Overture – Open-source Tools for Formal Modelling TR-2009-01
March 2009

User Manual for the Overture Combinatorial Testing Plug-in

by

Peter Gorm Larsen and Kenneth Lausdahl
Engineering College of Aarhus
Dalgas Avenue 2, DK-8000 Århus C, Denmark

Contents

ABSTRACT

This document provides the basic understanding of the Combinatorial Testing (CT) component from the Overture open source initiative enabling users to benefit from this plug-in. Combinatorial testing enables test automation of a huge collection of test cases which each follow a test template provided in the form of trace definitions. The trace definitions are formed as a kind of regular expression enabling testing of sequences of operation calls. In order to manage the large number of test cases generated, this tool contains a filtering feature that is able to avoid repeating the execution of test cases where the same prefix of operation calls already have resulted in a run-time error.

1 Introduction

This plug-in for the Overture open source tool initiative started off with the MSc thesis of Adriana Succena Santos [?]. This work is based on the work conducted on TOBIAS in the research group from Yves Ledru [?, ?, ?, ?]. Combinatorial testing has drawn attention from many other researchers and it is also connected to most of the work conducted on generating test cases from model checkers [?, ?, ?, ?]. However their work can be characterised as using symbolic evaluation whereas the work here is using evaluation with ordinary VDM values.

This tool supports the VDM++ language as it is supported by the Overture open source initiative[1]. It is possible to use this plug-in both together with the interpreter that is included in the Overture platform on top of eclipse as well as VDMTools [?].

An extension has been made to VDM++ enabling a section about trace definitions to appear inside each VDM++ class. This tool makes use of that extension. Briefly speaking it expands the trace definitions made in all classes into a collection of test cases. Each of these test cases essentially are a sequence of operations to be called after each other with different arguments. This means that each class that contains at least one trace definition will get test cases generated for each of these definitions. This process is illustrated with small examples below.

The basic principles behind the trace definitions is to expand patterns of operation calls to be tested. Those familiar with the "generative" shell expansion patterns in may think about this when we talk about expansion in the remaining part of this user manual. At a shell in Unix (or Linux) you could do the following:

```
$ echo abc{d,e,f}ghi
abcdghi abceghi abcfghi

$ echo {a,b,c}{d,e,f}
ad ae af bd be bf cd ce cf
```

which essentially generate all the strings that follow the pattern provided for the `echo` command. Note how the comma separated list inside the curly brackets are considered as alternative choices. We will see how to do something similar in the trace definitions in a VDM++ model.

2 A Guided Tour with a Small Example

In order to illustrate the combinatorial testing approach it makes sense to take a small example to illustrate its usage. The example used in this section is virtually copied from [?] where it was formulated in VDM-SL and JML. This is a small example with numbers of elements in buffers. In VDM++ it can be formulated as:

[1]This language is known as the Overture Modelling Language (OML) but since it is so close to VDM++ we will refer to VDM++ throughout this user manual.

```
class Buffers

instance variables

  b1 : nat := 0;
  b2 : nat := 0;
  b3 : nat := 0;

inv b1 + b2 + b3 <= 40 and b1 <= b2 and b2 <= b3 and b3 - b1 <= 15

operations

public Add: nat ==> ()
Add(x) ==
  if x + b1 < b2
  then b1 := b1 + x
  elseif b2 + x <= b3
  then b2 := b2 + x
  else b3 := b3 + x
pre x <= 5 and b1 + b2 + b3 + x <= 40
post b1 + b2 + b3 = b1~ + b2~ + b3~ + x;

public Remove: nat ==> ()
Remove(x) ==
  if x + b2 <= b3
  then b3 := b3 - x
  elseif x + b1 <= b2
  then b2 := b2 - x
  else b1 := b1 - x
pre x <= 5 and x <= b1 + b2 + b3
post b1 + b2 + b3 + x = b1~ + b2~ + b3~;

end Buffers
```

So essentially there are two operations that can modify the state of the class:

- `Add(x)` which increases the total number of elements of the system of a strictly positive number (`x`) (i.e. it adds `x` elements to the buffers; these elements are distributed in `b1`, `b2` and `b3`).

- `Remove(x)` decreases the total number of elements in the system of a strictly positive number (`x`) (i.e. it removes `x` elements from the buffers).

Note that the post-conditions for `Add` and `Remove` allow implementation freedom in the sense that different explicit formulations will be able to live up to the requirements. For illustration purposes the small VDM++ model listed above has an error in the explicit part of the `Remove` operation where one of the three buffers may get a negative value. In this model the specifier has also taken the time to supply post-conditions and thus this makes it easier to validate whether the explicit part lives up to the overall requirements.

