CS412 Lab session 3

Using functions and generating proof obligations

In this lab session you'll have the opportunity to develop more abstract machines, this time making use of functions and dealing with nondeterminism too. Also, in later lectures we'll be going on to talk about proof so, in advance of that, this lab starts to consider how the toolkit deals with proof obligations.

Writing a *Projects* machine

Start up the B tool and create a new project for this week's session. Create a new machine called *Projects* to meet the following description.
The *Projects* machine keeps track of which project is allocated to which student and also handles the mark allocated. The sets STUDENT and PROJECT should be declared. There will be a maximum mark, *maxmark*, and marks may be awarded in the range $0\,..\,maxmark$ - this set should be referred to as MARK.
The following variables should be declared:
students - records those students so far registered for a project
projects - records which project students choose (domain is *students*)
marks - records students' marks
Your invariant should capture any important relationships between these. A student can sign up for only one project, but a project may be attempted by more than one student. Initially, all components are empty.
Provide the following operations:

name	**inputs**	**description**	**outputs**
signup	*ss pp*	Student signs up for a project	
assignproject	*ss*	Student without project is assigned one	*pp*
queryproject	*ss*	Query which project a student is doing	*pp*
entermark	*ss mm*	Add (or alter) a student's mark	
querymark	*ss*	Find out what mark a student has	*mm*

In what way is the *assignproject* operation different from operations defined in previous labs? Save and type check your machine to ensure you have no syntactic or type errors.

Mixing different structures

Write a specification to meet the following requirements. They're fairly general, so there may be many different interpretations. Choose a workable way of representing the requirements and supporting the operations stated.
A learning management system hosts a number of different modules. Each module is managed by an individual member of staff. A student can obtain

different qualifications in the system, with each qualification requiring successful completion of a specified set of modules. Some modules may contribute to a number of different qualifications. The modules and qualifications offered in the system are fixed (that is, they will not change). Both staff and students will register in the system. Once registered, a student may enrol to study a variety of modules. Upon successful completion of a module's assessment they will be noted as having passed that module. No student can be both registered as studying a module which they have now passed. The following are examples of operations to provide:

1. record a student starting to study a module;
2. record a student passing a module;
3. output modules a student is studying;
4. query whether a student has passed a particular qualification;
5. output what module a student still needs to pass for completion of a particular qualification;
6. change the manager of a module (input the new one);
7. change the manager of a module (get the system to choose a new one);
8. change the manager of a module (get the system to choose a member of staff who is currently one of the least lightly loaded).

Think of other operations that might be useful: try writing one deterministic one and one nondeterministic one.

Generating proof obligations

Having written specifications, you'll now be aware of some of the mistakes that can be made, preconditions which are needed to ensure the operation will preserve the invariant etc.
There is another version of the projects machine called *ProjectsLab*3 available on the module website. Copy this to the directory you are working in and have a look through. Note any differences to your specification and consider whether either, both or neither is right. (Don't change anything yet.) Working with the *ProjectsLab*3 machine, analyse to show it's syntactically and type correct.
We've talked in general about being able to prove properties about a specification, but have not yet seen how to do this. Next week we'll be looking at exactly what we can and should verify. The B tool can generate standard proof obligations which represent general "healthiness checks" for the specification. This includes things like making sure your initialisation establishes the invariant, and making sure each operation maintains the invariant. We'll look in detail later at what these are and how they are derived. For now, as preparation, let's get the tool to generate the proof obligations and see what can be proved automatically.

If you haven't already, it's probably easiest to toggle the "view " option (just above the components window in the main Atelier B view) to "Classical view". In this same view, the blue Po button at the top generates the proof obligations for a machine according to some basic consistency conditions. Make sure the ProjectsLab3 line is selected in the component window and that it has been successfully typechecked. Then click the Po button. How many proof conditions are generated for this small specification?

Viewing proof obligations

Having generated the obligations, click on the rightmost blue button which is labelled `Ip` for "Interactive proof". For now we'll just be looking at the obligations generated. A new Prover window will appear. In the list of POs click on the initialisation line to expand it and you should see a list of proof obligations, PO1 upwards. Double click on PO1 to see what it is. In the proof window you should see a description of what it's checking and the formal statement of this. Check you understand what this means (ask for help if not). How has is been generated from the specification? Have a look at some of the other obligations that have been generated. Although the exact details won't be familiar yet, you can see the sorts of things it's doing. The display of the POs just shows you the goal you're aiming at in each case. The specification itself will provide various assumptions you can work with to help in the proof which we'll talk more about another time.

Proving proof obligations

Most of these are trivial and we hope that the toolkit can prove them automatically for us. To try to prove the obligations, close down the interactive proof window, go back to the main Atelier B window and make sure that the `ProjectsLab3` line is selected in the component window. Now click on the blue `Fo` button at the top. This tries to prove the obligations. It may take a second or two, but you should find that all the obligations are successfully proved (it should go from all unproved to all proved). Congratulations - you've just performed your first verification of machine consistency! If some proof obligations are left over then you could try clicking the blue `F1` button which tries a bit harder. When that fails, some intervention will be required.
Now let's see what would happen if our specification were not correct. Edit the `ProjectsLab3` machine to comment out the final conjunct of the precondition for the `signup` operation. The operation cannot now guarantee to preserve the invariant because it could be trying to add another project for a student who already has one. Go to the main Atelier B window, type check the machine and generate the proof obligations. Try automatic proof at both levels - one PO will still be left. Use the blue `Ip` button to locate and view the offending proof obligation which will be marked in the Prover window with a red cross.
If a proof obligation fails to prove automatically it can be for a number of reasons. In this case, we know we've made a mistake in an operation, but in general we may need to look at the failed condition or delve into how the proof

fails to get more insight. If the toolkit can't prove an obligation it could be because:

- it's not true! The machine is inconsistent and needs to be changed;
- the toolkit isn't clever enough to work out what sequence of rules to apply and needs a bit of help;
- the toolkit hasn't got enough rules to deal with the situation and we must add some.

Later, we'll carry on with this and start interacting with the prover to gain information. For now, when you write a machine you can at least generate POs and use automatic proof plus viewing of unproved POs to guide your development.

Now go back to the original project machine that you wrote. Generate the proof obligations and try to prove them. If they don't prove, look at the problematic ones. Do they reveal any problems in your specification? If so, correct as necessary and try again.

Because we're looking at proof today we dived straight in to the prover to look for mistakes. But don't forget that animation is a good way to check for (un)expected behaviour and the facility to show a trace leading to the invariant being broken shows you've got a problem. Use this to guide you as well.