Overture Technical Report Series
No. TR-007

October 2021

Overture VSCode Extension Support: User Guide

Version 0.0.2

by

Tom Montout, Hugo Daniel Macedo, Jonas K. Rask, Peter Gorm Larsen
Aarhus University, Department of Engineering
Finlandsgade 22, DK-8000 Århus C, Denmark

Nick Battle
Fujitsu UK
Lovelace Road, Bracknell,
Berkshire. RG12 8SN, UK

Document history

Month	Year	Version	Version of Overture
August	2021	0.0.1	1.2.0
October	2021	0.0.2	1.2.1

Contents

ABSTRACT

This document is the user manual for the Overture VSCode Extension supporting the Vienna Development Method (VDM). It serves as a reference for anybody wishing to make use of the tool with one of the VDM dialects (VDM-SL, VDM++ or VDM-RT). The different dialects are controlled by a VDM language Board that evaluates possible Requests for Modifications. Overture tool support is built on top of the VSCode platform. The objective of the Overture initiative is to create and support an open source platform that can be used for both experimentation with new VDM dialects, as well as new features for analysing VDM models in different ways. The tool is entirely open source, so anybody can join the development team and influence future developments. The goal is to ensure that stable versions of the tool suite can be used for large scale industrial applications of VDM technology.

Chapter 1

Introduction

The Vienna Development Method (VDM) is one of the longest established model-oriented formal methods for the development of computer-based systems and software [Bjørner&78, Jones90, Fitzgerald&08a]. It consists of a group of mathematically well-founded languages for expressing system models during early design stages, before expensive implementation commitments are made. The construction and analysis of a model using VDM helps to identify areas of incompleteness or ambiguity in informal system specifications, and provides some level of confidence that a valid implementation will have key properties, especially those of safety or security. VDM has a strong record of industrial application, in many cases has been used by practitioners who were not specialists in the underlying formalism or logic [Larsen&96, Clement&99, Kurita&09]. Experience with the method suggests that the effort spend on formal modelling and analysis can be recovered in reduced rework costs arising from design errors.

VDM models can be expressed in a Specification Language (VDM-SL) which supports the description of data and functionality [ISOVDM96, Fitzgerald&98, Fitzgerald&09]. Data are defined by means of types built using constructors that define structured data and collections such as sets, sequences and mappings from basic values such as Booleans and natural numbers. These types are very abstract, allowing you to add any relevant constraints using data type invariants. Functionality is defined in terms of operations over these data types. Operations can be defined implicitly by preconditions and postconditions that characterise their behavior, or explicitly by means of specific algorithms. An extension of VDM-SL, called VDM++, supports object-oriented structuring of models and permits direct modelling of concurrency [Fitzgerald&05]. A further extension to VDM++, called VDM Real Time (VDM-RT[1]), includes support for discrete time models [Mukherjee&00, Verhoef&06]. The VDM-RT dialect is also used inside the Crescendo tool[2] supporting collaborative modelling and co-simulation [Fitzgerald&14]. All three VDM dialects are supported by Overture.

Since VDM modelling languages have a formal mathematical semantics, a wide range of analyses can be performed on models, both to check internal consistency and to confirm that models have emergent properties. Analyses may be performed by inspection, static analysis, testing or

[1] Formerly called VDM In a Constrained Environment (VICE).
[2] See `http://crescendotool.org/`.

mathematical proof. To assist in this process, Overture offers tool support for building models in collaboration with other modelling tools, to execute and test models and to carry out different forms of static analysis [Larsen&13]. It can be seen as an open source version of the closed (but now freely available) tool called VDMTools [Elmstrøm&94, Larsen01, Fitzgerald&08b].

This guide explains how to use the Overture VDM VSCode Extension for developing models for different VDM dialects. It starts with an explanation of how to get hold of the software in Chapter 2. This is followed in Chapter 3 with an introduction to the VSCode workspace terminology. Chapter 4 explains how projects are managed in the Overture VSCode Extension. Chapter 5 covers the features for creating and editing VDM models. This is followed in Chapter 6 with an explanation of the interpretation and debugging capabilities in Overture. Chapter 7 illustrates how test coverage information can be gathered when models are interpreted. Chapter 8 shows how models with test coverage information can be written as LaTeX and automatically converted to PDF format. Then, Chapters 9 illustrates the generation and the visualisation of a dependency graph. Chapters 10 to 12 cover various VDM specific features: Chapter 10 explains the notion of proof obligations and their support in Overture; Chapter 11 explains combinatorial testing and the automation support for that; Chapter 12 explains how it is possible automatically to generate executable code in programming languages such as Java for a subset of VDM models; Chapter 13 demonstrates how Java can be used in combination with VDM models. Appendixes A to F give complete lists of possible errors, warnings and proof obligation categories. Appendix G provides an overview of the VDM++/VDM-RT to UML mapping rules. Appendix H provides details about how to represent VDM values in order to combine Java with VDM. Finally, there is an index of significant terms used in this manual.

Chapter 2

Getting Hold of the Software

VSCode: This is a source code editor, available for Windows, macOS and Linux. One easy way to download this IDE is to go on their website and choose one version depending on your operating system. If you go to :

`https://code.visualstudio.com/Download`

you should be able to install it and after opening you should see the welcome screen (see Figure 2.1).

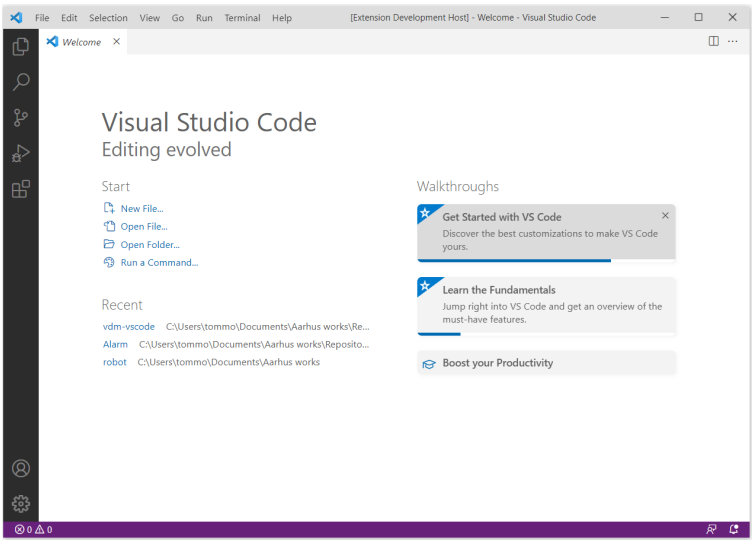

Figure 2.1: The VSCode Welcome Screen

Install the VDM VSCode Extension

After downloading the VSCode you should install the VDM VSCode extension available in the marketplace.

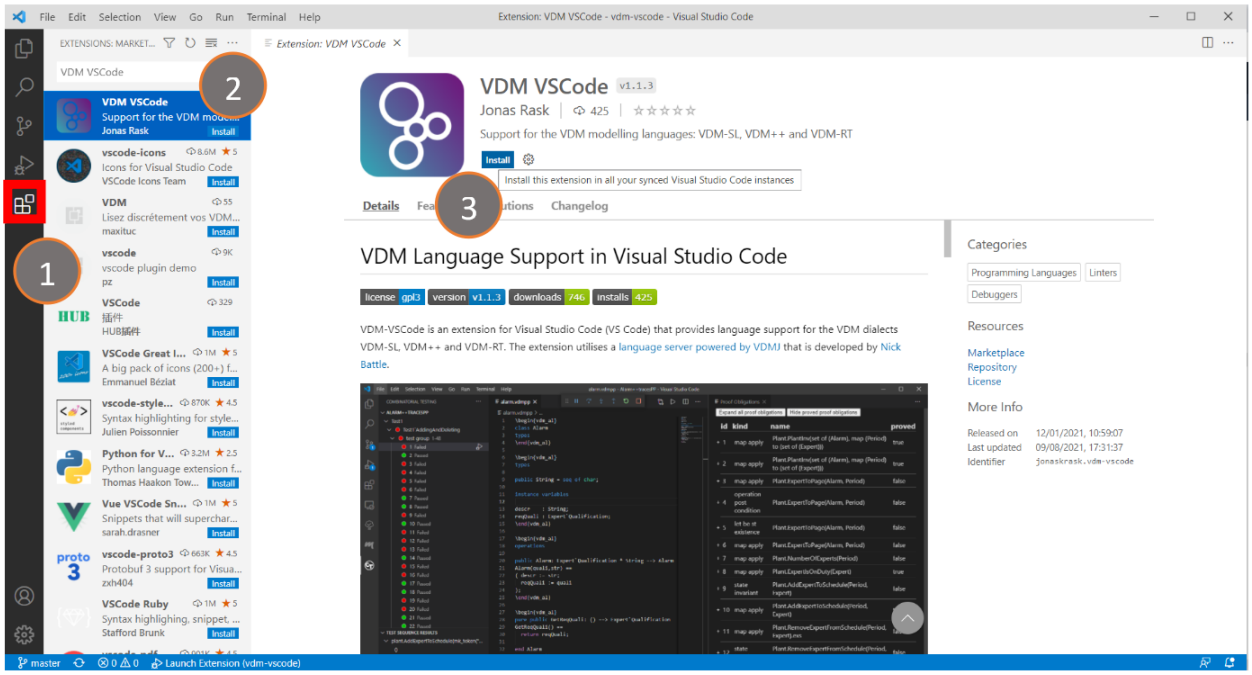

Figure 2.2: Adding VDM VSCode Extension

1. Click on the button ‘Extensions’(Ctrl+Shift+X) to have access to different extensions available in VSCode (red square)

2. Write ’VDM VSCode’ in the search bar and choose the first extension ‘VDM VSCode (Support for the VDM modelling languages)’

3. Install the extension by clicking on the button ‘Install’

After installing you can check changes and more information in the Extension view (see Figure 2.3).

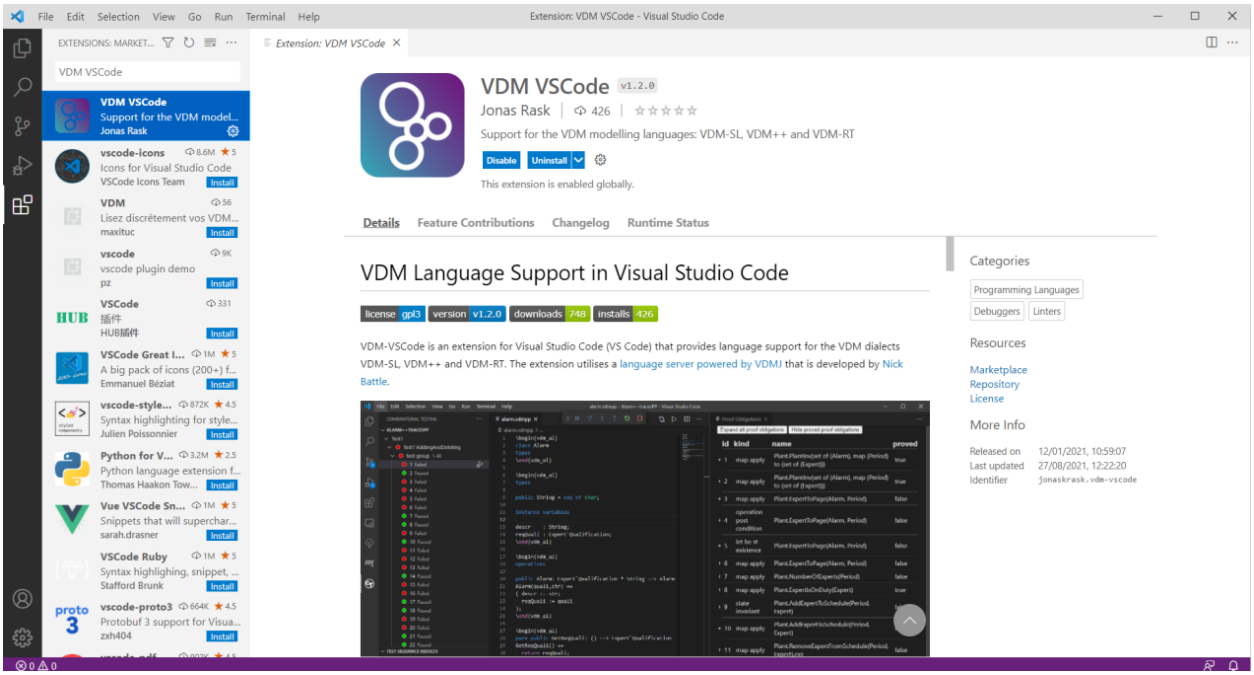

Figure 2.3: End of the Installation of the VDM VSCode Extension

Note that in order to be able to execute Overture you need to have Java Runtime Environment (minimum version 1.11) installed on your computer.

Finally, in order to make use of the test coverage feature described in Section 7 it is necessary to have the text processing system called LaTeX and its `pdflatex` feature. This can for example be obtained from:

- Windows: `http://miktex.org`
- Mac: `http://tug.org/mactex/`
- Linux: Most distributions offer LaTeX packages

Chapter 3

Using the VDM VSCode Extension

3.1 Understanding VSCode Terminology

VSCode is a free source code editor that uses a folder or workspace system for interacting with a project and a document system for handling the source code files in the project. If you are familiar with one VSCode product, for instance the built-in support for JavaScript and TypeScript you will generally find it easy to start using other products that use the same editor.

The VSCode workspace consists of several panels known as *areas*. A particular arrangement of areas is called a *user interface*, for example Figure 3.1 shows the standard VDM user interface. This consists of a set of areas for managing Overture projects and viewing and editing files in a project.

The *VDM Explorer* lets you create, select, and delete Overture projects and navigate between the files in these projects, as well as adding new files to existing projects. The first point shows the *Editor area* where you can see all the open files which can be edited. By default, the top of the Editor area is a set of tabs (one open file corresponds to one tab). The *Explorer* (left part) is composed of three separate parts : the Open Editors, the Project view, also named the project structure (point 2) and the Outline View (point 3). The first section shows the editor groups and the files contained in each in a tree format. The second one contains a view of the files and folders that constitute the current project. Finally, the last section of the Explorer display the contents of the current file in a hierarchical way. To finish, the *Proof Obligations view* (point 4) provides a list of the proof obligations for a specification, where expanding an element displays the actual proof obligation.

Dialect editors are sensitive to the keywords used in each particular dialect, and simplify the task of working on the specification.

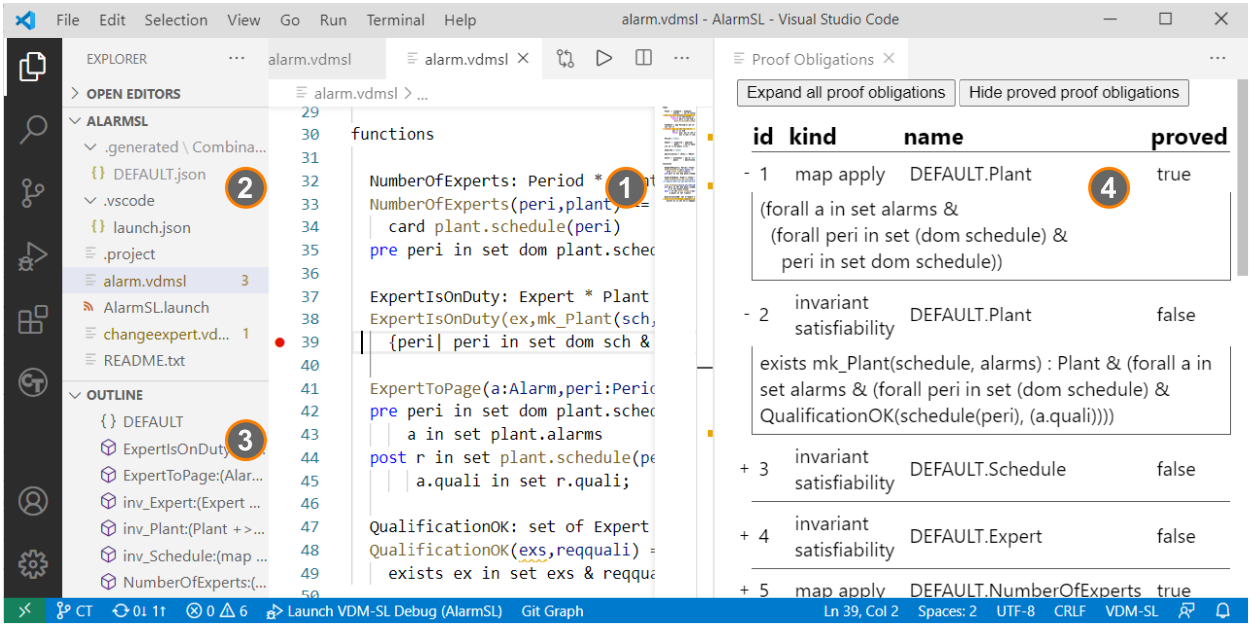

Figure 3.1: The VDM VSCode Extension User Interface

The *Outline View*, presents an outline of the file selected in the editor. The outline shows all VDM definitions, such as state definitions, values, types, functions and operations. The orange icon at the top represents the component itself (called 'CTDataProvider' in our example). The purple cube is used for the methods, and the blue blob inside square brackets for the variables. Moreover, the monkey brench illustrates an operation, a trace or can also be an class variable. Finally, the two orange windows define an enumeration. Figure 3.2 illustrates the different outline icons. At the top of the Outline View there are buttons to filter what is displayed and to sort the icons.

The *Status Bar* view at the bottom of Figure 3.1 displays information messages about the projects you are working on, such as warnings and syntax or type checking errors.

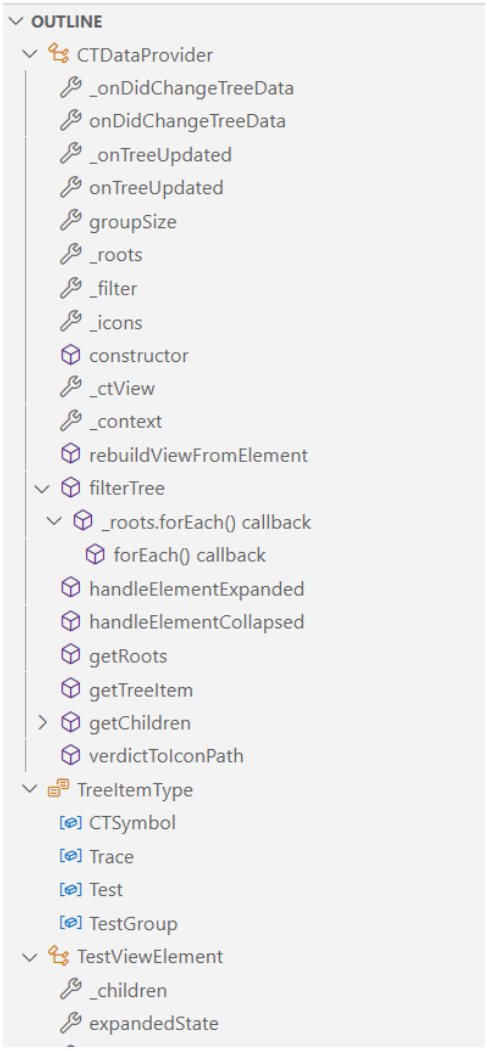

Figure 3.2: Outline View

Chapter 4

Managing Overture Projects

4.1 Importing VDM Projects

It is possible to automatically import a large collection of existing examples.

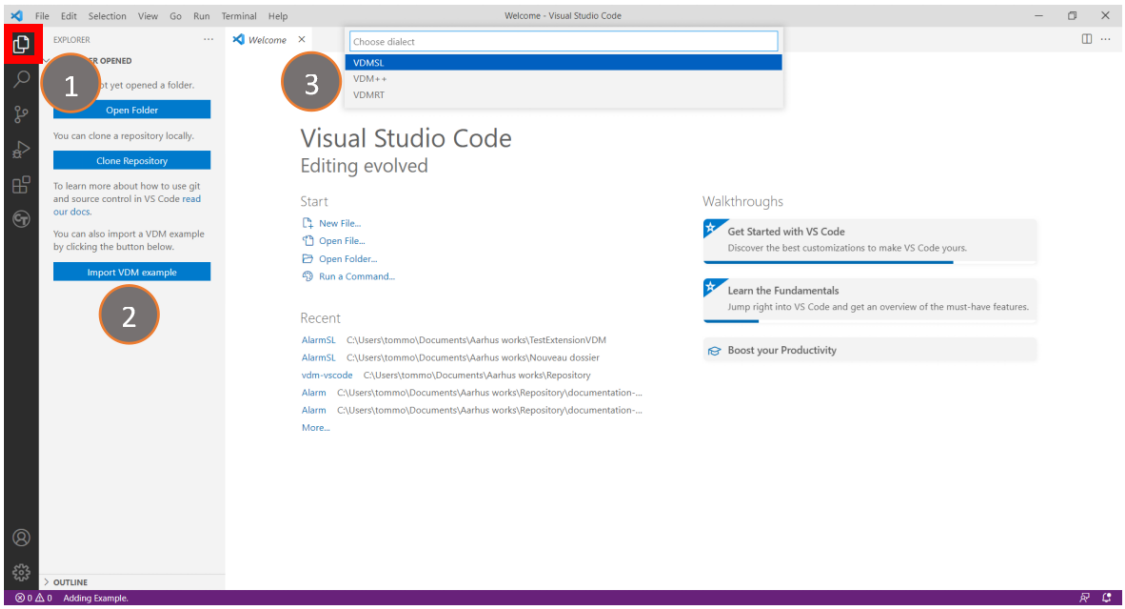

Figure 4.1: Import VDM Example

1. Click on the button 'Explorer'(Ctrl+Shift+E) to have access to the Explorer view (red square)
2. Click on the button 'Import VDM example'
3. Choose the dialect and then choose an example in the menu. Finally, select the folder where you want to save the example.

