15-398 Bug Catching: Automated Program Verification and Testing
Homework Assignment

October 18, 2002

1. The following SMV code models a simple in-order pipeline stage. Such a pipeline stage is part of all modern microprocessors. The instructions of a program are processed as a car in an assembly line.

   The signal that indicates that an instruction moves into the next stage is called *update enable* signal. The instruction in this next stage is overwritten, unless it also moves. This update enable signal is used as clock signal for the registers of the next stage.

   The module of the stage takes the following parameters: The parameter `first` is set if the stage is the first stage. This allows injection of new instructions. The parameter `prev_ue` is the update enable (i.e., clock) signal of the previous stage. The parameter `next_stall` is the stall signal of the next stage. The stall signals allow stopping the execution in arbitrary stages. A reason for such a stall condition might be slow external memory (or, in case of the assembly line, that a worker dropped his screw driver).

```
MODULE simple_stage(first, prev_ue, next_stall)
VAR
  full   : boolean;
  i_stall: boolean;
ASSIGN
  init(full):=first;
  next(full):=prev_ue | stall | first;
DEFINE
```

```
  ue:=full & !stall;
  stall:=(i_stall | next_stall) & full;
```

The module has two state variables: the variable `full` indicates that there is an instruction in the stage. If it is not set, the stage is empty. The variable `i_stall` allows nondeterministic stalls within the stage, as described above.

Describe - informally - the behavior of one stage.

2. The stages can be composed to a five stage pipeline as follows:

```
MODULE main
VAR
  stage0: simple_stage(1, 0,         stage1.stall);
  stage1: simple_stage(0, stage0.ue, stage2.stall);
  stage2: simple_stage(0, stage1.ue, stage3.stall);
  stage3: simple_stage(0, stage2.ue, stage4.stall);
  stage4: simple_stage(0, stage3.ue, 0);
```

$stage0.full$

$stage1.full$

$stage2.full$

$stage3.full$

$stagen.full$

Formalize and verify the following properties:

- **Liveness:** Eventually, every stage will be clocked (i.e., the update enable signal is active) infinitely often. In order to verify this claim, you have to make an assumption. What is the assumption? What goes wrong without it? How does this compare to the assembly line?
- **Nothing is overwritten:** If the instruction in a given stage is overwritten, it simultaneously moves into the next stage. For which stages does this hold? Make the comparison with the assembly line!
- **Nothing is lost:** If an instruction does not move into the next stage, it will be in the same stage in the next cycle.
- **Nothing is duplicated:** If an instruction in a given stage moves into the next stage, and no instruction moves into the given stage, the given stage will be empty in the next cycle.
- **No instruction appears without reason:** Any stage - other than stage 0 - is not full until an instruction moves into the stage.

3. The model of the bus protocol presented in the class is available on the class home page. Add a special node that is not communicating initially, but has no power. Later on, after the node is powered on, it tries to synchronize with the nodes that are already running. This process is called *integration*.

   - Extend the control automaton by a state `no_power` and a state `integrating`.
   - The automaton is initialized with the state `no_power`.
   - If in the state `no_power`, the node can stay there or go into the state `integrating`. This is chosen nondeterministically.
   - If in the state `integrating`, the node watches the bus. Upon activity, the node goes into the `busy` state.
   - Formalize (and verify) the following property: If the node gets power, it will be in the `busy` state eventually.
   - This requires help from the other nodes. Why? How can you ensure this help as part of the implementation, i.e., without a fairness constraint?