Determining whether a given explicit algorithm satisfies the post-condition can be done with different kinds of techniques. Absolute certainty could be obtained by formally proving the satisfiability proof obligation generated for examples such as these. Alternatively an increasing amount of confidence can be gained in its correctness using different kinds of testing techniques. For a VDM model such as the one presented above it would be easy to manually invent a number of test cases and measure the test coverage obtained with this. However, it can be time consuming to manually define and execute such test cases. The combinatorial testing feature of Overture is meant to assist users with extensive tests in desired areas in a fashion that have resemblances with model checking.

The basic idea is that instead of defining test cases one by one the user can define trace definitions that act as a kind of test schema and it defines a set of test cases when it gets expanded. Essentially such trace definitions are formulated as bounded regular expressions involving invocations of operators for specific instances of classes for which testing is desired. Typically trace definitions are written in a class that is using the class that one wish to test (sometimes this is actually an extra class only used for test purposes). Let us start considering a very basic trace definition such as:

```
class UseBuffers

instance variables

  b : Buffers := new Buffers()

traces

S1: let x in set {1,...,5} in b.Add(x)
...

end UseBuffers
```

This basic trace definition is expanded to 5 test cases. These looks like:

1. `let x = 1 in b.Add(x)`
2. `let x = 2 in b.Add(x)`
3. `let x = 3 in b.Add(x)`
4. `let x = 4 in b.Add(x)`
5. `let x = 5 in b.Add(x)`

Semantically reach of these would correspond to directly instantiating the variable `x` with the number e.g. `b.Add(3)`. It is also worthwhile to note that where the semantics of loose expression such as the *let* inside `S1` above in the VDM interpreters (both for VDMTools and VDMJ) yield only one value, the CT tool here yields all possible values. Note also that each of these test cases must be executed in a scope where a default instance of the `UseBuffers` class is in context[2]. It is only worthwhile noting that special constructors cannot be used for the tested class.

Whenever the test cases are to be executed this should be carried out using an interpreter where the checking of additional predicates (invariants, pre and post-conditions) is enabled. Naturally this means that some of the test cases may yield run-time errors. If all operations have post-conditions defined and no run-time error is raised for a test case this means that it is likely that it is a successful test case. However, even in this case it could be that the post-condition was simply formulated incorrectly and thus the operation was doing something different than intended, but from a test automation perspective the verdict from running such a test case would have to be that it passed. If a run-time error occurs however, it is less clear whether the test case succeeded or not. In case the run-time error was caused by a violation of a post-condition a bug have definitely been discovered. However, if the run-time error instead was the violation of a pre-condition it depends whether it is a pre-condition that is called from the test case directly or a pre-condition called inside one of these operations that is violated. In such cases it is not possible to automatically determine whether the test case have succeeded or not. Each test case may lead to either a *pass*, *fail* or *inconclusive* verdict. These will be used in different situations:

Pass: This verdict is reached whenever all operation calls in a test case successfully execute without violating any invariants, pre- or post-conditions.

Inconclusive: This verdict is reached if an operation is called outside the type for which it is defined or the parameters in the operation call violates the pre-condition of the operation.

Fail: This verdict is reached if the explicit definition of an operation yields a run-time error or violates the post-condition of the operation in cases where the pre-condition have been satisfied and the operation have been called within the types where it has been defined. Whenever this verdict has been reached a problem with the VDM++ model have been detected and it should be corrected.

Since `Add` and `Remove` does not yield any result and VDM++ models may not always have post-conditions for all operations it may be convenient to also have a query operation like `getBuffers`:

```
class Buffers
...
public getBuffers: () ==> nat * nat * nat
getBuffers() ==
  return mk_(b1,b2,b3)

end Buffers
```

In order to be able to inspect the state of the object which may be valuable if the user also wished to inspect the results of executing selected test cases. Using this one could extend `S1` like:

[2]In VDMTools this is enabled by having a "`push` *classname*" command and in VDMJ this is enabled by a special `runtrace` operation enabled for all classes that have trace definitions but all of this is taken care of by the CT tool.

```
class UseBuffers

instance variables

  b : Buffers := new Buffers()

traces

S1: let x in set {1,...,5} in b.Add(x); b.getBuffers()
...

end UseBuffers
```

where we have used the "`;`" as a sequencing operator for traces and the result of this is that a sequence of operation calls is made in each test case (i.e. calling `getBuffers` on the `b` object after the call of `Add`).

```
class UseBuffers
...
S1': b.Add(1); (b.Add(2) | b.Add(3)); b.Add(4)

end UseBuffers
```

The "`|`" means that the traces on each side are considered as alternatives. For this trace definition two different test cases will be generated in a way similar to the `echo` example from section 1. Here we get:

```
S1'-TC1: b.Add(1); b.Add(2); b.Add(4)
S1'-TC2: b.Add(1); b.Add(3); b.Add(4)
```

In the same way iterations can be specified by using a repeat pattern. So if we for example made a trace definition like:

```
class UseBuffers
...
S1'': b.Add(1){1,3}

end UseBuffers
```

