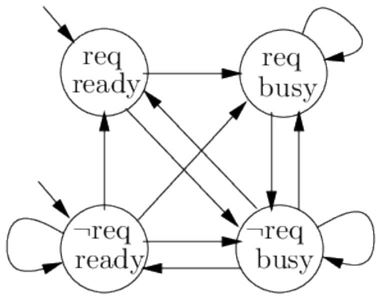

Figure 3.9. The model corresponding to the SMV program in the text.

easily verify that the specification of our module `main` holds of the model in Figure 3.9.

Modules in SMV SMV supports breaking a system description into several *modules*, to aid readability and to verify interaction properties. A module is instantiated when a variable having that module name as its type is declared. This defines a set of variables, one for each one declared in the module description. In the example below, which is one of the ones distributed with SMV, a counter which repeatedly counts from 000 through to 111 is described by three single-bit counters. The module `counter_cell` is instantiated three times, with the names `bit0`, `bit1` and `bit2`. The counter module has one formal parameter, `carry_in`, which is given the actual value 1 in `bit0`, and `bit0.carry_out` in the instance `bit1`. Hence, the `carry_in` of module `bit1` is the `carry_out` of module `bit0`. Note that we use the period '.' in `m.v` to access the variable `v` in module `m`. This notation is also used by Alloy (see Chapter 2) and a host of programming languages to access fields in record structures, or methods in objects. The keyword `DEFINE` is used to assign the expression `value & carry_in` to the symbol `carry_out` (such definitions are just a means for referring to the current value of a certain expression).

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
LTLSPEC
  G F bit2.carry_out
```

Exercises 3.3

1. Consider the model in Figure 3.9 (page 193).
   * (a) Verify that `G(req -> F busy)` holds in all initial states.
   (b) Does ¬(`req` U ¬`busy`) hold in all initial states of that model?
   (c) NuSMV has the capability of referring to the next value of a declared variable `v` by writing `next(v)`. Consider the model obtained from Figure 3.9 by removing the self-loop on state `!req & busy`. Use the NuSMV feature `next(...)` to code that modified model as an NuSMV program with the specification `G(req -> F busy)`. Then run it.
2. Verify Remark 3.11 from page 190.
* 3. Draw the transition system described by the ABP program.
   Remarks: There are 28 reachable states of the ABP program. (Looking at the program, you can see that the state is described by nine boolean variables, namely `S.st`, `S.message1`, `S.message2`, `R.st`, `R.ack`, `R.expected`, `msg_chan.output1`, `msg_chan.output2` and finally `ack_chan.output`. Therefore, there are $2^9 = 512$ states in total. However, only 28 of them can be reached from the initial state by following a finite path.)
   If you abstract away from the contents of the message (e.g., by setting `S.message1` and `msg_chan.output1` to be constant 0), then there are only 12 reachable states. This is what you are asked to draw.

Exercises 3.4

1. Write the parse trees for the following CTL formulas:
   * (a) $\mathrm{EG}\, r$
   * (b) $\mathrm{AG}\,(q \rightarrow \mathrm{EG}\, r)$
   * (c) $\mathrm{A}[p \ \mathrm{U}\ \mathrm{EF}\, r]$
   * (d) $\mathrm{EF}\,\mathrm{EG}\, p \rightarrow \mathrm{AF}\, r$, recall Convention 3.13
   (e) $\mathrm{A}[p \ \mathrm{U}\ \mathrm{A}[q \ \mathrm{U}\ r]]$
   (f) $\mathrm{E}[\mathrm{A}[p \ \mathrm{U}\ q] \ \mathrm{U}\ r]$
   (g) $\mathrm{AG}\,(p \rightarrow \mathrm{A}[p \ \mathrm{U}\ (\neg p \wedge \mathrm{A}[\neg p \ \mathrm{U}\ q])])$.
2. Explain why the following are not well-formed CTL formulas:
   * (a) $\mathrm{F}\,\mathrm{G}\, r$
   (b) $\mathrm{X}\,\mathrm{X}\, r$
   (c) $\mathrm{A}\neg\mathrm{G}\,\neg p$
   (d) $\mathrm{F}\,[r \ \mathrm{U}\ q]$
   (e) $\mathrm{EX}\,\mathrm{X}\, r$
   * (f) $\mathrm{AEF}\, r$
   * (g) $\mathrm{AF}\,[(r \ \mathrm{U}\ q) \wedge (p \ \mathrm{U}\ r)]$.
3. State which of the strings below are well-formed CTL formulas. For those which are well-formed, draw the parse tree. For those which are not well-formed, explain why not.