Overture – Open-source Tools for Formal Modelling TASKS
April 2013

Overture Task Suggestions

by

Peter Jørgensen   Luis Diogo Couto   Rasmus Lauriten

ABSTRACT

The success of Overture has provided the community with a dependable tool. To bring state of the art formal methods to this community Overture becomes a platform for a variety of formal methods tools. Therefore, its software architecture needs constantly to support extensibility and maintainability. This document discusses such architectural concerns as well as suggests several tasks and improvements to be considered for future versions of the tool.

Chapter 1

Code

1.1 Code Style

A strong asset with maintainable code is normalisation such that developers are familiar with the code-style being used. The first suggestion is thus to adopt a code-style for Overture that at least makes it follow the lines stipulated in `http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html`.

Suggestion 1.1.1 *Adopt a code-style for Overture, using those adopted by Google is an option.* *`http://code.google.com/p/java-coding-standards/wiki/CodeOrganisation`*

It is important that the code-style is advocated by all community members and developers. It should not only be document but also be explicitly visible in the code.

Suggestion 1.1.2 *Advocate code-style and make it visible in the source. Consider periodic reviews of core components.*

A central part of good code-style is documentation. All core components in Overture should be well documented in the source-code such that a coherent understanding of the code is obtainable from the source files alone.

Suggestion 1.1.3 *Thorough documentation of the code in the source files should be available, including examples.*

However, knowing how classes-work by them self is typically not enough to understand a software project as complex as Overture. Thus a description of the intended interplay between classes and which tasks they perform is needed. A well scoped package typically has a public factory class or similar point of access.

Suggestion 1.1.4 *Describe the intended interplay between* public *classes/interfaces and intended use of a coherent package in its public entry point classes.*

1.2 AST structure validation and bug fixing (PJ)

The AST provides support for things like finding the nearest ancestor of a node which is a sub class of a given type. For example, it is possible to find the class definition of a given type in the following way:

```
type.getAncestor(SClassDefinition.class)
```

Only few Overture plugins use this kind of functionality and there is no proper testing of it. Therefore, it would be a good place to look for bugs. This task suggests writing code that validates the structure of the AST. For example, one could write a visitor checking if the parent of each node has been properly set, i.e. not `null`. Similarly, other visitors can be made to check if certain properties hold for the AST.

Chapter 2

Functionality

The following sub-sections suggest development tasks for the Overture platform.

2.1 Quick interpreter improvements

Currently the error messages given by the Quick Interpreter are not very informative. For example evaluating the expression `1 + true` yields *"Fatal error"*, wheres the error message would be *"Right hand of* + *is not numeric. Actual: bool"* when using the VDM editor. It would be useful for the Quick Interpreter to show useful error messages like this. In addition, the Quick Interpreter could be extended to include for instance:

- A feature for clearing the screen.
- The possibility writing expressions/statements consisting of multiple lines.

2.2 Template improvements

Currently it is possible to use templates for operations, functions cases expressions etc. in the VDM editor by for instance writing *"ope"* followed by the `Ctrl+space` command. If the user chooses the explicit operation template the following is generated:

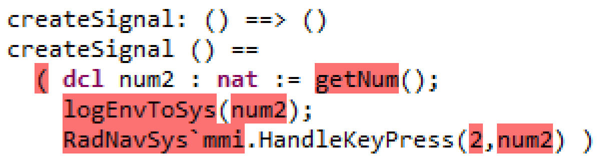
```
operationName (parameterNames) == statements;
```

Eclipse then assists the user in filling out the template, which is useful for a novice user. The drawback is that every element of the template must be filled out by the user manually. A more experienced user might prefer something which type checks right away. For example, an explicit operation on the form below may require less manual work by the user:

```
public op1 : () ==> () op1 () == skip;
```

2.3 Coverage coloring

Generated coverage is useful for getting an overview of which parts of a model that have been executed. However, it is possible to find situations where the coverage coloring could be improved. Take for example the generated coverage in figure 2.1.



```
createSignal: () ==> ()
createSignal () ==
  ( dcl num2 : nat := getNum();
    logEnvToSys(num2);
    RadNavSys`mmi.HandleKeyPress(2,num2) )
```

Figure 2.1: Coverage of an operation which has not been executed

It seems strange that only the first "(" of the block statement is colored. It may be more appropriate not to color the parentheses of a block statement at all. In addition, the `HandleKeyPress` operation is not colored in red. This task suggests finding problems with the coverage coloring and improving on this feature.

2.4 Improving robustness of the type checker

Consider the model below with recursive types.

```
types

public B = C | nat1; public C = B | nat1;

end A
```

Using this model we can ask if the union type `C` is a quote type in the following way:

```
PTypeAssistantTC.isType(C, AQuoteType.class)
```

Currently this causes the `isType` method to call itself recursively until it fails due to a stack overflow error. This task suggests improving the `isType` method to provide robustness for invocations like this.