Then we will get the `b.Add(1)` repeated 1 to 3 times. This means that here we will get the followinhg test cases:

```
S1''-TC1: b.Add(1);
S1''-TC2: b.Add(1); b.Add(1)
S1''-TC2: b.Add(1); b.Add(1); b.Add(1)
```

It is also possible to make a combination of alternatives and repetitions in trace definitions and thereby obtain more expressiveness:

```
class UseBuffers
...
S2: b.Add(2);
   ((let x in set {1,...,5} in b.Add(x)) |
    (let y in set {1,3,5} in b.Remove(y))){1,2};
    b.getBuffers()
end UseBuffers
```

All traces here will start by `Add`'ing 2 followed by a more complex trace. Here the trace bindings will have (5+3) 8 different instantiations. As above, the "`|`" means that the traces on each side are considered as alternatives and finally the `{1,2}` part means that the trace definitions inside is repeated 1 to 2 times. `S2` is expanded into 8 + (8*8) = 72 test cases:

```
S2-TC1: b.Add(2); let x = 1 in b.Add(x); b.getBuffers()
...
S2-TC8: b.Add(2); let y = 5 in b.Remove(y); b.getBuffers()
S2-TC9: b.Add(2); let x = 1 in b.Add(x); let x = 1 in b.Add(x); b.getBuffers()
...
S2-TC72: b.Add(2); let y = 5 in b.Remove(y); let y = 5 in b.Remove(y);
         b.getBuffers()
```

Naturally a number of the test cases generated here will violate pre-conditions of some of the operations called. These are places where the combinatorial testing feature are able to optimize the executions using a filtering approach that we will come back to later. For this VDM++ model we finally define two more trace definitions:

```
class UseBuffers
...
S3: b.Add(5){7};
    ((let x in set {1,...,5} in b.Add(x)) |
     (let y in set {1,3,5} in b.Remove(y))){1,2};
    b.getBuffers()

S4: let x in set {1,...,5} in b.Add(x);
    ((let x in set {1,...,5} in b.Add(x)) |
     (let y in set {1,3,5} in b.Remove(y))){1,3};
    b.getBuffers()

end UseBuffers
```

The trace definition `S3` aims at testing the behaviour of the VDM++ model at the "limits" i.e. when the buffer system is quite full. The trace definition `S4` was built to produce lots of test cases with test sequencing (a kind of "brute force approach"). In section 4 below we will come back to this example in the demonstration of how the tool can be used to analyse VDM++ models such as this one.

3 The Use of the Trace Definition Syntax

The syntax for trace definitions are defined as:

traces definitions = ‘traces’, { named trace } ;

named trace = identifier, { ‘/’, identifier }, ‘:’, trace definition list ;

The naming of trace definitions (with the “/” separator) is used for indicating the paths that are used for generated argument files for test cases (`.arg`) and the corresponding result files (`.res`)[3].

trace definition list = trace definition term, { ‘;’, trace definition term } ;

So the “`;`” operator is used for indicating a sequencing relationship between its *trace definition term*’s.

trace definition term = trace definition
| trace definition term, ‘|’, trace definition ;

So the “|” operator is used for indicating alternative choices between trace definitions.

trace definition = trace core definition
| trace bindings, trace core definition
| trace core definition, trace repeat pattern
| trace bindings, trace core definition, trace repeat pattern ;

Trace definitions can have different forms and combinations:

- Core definitions which includes application of operations and bracketed trace expressions.
- Trace bindings where identifiers can be bound to values and in case of looseness (**`let`** *bind* **`in set`** *setexpr* **`in`** *expr*) this will give raise to multiple test cases generated.
- Trace repeat patterns which are used whenever repetition is desired.

trace core definition = trace apply expression
| trace bracketed expression ;

trace apply expression = identifier, ‘.’, identifier, ‘(’, expression list, ‘)’ ;

[3]Currently the full path names are however not supported in an Overture context but this is envisaged in the future.

Trace apply expressions are the most basic element in trace definitions. The identifier before the "`.`" indicate an object for with the operation (listed after the "`.`") is to be applied with a list of arguments (the expression list inside the brackets). Note that with the current syntax for trace definitions apply expressions are limited to this form `instid.opid(args)` so it is for example not at the moment possible to call an operation in the same class directly as `opid(args)`. Nor is it possible with the current syntax to make use of a particular operation in a superclass in case of multiple possible ones which in VDM++ is would be written as `` instid.clid`opid(args). `` In the current version it is also not allowed to call functions here directly, although that may be changed at some stage in the future.

```
trace repeat pattern  =  '*'
                      |  '+'
                      |  '?'
                      |  '{', numeric literal, '}'
                      |  '{', numeric literal, ',' numeric literal, '}' ;
```

The different kinds of repeat patterns have the following meanings:

- '*' means 0 to n occurrences (n is tool specific).
- '+' means 1 to n occurrences (n is tool specific).
- '?' means 0 or 1 occurrences.
- '{', n, '}' means n occurrences.
- '{', n, ',' m '}' means between n and m occurrences.

```
trace bracketed expression  =  '(', trace definition list, ')' ;

trace bindings  =  trace binding, { trace binding } ;

trace binding  =  'let', local definitions, { ',', local definition }, 'in'
               |  'let', bind, 'in'
               |  'let', bind, 'be', 'st', expression, 'in' ;
```

The syntax for these is taken from the VDM++ language manual [**?**].

