CS412 Lab session 7

Starting refinement and adding rules

1 Refinement

This week's lab basically covers the machines from questions 1 and 3 in this week's exercise sheet. They're reproduced here. To write a refinement machine, click on the blue + button to add a component, select the Refinement type and choose the name of the abstract machine you're refining. Click next. You can then say what you want copied over from the specification (usually it's useful to select all the operations at least because this brings in the structure and you can then edit as required).

The Passengers machine (Q1 in exercises)

The following machine keeps track of passengers booking and cancelling for a coach tour. Provide a refinement which stores the bookings as a partial function with domain some subset of $1 \ldots 50$ and whose range is *pass*. The refinement should provide deterministic operations.

```
MACHINE         Passengers
SETS            PID
VARIABLES       pass
INVARIANT       pass : FIN(PID) & card(pass) <= 50
INITIALISATION  pass := {}
OPERATIONS
     book(pp) = PRE   pp:PID & pp /: pass & card(pass) < 50
                THEN  pass := pass \/ {pp}
                END;

     cancel(pp) =  PRE   pp:PID & pp : pass
                   THEN  pass := pass - {pp}
                   END;

     oo <-- query(pp) = PRE   pp:PID
                        THEN  oo := bool(pp : pass)
                        END;

     oo <-- spaces = oo := 50 - card(pass)
END
```

Write the refinement and investigate in the toolkit. If any POs do not autoprove for the refinement - why not?!

The OpenDay machine (Q3 in exercises)

Suppose the CS department keeps a record of up to 30 students who are willing to act as guides at open days. For each open day, 6 student guides are required. Suppose student identifiers are represented using *SID*.

1. Write an abstract machine for this situation which uses the set variables *volunteer* for the (up to) 30 possible guides and *chosen* for the 6 currently selected (although a selection cannot be made until at least 6 students have volunteered). It should include a suitable invariant and initialisation and the following operations:

   - *newvolunteer*(vv) - to add vv as a volunteer if max not reached;
   - *swap*($v1, v2$) - when the 30 max volunteers has been reached, this replaces one of the current 30 volunteers ($v1$) with a new volunteer ($v2$);
   - *newchoice* - to nondeterministically choose 6 volunteers (either to make the initial selection of 6 or, if there are at least 12 volunteers, to choose a new selection of 6 who aren't currently selected);
   - *query* - to output the current value of "chosen".

2. Suppose that a refinement is proposed in which the volunteers and the chosen set are to be represented as sequences of *SID*s. Write a suitable linking invariant for this refinement machine. Consider how instances of concrete and abstract states match up to check you understand the relationship given by your linking invariant.

3. Write a refinement machine which uses this linking invariant. Suppose the decision is made that, for the *newchoice* operation, the initial selection is to be made deterministic at this stage but subsequent choices are to be left nondeterministic.

4. Try different data refinements for the original machine and see how the linking invariant, initialisation and operations work out for your different approaches. Try some of these things out in the B tool.

2 Adding rules in the prover

In the previous lab we looked at interacting with the prover and saw how to find and apply rules to help it when necessary. However, the rule base is not complete and further rules may at times be necessary in order to verify some properties. You won't be required to do this, but the following section provides material and basic syntaax for doing this. So, if you're interested and would like to take the proof aspect a bit further, the following sections should provide a starter to get going. If you want to know more about the syntax for more complex rules and the many options for expressing guards, see the prover manuals.

When there aren't enough rules

The rule base covers many of the basics, but it is by no means complete. That is, there will be provable goals that you cannot verify. In this case you will need to add rules. Also, there may be rules that you want to add for convenience, particularly if a number of proofs in a development will want to use it. As discussed in lectures, this is a potentially dangerous activity and needs to be done with great care. As an example, consider the following spec:

```
MACHINE tryit
VARIABLES         mm
INVARIANT         mm : 0 .. 2
INITIALISATION    mm := 0
OPERATIONS
    testop(xx) = PRE xx:NAT THEN mm := xx mod 3 END
END
```

One PO doesn't autoprove ... When we try to prove it, after clicking dd, the first thing we get offered in the "View as list" view of the Theory window is *InSetXY.13*. It looks promising so we can apply it. We now see a goal of $0 <= xx \bmod 3$. Again, the first rule offered, *b1.59* looks good so apply that. Then once more the top rule offered seems to be what we need, *b1.46*. But when we apply this it doesn't seem to work.

Q0 Look at the rule. Why doesn't it apply in this situation? Does this mean that the rule is incorrect?

