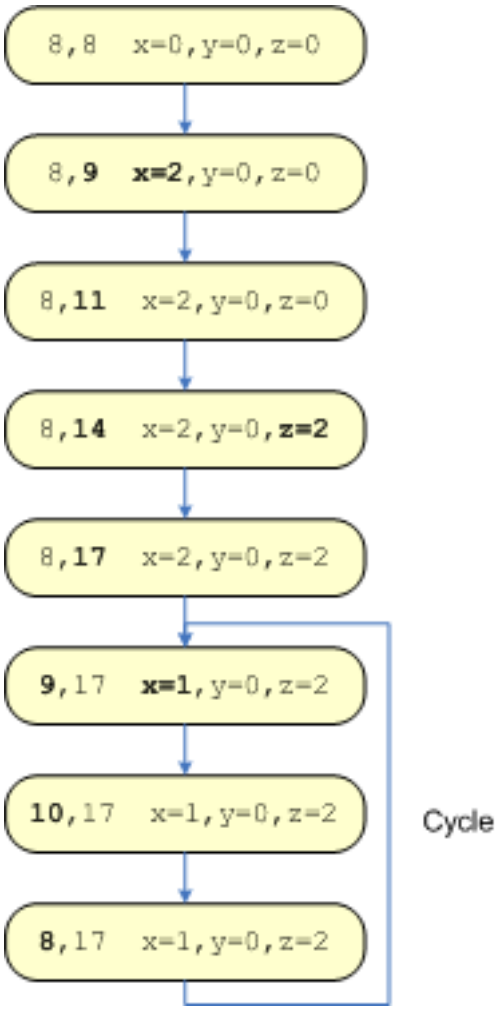


Figure 7.1: The shortest counter-example trace. For brevity, the local values of `me` are not shown, since they cannot vary.

Using temporal logic, we can create a formula that describe our desired behavior: "Eventually, each process gets to the critical section, but not both at the same time." One such formula is $([\texttt{U+25C7}]p \wedge [\texttt{U+25C7}]q \wedge \Box\neg(p \wedge q))$ where we define

```
#define p (user[0]@crit)
#define q (user[1]@crit)
```

and define the label `crit` in the critical section.

This is a Promela version of an algorithm once recommended by a major computer manufacturer. As you can see, like many other mutual exclusion algorithms that have been proposed, it is flawed.

Solution to Exercise 7.5 (p. 89)

The problem with the given code is that the concurrent `max` computations does not necessarily result in each element of `number[]` being unique. Uniqueness is assumed by the following conditional to prioritize the processes:

```
if ((number[j] > 0) &&
    (number[j] < number[i]))
```

One of the hints precludes changing the `max` calculation to produce uniqueness. So, we need a way to prioritize processes when they receive the same number. This is most easily accomplished by using their process IDs:

```
if ((number[j] > 0) &&
    ((number[j] < number[i]) ||
     ((number[j] == number[i]) && j<i)))
```

Of course, ...`j>i`... is also fine.

Glossary

A

Always

$\sigma_i \vDash \Box\phi$ iff $\forall j \geq i . \sigma_j \vDash \phi$

Example:

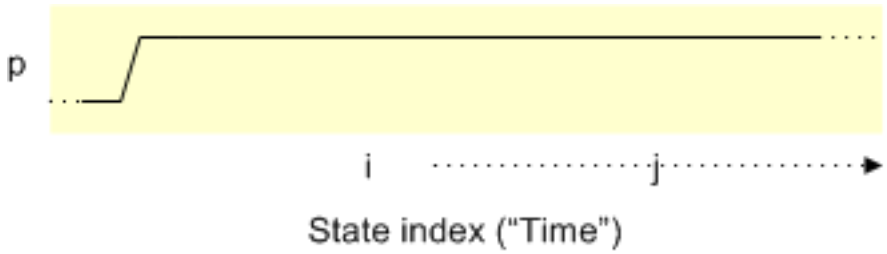


Figure 6.3: A generic trace σ satisfying $\sigma_i \vDash \Box p$. p is true in state i and all later states.

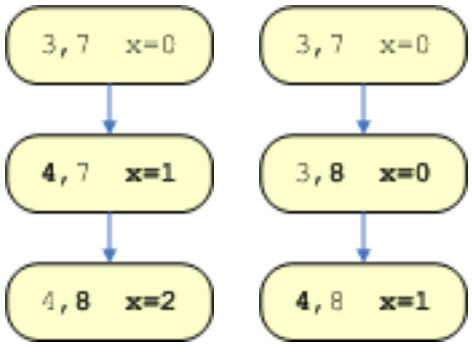
Atomic

An atomic operation is indivisible. A context switch to another process cannot happen during this operation.