4 The envisaged usage of this tool

When a user has defined a VDM++ model it is envisaged that trace definitions can be defined in selected classes indicating the scenarios that the user would like to have explored with test cases executing the VDM++ model. At first the trace definitions are expanded to a collection of test cases. Most users will then initially be interested in finding definitive errors in the model. This is most efficiently done by running all of the test cases. Then the tool will be able to list all the test cases that result in run-time errors that are not due to errors in the input data (i.e. the test cases that have the *fail* verdict).

For all these cases that fail there is a need to update the VDM++ model. Typically that is simply a matter of strengthening pre-conditions and/or invariants for types or instance variables. However, more serious errors may naturally also be detected. If none of these run-time errors are present the user can step though the test cases and the corresponding results one by one. For each of these it will be possible to accept or reject that the result is as expected. In this process it is possible to store the results as expected results. This can be used to form a regression test environment that can be used subsequently test the VDM++ model when further changes are made.

5 The graphical user interface

5.1 Installation

The Graphical User Interface (GUI) for the combinatorial testing plug-in is installed in Eclipse as an additional plug-in to Eclipse (requiring the main overture GUI first). Eclipse can be freely downloaded and installed from:

```
http://www.eclipse.org/downloads/
```

The Overture combinatorial testing plug-in can be installed from the update site[4]:

```
http://mt.lausdahl.com/CT/updatesite
```

If a version of The Overture Editor already is installed it can only be updated with the Overture combinatorial plug-in, if it is provided from the same update site as state above.

5.2 The Combinatorial Testing Perspective

When Eclipse is started up and an overture project is loaded the combinatorial testing perspective can be invoked using `Windows -> Open perspective -> Other -> Overture Combinatorial Testing` (see Figure 1). Then the layout will be like in Figure 2.

On the right-hand-side of Figure 2 there is a browser with the title "CT Overview" with the projects that contain all projects and all classes present in each project with a VDM++ model that contains trace definitions. At the othermost level it contains projects. At the second level one will find a list of all the classes that are included in the project that contains trace definitions. At the third level the trace definitions inside each class is present. Finally at the fourth level the test cases that each are expanded from a trace definition is present. The number of test cases is listed after the name of the trace definition.

[4]The release version of Overture combinatorial will later be available at: `http://www.overturetool.org/updatesite`

