

CSC2021 Software Engineering

Overture Tutorial 1

2019

1. Aims

This practical exercise introduces the Overture toolset for creating, editing and querying a VDM-SL model.

The aims are:

- To introduce Overture, show how to import a project and examine it
- To use the Overture console to interactively execute a model
- To use the debugger to examine a model
- To use a Run() operation to execute a model and write output to a file

VDM-SL Concepts explored: **state, invariant, set, mapping**

There are two tutorials to complete. Completing them, and submitting evidence to Ness, gains 2 marks per tutorial (4 marks total = 4%). A larger piece of modelling coursework will follow, which will be worth 16%.

Submission: This tutorial is to be completed, and evidence submitted to Ness, by **15:00 Friday 15 Feb.**

By working through all the steps in the tutorial, you will generate an output file which is to be submitted to Ness. See the submission instructions at the end of the tutorial.

2. Introduction

Overture is an open-source project to develop industrial strength support for the formal modelling languages VDM-SL and VDM++. The Overture tool is an Eclipse-based Integrated Development Environment.

Assumptions

The exercise assumes you are using a Windows PC in the USB level 3 teaching labs.

Overture for Windows, linux and MacOS can also be downloaded for your own use from

<http://overturetool.org/>

3. Getting started

Overture can be found in the **Simulation, Visualisation and Modelling** folder in the Windows Start Menu – or by simply pressing the Start key and typing **Overture**.

The first time Overture is run, it will ask where to store the *workspace* directory (folder). This will be used to store Overture project files.

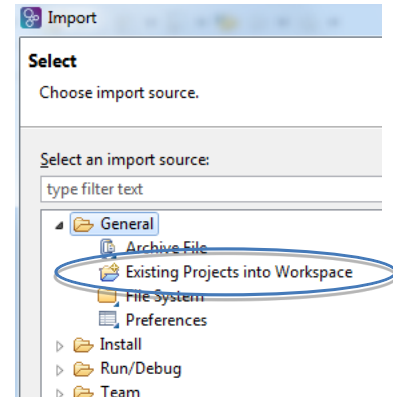
**** You should change the default workspace at this point. Do not use an existing Eclipse workspace: choose a new folder, to avoid possible conflicts with your Eclipse workspace. ****

Overture should now open with the default VDM perspective. This includes a number of views which will be used as you work on projects: on the left, the **VDM Explorer** shows open projects and specification; in the middle, the **editor** shows open specifications; on the right, the **Outline** lists the datatypes, operations, functions in the current specification; at the bottom is a **Console**/debug log area.

The rest of this exercise uses the material presented in the lectures on modelling, using a VDM model of the Chemical Plant Alarm System. In the following instructions, you'll use Overture to import the Alarm project, and use the toolset to explore the project files.

4. Importing a Project

- Download the project file `AlarmSL.zip` from Blackboard, and save it in a suitable folder on your H: drive. NB you do **not** need to unzip the file.
- Now import the project into Overture: from the **File** menu choose **Import**, then open **General** and select *Existing Projects into workspace*, as shown to the right.
- Click **Next**.



- On the next screen, choose *Select archive file*, and Browse to find the `AlarmSL.zip` file that you saved before.

- Ensure that the AlarmState project is selected, and click **Finish**.

- The VDM Explorer pane now lists the open project: click the arrow beside **AlarmState** to expand and show the files which are contained in the project. Double-click on `alarmSL.vdmsl` to open this file in the editor. The Outline pane shows defined state, types, operations, functions and values. Take a moment to examine the model, particularly the persistent state, datatypes and operations.

Lines 4-6 define the specification module (AlarmState) and connection to two other modules, IO and VDMUtil. Definitions for these modules are contained in the folder **lib**: they contain library operations and functions used for writing to an output file. You don't need to explore these files.

Some lines will be highlighted with yellow warnings "definition not used": these can be ignored for now.

