CS412 Lab session 4

Sequences and more investigation of a specification

The supermarket checkout specification

The following scenario follows on from the checkout idea in lectures.
Suppose a supermarket has 10 checkouts, each of which may be open or closed. Each open checkout has a queue (possibly empty) of customers. Write the static part of the machine *Checkout* to include the following variables:

checkouts - a suitable structure of 10 sequences
opened - the set of tills currently open
closed - the set of tills currently closed

The invariant should include the types of these and also state any relationships between them. Make sure no person can occur twice in your checkout system. Define the following operations:

name	**ins**	**description**	**outs**
join	*pp cc*	Person *pp* joins checkout *cc*	
is_served	*cc*	Checkout *cc* serves the next person, *oo*	*oo*
opentill	*cc*	Open checkout *cc*	
closetill	*cc*	Close checkout *cc*	
query_queue	*pp*	Find out which queue a customer is in	*oo*
query_shortest		Find the shortest queue (or *a* shortest queue if there are several)	*cc*

When your specification is analysed, animate it to investigate its behaviour. Use the information from the animation and from model checking on the invariant to correct your machine if necessary. You can also try generating POs but you are likely to find that the tool cannot prove them all automatically. As stated last week, this could mean either that your specification is wrong or that the spec is ok but the tool is not clever enough to find all the proofs on its own.

More investigation using proof obligations

If the POs cannot be proved automatically we need to consider what to do. We might not want to launch into manual proof immediately - this is hard, time-consuming and can be wasted effort if the spec was incorrect! A good first step is to investigate first of all to check as far as possible that the spec is doing what you want and that you believe it is correct. At your disposal you have ProB and the information from the prover.

As an example, go back to the version of the ProjectsLab3 machine that we put the known bug into last week. See below. Make sure it's type-checked then generate POs and try automatic proof. The relevant operation remains unproved. (Please note here and in future work that the exact presentation of POs and capabilities of the prover may vary between versions of the tool.) Go into the interactive prover by clicking the Ip button. The situation window has a red cross against the signup operation, and clicking on this will reveal which PO(s) for this operation is (are) the problem. You should click one requiring that adding the new pair should still ensure you have a function. Clicking on this PO causes the obligation to appear in the top centre window. You should see something like this:

```
"Invariant is preserved" => projects\/{ss|->pp}: STUDENT +-> PROJECT
```

The PO wants us to show that, when $ss \mapsto pp$ is added, *projects* is still a function. Do we believe this should follow from our specification or not? The most obvious thing to do (and probably should have tried before getting to this stage!) is finding an instance of the problem in ProB. If you open up this spec in ProB and model check to find an invariant violation it will show you one. The problem occurs after it's done a signup operation - and there is a comment in red to say that we have a student mapping to two possible values. This should be your first sort of investigation and you will be able to find a lot of specification errors in this way.
Suppose however that ProB doesn't find an error but you still have obligations you can't prove. Is it just that the prover wasn't clever enough - or is it that ProB hasn't sufficient scope to find a problem? Or maybe ProB does report a problem but you can't work out why the specification is wrong. The next thing is to investigate the POs.

Viewing hypotheses

Go back to the prover where you should have displayed the PO already. Before the goal there is an implication sign indicating that this goal should follow from the local definitions in the specification. To add these as hypotheses in the current proof click on the yellow *dd* button (either above or below the display window). The goal on its own is now printed. To see all hypotheses generated from the spec type $sh(a)$ in the command window which is underneath the proof display window. If you just wanted to see the hypotheses which involve things mentioned in the goal you can click the yellow rp_1 button. In order to try to prove a goal we can use both the specific hypotheses relating to this particular specification and any general rules (about logic, sets, etc) which can help.
At this stage we're just investigating rather than proving. However, if we want to see where the automatic prover gets stuck we can call it (either click the yellow *pr* button or type *pr* in the command window). The prover has worked towards the proof and now shows you the subgoal it got stuck on. Check you understand what the new goal says. For this to be true, we'd need to know either *ss* isn't in the domain of projects, or, if *ss* is there already then it must be mapping to *pp*. (We'll look at where this comes from later). When you're