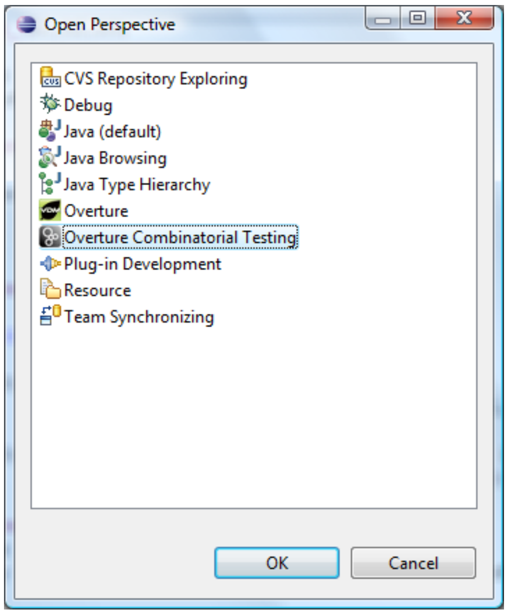


Figure 1: Select Combinatorial Testing Perspective

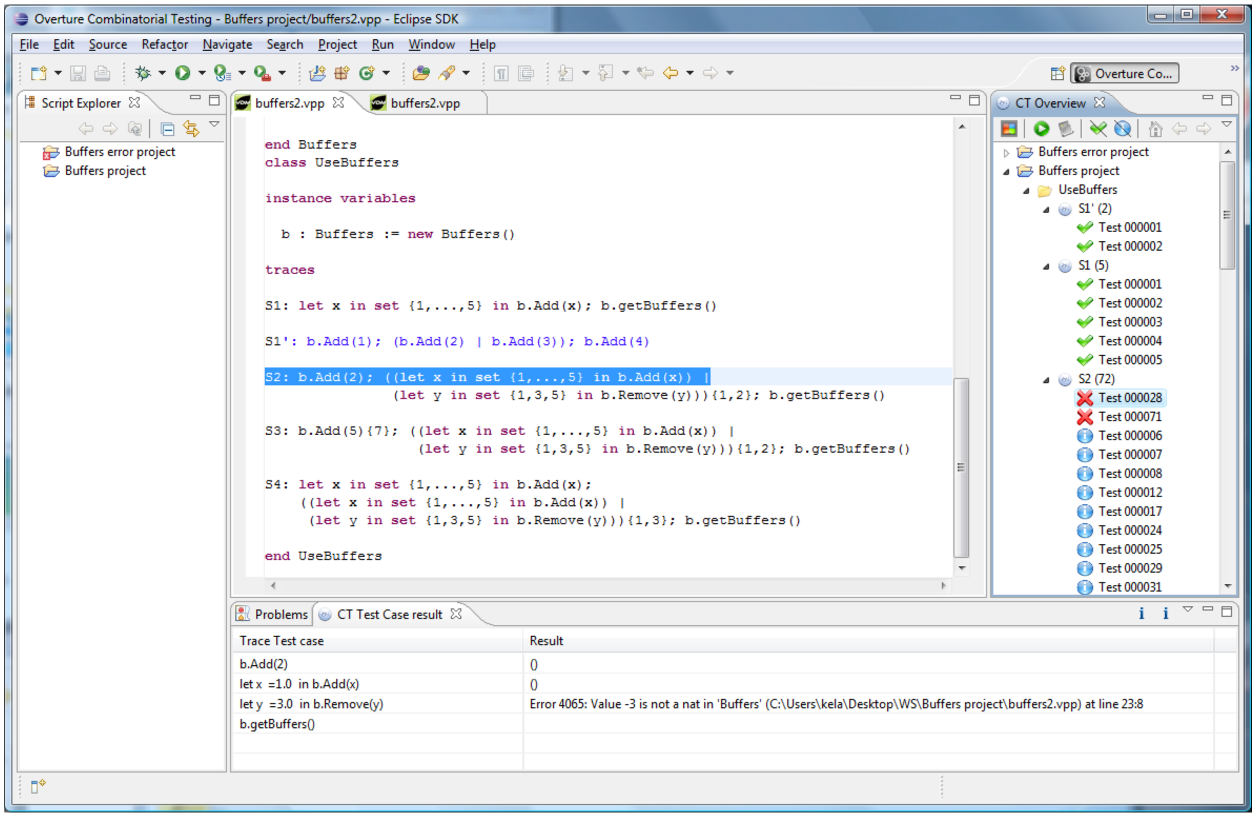
When the perspective is opened it will automatically attempt to expand all the trace definitions in the projects that are open. In rare cases run-time errors can appear in the expansion process. If this happens these will be listed in the "problems" view in Eclipse and the source view will have an error icon with the corresponding line in the trace definition that cannot be expanded. This is exactly the same as if syntax or type errors are discovered. In Figure 3 an example of a problems that can occur in the process of expanding trace definitions into a collection of test cases.

Note in the figure that there is a zero in brackets after the `DD` trace definition name in the CT Overview part of eclipse. This indicates the number of test cases that it has been expanded to and none gets produced when the expansion results in an error like in this case. In all cases the number in brackets after the name of a trace definition indicates the number of test cases that has been generated.

At all elements in the "CT Overview" browser it is possible to right click and then select the "Run selected" option (see the menu in Figure 4). As a consequence all the test cases under the selected item will be executed.

As an alternative to "run selected" one can select the "Run all" entry at the top of the "CT Overview" browser since that will run all the test cases in all the active projects. In case any of the test cases are able to discover errors in the VDM++ model the trace definitions with these test cases are opened in the browser with a red cross over them indicating a *fail* verdict for that test case. In each of these cases an error have been discovered in the VDM++ model. It makes sense to fix these errors before proceeding with using the remaining functionality in the CT perspective. Thus it is possible to move over to debugger perspective with a selected test case[5]. In this perspective it will

[5]This feature is not yet available though because the debugger perspective in Overture is still under development.



```
end Buffers
class UseBuffers

instance variables

  b : Buffers := new Buffers()

traces

S1: let x in set {1,...,5} in b.Add(x); b.getBuffers()

S1': b.Add(1); (b.Add(2) | b.Add(3)); b.Add(4)

S2: b.Add(2); ((let x in set {1,...,5} in b.Add(x)) |
               (let y in set {1,3,5} in b.Remove(y))){1,2}; b.getBuffers()

S3: b.Add(5){7}; ((let x in set {1,...,5} in b.Add(x)) |
                  (let y in set {1,3,5} in b.Remove(y))){1,2}; b.getBuffers()

S4: let x in set {1,...,5} in b.Add(x);
    ((let x in set {1,...,5} in b.Add(x)) |
     (let y in set {1,3,5} in b.Remove(y))){1,3}; b.getBuffers()

end UseBuffers
```

Trace Test case	Result
b.Add(2)	()
let x =1.0 in b.Add(x)	()
let y =3.0 in b.Remove(y)	Error 4065: Value -3 is not a nat in 'Buffers' (C:\Users\kela\Desktop\WS\Buffers project\buffers2.vpp) at line 23:8
b.getBuffers()	

Figure 2: Combinatorial Testing Perspective

be much easier to find out why a particular test case does not behave as expected.