2.5 Improving the Overture debugging features

This task suggests improving debugging in Overture by for instance making it possible to:

- Set break points that will suspend threads at permission predicates.
- Inspect the value of history counters during debugging.
- Use the debugging expression evaluator for all kinds of expressions.
  - For example, evaluating the expression **`dom`** `someMap` currently evaluates to “set” instead of actually showing what is contained in the set.
  - etc.

The documentation example model of the POP3 protocol would serve as a good starting point for this task.

2.6 RT Trace Viewer improvements

When executing a VDM-RT model, a series of events are logged and timestamped by a component of the Overture tool, called the RT Logger. During execution, these events are triggered by various actions such as function calls, object creations and thread activations. These events are logged to a text file and a binary file with .rt and .rtbin extensions, respectively. The text file can be inspected by the user using a text editor, while the RT Trace Viewer plugin enables graphical visualization of the binary trace file. This can be used for analyzing timing requirements.

This task suggests a series of nice-to-have features and optimization tasks identified by the plugin authors Martin Askov Andersen, Mads von Qualen and Peter Jørgensen:

Suggestions for nice to-have-features:

- A total overview of events for easy scrolling.
- Make “Move Next” and “Move Previous” jump to next/previous visible event instead of “Next” in terms of time.
- The generated conjecture file should be placed in same folder as the .rtbin file.
- The generated conjecture file should contain additional information which will enable a more precise placement of the conjecture violation circle in the RT log viewer. Preferably by supporting the concepts of relative time and absolute time like in the NextGen data structure. This makes it possible to infer the order of events happening at the same point in time.
- Maybe the best solution for conjectures would be to integrate the functionality with the NextGen data structure?

Optimizations:

- Change “jump in time” model. Currently we draw/parse everything for all CPUs when a new time in the future is selected. Introduce some sort of history model to save object states when they have been parsed.
- Refactor TraceFileRunner to move event-parsing (sorting of CPUs etc.) out.
- Refactor eventhandler base to move conjecture handling out.
- Cleanup “update CPU” hack in TraceFileRunner.

Chapter 3

Architecture

3.1 Migration to Interfaces

The Problem:

At the moment, various elements used to construct the AST are defined as classe. For example: LexLocation. When we extend Overture, this can cause quite a few problems. The COMPASS extensions have slightly different requirements for these elements so we have to redefine them. But because we depend on Overture, the original versions are also available to us. This leads to multiple versions of the same class. Worse, we get the packages split across multiple bundles (the original versions in the overture.ide.core bundle and the changed versions in compassresearch.ide.cml.core bundle). In general, OSGi does not cope very well with this situaton. In fact, this was the cause of a very problematic java.lang.VerifyError bug in the COMPASS extension.

Proposed Solution:

The elements used to build the AST should be set up as interfaces. Overture implements its own version of these interfaces and the extensions can either reuse the Overture classes directly or implement their own versions as needed.

Ultimately, we should have all the classes in the preamble of the AST definition file set up as interfaces.

The first step towards this new set up is to define and create interfaces and convert the existing Overture classes over. LDC has begun this work when fixing the verify bug. At the moment, only 3 classes have been converted: LexIdentifierToken, LexNameToken and LexToken. For the most part, the conversion work isn't very difficult. Just time consuming, since there are a lot of small errors that need fixing.

3.2 Reorganize Packages

The package structure for Overture is used for internal modularisation which is conflicting with the scoping mechanisms in Java. Packages were never intended to be used that way. The proper way to have internal modularisation is through source folders.

Suggestion 3.2.1 *Reduce Overture to a few packages with high cohesion. The modularisation reflected in existing packages is good but should be realised through source folders.*

With a few well defined packages it becomes feasible to use package-scoping to create a narrow well scoped interface exposed by the Overture code base.

Suggestion 3.2.2 *Reduce scoping on classes to package scope where possible.*

3.3 OSGi Refactoring

The Problem:

Currently, the Overture (and also COMPASS) OSGi setup is mostly based on the Require-Bundle style. This means that each bundle imports the whole bundle for everything it needs. Use of require bundle is generally discouraged [1]. The recommended best practice is to use the Import-Package and Export-Package constructs. Migratation to this style is possible but would be greatly hindered by the current Overture package structure.

Proposed Solution:

After reducing the number of packages in Overture, the manifest files on each project can be changed to use the package constructs. This work is not particularly complex but will be rather time consuming since there are many packages that need to be refactored. Also, these kind of changes have a high potential for conflicts so special care must be taken with that aspect of the task.

Bibliography

[1] OSGi Community Wiki, *Require-Bundle*. `http://wiki.osgi.org/wiki/Require-Bundle`, *Retrieved April 2013*