Example: Statement-level Atomicity Let's examine the possible interleavings of the code fragments for two threads. First, let's assume that each assignment statement is atomic.

```
1  thread0:
2  {
3    x=x+1
4  }

5  thread1:
6  {
7    x=x*2
8  }
```



(a) (b)

Figure 2.1: There are two possible traces, as either thread's statement could execute first. Bold type highlights anything modified or assigned to each step.

Here, each trace is a timeline, representing one possible interleaving of operations. From top to bottom, each state represents one point in time, with a current line number for each thread, plus the value of each variable. The first state is the starting point, before execution begins. Here, we assume that `x` is initially zero. We will use traces and their timeline diagrams throughout this module to understand concurrent programs.

Example: Instruction-level Atomicity For contrast, let's change what is atomic. Now, assume that each machine instruction operation, such as loading, adding, or storing a register, is atomic. Again, we assume `x` is initially zero. Observe that with finer-grained atomicity, there are many more ways to interleave the threads, and (in this case) more possible result values for `x`.

```
 1  thread0:
 2  {
 3    r0 =  x
 4    r0 += 1
 5    x  =  r0
 6  }

 7  thread1:
 8  {
 9    r1 =   x
10    r1 ≪= 1  /* Shifting left one position multiplies by 2. */
11    x  =   r1
12  }
```

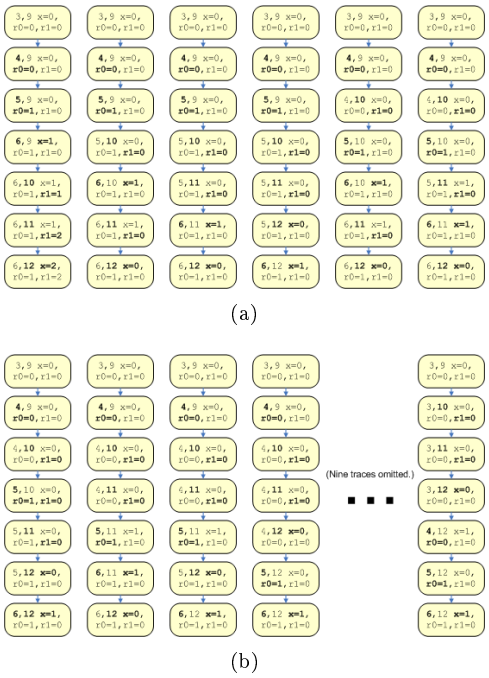


Figure 2.2: With finer-grained atomicity, there are many more ways to interleave the threads.

D Deadlock

(Informal) Deadlock is when two or more threads stop and wait for each other.

E Enabled

A guarded statement is enabled if its guard evaluates to true.

Eventually

$\sigma_i \vDash$ `[U+25C7]`ϕ iff $\exists k \geq i . \sigma_k \vDash \phi$

Example:

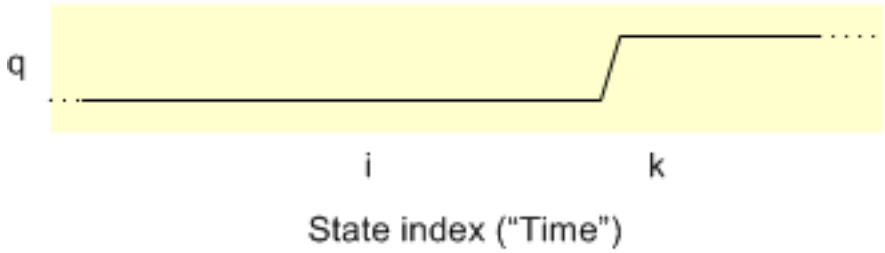


Figure 6.5: A generic trace σ satisfying $\sigma_i \vDash$ `[U+25C7]`q. q is true in some state k after state i.

F Fairness

(Informal) Fairness is the idea that each thread gets a turn to make progress.

G Guarded statement

A guarded statement, written in Promela as `guard -> body`, consists of two parts. The **guard** is a boolean expression. The body is a statement. Execution of the body will **block**, or wait, doing nothing, until the guard becomes true.