working with a proof, consider that you may be trying to prove something that is false! If you inspect the goal and realise this, then it's time to go back to the specification and decide how to correct it. If you're still wondering whether it is provable, it may be useful to view the hypotheses and decide whether we believe the goal should be provable from these with a bit of help or not.
The fact that there doesn't seem to be anything in the hypotheses to ensure the goal we want may perhaps lead us to realise our mistake. But suppose we are still unsure. Another thing to try is to notice that we have only tried the basic automatic proof level. Can the automatic prover do better if we increase the level? First, reset the proof (blue button to LHS of the "one step back" button). Go to the far RHS of the tool bar and you should see a a Force selector. Click this to a higher level. You should see that it has been able to break things down further and a new subgoal has been displayed as the stopping point. If you click rp_1 you may notice that some more hypotheses were added by the steps of the proof the new force level has allowed - we are now in a branch of the proof which is assuming that *pp* is in the domain of *projects* (that is, there is already a project registered for *pp* before the signup operation). We'll talk about how this happens later. For now, notice that the generated subgoal is clearly saying that we'd need to know that, at the given stage in the proof, adding the pair $ss \mapsto pp$ needs to ensure that *projects* component before the operation maps *ss* to *pp*. remains functional. There is nothing in the hypotheses to help us. The fact that this is looking at cases with *pp* in the domain of *projects* should help us to realise that *signup* is not ruling out that possibility. The most likely solution is that we should just check that *pp* does not already have a project although there would be other options as well such as overwriting the old one. Let's assume we have spotted the error. Exit the proof and go back to the spec to alter the precondition of *signup* by uncommenting the check. Once it's saved and type checked again you can click either the blue F0 or blue F1 button for automatic proof (F1 tries a bit harder). All POs should now be proved.

Viewing the rulebase

Finally for now, let's have a quick look at how the subgoals are generated. Click Component → Proof → Unprove to reset. Then click Ip,select the signup operation and click on the PO for this operation that we were looking at before. Click *dd* to generate the current hypotheses. What does the prover use to generate subgoals from your main goal? The answer is that there is a rule base which can be used to find applicable rules. Make sure you have the Theory List showing (if you can't see it, click the View button and check the Theory List box). The theory list has two option: View as tree shows you all the rules it knows about. They are grouped in vaguely helpful "theories" (eg: relating to a single logical operator). You can have a browse to see the sorts of things there. However, if you choose the View As List option it will show you only the rules that are applicable to your current goal. There are still quite a few. We're trying to find something helpful about a set union. If you click on the first name in the list (CommutativityXY.4) it expands to show a rule that seems true - but not very helpful. Similarly for the next two rules. But when you get

to b1.6 it looks quite useful. Its goal (last line) is the same format as ours. And the subgoals it generates (first two lines) look like things we should be able to prove. If you applied it the tool would pattern-match your goal to the bottom line of the rule and then use the lines above the implication sign to generate subgoals. We'll look at applying rules next time.
If you have time, go back to your *Checkouts* spec and examine the unprovable POs. Do you think they should be provable if we give the prover a bit of help?

More sequence practice

Here is another specification you can try writing for more practice with sequences. Write an abstract machine, *Calculator*, which keeps as its state *nseq*, the sequence of all natural numbers entered by the user and *tot*, the number of elements in the sequence. The calculator cannot store more than *nmax* numbers. It has the following operations.

Name	**Inputs**		**Outputs**
num_in	$ii : \mathbb{N}$	Adds the input to the sequence	
sum_all		Outputs the sum of all numbers in *nseq*	$oo : \mathbb{N}$
clear_last		Removes last number entered	
clear_all		Removes all entries	
how_many	$ii : \mathbb{N}$	Outputs number of times *ii* occurs in the sequence	$oo : \mathbb{N}$
order_seq		Outputs the sequence sorted in ascending order	$ss : \mathrm{seq}\,\mathbb{N}$
find_num	$ii : \mathbb{N}$	Outputs a position in the sequence where *ii* occurs (0 if not there)	$oo : \mathbb{N}$

```
/* This is the ProjectsLab3 machine with deliberate error */
MACHINE           ProjectsLab3

SETS              STUDENT; PROJECT

CONSTANTS         maxmark, MARK

PROPERTIES        maxmark : NAT1 & MARK = 0..maxmark

VARIABLES         students, projects, marks

INVARIANT         students <: STUDENT &
                  projects : STUDENT +-> PROJECT &
                  dom(projects) = students &
                  marks : STUDENT +-> MARK &
                  dom(marks) <: students

INITIALISATION    students := {} || projects := {} || marks := {}

OPERATIONS

 signup(ss,pp) =
  PRE
     ss:STUDENT & pp:PROJECT /* & ss /: dom(projects)  */
  THEN
     students := students \/ {ss} ||
     projects := projects \/{ss |-> pp}
  END;

pp <-- assignproject(ss) =
  PRE  ss:STUDENT
  THEN
     ANY ppx
       WHERE ppx : PROJECT
       THEN pp := ppx
            || students := students \/ {ss}
            || projects(ss) := ppx
       END
  END;

pp <-- queryproject(ss) =
  PRE
    ss:STUDENT & ss : dom(projects)
  THEN
    pp := projects(ss)
  END;
```

```
  entermark(ss,mm) =
   PRE
     ss:STUDENT & mm:MARK & ss : dom(projects)
   THEN
     marks(ss) := mm
   END;

mm <-- querymark(ss) =
  PRE
    ss:STUDENT & ss : dom(marks)
  THEN
    mm := marks(ss)
  END


END
```