4.2 Adding Standard Libraries

It is possible to add existing standard libraries. This can be done by right-clicking on the Explorer view where the library is to be added and then selecting *Add VDM Library*. That will make a new window as shown in Figure 4.2. Here the different standard libraries provide different standard functionalities.

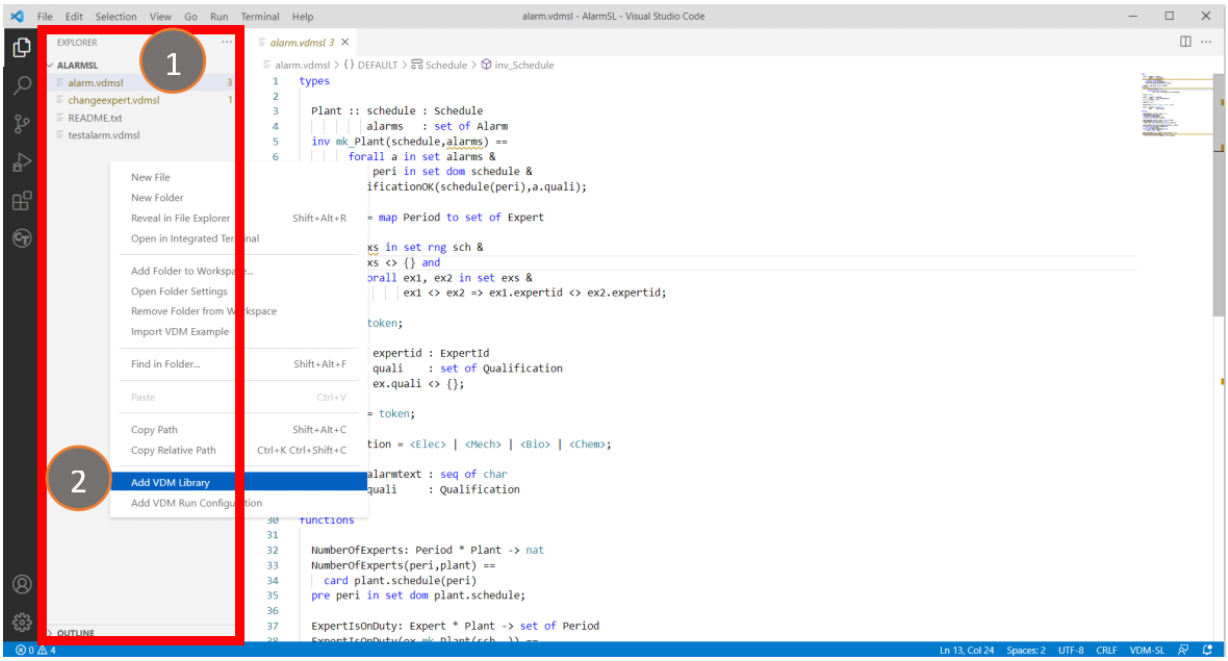

Figure 4.2: Adding VDM Libraries

1. Right click in the Explorer view (red square) to display the different available options
2. Click on ‘Add VDM Library’ (highlighted in blue)

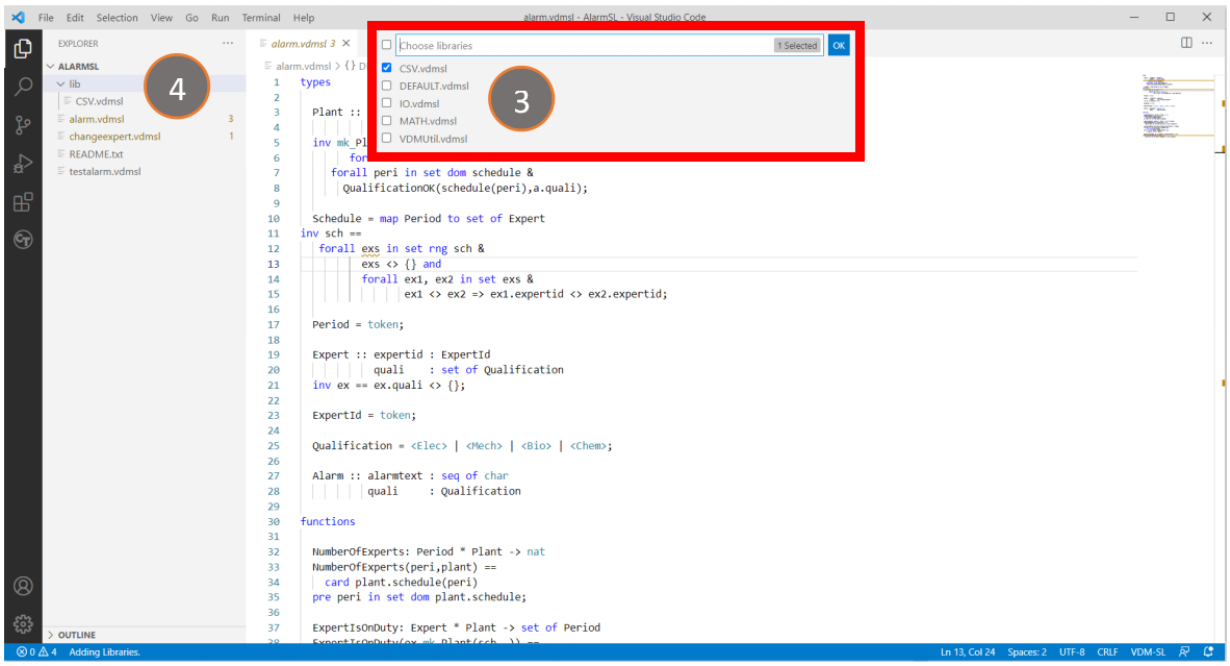

Figure 4.3: Choosing VDM Libraries

3. Click on the library that you want to add to your project (possibility of adding several libraries)

4. Now you can see all the libraries added in the folder 'lib'.

If you inspect the added files, you will notice that the body of many of these functions/operations are declared as "**`is not yet specified`**" but the actual functionality for all of these are hard-coded into Overture so the user can get access to this when the respective standard libraries are included. This can be summarised as:

IO: This library provides functionality for input and output from/to files and the standard console.

Math: This library provides functionality for standard mathematical functions such as sine and cosine.

Util: This library provides functionality for converting different kind of VDM values mainly to and from files and strings.

CSV: This library is an extension of the IO library which provides additional functionality for saving and reading VDM values to/from comma separate format used by excel spreadsheets.

VDM-Unit: This library provides functionality for unit testing of VDM models similar to the well-known JUnit library.

All these libraries except `VDM-Unit` are available for all VDM dialects also when a flat VDM-SL specification is used. `VDM-Unit` use object-orientation and thus it cannot be used with VDM-SL.

4.3 Setting Project Options

There are various VDM specific settings for an Overture project. You can change these by going in *File*, selecting *Preferences* and finally clicking on *Settings*, See Figure 4.4. The options that can be set for each VDM project are:

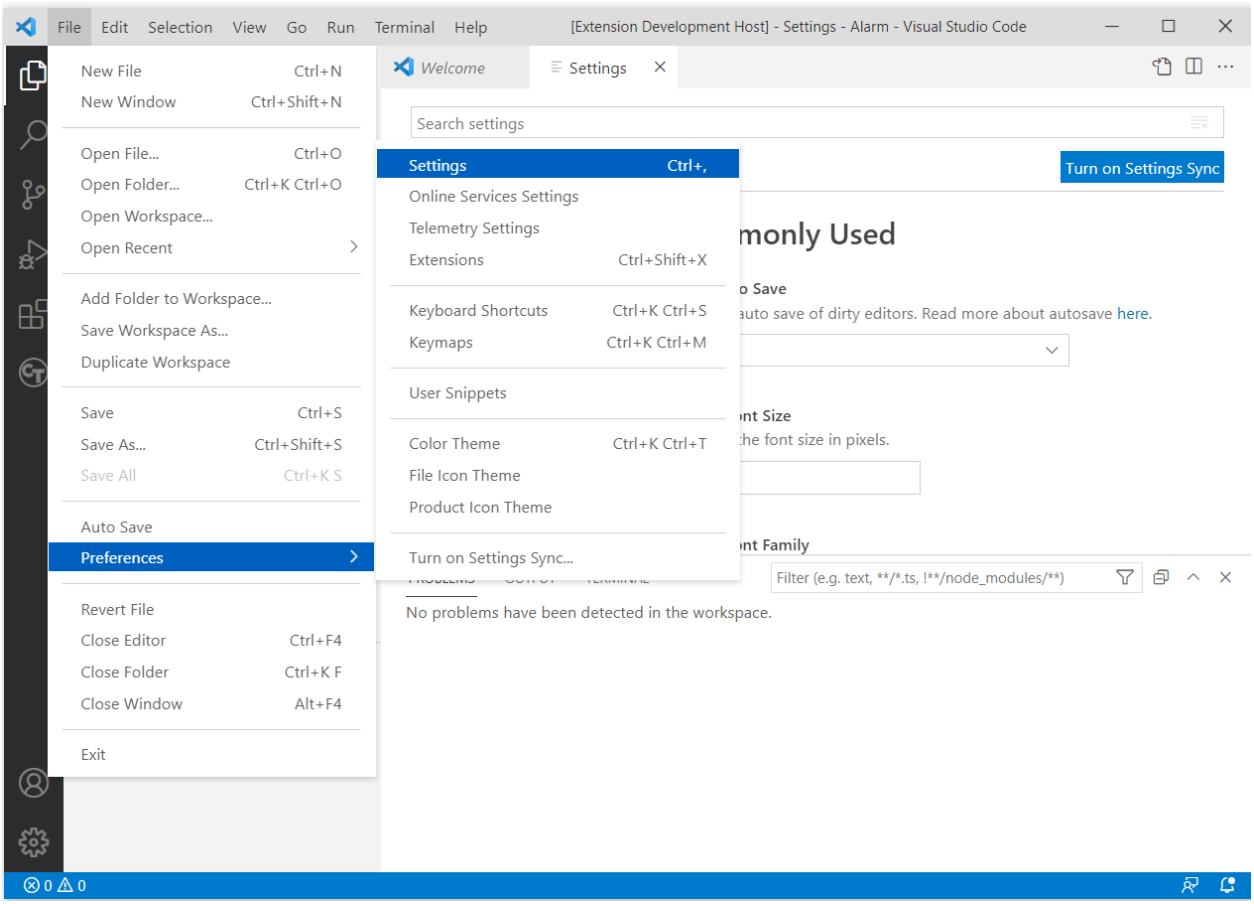

Figure 4.4: Path to go to Settings

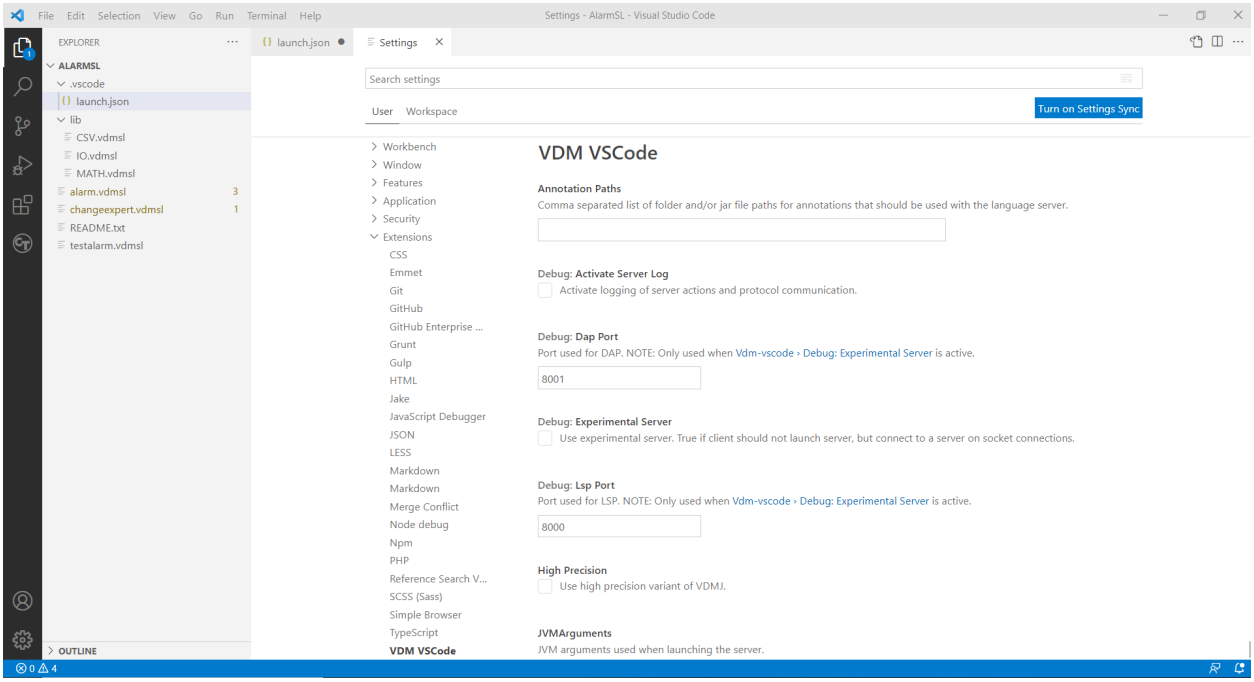

Figure 4.5: VDM VSCode Settings

Chapter 5

Editing VDM Models

5.1 VDM Dialect Editors

VDM model files are always ment to be changed in the dialect Editor area. Syntax checking is carried out continuously as source files are changed (even before the files are saved). Whenever files are saved, assuming there are no syntax errors, a full type check of the *entire* VDM model is performed. Problems and warnings will be listed in the Problems window as well as being highlighted directly in the Editor area where the problems have been identified.

5.2 Using Templates

VSCode templates can be particularly useful when you are new to writing VDM models. If you press *CTRL+space* after typing the first few characters of a template name, Overture will offer a proposal. For example, if you type ”fun” followed by *CTRL+space*, the IDE will propose the use of an implicit or explicit function template as shown in Figure 5.1. The IDE includes several templates: cases, quantifications, functions (explicit/implicit), operations (explicit/implicit) and many more. The use of templates makes it much easier for users to create models, even if they are not deeply familiar with the VDM syntax.

```
41  \begin{vdm_al}
42  pure public NumberOfExperts: Period ==> nat
43  NumberOfExperts(p) ==
44    return card schedule(p)
45  pre p in set dom schedule;
46
47  pure public ExpertIsOnDuty: Expert ==> set of Period
48  ExpertIsOnDuty(ex) ==
49    return {p | p in set dom schedule &
50                ex in set schedule(p)};
51
52  functionName : parameterTypes ==> resultType
53  functionName (parameterTypes) == expression
54  pre preCondition
55  post postCondition;
```

Figure 5.1: Explicit function template

Chapter 6

Interpretation and Debugging in Overture

This chapter describes how to run and debug a model using the Overture VSCode Extension.

6.1 Run and Debug Launch Configurations

To execute or debug a VDM model, you must first create a launch configuration. To do this, follow the instructions below.

1. Right click in the EXPLORER (red square) to display the different available options
2. Click on 'Add VDM Run Configuration' (highlighted in blue)
3. Write your input entry point module ('DEFAULT' in our case)
4. Write your input entry point function or operation ('Run(e1)' in our case with 'Run' the function and 'e1' the argument)

Finally, you can see in Figure 6.4 the newly created configuration file.