L Livelock

(Informal) Livelock is when two or more threads continue to execute, but make no progress toward the ultimate goal.

R Race Condition

A race condition is when some possible interleaving of threads results in an undesired computation result.

Example: Returning to our airline reservation system example (Example 2.2), suppose Joe and Sue each see that one seat is left. It is a race condition error if the system allows both of them to successfully reserve the one seat.

S Starvation

(Informal) Starvation is when some thread gets deferred forever.

State

A snapshot of the entire program. In Promela, this is an assignment to all the program's variables, both global and local, plus the program counter, or line number, of each process. In other languages, the state also includes information such as each process' stack contents.

State

A state captures all the current information in a program. This includes all local and global variables' values and each thread's current program counter. More generally, this includes all the memory and register contents.

State space

A directed graph with program states as nodes. The edges represent program statements, since they transform one state to another. There is exactly one state, the **initial state**, which represents the program state before execution begins.

State Space Automaton

A State Space Automaton A is a tuple $< S, s_0, T, \text{Prop}, P >$ where:

- S is a set of states. As before (Definition: "State", p. 17), a **state** is an assignment to all program variables (both local an global), including the program counter (line number) for each process.
- $s_0 \in S$ is the **initial state** or **start state**.
- $T \subseteq S \times S$ is the **transition relation**. Equivalently, this is the edge relation of the state space (Definition: "State space", p. 17).
- Prop, a set of propositions.
- $P : S \to 2^{\text{Prop}}$ is a mapping which, given a state, indicates all the properties that are true in that state.

Strong Until

$\sigma_i \vDash (\phi U \psi)$ iff $\exists k \geq i. (\sigma_k \vDash \psi \wedge \forall j. (i \leq j < k \rightarrow \sigma_j \vDash \phi))$.

Example:

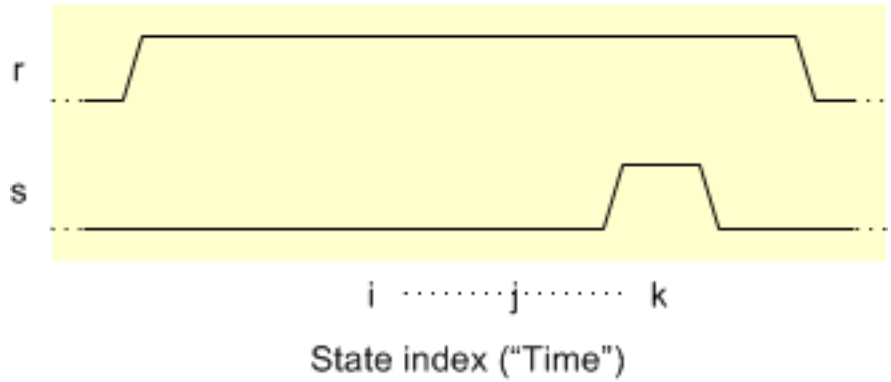


Figure 6.6: A generic trace σ satisfying $\sigma_i \vDash (rUs)$. r is true in states i through $(k-1)$.

Stutter-Extend

The finite trace $\sigma_0,\ldots,\sigma_n$ can be stutter-extended to the infinite trace $\sigma_0,\ldots,\sigma_n,\sigma_n,\sigma_n,\ldots$.

T

Trace

A sequence — possibly infinite — of successive states, corresponding to one particular run of a program.

Trace

A trace is the sequence of operations (and their data) performed in a particular run of a concurrent program. Equivalently, it is the sequence of states during a run — *i.e.*, the collection of variable and program counter values.

Trace

Given a state space automaton, a **trace** σ (sometimes called an ω-**trace**) is a (possibly infinite) sequence of states σ_0, σ_1, σ_2, ... which respects the automaton's transtion relation T.

V

Valid end state

(Informal) Some states of our state space may be designated valid end states. These represent points where all of the threads have completed execution.

W

Weak Fairness

Each statement that becomes enabled and remains enabled thereafter will eventually be scheduled.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Module: "Using Temporal Logic to Specify Properties: Homework Exercises"
By: John Greiner, Ian Barland, Moshe Vardi
URL: http://cnx.org/content/m12940/1.3/
Pages: 87-93

Model Checking Concurrent Programs
Introduction to the use of Promela and SPIN as a model-checker for verifying concurrent programs. The underlying transition system is used as a model for the interpretation of Linear Temporal Logic (LTL) formulas.