# CS412/932 Lab session 2

## Using relations in a specification

In this lab session you'll have the opportunity to look at and develop more abstract machines, this time making use of relations.

## Writing the *docsys* machine

The document machine, *docsys*, specifies a system for keeping track of secure documents and the users who view them. To view a document a person must be registered in the system and they must have permission to view the document. The first part of the machine is given below. *PID* and *DOC* are deferred sets which represent identifiers for people and documents respectively. The variables are:

- *reg* the set of currently registered users;
- *allowed* a relation (set of pairs) representing the permissions (that is, which documents users are allowed to see);
- *viewing* a relation (set of pairs) representing who is currently viewing which documents.

Your task will be to continue writing this machine, adding a variety of operations.

```
MACHINE   docsys
SETS      PID; DOC
VARIABLES reg, allowed, viewing
INVARIANT reg <: PID & allowed:PID <-> DOC &  viewing:PID <-> DOC & ...
END
```

Start up Atelier B (type *startAB* in a command line) and create a new machine called *docsys* (decide how you want to organise things: maybe you want a new workspace for lab2?). Use the definitions above to get started with writing the new abstract machine. As you add to the machine, save and type check as you go along to catch any basic errors.

### Complete the invariant

What other requirements would it be appropriate to place in the invariant? Write these for your abstract machine.

**Initialisation**

Add an appropriate initialisation.

**Operations**

Provide the following operations:

- to output the documents currently being viewed;
- to output the people allowed to access a particular document;
- to register a person in the system;
- person pp starts to view document dd;
- person pp stops viewing document dd;
- person pp stops viewing all documents;
- to remove from a person the right to access a particular document;
- to allow all registered people the permission to access a particular document;
- to give everyone in a group of people the same access based on the principle that if anyone had access to a document initially, the operation allows everyone to have access to the document.

*This may require a bit more thought. Perhaps come back to it later.* Suppose in addition that a type of "Chinese Wall" policy is in operation. That is: a set of documents is identified with the restriction that no user can view more than one of these particular documents at any one time. Make alterations where necessary within the specification to reflect this. New operations will be required to add or remove documents from the restricted set. Old operations will need to be updated.

## A bit more about animation

Start up ProB to animate this specification and click the "Enabled operation" to initialise. As you may have noticed from last week, a green OK sign should be visible in the state window. This is telling you that in the current state the invariant is true. If not, you need to revisit the specification, work out why and correct the problem.
Your operation allowing someone to start viewing a document should include a precondition to check that the access is allowed. Modify your specification so that it no longer makes this check. Reload the spec (File $\rightarrow$ Save and Reopen) and animate. After initialisation, register a user and then choose the operation for that user to start viewing a document. You will see the green OK turn to a red KO. This is because the operation has led to the invariant becoming false. In this case, a user is accessing a document they do not have permission to see. If the invariant can become false you need to go back and modify your specification. Inspecting the state and the history of events will help locate the problem.

Playing around with the specification may help you find errors - but only if you happen upon one of the "bad" sequences of operations. You can get ProB to search and see if it can find one for you. Click Animate → Reset to start afresh. Now click Verify → Model Check. Check the box for "Find invariant violations" and then click on Model Check. Y ou should get a message saying it's found an invariant violation. When you click Ok it will display a history of how this happened. You can use this information to guide you in correcting the spec. Remember, you can use the blue arrows at the top of the "Enabled operations" box to go back to the start of the history and then step forward through the trace to see how the problem arises. Once again, you need to be clear what this is achieving. If it finds a violation you know you have a problem. But what if it doesn't find a "bad" sequence? Because it can only search a small, finite state space you can only infer that no problems exist **within the limited space searched**. There could be problems which only show up in larger models. If you go back to your original spec and reopen it you'll notice in the proB state window that it's supposing PID and DOC each have 2 members only, and also setting the integer range checked as -1...8. You can increase the size of the search to see if problems arise in a larger scope. For instance, here you could alter the number of elements in DOC and PID by adding a DEFINITIONS section to your spec and saying, eg, $scope\_DOC == 6$; $scope\_PID == 8$. Another thing you may sometimes want to do is alter internal limits such as MAXINT. You can do this in the DEFINITIONS section too with, for example, $SET\_PREF\_MAXINT == 15$
Increasing sizes means that more values are checked (and checking becomes slower) but it's only made the state space a bit bigger (and a lot longer to search) and there could still be errors which only show up in a bigger state space still. So, how can you know a property is true for all cases? You'll have to verify it with a formal proof!
Also, remember that ProB is only a tool to help investigate the specification. A definition such as that to set the size of DOC and PID is simply helping with your investigation - it's not something you're aiming to implement.

## Defining an abstract machine from requirements

From the requirements below, decide on an appropriate representation of the static part (that is, not including the operations) of a suitable abstract machine. Use the invariant to state useful required properties for the specification. Then define some appropriate operations for the machine. Ensure that they maintain the invariant. That is, if the invariant is true before the operation starts, it should be true when the operation ends.
In a university, students may register to take modules. Each module offered is managed by one or more members of staff. No student can be registered for more than 6 modules. No member of staff will be involved with more than 3 modules. An abstract machine is required to keep track of this. Students cannot be registered for a module unless it is actually being offered! Operations might include: a student registering for a module; a student deregistering for a module; allocating an additional manager to a module offered; output all students taking a particular module; output all staff who are currently associated with the maximum of 3 modules; introducing a new module; withdrawing an old module etc.