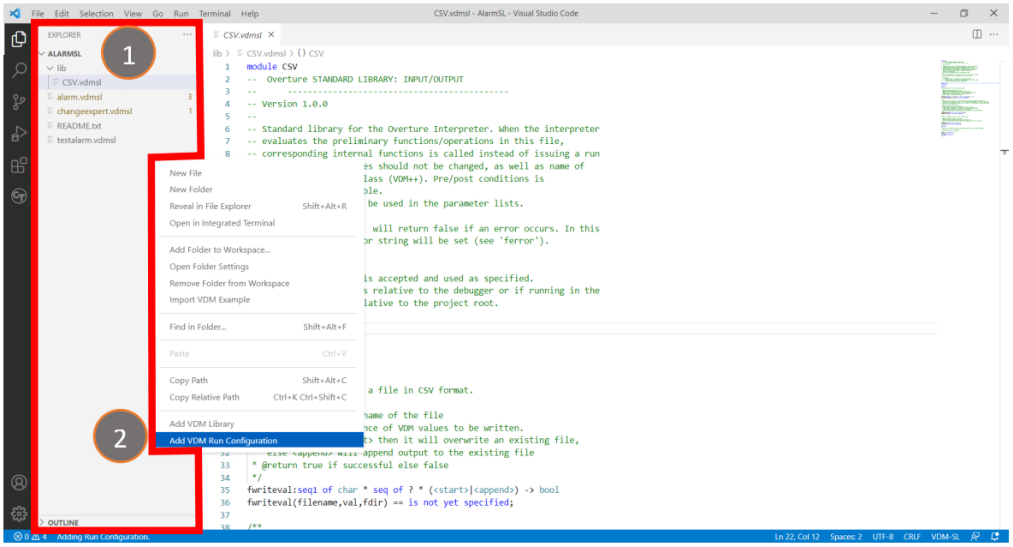

Figure 6.1: Adding VDM Run Configuration

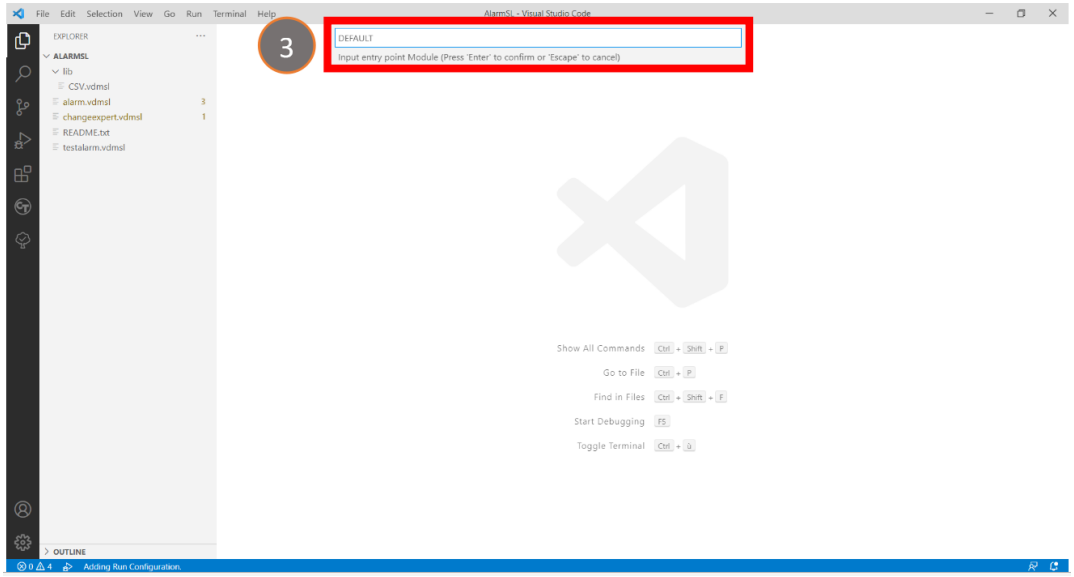

Figure 6.2: Input Entry Point Module

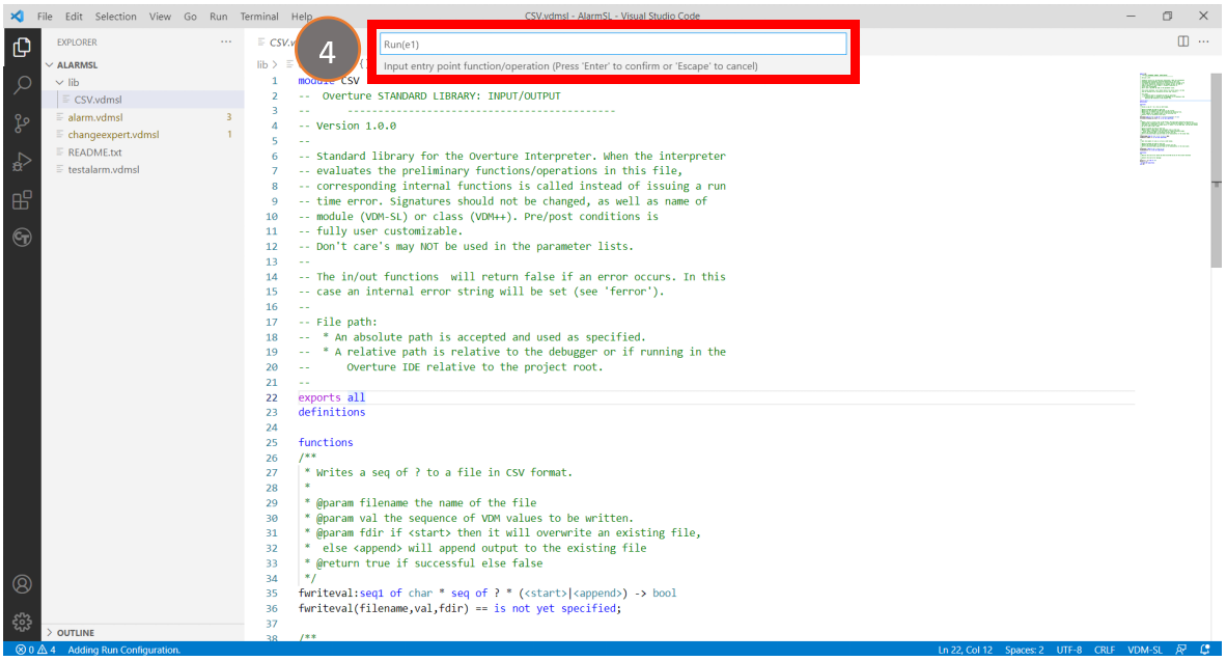

Figure 6.3: Input Entry Point Function/Operation

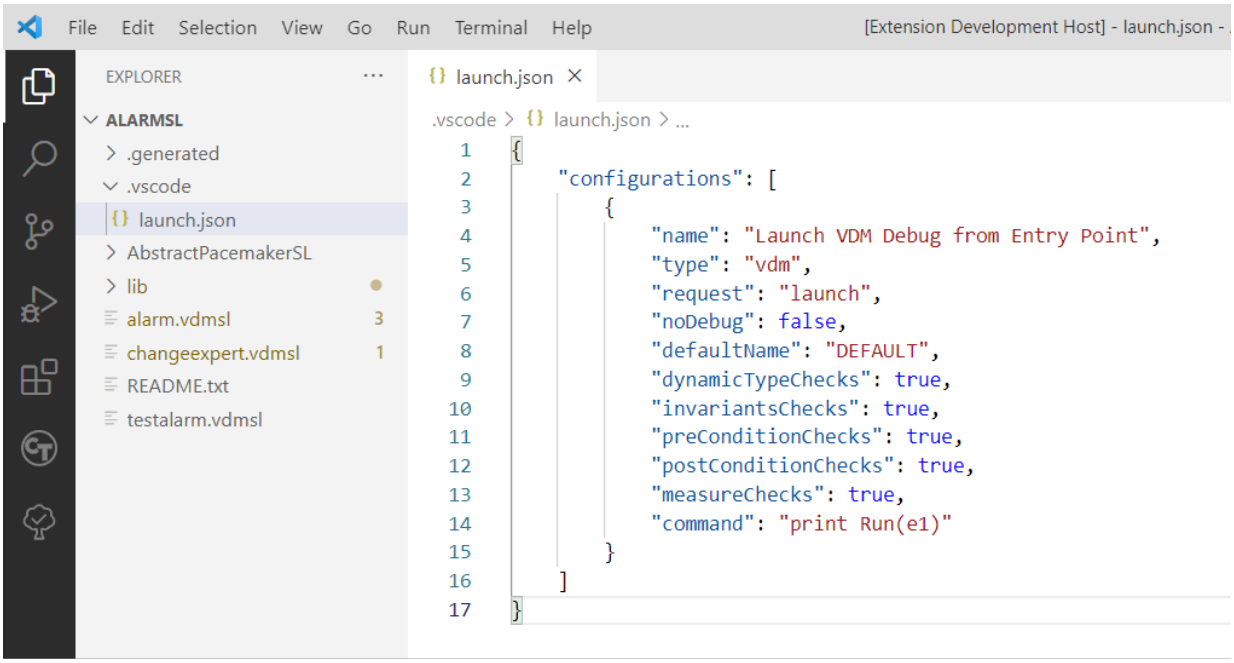

Figure 6.4: New launch.json file with the configuration

On the other hand, you can also create a launch.json file by default without entering module or function, see Figure 6.5.

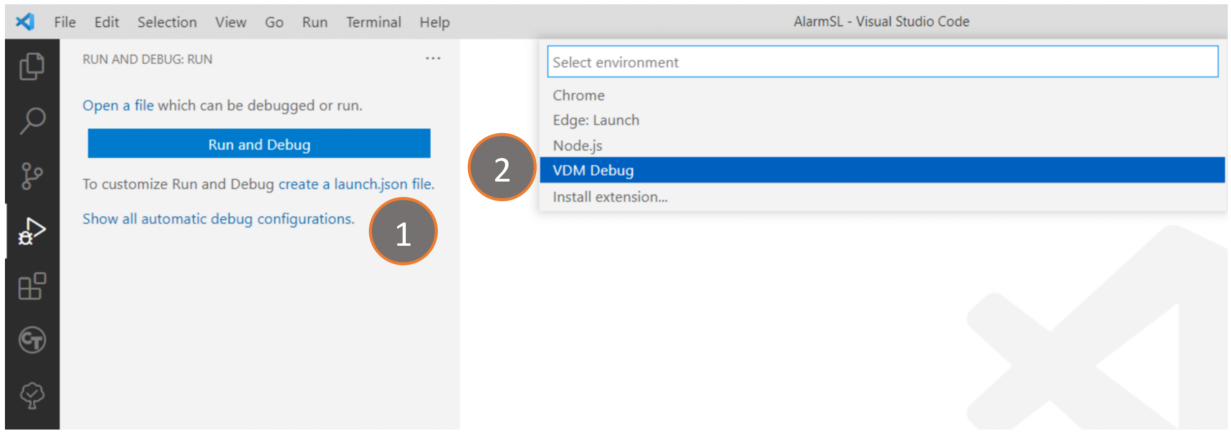

Figure 6.5: Creating a launch.json file by default

1. Click on 'create a launch.json file'

2. Then a window appear, thus select an environment ('VDM Debug' if you want to configure a VDM project)

Whenever a launch configuration is started up it is also possible to decide upon which additional run-time checks to carry out. By default all possible run-time checks are swiched on (true) but if desired (some of) these can be swiched off (false) by modifying the different options by false in the launch.json file (see Figure 6.5). The different run-time checks that can be performed are:

- **Dynamic type checks:** This is an option for the interpreter (default true) to continuously type check values during interpretation of a VDM model. It is possible to switch off the check here [1].

- **Invariant checks:** This is an option for the interpreter (default on) to continuously check both state and type invariants. It is possible to switch off this check here, but note that option requires dynamic type checking also to be switched off.

- **Pre condition checks:** This is an option for the interpreter (default on) to continuously check pre-conditions for all functions and operations during interpretation of a VDM model. It is possible to switch off this check here.

[1] However a consequence of doing that is that you may get internal Java errors (null pointer or class cast exceptions typically) rather than nice clean VDM type errors about mismatched types.

- **Post condition checks:** This is an option for the interpreter (default on) to continuously check post-conditions for all functions and operations during interpretation of a VDM model. It is possible to switch off this check here.

- **Measure checks:** This is an option for the interpreter (default on) to continuously check recursive functions, for which a measure function has been defined. It is possible to switch off this check here [2].

In addition, you have the possibility to add other configurations. You just have to go in the Run view and follow the instructions below.

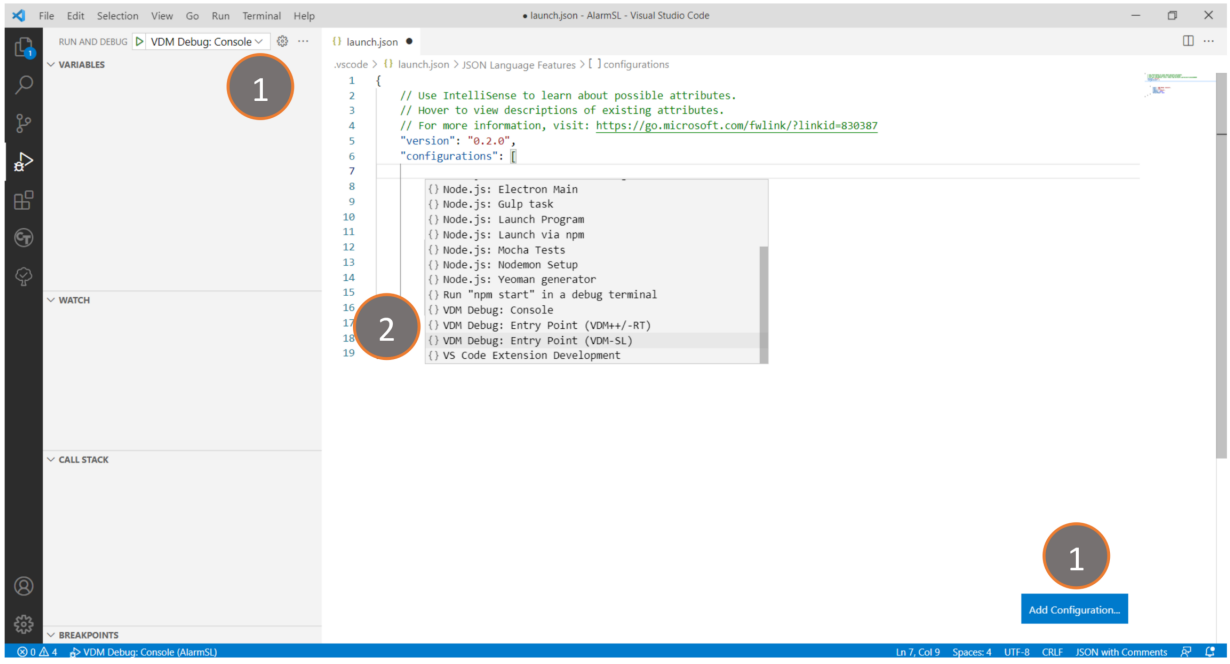

Figure 6.6: Adding an other VDM Run Configuration

1. Click on the button 'Add Configuration...' (bottom right corner), or on the arrow facing downwards and press 'Add Configuration...'

2. Select the configuration that you want to add to your project ('VDM Debug: Entry Point (VDM-SL)' in our case, highlighted in grey)

[2]Note that this feature may not work correctly with the presence of mutually recursive function definitions.

6.2 The Run View

The Run view contains all the views commonly needed for debugging in VDM. Breakpoints can easily be set in the model by clicking in the left margin of the Editor view at the chosen line. Then a red dot will appear next to the number of the selected line. When the debugger reaches the location of a breakpoint and stops, you can inspect the values of different identifiers and step through the VDM model line by line.

The Run view is illustrated in Figure 6.7.

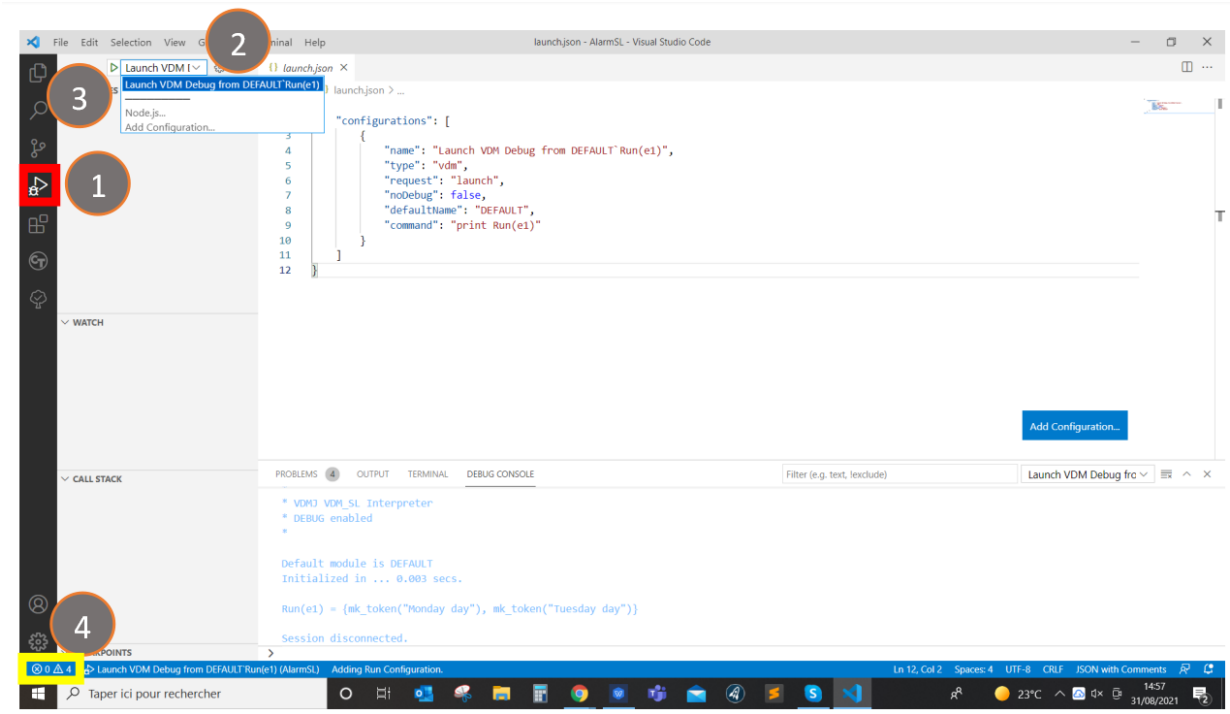

Figure 6.7: Run and Debug a .json File

1. Click on the button ‘Run and Debug’(Ctrl+Shift+D) to have access to the Run view (red square)
2. Select with the arrow facing downwards which configuration you want to launch
3. Finally, click on the ▷ (‘Start Debugging’ (F5)) to run and debug the configuration selected
4. If you want to display the debug console, click in the yellow square and click on ‘Debug Console’

Chapter 7

Collecting Test Coverage Information

When a VDM model is being interpreted, it is possible to automatically collect test coverage information. Test coverage measurements help you to see how well a given test suite exercises your VDM model.

Before generating coverage, there is one prerequisite, you have to run and debug a VDM model. You can see more details about running and debugging a VDM model in the Figure 6.7.

In order to enable the collection of test coverage data, right click on a file in the Explorer and select the *Generate coverage* option (See Figure 7.1). After launching this coverage, a new file with a `.covtbl` extension will be created. This file is written into a project subfolder named `.generated/coverage <date and time>`.

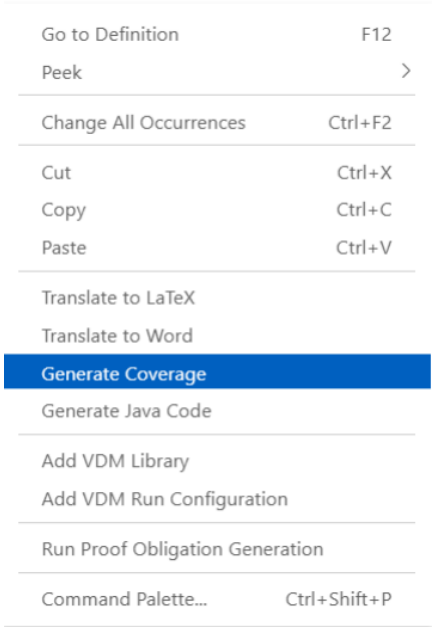

Figure 7.1: Path to Generate Coverage

Thus, all highlighted text represents the code used for running the test. We can see that in our case the operation 'NumberOfExperts' is not used in our test because this part of the code is not highlighted in blue (see Figure 7.2).

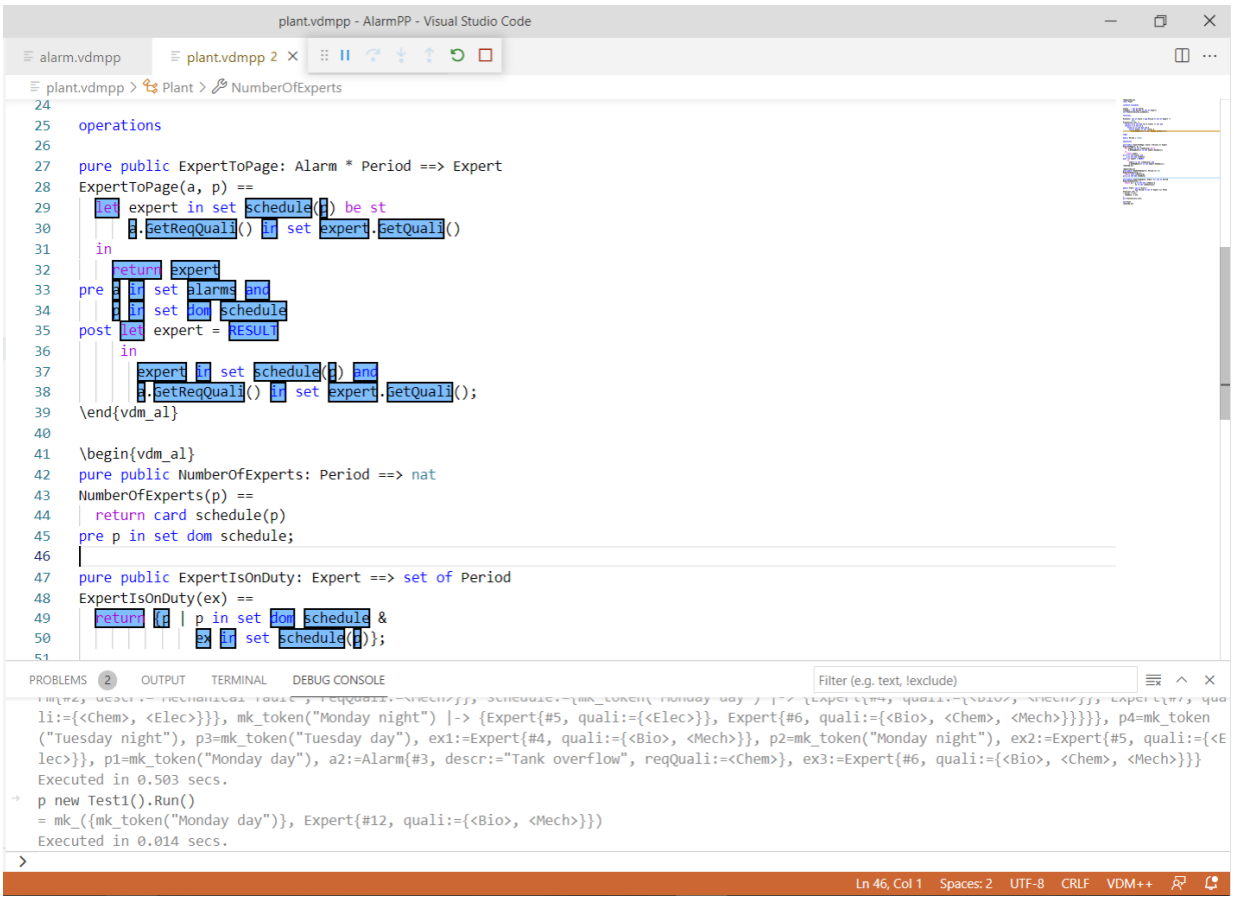

Figure 7.2: Test Coverage

Chapter 8

Pretty Printing to LaTeX

It is possible to use literate programming/specification [Johnson96] with Overture just as you can with VDMTools. To take advantage of this, you need to use the LaTeX text processing system with plain VDM models mixed with textual documentation. The VDM model parts must be enclosed within "`\begin{vdm_al}`" and "`\end{vdm_al}`". The text-parts outside these specification blocks are ignored by the VDM parser, though note that each source file must start with a recognizable LaTeX construct: a `\documentclass, \section, \subsection` or a LaTeX comment.

To use this functionality, you just have to right click on a file and select 'Translate to LaTeX'. Thus, a new file with a `.tex` extension will be created. This file is written into a project subfolder named `.generated/latex <date and time>`.

Chapter 9

Generating Dependency Graph

A dependency graph represents the dependencies of several objects towards each other. Thus, it is considered as a directed graph where each node points to the node on which it depends. In some cases you can add some conditions set on the different connections between the nodes. Moreover, each shape represents a node (usually ellipses or circles) and each connector, composed of one or two arrow heads, indicates the direction of the dependencies. You have also the possibility to add labels on connectors to specify the relation between two nodes. To finish, the main usage of a dependency graph consists of describing processes to make it easier for the developer to understand, reuse and maintain his code.

