CS412 Lab session 9

Implementation machines

This week's lab looks at the development of a program from the abstract machine stage to an implementation machine. The most important aspects are writing implementation machines and seeing how an interface can be incorporated in the process. We'll also mention the issues involved with generating code from implementation machines. As usual, start Atelier B.

1 Writing B0 implementations

This is the approach where you use the concrete subset of the B language.

1.1 A first B0 example

Start a new project for this development (eg: *testimpl*). (It's probably best to start each separate development as a new project so you can generate code for it all together but with nothing extraneous.) Copy the bounds machine below and then provide a B0 implementation for it. Using an IF instead of the PREs (so that it skips otherwise) would be ok. You can check the implementation is B0 by selecting that component in the main B window and then clicking Component → B0 check. A B0 should appear to the RHS of the line for the *bounds_i* machine. A machine couldn't be translated as B0 until it passes this check. If there were separate components, all would have to be B0.

1.2 Another B0 example: the registration machine

The *CS*400*Reg* abstract machine below records students who register for CS400. It refers to another abstract machines, *Courses*, which is also given. Copy these machines, check out what they do and analyse them. Suppose we wish to implement *CS*400*Reg*. To do this we need to implement every machine in the development.

Courses

Write an implementation for *Courses* choosing a suitable concrete value for the constant.

*CS*400*Reg*

Suppose we implement the set using a B0 array representation with students added from the least position upwards as registrations occur. Use a new variable to record how much of the array is actually relevant. Suppose also that we simply represent students using numbers. Write an implementation that does this. For now, you can assume any preconditions are met.
Write an implementation which just outputs the spaces left. Check that your array approach works as a B0 implementation.

2 Library machines

To use the library machine approach to implementations you will need to add the B library project into your workspace. Download the `tarLIBRARY-comenc.arc` file from the CS412 website (linked next to the labnotes for this week). In Atelier B, choose 'Restore Project' from the Workspace menu. Select the library archive file as the archive path, a project name of 'LIBRARY' and then import the library. A new LIBRARY project should appear in your workspace containing the library components.
To us a library machine in your current project you can add it by right clicking on (in this case) "testimpl" in the workspaces panel of the main AtelierB window, then click on Add Component and select the machine you want from .../LIBRARY/bdp/src.
(NB: it should be possible to add the whole library as a dependency to the project that will use it but you may find it doesn't work. Right click on "testimpl" in the workspaces panel of the main AtelierB window and select the properties option from there. At the bottom of the dialogue box click the LIBRARY option from the Libraries section and add it to the project by clicking the left pointing arrow. Just below that, click on Add and choose the LIBRARY bdp file to add this to the definitions directory. Click OK to confirm. Now, back in the "testimpl" project space you should be able to see LIBRARY appearing in the Libraries section. And the LIBRARY bdp in the Definitions section. If this doesn't work for you - just adding the

component as above is fine. Or even just copy and paste the machine you need!)
Once you've obtained the library machines you'll be able to use them to directly implement sets, functions, etc. The *BASIC_IO* machine also allows you to write an interface. If you want to see what operations a library machine has, just open its specification up in Atelier B.

3 Implementation using imported library machines

This is the second way of approaching an implementation machine. We "farm out" responsibility for our state components to library machines that can deal with it for us. Unfortunately, the translation to code for these isn't working properly so we'll just concentrate on getting to the implementation machine level.

3.1 Another implementation for the registration machine

In this part we'll consider a different way of implementing the CS400Reg machine. Previously, we made a B0 implementation and ensured our operations were all concrete and directly implementable. This time let's suppose we wish to use one of the library machines to handle the state.

Copy the CS400Reg machine and implement it again, this time using a set library machine. The operations available for this machine were discussed in lectures. You can find full information on all library machines in the reusable components reference manual available on the CS412 website. Or just open the relevant library machine in the toolkit to inspect it.

Assume that the preconditions are met and implement the successful cases of the operations.

3.2 Using a different library

The numlist machine below provides some basic functionality for a list of numbers. Use the L_SEQUENCE library machine to write an implementation. You can find the details in the reusable components manual or by opening this machine in Atelier B.