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

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

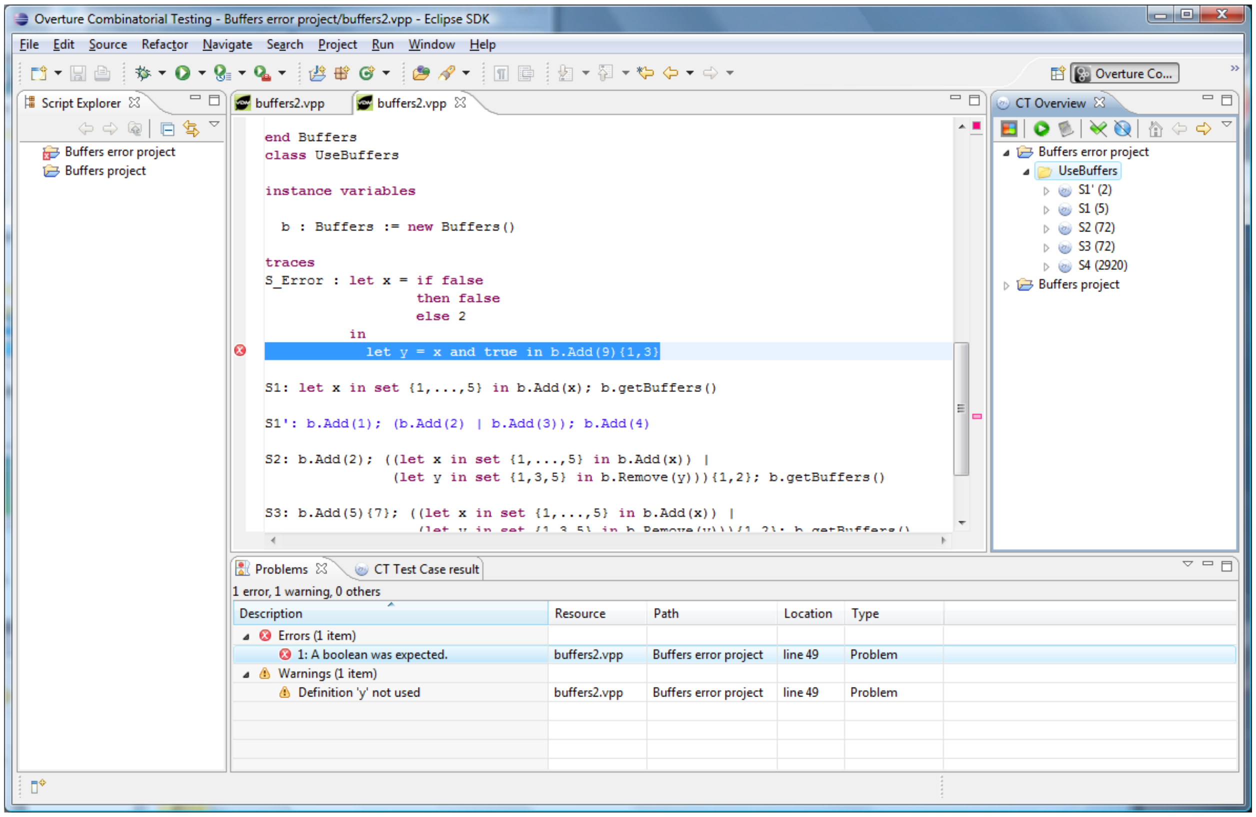
✓: This icon is used to indicate that the test case has a pass verdict.

ℹ: 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.

▷ S4 (2800 skipped 120): If test cases result in a run-time error other test cases with the same prefix will be filtered away and thereby skipped by in the test execution. The number of skipped test cases is indicated after number of test cases for the trace definition name.

Once no more errors can be discovered directly using the CT feature it is possible to manually step through the test cases individually. When a test case is selected in the CT Overview browser the contents of the test case and the corresponding output from the interpreter will be shown in the "CT Test case result" window. It is possible to move through the list of test cases and their results



```
end Buffers
class UseBuffers

instance variables

  b : Buffers := new Buffers()

traces
S_Error : let x = if false
                  then false
                  else 2
          in
            let y = x and true in b.Add(9){1,3}

S1: let x in set {1,...,5} in b.Add(x); b.getBuffers()

S1': b.Add(1); (b.Add(2) | b.Add(3)); b.Add(4)

S2: b.Add(2); ((let x in set {1,...,5} in b.Add(x)) |
               (let y in set {1,3,5} in b.Remove(y))){1,2}; b.getBuffers()

S3: b.Add(5){7}; ((let x in set {1,...,5} in b.Add(x)) |
```