9.1 Generating Dependency Graph

In order to generate a dependency graph, you just have to right-click on a file and select 'Generate Dependency Graph'. Thus a graphviz file (dependencies.dot) will be generated into a project subfolder named `.generated/graphviz <date and time>`. One example of the created file is shown below, see Figure 9.1

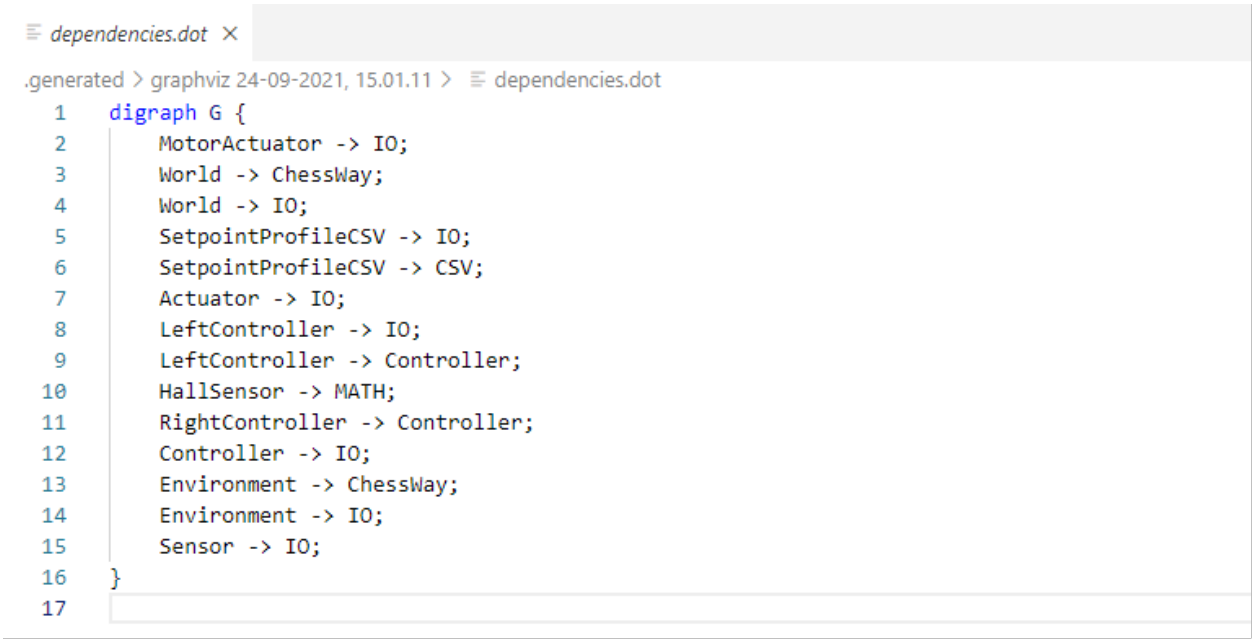

Figure 9.1: Generating a Dependency Graph

9.2 Visualising Dependency Graph

Now that the dependency graph is generated, you can display it thanks to a visualiser, by installing a graphviz extension such as Graphviz for VSCode :

`https://marketplace.visualstudio.com/items?itemName=joaompinto.vscode-graphviz`

Or with Graphviz Interactive Preview :

`https://marketplace.visualstudio.com/items?itemName=tintinweb.graphviz-interactive-preview`

However, you can simply add a graphviz extension by clicking on 'Extensions' (Ctrl+Shift+X), writing 'Graphviz' in the search bar, choosing *Graphviz (dot) language support vor Visual Studio Code* and finally installing the extension. See the Figure 9.2 for more details.

You are now able to visualise a dependency graph by opening the graphviz file, right-clicking in the Editor area, selecting 'Command Palette...' (Ctrl+Shift+P), writing Graphviz in the search bar and choosing if you want to 'Open Preview to the Side' or simply 'Toggle Preview' (Ctrl+Shift+V). Thus, the Figure 9.3 shows you an example of a dependency graph. If you need more information about the extension Graphviz for VSCode, you can visit their website :

`https://marketplace.visualstudio.com/items?itemName=joaompinto.vscode-graphviz`

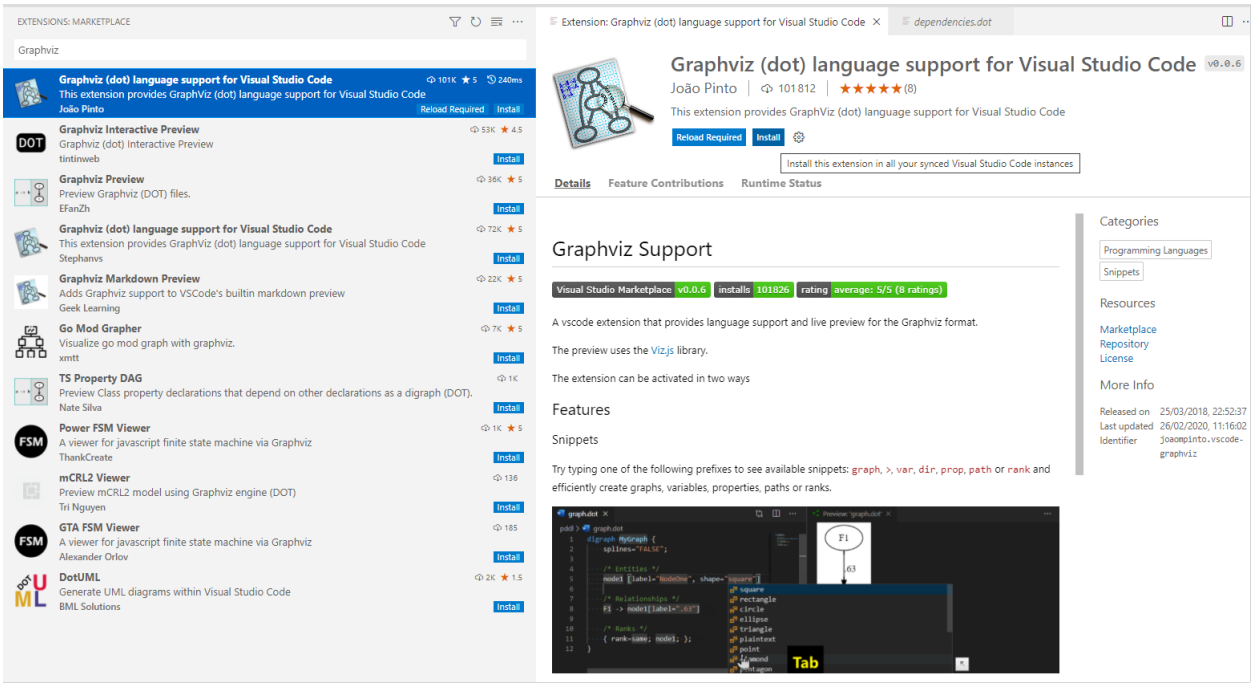

Figure 9.2: Installing Graphviz for VSCode

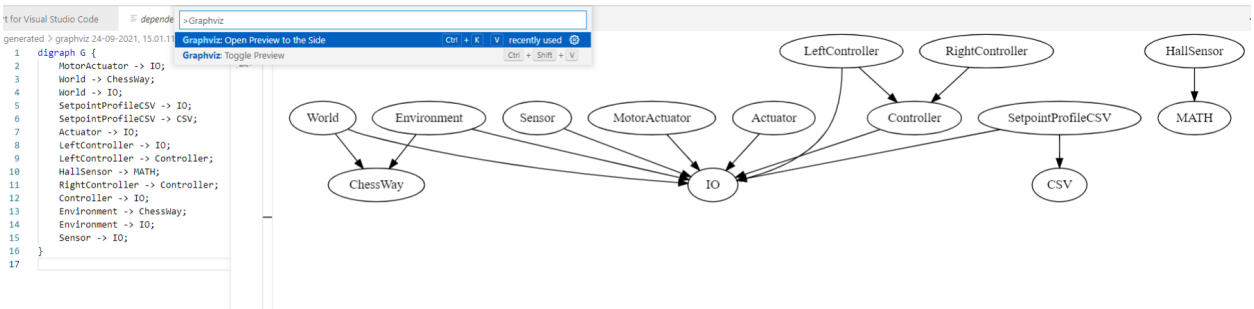

Figure 9.3: Visualising a Graphviz File

Chapter 10

Managing Proof Obligations

In all VDM dialects, Overture can identify places where run-time errors *could* potentially occur if the model was to be executed. The analysis of these areas can be considered as a complement to the static type checking that is performed automatically. Type checking accepts specifications that are *possibly* correct, but we also want to know the places where the specification could possibly fail.

Unfortunately, it is not always possible to statically check if such potential problems will *actually* occur at run-time error or not. So Overture creates *Proof Obligations* for all the places where run-time errors *could* occur. Each proof obligation (PO) is formulated as a predicate that must hold at a particular place in the VDM model if it is error-free, and so it may have particular context information associated with it. POs can be considered as constraints that will guarantee the internal integrity of a VDM model if they are all met. In the long term, it will be possible to prove these constraints with a proof component in Overture, but this is not yet available.

POs can be divided into different categories depending upon their nature. The full list of categories can be found in Appendix F along with a short description for each of them.

The proof obligation generator is invoked either on a VDM project (and then POs for all the VDM model files will be generated) or for one selected VDM file. Right-click the file in the Explorer and then select *Run Proof Obligation Generation*. Overture will change into a special *Proof Obligations* view as shown in Figure 10.1. Once you have generated POs for a VDM project for the first time, they will automatically be re-generated whenever the project is rebuilt as long as you stay in the *Proof Obligations* view.

Note that in the *Proof Obligation View* view, each proof obligation has four components:

- A unique number in the list shown;
- The proof obligation category (type);
- The name of the definition in which the proof obligation is located; and
- The proved proof obligations (true or false).

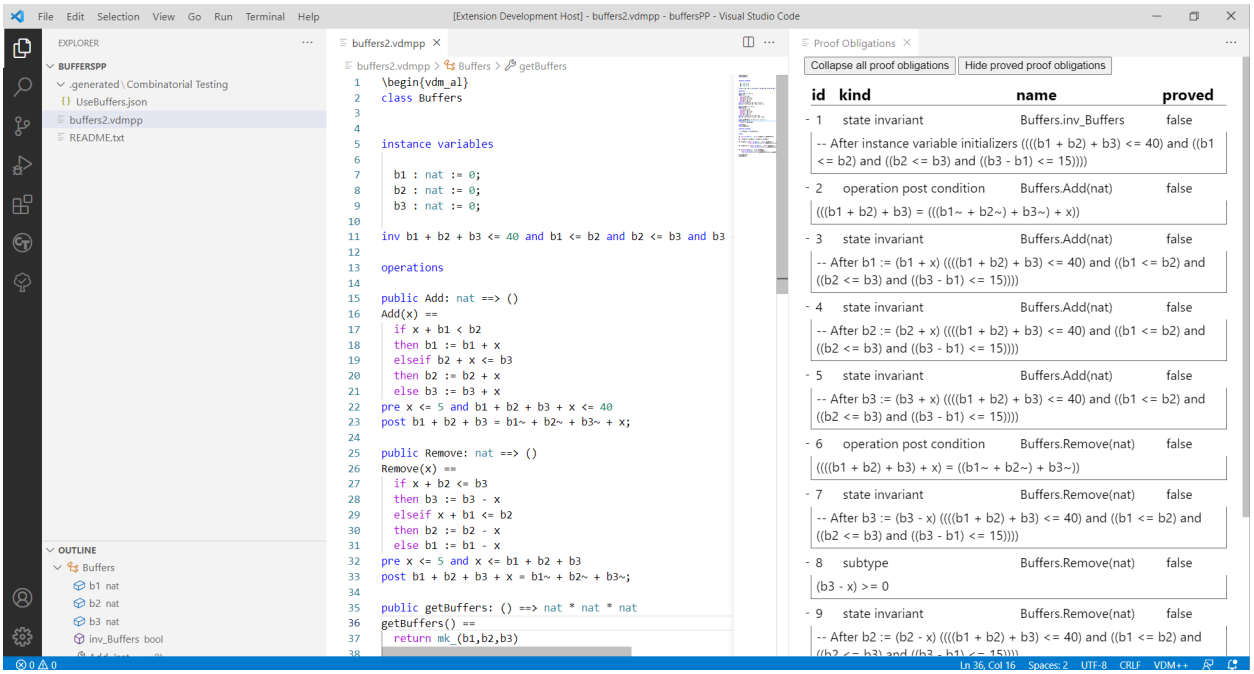

Figure 10.1: The Proof Obligation view

Moreover, you are able to expand or collapse all proof obligations with the buttons 'Expand all proff obligations' and 'Collapse all proof obligations'. And you can also display or hide proved proof obligations with the buttons 'Display proved proof obligations' and 'Hide proved proof obligations'.

Chapter 11

Combinatorial Testing

In order to better automate the testing process, a notion of test *traces* has been introduced into VDM++ (and subsequently VDM-SL and VDM-RT)[1]. Traces are effectively regular expressions that can be expanded to a collection of test cases. Each test case comprises a sequence of operation calls. If a user defines a trace it is possible to make use of a special *Combinatorial Testing* perspective to automatically expand the trace and execute all of the resulting test cases. Subsequently, the results from the tests can be inspected and erroneous test cases easily found. You can then fix problems and re-run the trace to check they are fixed.

11.1 Using the Combinatorial Testing GUI

The syntax for trace definitions is defined in the VDM-10 Language Manual [Larsen&10]. If you have created a traces entry for a module or class it can be executed via the *Combinatorial Testing* view, see Figure 11.1.

Different icons are used to indicate the verdict in a test case. These are:

: This icon is used to indicate that the test case has not yet been executed.

: This icon is used to indicate that the test case has a pass verdict.

: This icon is used to indicate that the test case has an inconclusive verdict.

: This icon is used to indicate that the test case has a fail verdict.

In the CT view, you can right-click on any individual test case, and then send it to the interpreter for execution ('Send test to interpreter'). This is particularly useful for failed test cases since the interpreter allows you to step through the evaluation to the place where it is failing. You can inspect the exact circumstances of the failure, including the values of the different variables in scope.

[1] Note that this is only available for VDM-SL and VDM-RT models if the VDM-10 language version has been selected.

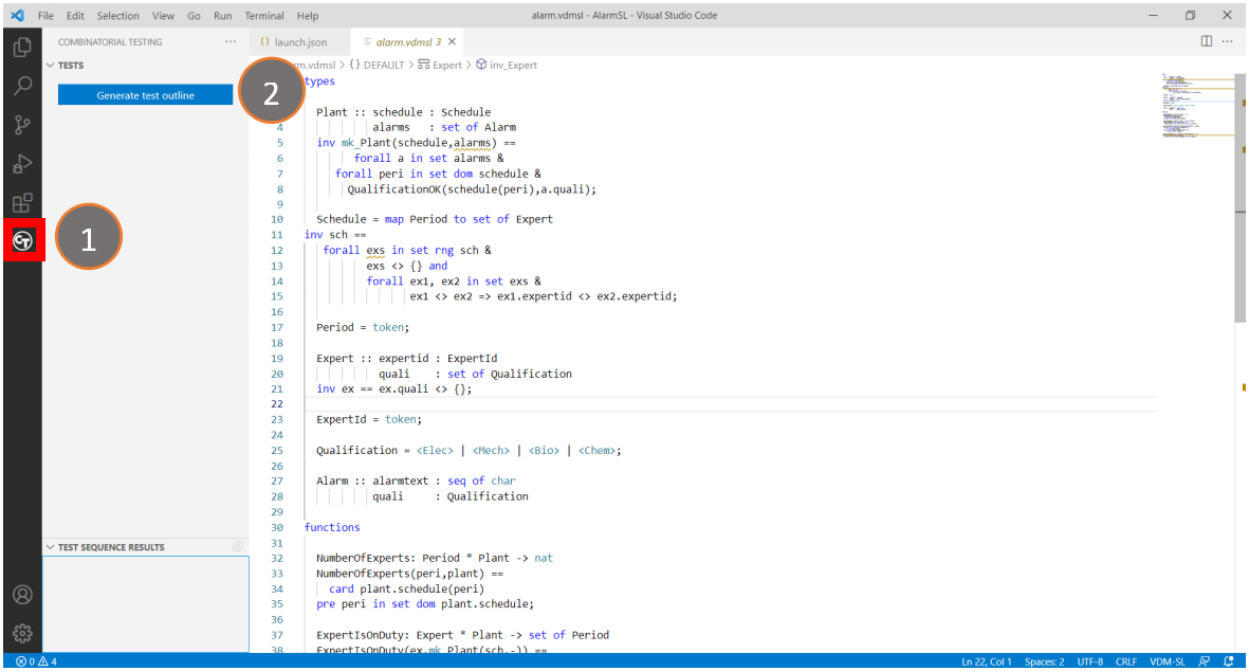

Figure 11.1: Using Combinatorial Testing

1. Click on the button ‘Combinatorial Testing’ to have access to the Combinatorial testing view (red square)

2. Click on the button ‘Generate test outline’

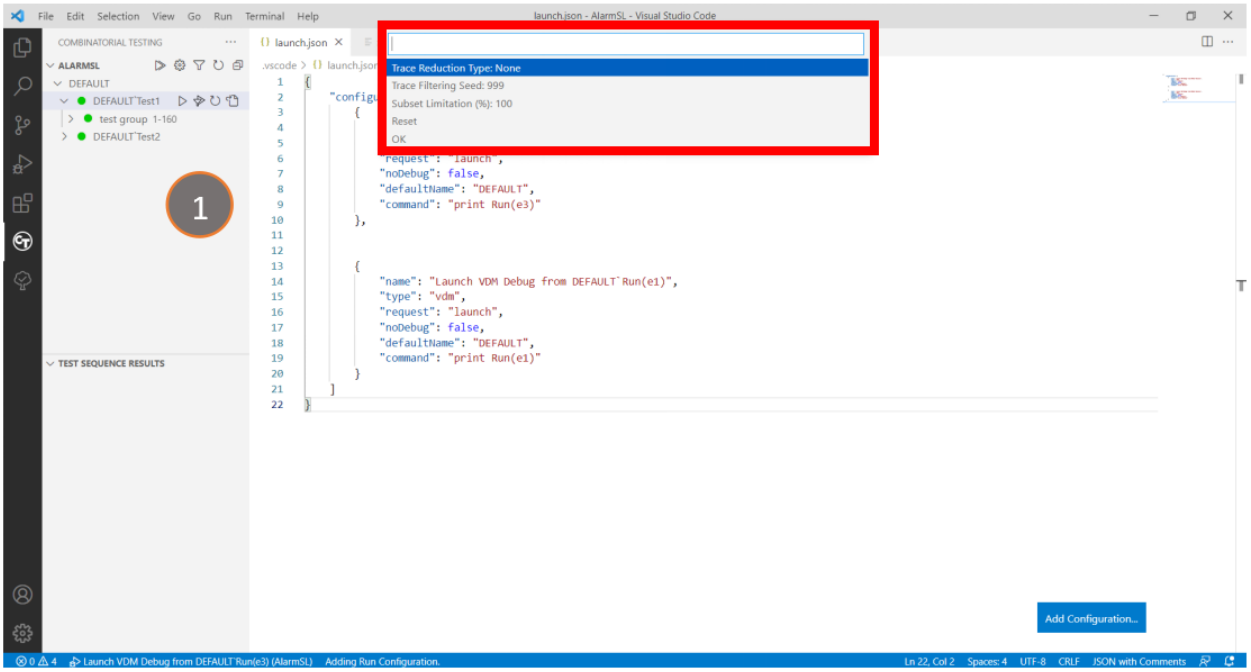

Figure 11.2: Launch of Combinatorial Tests

1. Click on the button (‘Execute All Tests’) to launch all the combinatorial tests. Or if you want to launch only one combinatorial test, you can click on (‘Full Evaluation’).
   If you want to add filtering options, click on the button (‘Set Filter Options’) and a window will appear (red square). Now you have the possibility to reduce the number of tests generated depending on the operation name or/and the variable name (’Trace Reduction Type’), to display the traces with a particular value in their seed (’Trace Filtering Seed’), to limit the number of executed tests (’Subset Limitation(%)’), to reset all the filters changed previously (‘Reset’) or just confirm your changes (‘OK’).

   If you want to filter the combinatorial tests in function of their result, you can click on (’Show selected tests’) and select one or many options between : ’Passed’, ’Failed’, ’Inconclusive’ and ’Filtered’.

   You can also rebuild trace outline with the button (‘Rebuild Trace Outline’), or collapse all tests with the button (‘Collapse All’).

   Finally, in each test, you have the possibility to filter evaluation with the button (‘Filtered Evaluation’), to restart the generation of tests with the button (‘Generate Tests’), or to go to the trace (‘Go to trace’).

Chapter 12

Automatic Generation of Code

It is possible to generate Java code for a large subset of VDM-SL and VDM++ models. In addition to Java, C and C++ code generators are currently being developed. Both these code generators are in the early stages of development. For comparison, code generation of VDM-SL and VDM++ specifications to both Java and C++ is a feature that is available in VDMTools [Java2VDMMan, CGMan, CGManPP]. The majority of this chapter focuses solely on the Java code generator available in VDM VSCode Extension.

12.1 Use of the Java Code Generator

The Java code generator can be launched via the context menu as shown in Figure 12.1. Alternatively, this can be done by highlighting the project in the VDM Explorer and typing one of the shortcuts associated to this plugin. To launch the Java Code Generation, right click on file and select 'Generate Java Code'.

Upon completion of the code generation process the status is output to the console as shown in Figure 12.2. In particular this figure shows the status of code generating the `AlarmPP` model available in the Overture standard examples. As indicated by the console output, the generated code is available as an VSCode project in the `generated/java` folder.

12.2 Limitations of the Java Code Generator

If the Java code generator encounters a construct that it cannot code generate it will report it as unsupported to the user and the user can then try to rewrite that part of the specification using other (supported) constructs. Reporting of unsupported constructs is done via the console output and using editor markers.