Tip: you can show line numbers in the editor window: from the **Window** menu, choose **Preferences, General, Editors, Text Editors**, then tick *Show line numbers*

5. Executing the model: Debug configuration

Overture allows you to execute parts of a VDM model using the initialised state and variables defined in the **values** section of the model. You will find the values towards the end of the specification file. Note that each value defines a variable name, type and value, for example:

```
p1:Period = mk_token("Monday day");
```

The model can be executed using a *Debug Configuration*, either by running commands in the console, or by giving an entry point in the specification for an operation to execute. We'll look at the first of these options here.

Under the **Run** menu, select **Debug Configurations**:

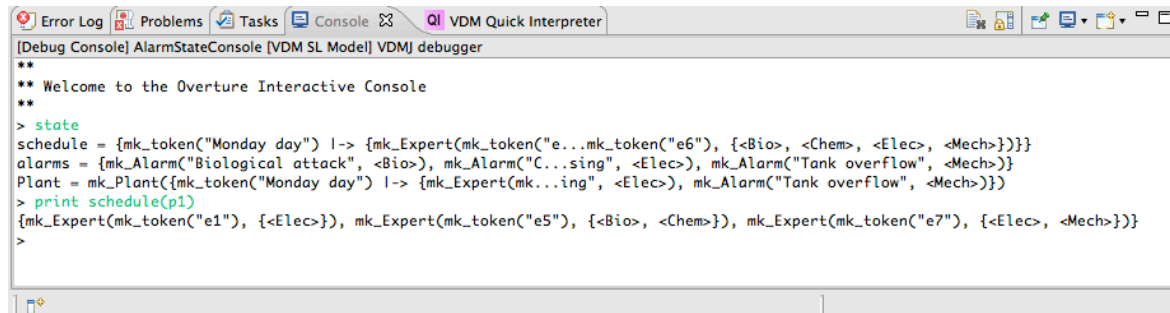
- In the dialog box, double-click **VDM-SL Model** from the left-hand pane to create a new configuration.
- Give the new configuration a Name (e.g. AlarmConsole) and Browse to select the **AlarmState** project.
- Select Launch Mode as **Console**
- Click on **Apply**, then **Debug** to start the Interactive Console with this model.
- You may get a Firewall warning the first time you run the console: this can be ignored.

The Console session will be opened with a prompt "Welcome to the Overture Interactive Console".

- In the Debug Console type the following commands:

```
state
print schedule(p1)
```

The output from these commands should be as shown below:



```
[Debug Console] AlarmStateConsole [VDM SL Model] VDM debugger
**
** Welcome to the Overture Interactive Console
**
> state
schedule = {mk_token("Monday day") l-> {mk_Expert(mk_token("e...mk_token("e6"), {<Bio>, <Chem>, <Elec>, <Mech>}})}
alarms = {mk_Alarm("Biological attack", <Bio>), mk_Alarm("C...sing", <Elec>), mk_Alarm("Tank overflow", <Mech>)}
Plant = mk_Plant({mk_token("Monday day") l-> {mk_Expert(mk...ing", <Elec>), mk_Alarm("Tank overflow", <Mech>)}}
> print schedule(p1)
{mk_Expert(mk_token("e1"), {<Elec>}), mk_Expert(mk_token("e5"), {<Bio>, <Chem>}), mk_Expert(mk_token("e7"), {<Elec>, <Mech>})}
>
```

From this you can see that the persistent state consists of the variables `schedule` and `alarms`, which together make up the `Plant`. The schedule provides a lookup (mapping) from time periods to experts, so the schedule for period `p1` gives a set of experts with qualifications as shown. The model's initial state defines a set of 4 alarms (identified by `a1`, `a2`, `a3`, `a4`) and a mapping of 4 time periods to different sets of experts.

