INTRODUCTION TO ELEVATOR CONTROL IN SPIN

The Promela system specification of the simple elevator control system starts with definitions of constants and declarations of global variables. In specifying an elevator that serves three floors, we might begin with

```
bit doorisopen[3];
chan opencloseddoor=[0] of {byte,bit};
```

The first declaration says that `doorisopen` is an array of three bits, where the i-th bit represents whether the door is open (`1`) or closed (`0`). The second says that `opencloseddoor` is a channel — it will provide communication between the elevator and the floor doors — with no buffering capacity (i.e., communication will be via rendezvous) that carries messages consisting of a byte (the floor number) and a bit (`1` for a command to open the door, `0` for a command to close the door).

Next, process types are declared. These declarations consist of the keyword `proctype`, followed by the name of the process type, and an argument list. For example, the process type `door` takes a floor number as an argument.

```
proctype door(byte i){
do
:: opencloseddoor?eval(i),1;
   doorisopen[i-1]=1;
   doorisopen[i-1]=0;
   opencloseddoor!i,0
od
}
```

The `do`-loop (terminated by `od`) is an infinite loop. The body of the loop says to wait for an order from the elevator to open this door, then open the door (with the result indicated by setting the appropriate bit to `1` in the `doorisopen` array), then close the door (with the result indicated by setting the appropriate bit to `0` in the `doorisopen` array), then to send a message to the elevator indicating the door has been closed. The `::` indicates the start of a command sequence block. In a `do`-loop, a non-deterministic choice will be made among the command sequence blocks. In this case, there is only one to choose from.

In general, command sequences can be preceded by *guards*, as in CSP.[1] The idea is that the choice of command sequence to execute is restricted to those with a guard that evaluates to “true”. The `elevator` process type illustrates the use of guards.

```
proctype elevator(){
show byte floor = 1;
do
:: (floor != 3) -> floor++
:: (floor != 1) -> floor--
:: opencloseddoor!floor,1;
   opencloseddoor?eval(floor),0
od
}
```

The first declaration in the body of the procedure type introduces a local variable, `floor`, that tracks the location of the elevator, which is initially the first floor. (Prefixing the declaration with `show` has no semantic effect. It just indicates that the value of this variable should be shown when message sequence charts are generated during simulation.) Then choice is made among three command sequences in an infinite loop. The first option is to go up a floor, provided the elevator is not already at the third floor. The second option is to go down a floor, provided the elevator is not already at the first floor. Finally, the elevator can send a message to the floor where it is located requesting that the door open, and then wait until it receives a message that the door has been closed.

Finally, the process types are instantiated and executed.

```
init{
  atomic{
    run door(1);
    run door(2);
    run door(3);
    run elevator()
  }
}
```

[1]Actually, guards are unnecessary in Promela, as all statements block when they are not executable and, in particular, Boolean statements block when they are false. This “feature” is an absolutely terrible design blunder. Figuring out where the program on the previous page may block requires reading every statement in the program; in a guarded command language, you just look at the statements on the left of the arrows. (In some sense, *every* statement is a guard, but that’s not a useful way of looking at things becaause most of them — all those assignment statements — are always executable.)

After looking at the history of the language, it’s pretty clear what happened. Promela has always had this block-on-nonexecutability “feature”. Users rightly complained that Promela programs were almost unreadable as a result. Gerald Holzmann, the creator of Spin and Promela, had a clever idea: by introducing ‘`->`’ as an alternate syntax for ‘`;`’, users could write Promela programs that look like programs written in a respectable CSP-like guarded command language, yet none of the old unreadable programs would be broken. The only downside of this minimal effort approach to fixing the problem is that nothing prevents people from writing more unreadable programs in the future. (In fact, new heights of unreadability could be reached by randomly changing semicolons to arrows.)

At least for this course — and forever after, if you’re smart — adopt the following rule: *Never write a Promela program that relies on the fact that false statements block until they are true.* If you’re using a statement as a guard, indicate that it is a guard by using the `->` notation.

Now that the elevator specification has been completed, Spin can be used to check it for syntactic errors, to simulate its execution, and to verify that it satisfies LTL formulas of interest.[2] Verification begins with definition of atomic proposition symbols, for example

```
#define open1 doorisopen[0]
#define closed1 !doorisopen[0]
#define open2 doorisopen[1]
#define closed2 !doorisopen[1]
#define open3 doorisopen[2]
#define closed3 !doorisopen[2]
```

where `!` is Spin's negation symbol. Formulas of interest, such as

```
[](open1 -> <> closed1)
```

where `[]` is Spin's notation for the `G` ("always") modality and `<>` is Spin's notation for the `F` ("sometime") modality, and

```
<> (open1 || open2 || open3)
```

where `||` is Spin's disjunction symbol, can then be syntactically checked and formally verified, or refuted, via model checking.

[2]Note that Spin does not support the `X` modality by default, because some of the techniques it uses to handle large state spaces require that the class of computations that satisfy the formula be closed under "stuttering", that is, inserting extra states in a computation where nothing happens.