4 Writing an interface

To allow us to write an interface as part of the B development we can introduce a top level abstract machine with just one operation. At the specification level, the operation won't actually do anything. We can refine it later to specify the required user interface. Suppose we're thinking of generating C code. Let's call the top level machine *Top* and the operation *main*. Write this machine in your development (see lectures if you're unsure).
A Top implementation won't actually be showing anything to the user until we tell it to. If we import the *BASIC_IO* library that will allow us to write to and read from the screen and we can use the operations it provides to specify the user interface. For example, going back to the *bounds* machine, if we just wanted to obtain the result of the length function and show the user we could edit the *Top* implementation so that it imports *BASIC_IO* and the *main* function uses screen output to show you the value. Eg:

```
main = VAR xx IN
             xx <-- length;
             STRING_WRITE("The length is ");
             INT_WRITE(xx);
             STRING_WRITE("\n")
          END
```

Try some of the following (note that if you read in a CHAR and want to test it, you have to use the ASCII value). Don't worry about implementing all of the suggested interfaces - as long as you've got the hang of it!

- An interface which allows the user to select whether they want the *length* operation or the *findone* operation and then shows them the output.
- An interface which obtains a new value that the user wants to set the minimum to and does this.
- An interface which allows the user the choice between all of the *bounds* operations and gets them to enter an input if appropriate.

5 Translation to code

Code generation is the final step in a process of development from abstract specification, through refinement and implementation machines, into generated code. Atelier B is supposed to provide a C translator and there is documentation in one of the help manuals. It comes with the warning that it is experimental, and there are currently issues which limit its use. It seems that the process to generate all the necessary files to compile the code doesn't work from within Atelier B. The actual production of code would be nice, but it's not something assessed in this module. The description below outlines a rather annoying sequence of steps that will allow you to work with the code generator in stand-alone mode within the department. This could be viewed as a proof of concept if you are interested to see the process.

To keep things as "clean" as possible for this trial it's a good idea to open a new project (eg *testimpl*1) and just copy the bounds machine and implementation there. To start with let's just try something simple to get the code generation working. Eg: add the usual Top machine and add a very simple Top implementation:

```
main = VAR xx IN xx <-- length END
```

Go to the directory where your *testimpl*1 project is kept. The machines and implementations for bounds and Top should be visible there.
The code generation program is called *b2c* but we'll need to run it using its full pathname which is: `/package/Atelier_B/bbin/b2c` so you might want to set up an alias. We can translate each machine separately (unless you can find an option which does everything under the top one - in which case let me know!). Here we have just two machines so we need to run:

```
b2c -D bdp/*.db -p C9X Top_i.imp
b2c -D bdp/*.db -p C9X bounds_i.imp
```

Here, `-D` points to the project database (there's only one .db file in the bdp directory), `-p C9X` gives it the C standard settings, and the final parameter is the implementation machine to translate. This will generate .c and .h files for these two machines. Now we need to direct the translator to our main function and generate a makefile:

```
b2c -D bdp/*.db -P -p C9X -m Top_i
```

You should get messages telling you it is generating CMakeLists.txt and the main file. If all is well, you can follow the usual CMake process, perhaps creating a subdirectory to build the project, eg:

```
  mkdir build
  cd build
  cmake ..
  make -j
```

You should now have an executable file, *testimpl*1 (or whatever you called

your project) which you can run using *./testimpl*1 and you should find that nothing happens! This is a good sign because if you've not got any errors, all is going well! All the main function did was write a value to a local variable, so we don't actually see anything. To do something about this we'll need to use the operations provided by the *BASIC_IO* mentioned in lectures.
Now add *BASIC_IO* to your *testimpl*1 project and edit the *Top* implementation again so that you have something that uses screen output such as:

```
main = VAR xx IN
           xx <-- length;
           STRING_WRITE("The length is ");
           INT_WRITE(xx);
           STRING_WRITE("\n")
        END
```