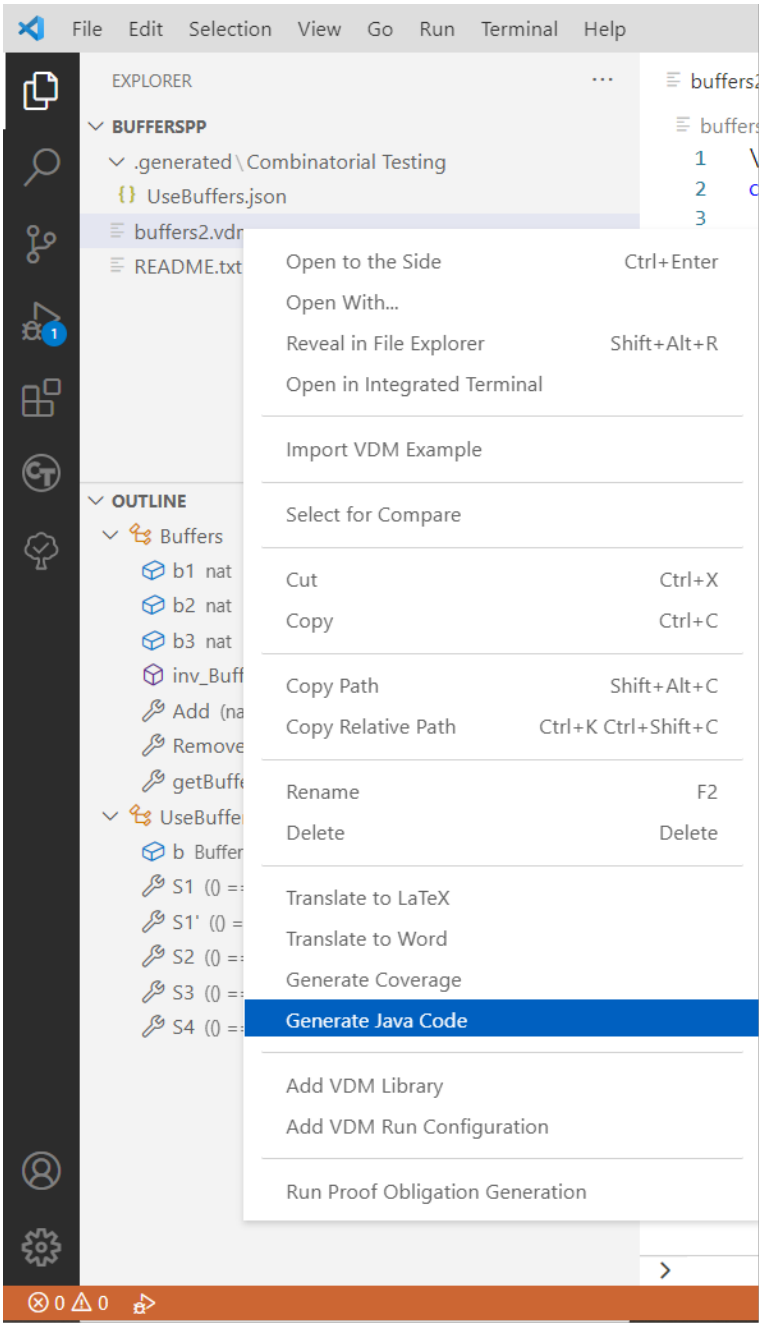

Figure 12.1: Launching the Java code generator.

Figure 12.2: The status of code generating the `AlarmPP` example.

The user will get similar messages and markers for other unsupported VDM constructs. To summarise, the Java code generator currently does not support code generation of multiple inheritance and neither does it support traces, type binds, invariant checks and pre and post conditions. Furthermore, let expressions appearing on the right-hand side of an assignment will also be reported as unsupported. The Java code generator also does not support every pattern. The patterns that are currently not supported are: object, map union, map, union, set, sequence, concatenation and match value.

12.3 The Code Generation Runtime Library

The generated code relies on a runtime library used to represent some of the types available in VDM (tokens, tuples etc.) as well as collections and support for some of the complex operators such as sequence modifications. For simplicity every project generated by the Java code generator contains the runtime library. More specifically, there is a copy of the runtime library containing only the binaries (`lib/codegen-runtime.jar`) as well as a version of the runtime library that has the source code attached (`lib/codegen-runtime-sources.jar`). The runtime library is imported by every code generated class using the Java import statement **`import`** `org.overture.codegen.runtime.*;` and in order to compile the generated Java code the runtime library must be visible to the Java compiler.

Similar to VDMTools the runtime library also provides implementation for subset of the functionality available in the standard libraries: The runtime library provides a full implementation of the `MATH` library, support for conversion of values into character sequences as provided by the `VDMUtil`, and finally functionality to write to the console as available in the `IO` library.

12.4 Configuration of the Java Code Generator

The Java code generator can be configured via a preference page as shown in Figure 12.4. The preference page can be accessed by :

1. Clicking on 'File' (left top corner), select 'Preferences', then Settings
2. Choosing 'Workspace' (not 'User')
3. Going below in 'Extensions' and 'VDM VSCode'

See Figure 12.3 for more details

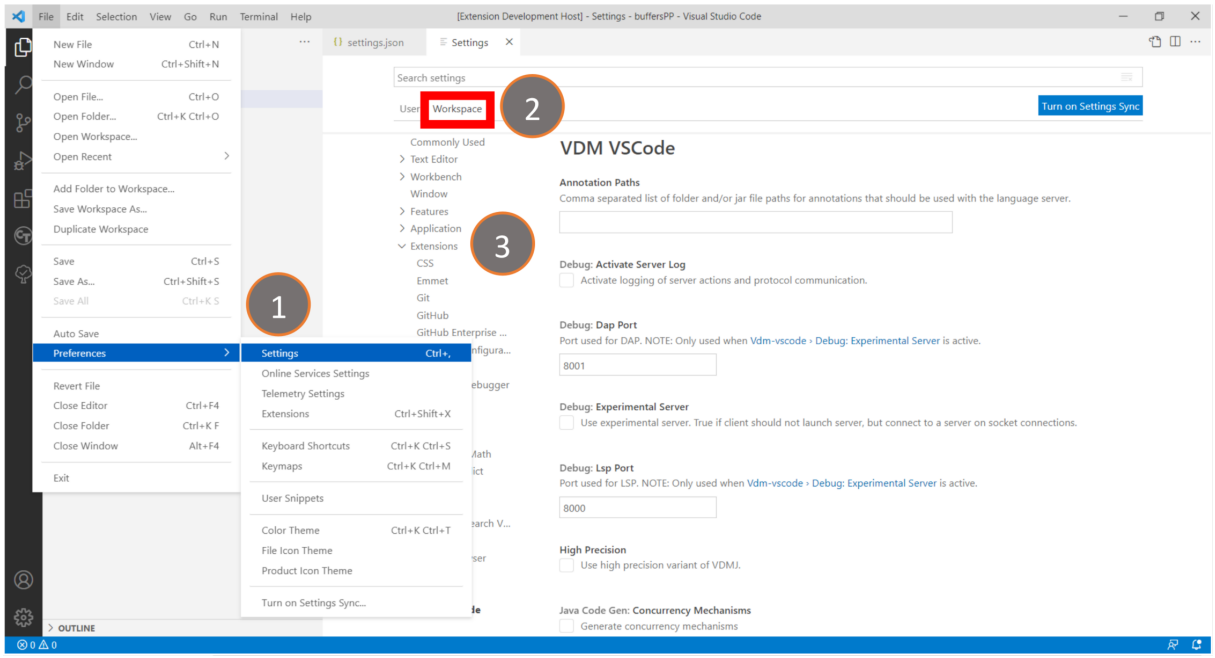

Figure 12.3: Path to go to Java Code Configuration

The Java code generator provides a few options that allows the user to configure the code generation process (see Figure 12.4). The subsections below treat each of these configuration parameters individually, in the order they appear in the preference page.

12.4.1 Disable cloning

In order to respect the value semantics of VDM the Java code generator sometime needs to perform deep copying of objects that represent composite value types (records, tuples, tokens, sets, sequences and maps). For example, in VDM a record is a value type, which means that occurrences of the record must be copied when it appears in the right-hand side of an assignment, it is passed as an argument or returned as a result. However, Java does not support composite value types like structs and records, and as a consequence record types must be represented using classes, which use reference semantics. This means that an object reference, which is used to represent a composite value type in the generated Java code must be deep copied when it appears in the right-hand side of an assignment, it is passed as an argument or returned as a result. For arbitrarily complex value types (such as records composed of record) deep copying may introduce a significant overhead in the generated code. If the specification subject to code generation does not truly rely on value semantics the user may wish to disable deep copying of value types in the generated code in order to remove this overhead. The user should, however, be aware that disabling of cloning may lead to code being generated that does not preserve the semantics of the input specification and in general disabling of cloning is discouraged. By default cloning is enabled.

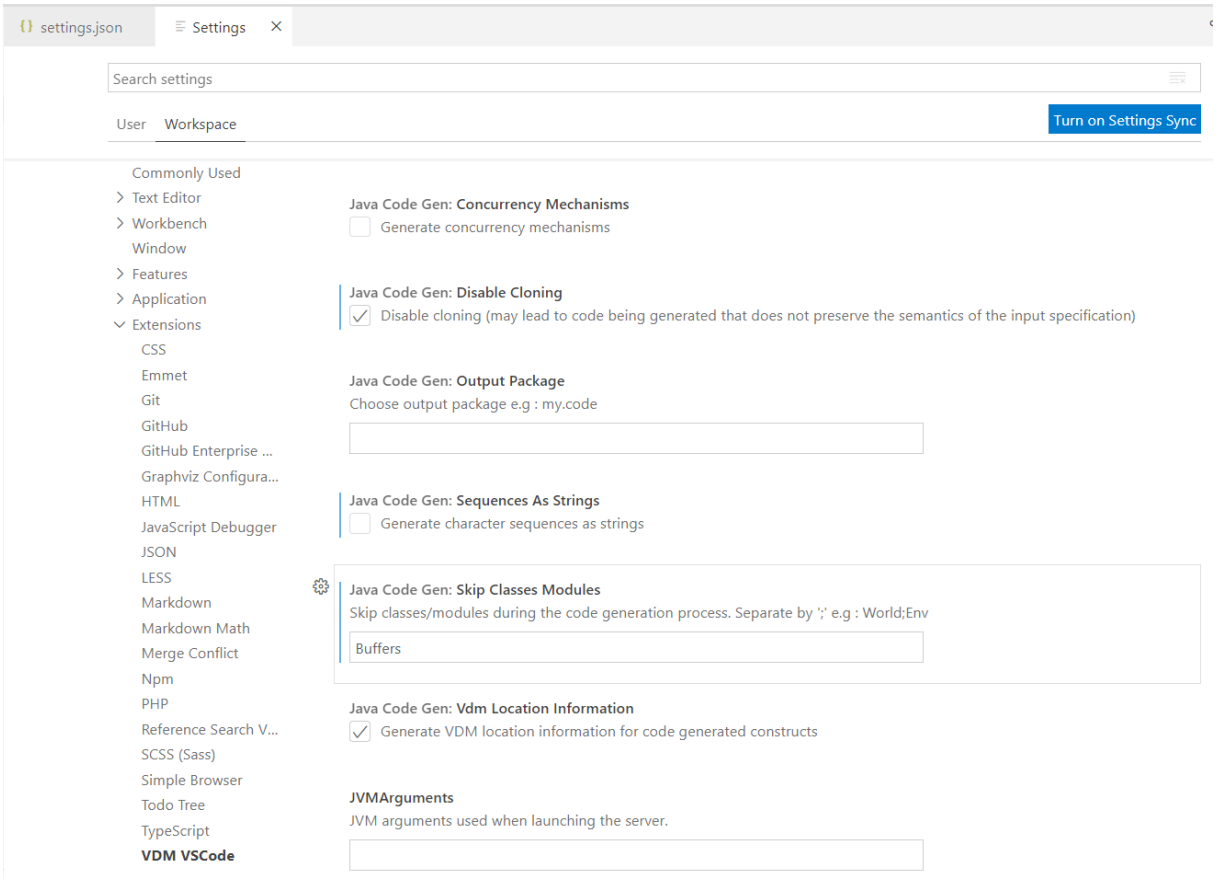

Figure 12.4: Configuration of the Java Code Generator

12.4.2 Generate character sequences as strings

In VDM a string is a sequence of characters and there is no notion of a string type. Java in particular works differently since it uses a separate type to represent a string. The default behaviour of the Java code generator is to code generate sequences of characters as strings and subsequently do the necessary conversion between between strings and sequences in the generated code. Another possibility is to treat a string literal for what it truly is, namely a sequence of characters, and thereby avoid any conversion between strings and sequences. In order to do that, i.e. not generating character sequences as strings, the corresponding option must be unchecked.

12.4.3 Generate concurrency mechanisms

If the user does not rely on the concurrency mechanisms of VDM++ and does not want to include support for them in the generated code the corresponding option in the preference page must be unchecked. By default the behaviour of the Java code generator is to not include support for the concurrency mechanisms of VDM++ in the generated code.

12.4.4 Generate VDM location information for code generated constructs

When a VDM model is code generated it can be helpful to know where the constructs in the generated code originate from. When this option is enabled the Java code generator will generate VDM location information for methods, statements and local declarations in the generated code. More specifically, the Java code generator will generate a Java source code comment containing the name of the VDM source file and the line number and the position, for each method, statement and local declaration. As an example, the code fragment below says that the Java return statement originates from a VDM construct at line 25, position 12 in `File.vdmsl`.

```
/* File.vdmsl 25:12 */
return 42;
```

12.4.5 Choose output package

The Java code generator allows the output package of the generated code to be specified. If the user does not specify a package, the code generator outputs the generated Java code to a package with the same name as the VDM project. If the name of the project is not a valid java package, then the generated code is output to the default Java package.

12.4.6 Skip classes/modules during the code generation process

It may not always make sense to code generate every class or module in a VDM project. A class or module can often be skipped if it acts as an execution entry point or it is used to load input for the specification. Classes or modules that the user wants to skip can be specified in the text box in the Java code generator preference page by separating the class/module names by a semicolon. As an example, `World;Env` makes the code generator skip code generation of `World` and `Env`, while generating code for any other module or class. For convenience the output of the Java code generator will also inform the user about what classes or modules are being skipped.

12.5 Translation of the VDM types and type constructors

Table 12.1 describes how the VDM type(s) in the left column are represented in the generated Java code (the right column). In this table `pack` is the user-specified root package of the generated Java code and `E`, `D` and `R` represent arbitrary VDM types. The type mapping in the last row is only used when the *Generate character sequences as strings* option is selected. Some of the types used to represent the VDM types are native Java types (from package `java.lang`), others are part of the Java code generator runtime library (from package `org.overture.codegen.runtime`), and some are generated.

VDM type(s)	Java type
`bool`	`java.lang.Boolean`
`nat`, **`nat1`**, **`int`**, **`rat`**, **`real`**	`java.lang.Number`
`char`	`java.lang.Character`
`token`	`org.overture.codegen.runtime.Token`
Tuple types (e.g. **`nat`** `*` **`nat`**)	`org.overture.codegen.runtime.Tuple`
Union types (e.g. **`nat`** `|` **`nat`**)	`java.lang.Object`
Quote type `<T>`	`pack.quotes.TQuote`
User-defined types `T = D`	Represented using the representation of type `D`
A class `C`	`pack.C`
Record type `R` defined in class or module `M`	Inner class `pack.M.R`
`set of` `E`	`org.overture.codegen.runtime.VDMSet`
`map` `D` **`to`** `R`, **`inmap`** `D` **`to`** `R`	`org.overture.codegen.runtime.VDMMap`
`seq of` `E`, **`seq1 of`** `E`	`org.overture.codegen.runtime.VDMSeq`
`seq of char`, **`seq1 of char`**	`java.lang.String`

Table 12.1: The type mappings used by the Java code generator.

Chapter 13

Expanding VDM Models Scope and Functionnality by Linking Java and VDM

13.1 Enabling Remote Control of the Overture Interpreter

In some situations, it may be valuable to establish a front end (for example a GUI or a test harness) for calling a VDM model. This feature corresponds roughly to the CORBA based API from VDMTools [APIMan].

Remote control should be understood as a delegation of control of the interpreter, which means that the remote controller is in charge of the execution or debug session and is responsible for taking action and executing parts of the VDM model when needed. When finished, it should return and the session will stop. When a Remote controller is used, the Overture debugger continues working normally, so for example breakpoints can be used in debug mode. Moreover, all dialects (VDM-SL, VDM++ and VDM-RT) support Remote Control. A new configuration with the use of a remote controller can be started by (see Figure 13.1 for more details):

1. Clicking on the button ”Add Configuration...”
2. Selecting ”VDM Debug: Remote Control (VDM-SL/++/RT)

Then a new snippet (see Figure 13.2) will be created with the remoteControl option. And you simply have to write the full package/class name of the Remote Control.

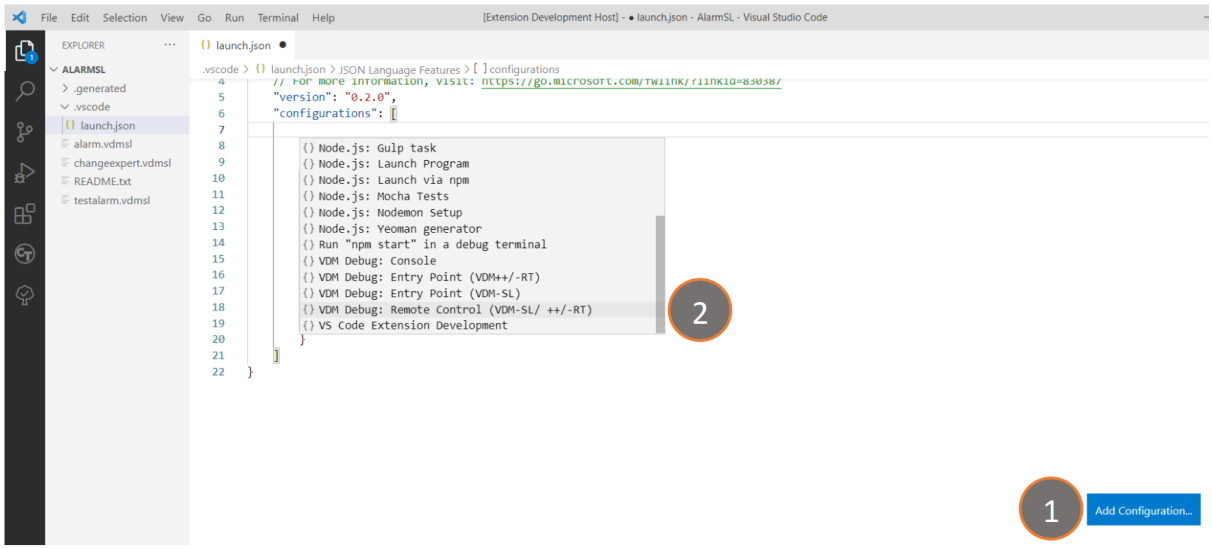

Figure 13.1: Path to add a New Configuration using Remote Control

```
"version": "0.2.0",
"configurations": [
    {
        "name": "Launch VDM Debug with Remote Control",
        "type": "vdm",
        "request": "launch",
        "noDebug": false,
        "dynamicTypeChecks": true,
        "invariantsChecks": true,
        "preConditionChecks": true,
        "postConditionChecks": true,
        "measureChecks": true,
        "defaultName": null,
        "remoteControl": "Full package/class name (e.g : 'com.fujitsu.ctrl.GUI')"
    },
```

Figure 13.2: New Snippet with Remote Control Option

References

[Bjørner&78] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*. Volume 61 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978.

This was the first monograph on *Meta-IV*. See also entries: [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]

[CGMan] The VDM Tool Group. *The VDM-SL to C++ Code Generator*. Technical Report, CSK Systems, January 2008.

[CGManPP] The VDM Tool Group. *The VDM++ to C++ Code Generator*. Technical Report, CSK Systems, January 2008.

[Clement&99] Tim Clement and Ian Cottam and Peter Froome and Claire Jones. The Development of a Commercial "Shrink-Wrapped Application" to Safety Integrity Level 2: the DUST-EXPERT Story. In *Safecomp'99*, Springer Verlag, Toulouse, France, September 1999. LNCS 1698, ISBN 3-540-66488-2.

[Elmstrøm&94] René Elmstrøm and Peter Gorm Larsen and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994. 4 pages.

[Fitzgerald&05] John Fitzgerald and Peter Gorm Larsen and Paul Mukherjee and Nico Plat and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[Fitzgerald&08a] J. S. Fitzgerald and P. G. Larsen and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. 11 pages. edited by Benjamin Wah, John Wiley & Sons, Inc..

[Fitzgerald&08b] John Fitzgerald and Peter Gorm Larsen and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008. 8 pages.

[Fitzgerald&09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[Fitzgerald&14] John Fitzgerald and Peter Gorm Larsen and Marcel Verhoef, editors. *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014.

[Fitzgerald&98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[ISOVDM96] Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. December 1996.

[Java2VDMMan] The VDM Tool Group. *The Java to VDM++ User Manual*. Technical Report, CSK Systems, January 2008.

[Johnson96] C.W. Johnson. Literate Specifications. *Software Engineering Journal*, 225–237, July 1996.

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. 333 pages. ISBN 0-13-880733-7.