The operations on the state can be executed using the `print` command. For example, try the following commands:

<i>To find out how many experts are on duty at a particular time:</i>	<code>print NumberOfExperts(p1)</code>
<i>To find out when a given expert is on duty:</i>	<code>print ExpertIsOnDuty(e1)</code>
<i>To find a qualified, available expert for a given alarm and time:</i>	<code>print ExpertToPage(a1,p1)</code>

6. Modifying the State

During a console session, state variables can be modified by operations. For this model, the operations `AddExpert`, `RemoveExpert` modify the experts on duty at a given time period.

The schedule has a restriction, which states that for all types of alarm, there must always be an expert on duty who is qualified to deal with that alarm. This is recorded as an **invariant** in the state definition.

Try the following commands:

To add expert e4 to time period p1, and confirm the change in state:

```
print AddExpert(p1,e4)
print ExpertIsOnDuty(e4)
```

To remove expert e4 from time period p1 and confirm:

```
print RemoveExpert(p1,e4)
print ExpertIsOnDuty(e4)
```

To remove expert e4 from time period p2 and confirm:

```
print RemoveExpert(p2,e4)
print ExpertIsOnDuty(e4)
```

Note that expert e4 was not in that time period, so there is no change.

To remove expert e5 from time period p1:

```
print RemoveExpert(p1,e5)
```

This last example should lead to a run-time error, because the removal of expert `e5` means that the state invariant is no longer satisfied. Overture will offer to open the Debug perspective (click Yes to do this when prompted), giving you the chance to investigate state variables by following the steps in the next section.

7. Debug Perspective

The Debug Perspective allows you to explore the current state of the model. Two extra panes have opened. On the top left, a **Debug** pane shows the current execution stack. On the right, an **inspection** pane allows you to drill down into the values of variables currently in scope.

Since you entered the Debug Perspective after entering the command **print RemoveExpert(p1,e5)**, you should see the two variables **ex** and **peri** which are in scope in the **RemoveExpert** operation. Click on the variable name to show its value beneath the variable list.

Expanding the **Instance Variables** will show the state (**Plant**) and its constituents **alarms** and **schedule**. Expanding **schedule** will show a list of maplets: drilling down to examine Maplet1 will show that this is the value that has caused the error: the set of Experts in the range of the mapping does not include an Expert with qualifications **<Bio>** or **<Chem>**. This breaks the invariant, which states that there must always be a qualified expert available for all types of alarm.

Once you've checked this, terminate the debug session (click the red 'stop' button or select **Terminate** from the **Run** menu) and return to the VDM perspective (by clicking on the Overture symbol at the top right).

8. Executing the model: entry point

The result of a console session is not stored after it has been terminated. A repeatable execution of the model is achieved by writing an operation – normally called **Run** – which is then used as an entry point for the model. This is similar to writing a **main** method in java. This operation can consist of a sequence of operations which modify the state, and may also generate output which can be written to a file.

The alarmSL model has a Run operation which executes an AddExpert and RemoveExpert call and appends the results to a file **VDM-out.txt** using standard output operations **fprint**, **fprintln**, **fprintState** and **toString** – these are defined at the end of the file.

Setting a debug configuration for the Run operation is similar to setting a console session:

Under the **Run** menu, select **Debug Configurations**:

- In the dialog box, double-click **VDM-SL Model** from the left-hand pane to create a new configuration.
- Give the new configuration a Name (e.g. AlarmRun) and Browse to select the **AlarmState** project.
- Select Launch Mode as **Entry Point**
- In the next section click the Search button to expand the **AlarmState** module and find the operation **Run()**
- Click on **Apply**, then **Debug** to run the operation.

You should see a new file, **VDM-out.txt** appear in the explorer window. Double click on this to open it and check the output produced by the operation.

9. Exercise submission

This completes the tutorial. To gain a mark for completing it, you need to find the file **VDM-out.txt** which was generated in the previous step. Submit this to the Ness exercise "**Tutorial 1**" by the deadline. 2 marks will be gained for a VDM-out.txt file containing the output from the Run command generated in the previous step.

Steve Riddle, Jan 2019

Parts adapted from Larsen et al, "Tutorial to Overture/VDM-SL"