April 17, 2013

Authors: Peter Jørgensen

Overture task suggestions

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

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.

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:

```
private operationName : parameterTypes ==> resultType
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;
```

AST structure validation and bug fixing

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.

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 1.

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

Figure 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.

Improving robustness of the type checker

Consider the model below with recursive types.

```
class A

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.

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.

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.

Traces

If you execute traces in the Combinatorial Testing perspective and the test invokes an operation with a failing precondition the verdict is *"nconclusive"*. This is how it should be. What seems strange, is that if you invoke a function with a failing precondition the verdict is *"failed"*. This task suggests changing this so that functions with failing preconditions are considered inconclusive as well.