This book deals with the Vienna Development Method. The approach explains formal (functional) specifications and verified design with an emphasis on the study of proofs in the development process.

[Kurita&09] T. Kurita and Y. Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3):343–355, October 2009. 13 pages.

[Larsen01] Peter Gorm Larsen. Ten Years of Historical Development: "Bootstrapping" VDMTools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.

| http://www.jucs.org/jucs_7_8/ten_years_of_historical—

[Larsen&10] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle. *The VDM-10 Language Manual*. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010.

[Larsen&13] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle and John Fitzgerald and Sune Wolff and Shin Sahara. *VDM-10 Language Manual*. Technical Report TR-001, The Overture Initiative, www.overturetool.org, April 2013. 208 pages.

[Larsen&96] Peter Gorm Larsen and Bo Stig Hansen. Semantics for Underdetermined Expressions. *Formal Aspects of Computing*, 8(1):47–66, January 1996.

[Mukherjee&00] Paul Mukherjee and Fabien Bousquet and Jérôme Delabre and Stephen Paynter and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[Verhoef&06] Marcel Verhoef and Peter Gorm Larsen and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra and Tobias Nipkow and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162, Springer-Verlag, 2006.

Appendix A

Internal Errors

This appendix gives a list of the internal errors in Overture and the circumstances under which each internal error can be expected. Most of these errors should *never* be seen, so if they appear please report the occurrence via the Overture bug reporting utility (`https://github.com/overturetool/overture/issues/new`).

0000: `File IO errors, eg. "File not found"` This typically occurs if a specification file is no longer present.

0001: `"Mark/reset not supported - use push/pop"`

0002: `"Cannot change type qualifier: <name><qualifiers> to <qualifiers>"`

0003: `"PatternBind passed <class name>"`

0004: `"Cannot get bind values for type <type>"`

0005: `"Illegal clone"`

0006: `"Constructor for <class> can't find <member>"`

0007: `"Cannot write to IO file <name>"`

0009: `"Too many syntax errors"` This error typically occurs if one have included a file that is in a non VDM format and by mistake have given it a vdm file extension (`vdmsl`, `vdmpp` or `vdmrt`).

0010: `"Too many type checking errors"`

0011: `"CPU or BUS creation failure"`

0052: `"Cannot set default name at breakpoint"`

0053: `"Unknown trace reduction type"`

0054: "Cannot instantiate native object: <reason>"

0055: "Cannot access native object: <reason>"

0056: "Native method cannot use pattern arguments: <sig>"

0057: "Native member not found: <name>"

0058: "Native method does not return Value: "

0059: "Failed in native method: <reason>"

0060: "Cannot access native method: <reason>"

0061: "Cannot find native method: <reason>"

0062: "Cannot invoke native method: <reason>"

0063: "No delegate class found: <name>"

0064: "Native method should be static: <name>"

0065: "Illegal Lock state"

0066: "Thread is not running on a CPU"

0067: "Exported type <name> not structured"

0068: "Periodic threads overlapping"

Appendix B

Lexical Errors

When a VDM model is parsed, the first phase is to gather the single characters into tokens that can be used in the further processing. This is called a lexical analysis and errors in this area can be as follows:

1000: "Malformed quoted character"

1001: "Invalid char <ch> in base <n> number"

1002: "Expecting '|->'"

1003: "Expecting '...'"

1004: "Expecting '<-:'"

1005: "Expecting close double quote"

1006: "Expecting close quote after character"

1007: "Unexpected tag after '#'"

1008: "Malformed module`name"

1009: "Unexpected character 'c'"

1010: "Expecting <digits>[.<digits>][e<+-><digits>]"

1011: "Unterminated block comment"

Appendix C

Syntax Errors

If the syntax of the file you have provided does not meet the syntax rules for the VDM dialect you wish to use, syntax errors will be reported. These can be as follows:

2000: "Expecting 'in set' or 'in seq' after pattern in binding"

2001: "Expecting 'in set' or 'in seq' in bind"

2002: "Expecting ':' in type bind"

2003: "Expecting 'in set' or 'in seq' after pattern in binding"

2004: "Expecting 'in set', 'in seq' or ':' after patterns"

2005: "Expecting list of 'class' or 'system' definitions"

2006: "Found tokens after class definitions"

2007: "Expecting 'end <class>'"

2008: "Class does not start with 'class'"

2009: "Can't have instance variables in VDM-SL"

2010: "Can't have a thread clause in VDM-SL"

2011: "Only one thread clause permitted per class"

2012: "Can't have a sync clause in VDM-SL"

2013: "Expected 'operations', 'state', 'functions', 'types' or 'values'"

2014: "Recursive type declaration" This is reported in type definitions such as `T = T`.

2015: "Expecting =<type> or ::<field list>"

2016: "Function name cannot start with 'mk_'"

2017: "Expecting ':' or '(' after name in function definition"

2018: "Function type is not a -> or +> function"

2019: "Expecting identifier <name> after type in definition"

2020: "Expecting '(' after function name"

2021: "Expecting ':' or '(' after name in operation definition"

2022: "Expecting name <name> after type in definition"

2023: "Expecting '(' after operation name"

2024: "Expecting external declarations after 'ext'"

2025: "Expecting <name>: exp->exp in errs clause"

2026: "Expecting 'rd' or 'wr' after 'ext'"

2027: "-"

2028: "Expecting 'per' or 'mutex'"

2029: "Expecting <set bind> = <expression>"

2030: "Expecting simple field identifier"

2031: "Expecting field number after .#"

2032: "Expecting field name"

2033: "Expected 'is not specified' or 'is subclass responsibility'"

2034: "Unexpected token in expression"

2035: "Tuple must have >1 argument"

2036: "Expecting mk_<type>"

2037: "Malformed mk_<type> name <name>"

2038: "Expecting is_<type>"

2039: "Expecting maplet in map enumeration"

2040: "Expecting 'else' in 'if' expression"

2041: "Expecting two arguments for 'isofbase'"

2042: "Expecting (<class>,<exp>) arguments for 'isofbase'"

2043: "Expecting two arguments for 'isofclass'"

2044: "Expecting (<class>,<exp>) arguments for 'isofclass'"

2045: "Expecting two expressions in 'samebaseclass'"

2046: "Expecting two expressions in 'sameclass'"

2047: "Can't use history expression here"

2048: "Expecting #act, #active, #fin, #req or #waiting"

2049: "Expecting 'end <module>'"

2050: "Expecting library name after 'uselib'"

2051: "Expecting 'end <module>'"

2052: "Expecting 'all', 'types', 'values', 'functions' or 'operations'"

2053: "Exported function is not a function type"

2054: "Expecting types, values, functions or operations"

2055: "Imported function is not a function type"

2056: "Cannot use module'id name in patterns"

2057: "Unexpected token in pattern"

2058: "Expecting identifier"

2059: "Expecting a name"

2060: "Found qualified name <name>. Expecting an identifier"

2061: "Expecting a name"

2062: "Expected 'is not specified' or 'is subclass responsibility'"

2063: "Unexpected token in statement"

2064: "Expecting <object>.identifier(args) or name(args)"

2065: "Expecting <object>.name(args) or name(args)"

2066: "Expecting object field name"

2067: "Expecting 'self', 'new' or name in object designator"

2068: "Expecting field identifier"

2069: "Expecting <identifier>:<type> := <expression>"

2070: "Function type cannot return void type"

2071: "Expecting field identifier before ':'"

2072: "Expecting field name before ':-'"

2073: "Duplicate field names in record type"

2074: "Unexpected token in type expression"

2075: "Expecting 'is subclass of'"

2076: "Expecting 'is subclass of'"

2077: "Expecting 'end' after class members"

2078: "Missing ';' after type definition"

2079: "Missing ';' after function definition"

2080: "Missing ';' after state definition"

2081: "Missing ';' after value definition"

2082: "Missing ';' after operation definition"

2083: "Expecting 'instance variables'"

2084: "Missing ';' after instance variable definition"

2085: "Missing ';' after thread definition"

2086: "Missing ';' after sync definition"

2087: "Expecting '==' after pattern in invariant"

2088: "Expecting '@' before type parameter"

2089: "Expecting '@' before type parameter"

2090: "Expecting ']' after type parameters"

2091: "Expecting ')' after function parameters"

2092: "Expecting '==' after parameters"

2093: "Missing colon after pattern/type parameter"

2094: "Missing colon in identifier/type return value"

2095: "Implicit function must have post condition"

2096: "Expecting <pattern>[:<type>]=<exp>"

2097: "Expecting 'of' after state name"

2098: "Expecting '==' after pattern in invariant"

2099: "Expecting '==' after pattern in initializer"

2100: "Expecting 'end' after state definition"

2101: "Expecting ')' after operation parameters"

2102: "Expecting '==' after parameters"

2103: "Missing colon after pattern/type parameter"

2104: "Missing colon in identifier/type return value"

2105: "Implicit operation must define a post condition"

2106: "Expecting ':' after name in errs clause"

2107: "Expecting '->' in errs clause"

2108: "Expecting <pattern>=<exp>"

2109: "Expecting <type bind>=<exp>"

2110: "Expecting <pattern> in set|seq <exp>"

2111: "Expecting <pattern> in set|seq <exp>"

2112: "Expecting '(' after periodic"

2113: "Expecting ')' after period arguments"

2114: "Expecting '(' after periodic(...)"

2115: "Expecting (name) after periodic(...)"

2116: "Expecting <name> => <exp>"

2117: "Expecting '(' after mutex"

2118: "Expecting ')' after 'all'"

2119: "Expecting ')'"

2120: "Expecting 'e1,...,e2' in subsequence"

2121: "Expecting ')' after subsequence"

2122: "Expecting ')' after function args"

2123: "Expecting ']' after function instantiation"

2124: "Expecting ')'"

2125: "Expecting 'is not yet specified"

2126: "Expecting 'is not yet specified"

2127: "Expecting 'is subclass responsibility'"

2128: "Expecting comma separated record modifiers"

2129: "Expecting <identifier> |-> <expression>"

2130: "Expecting ')' after mu maplets"

2131: "Expecting ')' after mk_ tuple"

2132: "Expecting is_(expression, type)"

2133: "Expecting ')' after is_ expression"

2134: "Expecting pre_(function [,args])"

2135: "Expecting '}' in empty map"

2136: "Expecting '}' after set comprehension"

2137: "Expecting 'e1,...,e2' in set range"

2138: "Expecting '}' after set range"

2139: "Expecting '}' after set enumeration"

2140: "Expecting '}' after map comprehension"

2141: "Expecting '}' after map enumeration"

2142: "Expecting ']' after list comprehension"

2143: "Expecting ']' after list enumeration"

2144: "Missing 'then'"

2145: "Missing 'then' after 'elseif'"

2146: "Expecting ':' after cases expression"

2147: "Expecting '->' after others"

2148: "Expecting 'end' after cases"

2149: "Expecting '->' after case pattern list"

2150: "Expecting 'in' after local definitions"

2151: "Expecting 'st' after 'be' in let expression"

2152: "Expecting 'in' after bind in let expression"

2153: "Expecting '&' after bind list in forall"

2154: "Expecting '&' after bind list in exists"

2155: "Expecting '&' after single bind in exists1"

2156: "Expecting '&' after single bind in iota"

2157: "Expecting '&' after bind list in lambda"

2158: "Expecting 'in' after equals definitions"

2159: "Expecting '(' after new class name"

2160: "Expecting '(' after 'isofbase'"

2161: "Expecting ')' after 'isofbase' args"

2162: "Expecting '(' after 'isofclass'"

2163: "Expecting ')' after 'isofclass' args"

2164: "Expecting '(' after 'samebaseclass'"

2165: "Expecting ')' after 'samebaseclass' args"

2166: "Expecting '(' after 'sameclass'"

2167: "Expecting ')' after 'sameclass' args"

2168: "Expecting <#op>(name(s))"

2169: "Expecting <#op>(name(s))"

2170: "Expecting 'module' at module start"

2171: "Expecting 'end' after module definitions"

2172: "Expecting 'dlmodule' at module start"

2173: "Expecting 'end' after dlmodule definitions"

2174: "Malformed imports? Expecting 'exports' section"

2175: "Expecting ':' after export name"

2176: "Expecting ':' after export name"

2177: "Expecting ':' after export name"

2178: "Expecting 'imports'"

2179: "Expecting 'from' in import definition"

2180: "Mismatched brackets in pattern"

2181: "Mismatched braces in pattern"

2182: "Mismatched square brackets in pattern"

2183: "Expecting '(' after mk_ tuple"

2184: "Expecting ')' after mk_ tuple"

2185: "Expecting '(' after <type> record"

2186: "Expecting ')' after <type> record"

2187: "Expecting 'is not yet specified"

2188: "Expecting 'is not yet specified"

2189: "Expecting 'is subclass responsibility'"

2190: "Expecting 'exit'"

2191: "Expecting 'tixe'"

2192: "Expecting '{' after 'tixe'"

2193: "Expecting '|->' after pattern bind"

2194: "Expecting 'in' after tixe traps"

2195: "Expecting 'trap'"

2196: "Expecting 'with' in trap statement"

2197: "Expecting 'in' in trap statement"

2198: "Expecting 'always'"

2199: "Expecting 'in' after 'always' statement"

2200: "Expecting '||'"

2201: "Expecting '(' after '||'"

2202: "Expecting ')' at end of '||' block"

2203: "Expecting 'atomic'"

2204: "Expecting '(' after 'atomic'"

2205: "Expecting ')' after atomic assignments"

2206: "Expecting '(' after call operation name"

2207: "Expecting '(' after new class name"

2208: "Expecting 'while'"

2209: "Expecting 'do' after while expression"

2210: "Expecting 'for'"

2211: "Expecting 'in set' after 'for all'"

2212: "Expecting 'in set' after 'for all'"

2213: "Expecting 'do' after for all expression"

2214: "Expecting 'in' after pattern bind"

2215: "Expecting 'do' before loop statement"

2216: "Expecting '=' after for variable"

2217: "Expecting 'to' after from expression"

2218: "Expecting 'do' before loop statement"

2219: "Missing 'then'"

2220: "Missing 'then' after 'elseif' expression"

2221: "Expecting ':=' in object assignment statement"

2222: "Expecting ':=' in state assignment statement"

2223: "Expecting ')' after map/seq reference"

2224: "Expecting statement block"

2225: "Expecting ';' after statement"

2226: "Expecting ')' at end of statement block"

2227: "Expecting ';' after declarations"

2228: "Expecting name:type in declaration"

2229: "Expecting 'return'"

2230: "Expecting 'let'"

2231: "Expecting 'in' after local definitions"

2232: "Expecting 'st' after 'be' in let statement"

2233: "Expecting 'in' after bind in let statement"

2234: "Expecting 'cases'"

2235: "Expecting ':' after cases expression"

2236: "Expecting '->' after case pattern list"

2237: "Expecting '->' after others"

2238: "Expecting 'end' after cases"

2239: "Expecting 'def'"

2240: "Expecting 'in' after equals definitions"

2241: "Expecting '['"

2242: "Expecting ']' after specification statement"

2243: "Expecting 'start'"

2244: "Expecting 'start('"

2245: "Expecting ')' after start object"

2246: "Expecting 'startlist'"

2247: "Expecting 'startlist('"

2248: "Expecting ')' after startlist objects"

2249: "Missing 'of' in compose type"

2250: "Missing 'end' in compose type"

2251: "Expecting 'to' in map type"

2252: "Expecting 'to' in inmap type"

2253: "Expecting 'of' after set"

2254: "Expecting 'of' after seq"

2255: "Expecting 'of' after seq1"

2256: "Bracket mismatch"

2257: "Missing close bracket after optional type"

2258: "Expecting '==>' in explicit operation type"

2259: "Operations cannot have [@T] type parameters"

2260: "Module starts with 'class' instead of 'module'"

2261: "Missing comma between return types?"

2262: "Can't have traces in VDM-SL"

2263: "Missing ';' after named trace definition"

2264: "Expecting ':' after trace name"

2265: "Expecting 'n1, n2' after trace definition"

2266: "Expecting 'n' or 'n1, n2' after trace definition"

2267: "Expecting 'obj.op(args)' or 'op(args)'"

2268: "Expecting 'id.id(args)'"

2269: "Expecting '(trace definitions)'"

2270: "Only value definitions allowed in traces"

2271: "Expecting 'duration'"

2272: "Expecting 'duration('"

2273: "Expecting ')' after duration"

2274: "Expecting 'cycles'"

2275: "Expecting 'cycles('"

2276: "Expecting ')' after cycles"

2277: "Can't have state in VDM++"

2278: "Async only permitted for operations"

2279: "Invalid breakpoint hit condition"

2280: "System class cannot be a subclass"

2290: "System class can only define instance variables and a constructor"

2291: "'reverse' not available in VDM classic"

2292: "Expecting '|| (...)'"

2293: "Expecting '|| (a, b ,...)'"

2294: "Expecting ')' ending || clause"

2295: "Can't use old name here"

2296: "Block cannot be empty"

2297: "Expecting '|->' in map pattern"

2298: "Map patterns not available in VDM classic"

2299: "Expecting {|->} empty map pattern"

2300: "mk_<type> must have a single argument"

2301: "Expecting narrow_(expression, type)"

2302: "Expecting ')' after narrow_ expression"

2303: "Narrow not available in VDM classic"

2304: "'stop' not available in VDM classic"

2305: "'stoplist' not available in VDM classic"

2306: "Expecting 'stop'"

2307: "Expecting 'stop('"

2308: "Expecting ')' after stop object"

2309: "Expecting 'stoplist'"

2310: "Expecting 'stoplist('"

2311: "Expecting ')' after stoplist objects"

2312: "Expecting '(' after sporadic"

2313: "Expecting ')' after sporadic arguments"

2314: "Expecting '(' after sporadic(...)"

2315: "Expecting (name) after sporadic(...)"

2316: "Periodic threads only available in VDM-RT"

2317: "Sporadic threads only available in VDM-RT"

2318: "Unexpected token after flat definitions"

2319: "Expecting class name after obj_ in object pattern"

2320: "Expecting '(' after obj_ pattern"

2321: "Expecting '|->' in object pattern"

2322: "Expecting ')' after obj_ pattern"

2323: "Object patterns not available in VDM classic"

2324: "Pure only permitted for operations"

2325: `"Pure operations are not available in classic"`

2326: `"Expecting 'of' after set1"`

2327: `"Type set1 is not available in classic"`

2328: `"Sequence binds are not available in classic"`

2331: `"Expecting inv, eq or ord clause"`

2332: `"Duplicate inv clause"`

2333: `"Type eq/ord clauses not available in classic"`

Appendix D

Type Errors and Warnings

If the syntax rules are satisfied, it is still possible to get errors from the type checker. The errors can be as follows:

3000: `"Expression does not match declared type"`

3001: `"Class inherits thread definition from multiple supertypes"`

3002: `"Circular class hierarchy detected: <name>"`

3003: `"Undefined superclass: <supername>"`

3004: `"Superclass name is not a class: <supername>"`

3005: `"Overriding a superclass member of a different kind: <member>"`

3006: `"Overriding definition reduces visibility"` This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3007: `"Overriding member incompatible type: <member>"`

3008: `"Overloaded members indistinguishable: <member>"`

3009: `"Circular class hierarchy detected: <class>"`

3010: `"Name <name> is ambiguous"`

3011: `"Name <name> is multiply defined in class"`

3012: `"Type <name> is multiply defined in class"`

3013: `"Class invariant is not a boolean expression"`

3014: `"Expression is not compatible with type bind"`

3015: "Set/seq bind is not a set/seq type?"

3016: "Expression is not compatible with set/seq bind"

3017: "Duplicate definitions for <name>"

3018: "Function returns unexpected type"

3019: "Function parameter visibility less than function definition"
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3020: "Too many parameter patterns"

3021: "Too few parameter patterns"

3022: "Too many curried parameters"

3023: "Too many parameter patterns"

3024: "Too few parameter patterns"

3025: "Constructor operation must have return type <class>"

3026: "Constructor operation must have return type <class>"

3027: "Operation returns unexpected type"

3028: "Operation parameter visibility less than operation definition"
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3029: "Function returns unexpected type"

3030: "Function parameter visibility less than function definition"
This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3031: "Unknown state variable <name>"

3032: "State variable <name> is not this type"

3033: "Polymorphic function has not been instantiated: <name>"

3034: "Function is already instantiated: <name>"

3035: "Operation returns unexpected type"

3036: "Operation parameter visibility less than operation definition" This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3037: "Static instance variable is not initialized: <name>"

3038: "<name> is not an explicit operation"

3039: "<name> is not in scope"

3040: "Cannot put mutex on a constructor"

3041: "Duplicate mutex name"

3042: "<name> is not an explicit operation"

3043: "<name> is not in scope"

3044: "Duplicate permission guard found for <name>"

3045: "Cannot put guard on a constructor"

3046: "Guard is not a boolean expression"

3047: "Only one state definition allowed per module"

3048: "Expression does not return a value"

3049: "Thread statement/operation must not return a value"

3050: "Type <name> is infinite"

3051: "Expression does not match declared type"

3052: "Value type visibility less than value definition" This error message typically are caused by using a more restrictive access modifier (or none which is interpreted as private) at this place compared to for example an inherited definition.