Now, in the main project directory you can generate code using bdp as above. Unfortunately, this is where it gets a bit messy. The *BASIC_IO.c* and *BASIC_IO.h* files are not in your *../testimpl*1 directory. You'll find them in the *../LIBRARY/lang/c* directory and will need to copy them across. Check that:
BASIC_IO.c has the line: `void BASIC_IO__INITIALISATION(void) {}`
BASIC_IO.h has the line: `void BASIC_IO__INITIALISATION(void);`
and if not, add them.
Finally, edit your *CMakeLists.txt* file to include *BASIC_IO.c* in the SOURCES and *BASIC_IO.h* in the HEADERS.
Now go back to your build directory:

```
cmake ..
make -j
```

Finally you should get some output! Having achieved that, everything is in place to generate more adventurous interfaces.

```
MACHINE
    bounds

VARIABLES
    minb, maxb

INVARIANT
    minb:NAT1 & maxb:NAT1 & minb < maxb

INITIALISATION
    ANY xx, yy WHERE xx:NAT1 & yy:NAT1 & xx < yy
    THEN minb:= xx || maxb:=yy
    END

OPERATIONS
   oo <-- querymin = oo := minb;

   oo <-- querymax = oo := maxb;

  resetmin(bb) =
      PRE  bb:NAT1 & bb < maxb
      THEN minb := bb
      END;

  resetmax(bb) =
      PRE  bb:NAT1 & bb > minb
      THEN maxb := bb
      END;

  nn <-- length = nn := maxb - (minb - 1) ;

  nn <-- findone =
      ANY xx WHERE xx:NAT & xx >= minb & xx <= maxb
      THEN nn := xx
      END

END
```

```
MACHINE          Courses

SETS             COURSE = {cs400,cs401,cs402,cs403}

CONSTANTS        maxcls   /*The maximum size of any class */

PROPERTIES       maxcls:NAT1 & maxcls < MAXINT

END


MACHINE          CS400Reg

SEES             Courses

SETS             STUDENT; REPLY = {yes,no}

VARIABLES        classreg

INVARIANT        classreg : FIN(STUDENT) & card(classreg) <= maxcls

INITIALISATION   classreg := {}

OPERATIONS

  register(ss) =
     PRE     ss:STUDENT & ss /: classreg & card(classreg) < maxcls
     THEN    classreg := classreg \/ {ss}
     END;

  oo <-- regquery(ss) =
     PRE     ss:STUDENT
     THEN    IF      ss : classreg
             THEN    oo := yes
             ELSE    oo:= no
             END
     END;
  oo <-- spaces = oo := maxcls - card(classreg)

END
```

```
MACHINE      numlist

CONSTANTS    maxlen

PROPERTIES   maxlen : NAT1

VARIABLES    nums

INVARIANT    nums:seq(NAT) & size(nums) <= maxlen

INITIALISATION
  nums := []

OPERATIONS

  addnum(nn) =
      PRE  nn:NAT & size(nums) < maxlen
      THEN nums := nums <- nn
      END;
  /* Add nn to list */

  removefirst =
      IF   size(nums) > 0
      THEN nums := tail(nums)
      END;
  /* Remove first number in list */

  removeall = nums := [];
  /* Remove all the numbers */

  removefirstn(nn) =
      PRE  nn:NAT1
      THEN nums := nums \|/ nn
      END;
  /* Remove first nn numbers */

  removefromposition(ii) =
      PRE  ii:NAT & ii:dom(nums)
      THEN nums := (nums /|\ (ii-1)) ^ (nums \|/ ii)
      END;
  /* Number at position ii is removed */

   ii <-- isinnums(nn) =
      PRE  nn : NAT
      THEN IF   nn : ran(nums)
           THEN ii:= min(dom(nums|>{nn}))
           ELSE ii:= 0
           END
      END;
  /* Outputs the position pp is at (0 if not there) */

  nn <-- sumnums = nn := SIGMA(ii).(ii:1..size(nums)|nums(ii));
  /* Outputs the sum of the numbers in the list */

  nn <-- howmany(ii) =
       PRE  ii:NAT
       THEN nn := card(nums |> {ii})
       END
  /* How many times does ii occur in the list */

END
```