Translating pseudocode to SMV

In the reference text, *Model Checking*, Clarke, Grumberg, and Peled describe a procedure for translating pseudocode programs into SMV models. I sketch this translation below with Deitel's second attempt at *mutual exclusion*. I do not go into great detail because it might be taken as suggesting that you perform your translations in a mechanical fashion. The goal at this stage is to develop intuition about modeling in SMV, so I hope this sketch will simply give you some ideas about how to approach the model-building exercise.

A couple of things to keep in mind are:

- The SMV language is not "structured" so hierarchical programs must be translated into something closer to assembly language.
- All program variables are global—essentially they are also processes—so the effects of assignments are modeled remotely from the points of control where those assignments take place

We begin with Deitel's pseudocode representation of a process.

```
bool T[2] = 0, 0;

procedure p[i]
 {
  while (1) do
   {
    NONCRITCIAL;
    while T[1-i] do {/* nothing */;}
    T[i] := 1;
    CRITICAL;
    T[i] := 0;
   }
 }
```

- First, *label* each statement.

```
bool T[2] = 0, 0;

procedure p[i]
 {
  while (1) do
   {
   N: NONCRITCIAL;
   W: while T[1-i] do {/* nothing */;}
    I: T[i] := 1;
   C: CRITICAL;
   X: T[i] := 0;
   }
 }
```

- Each program variable has a corresponding variable in SMV. In addition, add a variable `pc` for each process to hold it's control-state. Usually, `pc` will be a local `MODULE` variable.

```
VAR
  T1:  boolean;
  T2:  boolean;
  p1:  process p(T1, T2);
  p2:  process p(T2, T1);

⋮

MODULE p(mybit, otherbit)

  VAR
    pc:  N, W, I, C, X;

  ⋮
```

- Initialize all the variables properly, and start building their *next-state* expressions.

```
VAR
  T1:  boolean;
  T2:  boolean;
  p1:  process p(T1, T2);
  p2:  process p(T2, T1);

ASSIGN
  init(T1) := 0;
  next(T1) :=
    case
    ⋮
    1:  T1;
    esac;

  MODULE p(mybit, otherbit)

    VAR
      pc:  {N, W, I, C, X};

    ASSIGN
      init(pc) := N;
      next(pc) :=
        case
        ⋮
        1:  pc;
        esac;
      ⋮
```

- Continue to build the `case` statements by adding clauses according to the pseudocode fragments.
  - For fragment N: `NONCRITICAL;` W:... add the clause

```
next(pc) :=
  case
  (pc = N): W;
  ⋮
```

NOTE: If we want to model the possibility that a process can nondeterministically stay non-critical forever, we would use {`N, W`} *here instead of simply* `W`.

  - For fragment W: `while otherbit do skip;` I:... add the clauses

```
next(pc) :=
  case
  (pc = W) & (otherbit = 1):  W;
  (pc = W) & (otherbit = 0):  E;
  ⋮
```

  - For fragment I: `T[i] := 1;` C:... add two clauses, one to model program execution and the other to model the effect of the assignment.

```
next(pc) :=
  case
  (pc = I): C;
  ⋮

next(mybit) :=
  case
  (pc = I): 1;
  ⋮
```

  - And so on.