3053: "Argument of 'abs' is not numeric"

3054: "Type <name> cannot be applied"

3055: "Sequence selector must have one argument"

3056: "Sequence application argument must be numeric"

3057: "Map application must have one argument"

3058: "Map application argument is incompatible type"

3059: "Too many arguments"

3060: "Too few arguments"

3061: "Inappropriate type for argument <n>"

3062: "Too many arguments"

3063: "Too few arguments"

3064: "Inappropriate type for argument <n>"

3065: "Left hand of <operator> is not <type>"

3066: "Right hand of <operator> is not <type>"

3067: "Argument of 'card' is not a set"

3068: "Right hand of map 'comp' is not a map"

3069: "Domain of left should equal range of right in map 'comp'"

3070: "Right hand of function 'comp' is not a function"

3071: "Left hand function must have a single parameter"

3072: "Right hand function must have a single parameter"

3073: "Parameter of left should equal result of right in function 'comp'"

3074: "Left hand of 'comp' is neither a map nor a function"

3075: "Argument of 'conc' is not a seq of seq"

3076: "Argument of 'dinter' is not a set of sets"

3077: "Merge argument is not a set of maps"

3078: "dunion argument is not a set of sets"

3079: "Left of '<-:' is not a set"

3080: "Right of '<-:' is not a map"

3081: "Restriction of map should be set of <type>"

3082: "Left of '<:' is not a set"

3083: "Right of '<:' is not a map"

3084: "Restriction of map should be set of <type>"

3085: "Argument of 'elems' is not a sequence"

3086: "Else clause is not a boolean"

3087: "Left and right of '=' are incompatible types"

3088: "Predicate is not boolean"

3089: "Predicate is not boolean"

3090: "Unknown field <name> in record <type>"

3091: "Unknown member <member> of class <class>"

3092: "Inaccessible member <member> of class <class>"

3093: "Field <name> applied to non-aggregate type"

3094: "Field #<n> applied to non-tuple type"

3095: "Field number does not match tuple size"

3096: "Argument to floor is not numeric"

3097: "Predicate is not boolean"

3098: "Function value is not polymorphic"

3099: "Polymorphic function is not in scope"

3100: "Function has no type parameters"

3101: "Expecting <n> type parameters"

3102: "Parameter name <name> not defined"

3103: "Function instantiation does not yield a function"

3104: "Argument to 'hd' is not a sequence"

3105: "<operation> is not an explicit operation"

3106: "<operation> is not in scope"

3107: "Cannot use history of a constructor"

3108: "If expression is not a boolean"

3109: "Argument to 'inds' is not a sequence"

3110: "Argument of 'in set' is not a set"

3111: "Argument to 'inverse' is not a map"

3112: "Iota set/seq bind is not a set/seq"

3113: "Unknown type name <name>"

3114: "Undefined base class type: <class>"

3115: "Undefined class type: <class>"

3116: "Argument to 'len' is not a sequence"

3117: "Such that clause is not boolean"

3118: "Predicate is not boolean"

3119: "Map composition is not a maplet"

3120: "Argument to 'dom' is not a map"

3121: "Element is not of maplet type"

3122: "Argument to 'rng' is not a map"

3123: "Left hand of 'munion' is not a map"

3124: "Right hand of 'munion' is not a map"

3125: "Argument of mk_<type> is the wrong type"

3126: "Unknown type <type> in constructor"

3127: "Type <type> is not a record type"

3128: "Record and constructor do not have same number of fields"

3129: "Constructor field <n> is of wrong type"

3130: "Modifier for <tag> should be <type>"

3131: "Modifier <tag> not found in record"

3132: "mu operation on non-record type"

3133: "Class name <name> not in scope"

3134: "Class has no constructor with these parameter types"

3135: "Class has no constructor with these parameter types"

3136: "Left and right of '<>' different types"

3137: "Not expression is not a boolean"

3138: "Argument of 'not in set' is not a set"

3139: "Left hand of <operator> is not ordered"

3140: "Right hand of <operator> is not ordered"

3141: "Right hand of '++' is not a map"

3142: "Right hand of '++' is not a map"

3143: "Domain of right hand of '++' must be nat1"

3144: "Left of '++' is neither a map nor a sequence"

3145: "Argument to 'power' is not a set"

3146: "Left hand of <operator> is not a set"

3147: "Right hand of <operator> is not a set"

3148: "Left of ':->' is not a map"

3149: "Right of ':->' is not a set"

3150: "Restriction of map should be set of <type>"

3151: "Left of ':>' is not a map"

3152: "Right of ':>' is not a set"

3153: "Restriction of map should be set of <type>"

3154: "<name> not in scope"

3155: "List comprehension must define one ordered bind variable"

3156: "Predicate is not boolean"

3157: "Left hand of '$\hat{}$' is not a sequence"

3158: "Right hand of '$\hat{}$' is not a sequence"

3159: "Predicate is not boolean"

3160: "Left hand of '\' is not a set"

3161: "Right hand of '\' is not a set"

3162: "Left and right of '\' are different types"

3163: "Left hand of <operator> is not a set"

3164: "Right hand of <operator> is not a set"

3165: "Left and right of intersect are different types"

3166: "Set range type must be an number"

3167: "Set range type must be an number"

3168: "Left hand of <operator> is not a set"

3169: "Right hand of <operator> is not a set"

3170: "Map iterator expects nat as right hand arg"

3171: "Function iterator expects nat as right hand arg"

3172: "'**' expects number as right hand arg"

3173: "First arg of '**' must be a map, function or number"

3174: "Subsequence is not of a sequence type"

3175: "Subsequence range start is not a number"

3176: "Subsequence range end is not a number"

3177: "Left hand of <operator> is not a set"

3178: "Right hand of <operator> is not a set"

3179: "Argument to 'tl' is not a sequence"

3180: "Inaccessible member <name> of class <name>"

3181: "Cannot access <name> from a static context"

3182: "Name <name> is not in scope"

3183: "Exported function <name> not defined in module"

3184: "Exported <name> function type incorrect"

3185: "Exported operation <name> not defined in module"

3186: "Exported operation type does not match actual type"

3187: "Exported type <type> not defined in module"

3188: "Exported value <name> not defined in module"

3189: "Exported type does not match actual type"

3190: "Import all from module with no exports?"

3191: "No export declared for import of type <type> from <module>"

3192: "Type import of <name> does not match export from <module>"

3193: "No export declared for import of value <name> from <module>"

3194: "Type of value import <name> does not match export from <module>"

3195: "Cannot import from self"

3196: "No such module as <module>"

3197: "Expression matching set/seq bind is not a set/seq"

3198: "Type bind not compatible with expression"

3199: "Set/seq bind not compatible with expression"

3200: "Mk_ expression is not a record type"

3201: "Matching expression is not a compatible record type"

3202: "Record pattern argument/field count mismatch"

3203: "Sequence pattern is matched against <type>"

3204: "Set pattern is not matched against set type"

3205: "Matching expression is not a product of cardinality <n>"

3206: "Matching expression is not a set type"

3207: "Object designator is not an object type"

3208: "Object designator is not an object type"

3209: "Member <field> is not in scope"

3210: "Object member is neither a function nor an operation"

3211: "Expecting <n> arguments"

3212: "Unexpected type for argument <n>"

3213: "Operation <name> is not in scope"

3214: "Cannot call <name> from static context"

3215: "<name> is not an operation"

3216: "Expecting <n> arguments"

3217: "Unexpected type for argument <n>"

3218: "Expression is not boolean"

3219: "For all statement does not contain a set type"

3220: "From type is not numeric"

3221: "To type is not numeric"

3222: "By type is not numeric"

3223: "Expecting sequence type after 'in'"

3224: "If expression is not boolean"

3225: "Such that clause is not boolean"

3226: "Incompatible types in object assignment"

3228: "<name> is not in scope"

3229: "<name> should have no parameters or return type"

3230: "<name> is implicit"

3231: "<name> should have no parameters or return type"

3232: "<name> is not an operation name"

3233: "Precondition is not a boolean expression"

3234: "Postcondition is not a boolean expression"

3235: "Expression is not a set of object references"

3236: "Class does not define a thread"

3237: "Class does not define a thread"

3238: "Expression is not an object reference or set of object references"

3239: "Incompatible types in assignment"

3241: "Body of trap statement does not throw exceptions"

3242: "Map element assignment of wrong type"

3243: "Seq element assignment is not numeric"

3244: "Expecting a map or a sequence"

3245: "Field assignment is not of a record type"

3246: "Unknown field name, <name>"

3247: "Unknown state variable <name> in assignment"

3248: "Cannot assign to 'ext rd' state <name>"

3249: "Object designator is not a map, sequence, function or operation"

3250: "Map application must have one argument"

3251: "Map application argument is incompatible type"

3252: "Sequence application must have one argument"

3253: "Sequence argument is not numeric"

3254: "Too many arguments"

3255: "Too few arguments"

3256: "Inappropriate type for argument <n>"

3257: "Too many arguments"

3258: "Too few arguments"

3259: "Inappropriate type for argument <n>"

3260: "Unknown class member name, <name>"

3261: "Unknown field name, <name>"

3262: "Field assignment is not of a class or record type"

3263: "Cannot reference 'self' from here"

3264: "At least one bind cannot match set/seq"

3265: "At least one bind cannot match this type"

3266: "Argument is not an object"

3267: "Empty map cannot be applied"

3268: "Empty sequence cannot be applied"

3269: "Ambiguous function/operation name: <name>"

3270: "Measure <name> is not in scope"

3271: "Measure <name> is not an explicit function"

3272: "Measure result type is not a nat, or a nat tuple"

3273: "Measure not allowed for an implicit function"

3274: "External variable is not in scope: <name>"

3275: "Error clause must be a boolean"

3276: "Ambiguous names inherited by <name>"

3277: "Trace repeat illegal values"

3278: "Cannot inherit from system class <name>"

3279: "Cannot instantiate system class <name>"

3280: "Argument to deploy must be an object"

3281: "Arguments to duration must be nat"

3282: "Arguments to cycles must be nat"

3283: "System class constructor cannot be implicit"

3284: "System class can only define instance variables and a constructor"

3285: "System class can only define a default constructor"

3286: "Constructor cannot be 'async'"

3287: "Periodic/sporadic thread must have <n> argument(s)"

3288: "'-' expression must be numeric"

3289: "'+' expression must be numeric"

3290: "Argument to setPriority must be an operation"

3291: "Argument to setPriority cannot be a constructor"

3292: "Constructor is not accessible"

3293: "Asynchronous operation <name> cannot return a value"

3294: "Only one system class permitted"

3295: "Argument to 'reverse' is not a sequence"

3296: "Cannot use '" + typename + "' outside system class"

3297: "Cannot use default constructor for this class"

3298: "Cannot inherit from CPU"

3299: "Cannot inherit from BUS"

3300: "Operation <type> cannot be called from a function"

3301: "Variable <name> in scope is not updatable"

3302: "Variable <name> cannot be accessed from this context"

3303: "Measure parameters different to function"

3304: "Recursive function cannot be its own measure"

3305: "CPU frequency to slow: <speed> Hz"

3306: "CPU frequency to fast: <speed> Hz"

3307: "Errs clause is not bool -> bool"

3308: "Cannot mix modules and flat specifications"

3309: "Measure must not be polymorphic"

3310: "Measure must also be polymorphic"

3311: "Pattern cannot match"

3312: "Void operation returns non-void value"

3313: "Operation returns void value"

3314: "Map pattern is not matched against map type"

3315: "Matching expression is not a map type"

3316: "Expecting number in periodic/sporadic argument"

3317: "Expression can never match narrow type"

3318: "Measure's type parameters must match function's"

3319: "'in set' expression is always false"

3320: "'not in set' expression is always true"

3321: "Type component visibility less than type's definition"

3322: "Duplicate patterns bind to different types"

3323: "Overloaded operation cannot mix static and non-static"

3324: "Operation <name> is not static"

3325: "Mismatched compose definitions for <type>"

3326: "Constructor can only return 'self'"

3327: "Value is not of the right type"

3328: "Statement may return void value"

3329: "Abstract function/operation must be public or protected"

3330: "Cannot instantiate abstract class <name>"

3331: "obj_ expression is not an object type"

3332: "Object pattern cannot be used from a function"

3333: "Matching expression is not a compatible object type"

3334: "<name>is not a matchable field of class <class>"

3335: "Subset will only be true if the LHS set is empty"

3336: "Illegal use of RESULT reserved identifier"

3337: "Cannot call a constructor from here"

3350: "Polymorphic function has not been instantiated"

3351: "Type parameter '<name>' cannot be used here"

3352: "Exported <name> function has no type paramaters"

3353: "Exported <name> function type parameters incorrect"

3354: "Function argument must be instantiated"

Warnings from the type checker include:

5000: "Definition <name> not used"

5001: "Instance variable is not initialized: <name>"

5002: "Mutex of overloaded operation" This warning is provided if one defined a mutex for an operation that is defined using overloading. The users needs to be aware that all of the overloaded operations will now by synchronisation controlled by this constraint.

5003: "Permission guard of overloaded operation"

5004: "History expression of overloaded operation"

5005: "Should access member <member> from a static context"

5006: "Statement will not be reached"

5007: "Duplicate definition: <name>"

5008: "<name/location> hides <name/location>"

5009: "Empty set/sequence used in bind"

5010: "State init expression cannot be executed"

5012: "Recursive function has no measure" Whenever a recursive function is defined the user have the possibility defining a measure (i.e. a function that takes the same parameters as the recursive function and returns a natural number that should decrease at every recursive call). If such measures are included the proof obligation generator can provide proof obligations that will ensure termination of the recursion.

5014: "Uninitialized BUS ignored" This warning appears if one has defined a BUS that is not used.

5015: "LaTeX source should start with %comment, \document, \section or \subsection"

5016: "Some statements will not be reached"

5018: "Field has ':-' for type with eq definition"

5019: "Order of union member <name> will be overridden"

5020: "Equality of union member <name> will be overridden"

5021: "Multiple union members define eq clauses, <types>"

5022: "Strict: expecting semi-colon between exports"

5023: "Strict: expecting semi-colon between imports"

5024: "Strict: order should be imports then exports"

5025: "Strict: expecting 'exports all' clause"

5026: "Strict: order should be inv, eq, ord"

5027: "Strict: order should be inv, init"

5028: "Strict: expecting semi-colon between traces"

5029: "Strict: unexpected trailing semi-colon"

5030: "Annotation is not followed by bracketed sub-expression"

5031: "Strict: impure operation '<name>' cannot be called from here"

5032: "Cannot use 'threadid' in a functional context"

5033: "Cannot use 'new' in a functional context"

5034: "Cannot use 'time' in a functional context"

5037: "Function equality cannot be reliably computed"

Appendix E

Run-Time Errors

When using the interpreter/debugger it is possible to get run-time errors, even if there are no type checking errors. The possible errors are as follows:

VDMJ Error and Warning Messages

4000: "Cannot instantiate abstract class <class>"

4002: "Expression value is not in set/seq bind"

4003: "Value <value> cannot be applied"

4004: "No cases apply for <value>"

4005: "Duplicate map keys have different values"

4006: "Type <type> has no field <field>"

4007: "No such field in tuple: #<n>"

4008: "No such type parameter @<name> in scope"

4009: "Type parameter/local variable name clash, @<name>"

4010: "Cannot take head of empty sequence"

4011: "Illegal history operator: <#op>"

4012: "Cannot invert non-injective map"

4013: "Iota selects more than one result"

4014: "Iota does not select a result"

4015: "Let be st found no applicable bindings"

4016: "Duplicate map keys have different values: <domain>"

4017: "Duplicate map keys have different values: <domain>"

4018: "Maplet cannot be evaluated"

4019: "Sequence cannot extend to key: <index>"

4020: "State value is neither a <type> nor a <type>"

4021: "Duplicate map keys have different values: <key>"

4022: "mk_ type argument is not <type>"

4023: "Mu type conflict? No field tag <tag>"

4024: "'not yet specified' expression reached"

4025: "Map key not within sequence index range: <key>"

4026: "Cannot create post_op environment"

4027: "Cannot create pre_op environment"

4028: "Sequence comprehension pattern has multiple variables"

4029: "Sequence comprehension bindings must be numeric"

4030: "Duplicate map keys have different values: <key>"

4031: "First arg of '**' must be a map, function or number"

4032: "'is subclass responsibility' expression reached"

4033: "Tail sequence is empty"

4034: "Name <name> not in scope"

4035: "Object has no field: <name>"

4036: "ERROR statement reached"

4037: "No such field: <name>"

4038: "Loop, from <value> to <value> by <value> will never terminate"

4039: "Set/seq bind does not contain value <value>"

4040: "Let be st found no applicable bindings"

4041: "'is not yet specified' statement reached"

4042: "Sequence does not contain key: <key>"

4043: "Object designator is not a map, sequence, operation or function"

4045: "Object does not contain value for field: <name>"

4046: "No such field: <name>"

4047: "Cannot execute specification statement"

4048: "'is subclass responsibility' statement reached"

4049: "Value <value> is not in set/seq bind"

4050: "Value <value> is not in set/seq bind"

4051: "Cannot apply implicit function: <name>"

4052: "Wrong number of arguments passed to <name>"

4053: "Parameter patterns do not match arguments"

4055: "Precondition failure: <pre_name>" This error occurs if a pre-condition to a function or operation is violated.

4056: "Postcondition failure: <post_name>" This error occurs if a post-condition to a function or operation is violated.

4057: "Curried function return type is not a function"

4058: "Value <value> is not a nat1"

4059: "Value <value> is not a nat"

4060: "Type invariant violated for <type>"

4061: "No such key value in map: <key>"

4062: "Cannot convert non-injective map to an inmap"

4063: "Duplicate map keys have different values: <domain>"

4064: "Value <value> is not a nat1 number"

4065: "Value <value> is not a nat"

4066: "Cannot call implicit operation: <name>"

4067: "Deadlock detected"

4068: "Wrong number of arguments passed to <name>"

4069: "Parameter patterns do not match arguments"

4071: "Precondition failure: <pre_name>"

4072: "Postcondition failure: <post_name>"

4073: "Cannot convert type parameter value to <type>"

4074: "Cannot convert <value> to <type>"

4075: "Value <value> is not an integer"

4076: "Value <value> is not a nat1"

4077: "Value <value> is not a nat"

4078: "Wrong number of fields for <type>"

4079: "Type invariant violated by mk_ arguments"

4080: "Wrong number of fields for <type>"

4081: "Field not defined: <tag>"

4082: "Type invariant violated by mk_ arguments"

4083: "Sequence index out of range: <index>"

4084: "Cannot convert empty sequence to seq1"

4085: "Cannot convert tuple to <type>"

4086: "Value of type parameter is not a type"

4087: "Cannot convert <value> (<kind>) to <type>"

4088: "Set not permitted for <kind>"

4089: "Can't get real value of <kind>"

4090: "Can't get rat value of <kind>"

4091: "Can't get int value of <kind>"

4092: "Can't get nat value of <kind>"

4093: "Can't get nat1 value of <kind>"

4094: "Can't get bool value of <kind>"

4095: "Can't get char value of <kind>"

4096: "Can't get tuple value of <kind>"

4097: "Can't get record value of <kind>"

4098: "Can't get quote value of <kind>"

4099: "Can't get sequence value of <kind>"

4100: "Can't get set value of <kind>"

4101: "Can't get string value of <kind>"

4102: "Can't get map value of <kind>"

4103: "Can't get function value of <kind>"

4104: "Can't get operation value of <kind>"

4105: "Can't get object value of <kind>"

4106: "Boolean pattern match failed"

4107: "Character pattern match failed"

4108: "Sequence concatenation pattern does not match expression"

4109: "Values do not match concatenation pattern"

4110: "Expression pattern match failed"

4111: "Integer pattern match failed"

4112: "Quote pattern match failed"

4113: "Real pattern match failed"

4114: "Record type does not match pattern"

4115: "Record expression does not match pattern"

4116: "Values do not match record pattern"

4117: "Wrong number of elements for sequence pattern"

4118: "Values do not match sequence pattern"

4119: "Wrong number of elements for set pattern"

4120: "Values do not match set pattern"

4121: "Cannot match set pattern"

4122: "String pattern match failed"

4123: "Tuple expression does not match pattern"

4124: "Values do not match tuple pattern"

4125: "Set union pattern does not match expression"

4126: "Values do not match union pattern"

4127: "Cannot match set pattern"

4129: "Exit <value>"

4130: "Instance invariant violated: <inv_op>"

4131: "State invariant violated: <inv_op>"

4132: "Using undefined value"

4133: "Map range is not a subset of its domain: <key>"

4134: "Infinite or NaN trouble"

4135: "Cannot instantiate a system class"

4136: "Cannot deploy to CPU"

4137: "Cannot set operation priority on CPU"

4138: "Cannot set CPU priority for operation"

4139: "Multiple BUS routes between CPUs <name> and <name>"

4140: "No BUS between CPUs <name> and <name>"

4141: "CPU policy does not allow priorities"

4142: "Value already updated by thread <n>"

4143: "No such test number: <n>"

4144: "State init expression cannot be executed"

4145: "Time: <n> is not a nat1"

4146: "Measure failure: f(args), measure <name>, current <value>, previous <value>"

4147: "Polymorphic function missing @T"

4148: "Measure function is called recursively: <name>"