At this point we may decide we want to add a rule which is both true and suitable for current purposes. Let's suppose we decide to add the following rule:

```
(no antecedents)
=>
x mod 3 : 0..2
```

For each component we can add our own rules in the componentname.pmm file. In the prover, click on Proof → Open pmm. The first time you do this for a component, it'll offer you the chance to create the pmm file. In this file you can organise your new rules into theories just like the ones in the prover, with rules separated by semi-colons. Add the rule by typing and saving:

```
THEORY mymodrules IS

  btrue
  =>
  x mod 3 : 0..2

END
```

with the theory name whatever you choose. The btrue is needed as a formality - even though the goal is true without any subgoals needed, the syntax of rules doesn't allow you to leave the antecedents blank.

We should then be able to load the new theory by going back to the prover window and typing *pc*. If you pmm file is syntactically ok it will be loaded, but you get an error message otherwise (and will then to correct the rule).
Select the troublesome PO, click on dd and then when you look at the applicable rules on the RHS you should see your new theory and the added rule. Use a proof step of *ar* to apply this. This should discharge the obligation.

Adding hypotheses

Go back to the tryit spec and change the invariant to $mm : 0..100$ then go in to the prover again and click dd. We are now left trying to prove that $xx \bmod 3 : 0..100$. In fact, we could do this using the existing rules from the prover as above. But suppose we wanted to use our new rule. The problem is that the added rule doesn't match the goal so you can't apply it. However, it's clear to see that our rule would *imply* the goal you want - that is, if xx is between 0 and 2 it's certainly between 0 and 100.
In a case like this, as we saw in the previous lab sheet, we can add a hypothesis (which will be justified by our rules and existing hypotheses). In this case we can add $xx \bmod 3 : 0..2$. We can then use the new hypothesis to prove our required goal. Step through to see how this works out:

1. add the new hypothesis using $ah(xx \bmod 3 : 0..2)$
2. the prover then presents this to you as a goal because you must *prove* it's legitimate to add it as a hypothesis
3. prove it by invoking the rule you added $ar(mymodrules.1, Once)$
4. the prover now presents the goal of proving the original goal using the intermediate step: goal is $xx \bmod 3 : 0..2 => xx \bmod 3 : 0..100$
5. click dd to get the RHS of this as your goal with LHS added as a hyp
6. try sh(a) to see it's now in the hyps
7. goal is now once again $xx \bmod 3 : 0..100$ - but with $xx \bmod 3 : 0..2$ in the hyps. It will be able to prove this with *pr*, or if you want to do it yourself, take a look at the rule InSetTryXY.1 Applying this will match the binhyp to your added hypothesis and will automatically check that 0..2 is a subset of 0..100.

In effect, this uses a step of "forwards proof". We noted that $xx \bmod 3 : 0..2$ would be a good intermediate step and so we showed that this could be derived from the existing hypotheses. It is a useful technique when you can see that a property derivable from the hyps will help prove your goal.

Another variant

Edit the tryit invariant back to $mm : 0..2$. The rule that we added before was specific for the divisor 3. Let's suppose we'd like to to fix the problem by adding

a rule which was rather more general which would work for any positive integer divisor.
Q1 Add another rule to your theory in the pmm file to do this. Reset your proof so you can start it again and this time prove it using the more general rule. Remember: the syntactic matching is very precise. If something doesn't seem to apply when you expect it to look closely to check there is an exact syntactic match. Check your rule also works for the variant of the spec where $mm : 0..100$

The guard language

The full syntax for writing rules is quite detailed - and can be rather subtle. For current purposes we will look at just a few of the possible instructions as given in lectures. If you're interested in finding out more, consult the prover manuals.

Looking in the hypotheses

Suppose you want to write a rule saying that if you have an injective function to start with and you then remove a singleton pair the result is guaranteed to still be an injective function. Suppose you also want it to be of a form which instructs the prover to automatically discharge the antecedents from the hypotheses.
Q2 Write this rule, adding it in the prover and loading to make sure it compiles.

Using a rewrite rule

These are similar in format but with an equivalence in the conclusion.
Q3 What is wrong with the following rule?

```
binhyp(s <: t) &
t = {}
=>
s == {}
```

Give an example to illustrate the problem and show how to correct the rule.

A word of warning

Allowing rules to be added like this is a double edged sword. It's useful in that you are not forced to derive everything from first principles but can enter facts which are obviously true and use them in your proofs. But there's nothing to stop you from adding complete rubbish which simply allows you to derive whatever you want. Beware!
You can now check out the machines you write using the proof environment. Obviously it won't always be easy, but you can help make it easier by thinking about proof from the start. Smaller, simply expressed units of specification are easier to verify than monolithic epics. Think about proof as you write the

machine - is your invariant suitable and well-expressed? Will the operations really preserve it? Can you sort out some problems by investigating with Prob?