1 error, 1 warning, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
1: A boolean was expected.	buffers2.vpp	Buffers error project	line 49	Problem
Warnings (1 item)				
Definition 'y' not used	buffers2.vpp	Buffers error project	line 49	Problem

Figure 3: Example for a problem detected in expanding test cases

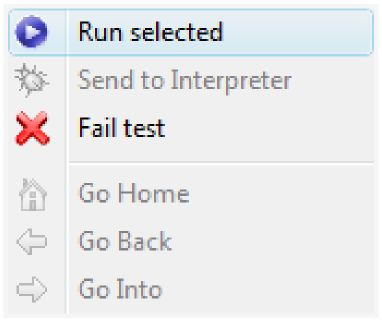


Figure 4: Menu coming up by right-clicking in the CT browser

in the “CT Overview” browser and whenever a test case is spotted where the verdict made by the automatic CT testing analysis is wrong it is possible to change it by right clicking on the test case and selecting the right verdict manually (see the menu in Figure 5). In this process the tool will skip all test cases that have been filtered away in the process because these does not supply any new information. In Figure 5 you can also see different icons (and at the top that enables

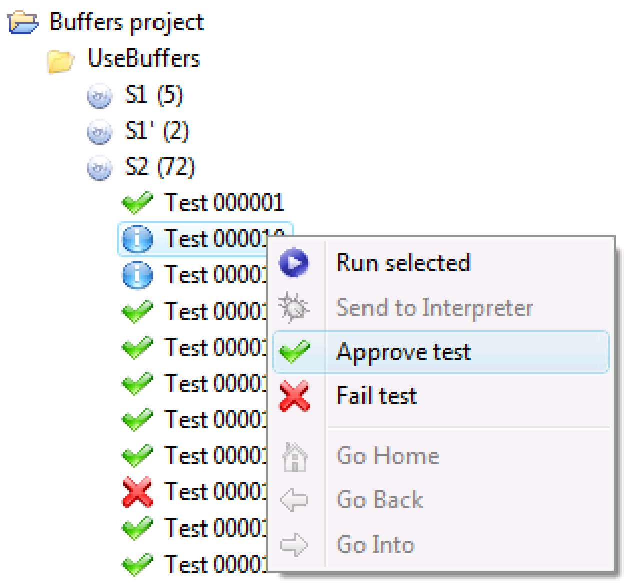


Figure 5: Menu for manual selecting verdict in the CT browser

filtering away of test cases that have a pass verdict as well as test cases that have an inconclusive verdict in the lists of test cases shown in the browser. This can be convenient to use when one have a large number of test cases and one wish to focus first on a particular kind of verdicts. Finally it is also possible to sort the test cases such that all the failed test cases are shown first, followed by the inconclusive ones and finally the passed ones. This is done with the sorting icon at the top of the "CT Overview" browser (▦).

After running all the test cases selected it is also possible to jump over to the test coverage information perspective of Overture in order to be able to see the details of the coverage of the VDM++ model[6]. Here the user may be able to detect uncovered parts of the VDM++ model and this can give inspiration to defining new trace definitions that we will able to explore more parts of the VDM++ model.

Finally a regression test environment that can be used subsequently for testing the VDM++ model when further changes are made can be produced by pressing the small "save results of trace test" icon (see Figure 6) at the top of the "CT Overview browser". When this is done argument files (with the `.arg` extension) are produced for all test cases and corresponding result files (with the `.res` extension) are produced. The root path of all these test case is determined by the user in a small menu that comes up when the icon is pressed (see Figure 7). The test cases and their results will then be produced in a directory structure relative to root for the project, the class they originally came from and finally the name of the trace definition.

[6]This feature is not yet available because the test coverage perspective is not yet enabled in Overture.

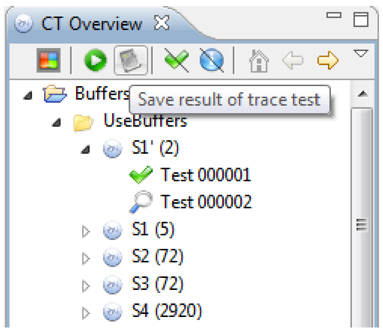


Figure 6: Icon for “Save results of trace test”

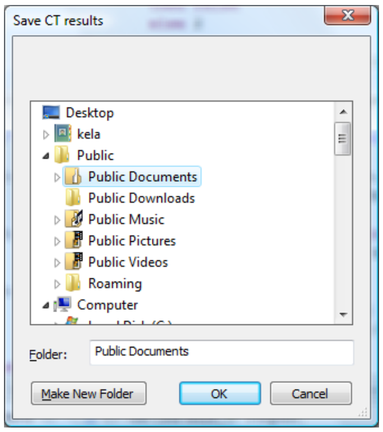


Figure 7: Determining the root for where to save files

5.2.1 Toolbox selection

The toolbox used when testing the test cases can be selected from two the two supported tools:

- Overture VDMJ
- The VDM Toolbox v8.2 or later.