4149: "CPU frequency to slow: <speed> Hz"

4150: "CPU frequency to fast: <speed> Hz"

4151: "Cannot take dinter of empty set"

4152: "Wrong number of elements for map pattern"

4153: "Values do not match map pattern"

4154: "Cannot match map pattern"

4155: "Map union pattern does not match expression"

4156: "Cannot match map pattern"

4157: "Expecting +ive integer in periodic/sporadic argument <n>"

4158: "Period argument must be non-zero"

4159: "Delay argument must be less than the period"

4160: "Object <#n> is not running a thread to stop"

4161: "Cannot stop object <#n> on CPU <name> from CPU <name>"

4162: "The RHS range is not a subset of the LHS domain"

4163: "Cannot inherit private constructor"

4164: "Compose function cannot be restricted to <type>"

4165: "Cannot convert <type> to <type>"

4168: "Arguments do not match parameters: <name(params)>"

4170: "Cannot convert empty set to set1"

4171: "Values cannot be compared: <first>, <second>"

Appendix F

Categories of Proof Obligations

This appendix provides a list of the different proof obligation categories generated by Overture, and an explanation of the circumstances under which each category can be expected.

cases exhaustive: If a cases expression does not have an `others` clause it is necessary to ensure that the different case alternatives catch all values of the type of the expression used in the case choice.

finite map: If a type binding to a type that potentially has infinitely many elements is used inside a map comprehension, this proof obligation will be generated because all mappings in VDM must be finite.

finite set: If a type binding to a type that potentially has infinitely many elements is used inside a set comprehension, this proof obligation will be generated because all sets in VDM must be finite.

function apply: Whenever a function application is used you need to be certain that the list of arguments to the function satisfies the pre-condition of the function, assuming such a predicate is present.

function compose: When using a function composition (f comp g), this ensures that the precondition of g implies the precondition of f applied to the result of g.

function iteration: When using a function iteration, for the function we are iterating with, this ensures that the precondition on an argument implies the precondition on the result.

function parameter patterns: When using a pattern as a function parameter, this ensures that all values in the parameter type for the function can match the pattern.

function postcondition satisfiable: Whenever a function has a post condition this checks that the precondition of the function implies the post condition.

function satisfiability: For all implicit function definitions this proof obligation will be generated to ensure that it is possible to find an implementation satisfying the post-conditions for all arguments satisfying the pre-conditions.

legal map application: Whenever a map application is made you need to be certain that the argument is in the domain of the map.

legal sequence application: Whenever a sequence application is used you need to be certain that the argument is within the indices of the sequence.

let be st existence: Whenever a let-be-such-that expression/statement is used you need to be certain that at least one value will match the such-that expression.

map compose: When composing 2 maps, ensures that the range of map2 is a subset of the domain of map1.

map compatible: Mappings in VDM represent a unique relationship between the domain values and the corresponding range values. Proof obligations in this category are meant to ensure that such a unique relationship is guaranteed.

map iteration: When performing a map iteration, ensures the iteration count expression is either 0 or 1 or if it's greater than 1 then the map's range is a subset of its domain.

map sequence compatible: When defining a map with enumeration, ensures that any two equal elements in the domain map to the same element in the range.

map set compatible: When merging a set of maps, any two equal elements in the domains of each map map to the same element in the range.

non-empty sequence: This kind of proof obligation is used whenever non-empty sequences are required (eg. taking the head of a sequence)

non-empty set: This kind of proof obligation is used whenever non-empty sets are required.

non-zero: This kind of proof obligation is used whenever zero cannot be used (e.g. in division).

operation parameter patterns: When using a pattern as an operation parameter, ensures that all values in the operation parameter type can match the pattern.

operation post condition: Whenever an explicit operation has a post-condition there is an implicit proof obligation generated to remind the user that you have to ensure that the explicit body of the operation satisfies the post-condition for all possible inputs.

operation satisfiability: For all implicit operation definitions this proof obligation will be generated to ensure that it is possible to find an implementation satisfying the post-condition for all arguments satisfying the pre-conditions.

recursive function: This proof obligation makes use of the `measure` construct to ensure that a recursive function will terminate.

sequence modification: Whenever a sequence modification is used, this ensures the domain of the modification map is a subset of the indices of the sequence.

state invariant: If a state (including instance variables in VDM++) has an invariant, this proof obligation will be generated whenever an assignment is made to a part of the state.

subtype: This proof obligation category is used whenever it is not possible to statically detect that the given value falls into the subtype required.

tuple selection: This proof obligation category is used whenever a tuple selection expression is used to guarantee that the length of the tuple is at least as long as the selector used.

type compatibility: Proof obligations from this category are used to ensure that when constructing values of types that have an invariant actually live up to that invariant.

type invariant satisfiable: Proof obligations from this category are used to ensure that invariants for elements of a particular type are satisfied.

unique existence binding: The `iota` expression requires one unique binding to be present and that is guaranteed by proof obligations from this category.

value binding: When binding a value to a pattern, ensures that the resulting value matches the pattern.

while loop termination: This kind of proof obligation is a reminder to ensure that a while loop will terminate.

Appendix G

Mapping Rules between VDM++/VDM-RT and UML Models

Transformation Rule 1
VDM classes are mapped as the UML meta-class `Class`

Transformation Rule 2
The visibility of VDM instance variables, values, functions and operations are mapped as a *subset* of the UML enumeration `VisibilityKind` comprising public, private and protected.

Transformation Rule 3
VDM static is mapped as the `isStatic` property of the UML meta-class `Class`, `Property` or `Operation` respectively.

Transformation Rule 4
Data type definitions are mapped as the UML meta-class `Class` and are referenced, and thus nested, through the meta-attribute `nestedClassifier` of the owning class. Notice that this rule is not specified or implemented.

Transformation Rule 5
Instance variable and value definitions are mapped as the UML meta-class `Association`, if:

5 a: The type is an *object reference type*, or

5 b: The type is *not* a basic *data type* [Fitzgerald&05, p64,71].

Transformation Rule 6
Instance variable and value definitions are mapped as the UML meta-class `Property`, if the type is a *basic data type* [Fitzgerald&05, p71]. Instance variables and values are distinguished by the meta-attribute `isReadOnly`. Notice: rule 10 is an exception to this rule.

VDM concept	`Property::isReadOnly`
Instance variables	`false`
Values	`true`

Table G.1: The meta-attribute `isReadOnly` distinguishes instance variables and values

Transformation Rule 7
The initial value of instance variables and values definitions are mapped as the property `default` of the UML meta-class `Property`.

Transformation Rule 8
The VDM optional type is mapped to the properties `lower = 0` and `upper = 1` of the UML meta-class.

Transformation Rule 9
The VDM constructs set, seq and seq1 is mapped as the UML meta-class `Association` which may be decorated with a textual constraint defined by the meta-attribute `isOrdered`[1] in addition to a multiplicity at both ends. Table G.2 shows how the above-mentioned VDM constructs are mapped.

VDM construct	Ordered	Target class Multiplicity
set	`false`	`0..*`
seq	`true`	`0..*`
seq1	`true`	`1..*`

Table G.2: Transformation rules for VDM constructs modeling collections

Transformation Rule 10
The VDM constructs map and inmap are mapped as the UML meta-class `Association` with a qualifier. The domain is specified by the qualifier, which is located at the source class. The range is specified by the target class. Notice, that if the range is specified by a *basic type* it is mapped as a separate class. This is an exception to rule 6.

VDM construct	Qualifier end `isUnique`	Target class end `isUnique`
map	`false`	`true`
inmap	`true`	`true`

Table G.3: Transformation rules for VDM constructs modeling relationships between two sets.

Transformation Rule 11
A VDM class with a thread compartment is mapped as the UML meta-class `Class` with the meta-attribute `isActive` set to `true`.

Transformation Rule 12
A VDM class with the keyword is subclass of followed by class-names is mapped as the UML meta-class `Generalization`, with the attributes `general` and `specific` referencing the superclass and subclass, respectively. More than one subclass results in more than one instance of `Generalization`.

Transformation Rule 13
A VDM class with the keyword is subclass responsibility as a function or operation body is mapped as the UML meta-class `Class` with the meta-attribute `isAbstract` set to `true`.

Transformation Rule 14
A VDM generic class maps to the UML meta-class `Class` with the attribute `templateSignature` referencing a set of `TemplateParameter` having the name property set to the name of the parameter.

Transformation Rule 15
A VDM operation and function are mapped to the UML meta-class `Operation` where the property `isQuery` determine whether the `Operation` represents a VDM function or operation:

true

- for a function.

false

- for a operation.

The return type of a function and operation is mapped collectively as the property `type` and the multiplicity[2]of the `Operation` meta-class. The parameters of the operation or function is mapped to the UML meta-class `Parameter` represented as the property `ownedParameters` of the `Operation` meta-class.
The name and type of a VDM parameter are mapped to the property `name`, `type` and the multiplicity[2] of the `Parameter` meta-class.

Appendix H

Using VDM Values in Java

Integration between Overture and Java code can be established, either by writing native libraries in Java that can be called from VDM, or by giving a Java program overall control of a VDM model by making calls to that model as a user interacts with a GUI.

In both cases, internal VDM values have to be handled by Java -– either because they are passed as arguments to a Java library, and returned as results to VDM, or because they are returned from a VDM model evalution to a controlling Java program.

This appendix describes the internal class hierarchy used by Overture to represent internal VDM model values, and describes how a Java program can convert these to Java values (int, long, String etc.) as well as creating internal values for returning to the VDM model (e.g. as the return value of library methods).

H.1 The Value Class Hierarchy

All internal VDM values in Overture are held by instances of the `Value` class with the fully qualified name, `org.overture.interpreter.values.Value`. The `Value` class itself is abstract, but subclasses can be instantiated to represent any VDM value, such as a "seq of char", "nat1'" or a value of an arbitrarily complex type. The hierarchy is shown in Figure H.1.

Generally, the name of the `Value` subclass for a VDM type is on the form <*name*>`Value`, for example `BooleanValue` or `SeqValue`.

The following sections describe how to obtain Java values from a `Value` object, and how to create `Value` objects from basic Java values (or iteratively from other values).

H.2 Primitive Values

Most primitive VDM types have subclasses with simple constructors that take a Java primitive type as an argument:

```
public BooleanValue(boolean value)
public CharacterValue(char value)
```

```
public RealValue(double value) throws Exception
public RationalValue(double value) throws Exception
public IntegerValue(long value)
public NaturalValue(long value) throws Exception
public NaturalOneValue(long value) throws Exception
public QuoteValue(String value)
public NilValue()
```

The constructors that throw exceptions are the ones for which some Java value does not match the VDM type concerned. For example, a `RealValue` or `RationalValue` cannot take the Java values `Double.NaN` or `Double.POSITIVE_INFINITY` as a constructor argument. Similarly, `NaturalValue` and `NaturalOneValue` cannot take a negative long as an argument.

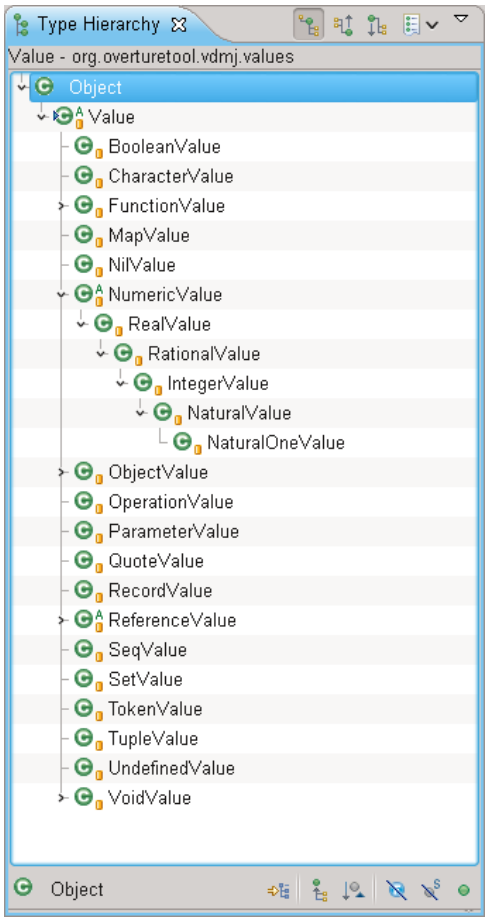

Figure H.1: Java Value Hierarchy

Note that a `QuoteValue` is constructed with a string. This is simply the string value that would appear between angle brackets in VDM, for example `<FAIL>` would be constructed with the Java string `"FAIL"`.

To convert a VDM value into a Java value, the `Value` class provides a number of conversion methods, each of which returns the corresponding Java primitive value, or throws an exception if the conversion cannot be made for the VDM type concerned:

```
public boolean boolValue(Context ctxt) throws ValueException
public char charValue(Context ctxt) throws ValueException
public double realValue(Context ctxt) throws ValueException
public double ratValue(Context ctxt) throws ValueException
public long intValue(Context ctxt) throws ValueException
public long natValue(Context ctxt) throws ValueException
public long nat1Value(Context ctxt) throws ValueException
public String quoteValue(Context ctxt) throws ValueException
```

Note that all of these conversion functions take a `Context` parameter as argument and potentially throw a `ValueException`. The `Context` parameter is used internally by Overture and represents the call stack during the evaluation of an expression. This parameter can be set to null when using these methods in Java code outside Overture. A `ValueException` is thrown if the VDM value cannot be converted into the Java type requested. For example, calling `booleanValue` on a `RealValue` object will raise a `ValueException` with the message text `"Can't get bool value of real"`.

H.3 Sets, Sequences and Maps

VDM allows primitive types to be built into more complex aggregations and collections, and these can also be converted to and from Java types, though the process is a little more involved. Three classes are provided to assist with this conversion: `ValueSet`, `ValueList` and `ValueMap` (all within the same `org.overture.interpreter.values` package). These classes represent, respectively, a Java `Set`, `List` and `Map` of VDM values:

```
public class ValueSet extends Vector<Value>
public class ValueList extends Vector<Value>
public class ValueMap extends LinkedHashMap<Value, Value>
```

Note that the `ValueSet` class is actually based on a Java `Vector`, not a Java `Set` type, though the class does have set semantics (no duplicates). This is an implementation detail and allows Overture to permute set orderings in certain circumstances.

These three classes have obvious constructors, and allow `Values` (or collections of them) to be added to the collection subsequently, using standard Java collection methods:

```
public ValueSet()
public ValueSet(int size)
public ValueSet(ValueSet from)
public ValueSet(Value v)

public ValueList()
public ValueList(ValueList from)
public ValueList(Value v)
```

```
public ValueList(int size)

public ValueMap()
public ValueMap(ValueMap from)
public ValueMap(Value k, Value v)
```

Using these three helper classes, it is now possible to create VDM set, sequence and map values, using constructors of the `SetValue`, `SeqValue` and `MapValue` classes:

```
public SetValue()
public SetValue(ValueSet values)

public SeqValue()
public SeqValue(ValueList values)
public SeqValue(String s)

public MapValue()
public MapValue(ValueMap values)
```

Note that there is a special constructor for `SeqValue` that takes a Java string. This creates a VDM "seq of char", but without the need to create a `ValueList` with `CharacterValue`s.

If the `ValueList` (or another) collection passed to these constructors contains a mixture of VDM types -– i.e. a mixture of VDM `Value` subclasses, such as a `BooleanValue` and a `NaturalOneValue` -– then the type of the constructed VDM value is the union of the various types passed, in this example "seq of (bool | nat1)". If this VDM type is not compatible with the use of a corresponding value in the VDM model a dynamic type exception occurs when the value is processed by the model.

Lastly, as before, to get the primitive Java values of a VDM collection, the following methods are provided:

```
public ValueList seqValue(Context ctxt) throws ValueException
public String stringValue(Context ctxt) throws ValueException
public ValueSet setValue(Context ctxt) throws ValueException
public ValueMap mapValue(Context ctxt) throws ValueException
```

Note that, as with the `SeqValue` constructor, there is a special method to return a Java String from a "seq of char" `SeqValue`, rather than a `ValueList` of `CharacterValue`s. As before, if the `Value` being used is not a sequence, set or map, then these methods will throw a `ValueException`.

H.4 Other Types

The sections above describe how to create or deconstruct simple VDM values in Java as well as simple collections of these. The remainder of this section describes the unusual cases, for more sophisticated types.

H.4.1 Function values

Overture has an internal `FunctionValue` class used for holding values of functions (e.g. the value of a "lambda" expression or the value of a function defined within a module). But as far as Java is concerned, these values are opaque -– there is no equivalent Java construct, and the only way to evaluate a VDM function is to let Overture perform that evaluation. Similarly, Java cannot construct a `FunctionValue`.

The only operation that Java can reasonably perform with a `FunctionValue` is to create a composite function (eg. "`f1 comp f2`" in VDM) or a function iteration (e.g. "`f ** 3`" in VDM) using existing `FunctionValues`. In order to do this, there are two subclasses of `FunctionValue`, called `CompFunctionValue` and `IterFunctionValue`, the constructors for which are as follows:

```
public CompFunctionValue(FunctionValue f1, FunctionValue f2)
public IterFunctionValue(FunctionValue function, long count)
```

These both create new `FunctionValues`, which when evaluated by Overture act as the composition and iteration of the arguments, respectively.

There is a method for obtaining a `FunctionValue` from a `Value`, but note that this is not an internal Java value (unlike other `Value` methods, like `realValue`). It is used as a more convenient way of casting the `Value` to a `FunctionValue`.

```
public FunctionValue functionValue(Context ctxt)
```

H.4.2 Object Values

When VDM++ and VDM-RT create new objects using the "new" operator, the resulting values are held as `ObjectValues` in Overture. These are complex types that involve function and operation definitions for the object as well as any type, value, sync, thread or traces sections defined. Therefore `ObjectValues` are really opaque to Java and cannot be used directly.

Like for `FunctionValue`, the `ObjectValue` class has a method for converting a `Value` into an `ObjectValue`:

```
public ObjectValue objectValue(Context ctxt)
```

H.4.3 Record Values

A VDM record is just a collection of typed field values. A `RecordValue` can be obtained from a `Value` using the following method, which returns a `RecordValue` rather than some other Java representation:

```
public RecordValue recordValue(Context ctxt)
```

To get individual field values from a `RecordValue`, two more Java helper types have to be introduced, called `FieldMap` and `FieldValue`. A `FieldValue` has the following constructor, and represents a record field:

```
public FieldValue(String name, Value value, boolean comparable)
```

The `comparable` argument indicates whether this field is used in the value comparison between record values. A field declared with "–" in VDM would have a false argument, but normally this argument would be true, and the value must match the record type being used. `FieldValues` are added to a `FieldMap`, which is just a Java `List` of `FieldValues`.

So given a `RecordValue`, its `FieldMap` can be obtained from a public final field in the object, called `fieldMap`[1], and from there, individual `FieldValues` can be accessed -– e.g. `fieldMap.get(0).name` and `fieldmap.get(0).value`.

To create a `RecordValue`, the record type is obtained from the `RemoteInterpreter`:

```
type = remoteInterpreter.getInterpreter().findType(typename)
```

The type is then passed to the `RecordValue` constructor, along with a `FieldMap` or a `ValueList` (of the fields in order).

```
public RecordValue(RecordType type,
                   ValueList values,
                   Context ctxt)
public RecordValue(RecordType type,
                   FieldMap mapvalues,
                   Context ctxt)
```

The `Context` parameter is needed to allow records with invariants to check the invariant before the value is constructed. Note that currently, record types with an invariant cannot be constructed in Java. The `Context` parameter can be passed as null from Java.

The caller is responsible for passing field values that match their expected type. If they do not match, Overture throws a dynamic type exception for subsequent evaluations.

H.4.4 Token Values

Token values are simply wrappers for normal VDM values, reflecting the way they are created in VDM, like `mk_token("hello")`, which would be a wrapper for a "seq of char". There is no special way of getting a `TokenValue` from a `Value`, other than casting it. Having casted the `Value`, the wrapped value can be obtained from the public final `Value` field called "`value`".

Constructing a `TokenValue` is just a matter of passing the `Value` required:

```
public TokenValue(Value exp)
```

[1] Really this ought to have a get method.

H.4.5 Tuple Values

A `TupleValue` in Overture is a wrapper for a `ValueList`. The following method and constructor can be used like one would expect:

```
public TupleValue(ValueList argvals)
public ValueList tupleValue(Context ctxt)
```

H.4.6 Invariant Values

A VDM type can be given a name and an invariant, e.g. when wrapping a primitive type without an invariant. Overture has a separate `Value` subclass for values of such types that simply combine the primitive `Value` with a `FunctionValue` for the invariant. However, as with `RecordValues` (which can also have invariants), it is not currently possible to create `InvariantValues` in Java for types that have an invariant.

For types without an invariant, the constructor is as follows:

```
public InvariantValue(NamedType type, Value value, Context ctxt)
```

The `NamedType` is obtained in a similar way to the `RecordType` above, using the `RemoteInterpreter`. Note that the caller is responsible for passing a `Value` that matches the expected type. If they do not match, Overture will throw a dynamic type exception for subsequent evaluations.

H.4.7 Void Values

Operations which do not return a value in VDM (i.e. `==> ()`) return an instance of `VoidValue` in Java. The constructor has no arguments.