To select a toolbox the dropdown menu in the CT Overview is used. It is marked as a down pointing triangle. The menu is shown in Figure 8. When selecting a new toolbox the CT Overview will be reset. If VDM Tools is chosen a dialog asking for the VDM Tools (vppgde.exe)[7] will pop-up asking for the path to the file.

[7]Only the Graphical version of VDM Tools is supported.

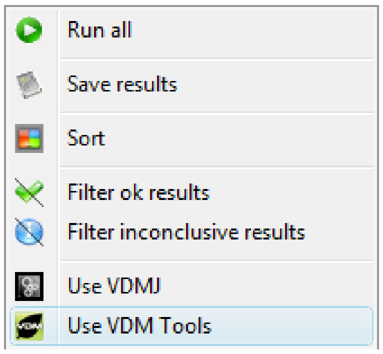


Figure 8: Select Toolbox type.

6 The command-line interface

In addition to the graphical user interface it is possible to make use of a command-line version of this tool. This is done from a command line using the following command:

`java -classpath` *`paths`* `org.overture.traces.jar` *`options specfiles`*

where the options that are enables are:

-outputPath: This will be the starting path for all the generated test cases and result placed in respectively `.arg` and `.res` files.

-c: Followed by a common separated list of the names of the classes that one would like the expansion for. If this option is not used all classes will be expanded and test cases will be executed.

-max: Maximum iterations used for star ($\star$) and plus ($+$) in expansion of traces including these operators.

-toolbox: The VDM++ interpreter which should be used is selected here. The options are:

> **VDMTools:** Requires VDMTools to be installed and an additional option `-VDMToolsPath` to be set to the directory with the executable file.
>
> **VDMJ:** Requires VDMJ.jar to be in the class path[8].

The command-line version of CT can be fetched as:

```
http://mt.lausdahl.com/CT/org.overture.traces-Cmd_release_
v1.zip
```

This includes a small `TestRun.bat` file and the buffers example from the appendix of this user manual as well.

[8]In the current version of the tool VDMJ is significantly faster than using VDMTools.

6.1 Example of usage

```
java -classpath dirs org.overture.traces.jar
     -outputPath c:\
     -c A,B
     -max 3
     -toolbox VDMJ
     a.vpp,b.vpp
```

This kind of usage will result in the trace definitions from the classes `A` and `B` being expanded and tested with the VDMJ interpreter. The argument and result files will be placed in the `c:\A` and `c:\B` directories. There will be one subdirectory here for each of the named trace definitions and each of these directories will have one argument file and one result file per test case.

A The Buffers Example in Full

```
class Buffers

instance variables

  b1 : nat := 0;
  b2 : nat := 0;
  b3 : nat := 0;

inv b1 + b2 + b3 <= 40 and b1 <= b2 and b2 <= b3 and b3 - b1 <= 15

operations

public Add: nat ==> ()
Add(x) ==
  if x + b1 < b2
  then b1 := b1 + x
  elseif b2 + x <= b3
  then b2 := b2 + x
  else b3 := b3 + x
pre x <= 5 and b1 + b2 + b3 + x <= 40
post b1 + b2 + b3 = b1~ + b2~ + b3~ + x;

public Remove: nat ==> ()
Remove(x) ==
  if x + b2 <= b3
  then b3 := b3 - x
  elseif x + b1 <= b2
  then b2 := b2 - x
  else b1 := b1 - x
pre x <= 5 and x <= b1 + b2 + b3
post b1 + b2 + b3 + x = b1~ + b2~ + b3~;

public getBuffers: () ==> nat * nat * nat
getBuffers() ==
  return mk_(b1,b2,b3)

end Buffers
```

```
class UseBuffers

instance variables

  b : Buffers := new Buffers()

traces

S0: let x = if false then false else 2 in let y = x and true in b.Add(9){1,3}

S1: let x in set {1,...,5} in b.Add(x); b.getBuffers()

S1a: b.Add(1); (b.Add(2) | b.Add(3)); b.Add(4)

S1b: b.Add(1){1,3}

S2: b.Add(2); ((let x in set {1,...,5} in b.Add(x)) |
               (let y in set {1,3,5} in b.Remove(y))){1,2}; b.getBuffers()

S3: b.Add(5){7}; ((let x in set {1,...,5} in b.Add(x)) |
                  (let y in set {1,3,5} in b.Remove(y))){1,2}; b.getBuffers()

S4: let x in set {1,...,5} in b.Add(x);
    ((let x in set {1,...,5} in b.Add(x)) |
     (let y in set {1,3,5} in b.Remove(y))){1,3}; b.getBuffers()

end UseBuffers
```