Enhancing Testing of VDM-SL models

Peter W. V. Tran-Jørgensen[1], René S. Nilsson[1,2], and Kenneth Lausdahl[3]

[1] Department of Engineering, Aarhus University, 8200 Aarhus N, Denmark
{pvj,rn}@eng.au.dk
[2] AGCO A/S, Dronningborg Allé 2, 8930 Randers NØ, Denmark
[3] Mjølner Informatics A/S, 8200 Aarhus N, Denmark
kgl@mjolner.dk

Abstract. We find that testing of VDM-SL models is currently a tedious and error-prone task due to lack of tool support for conveniently defining tests, executing tests automatically, and validating test results. In VDM++, test-driven development is supported by the VDMUnit framework, which offers many of the features one would expect from a modern testing framework. However, since VDMUnit relies on object-orientation and exception handling, this framework does not work for testing VDM-SL models. In this paper, we discuss the challenges of testing VDM-SL models, and propose a library extension of Overture/VDMUnit that improves this situation. We demonstrate usage of this library extension, and show how it also enables one to reuse tests to validate code-generated VDM-SL models.

Keywords: VDM, unit testing, continuous validation, code-generation

1 Introduction

Currently, there is a lack of tool support for unit and integration testing in VDM-SL [9], which we find makes testing tedious and error-prone due to lack of support for conveniently defining tests, executing them automatically, and validating the results. Concretely, testing of VDM-SL models requires a significant amount of extra boiler-plate code that must be added and maintained by the modeller throughout the development process. On the other hand, modern development environments often offer test support in the form of frameworks that reduce the time spent on validation.

Most popular programming languages are supported by one or more well-established unit and integration testing frameworks. Examples of these include the JUnit framework for Java [16], Google Test for C/C++ [12], and NUnit for C# [23]. All of these frameworks provide a convenient way to

- define tests (for example by annotating test methods, or by using special naming conventions),
- intercept and control the life-cycle of a test (for example using special "set up" and "tear down" methods to allocate and free resources before/after executing each test),
- check test results (by writing assertions),
- easily run groups of tests and,

– generate test reports.

Often testing frameworks are implemented using peculiarities of the language they support. For example, recent versions of JUnit (version 4 and 5) use *annotations* to mark test methods, whereas NUnit uses C# *attributes*, and Google Test uses *macros*.

While the unit testing frameworks described above are used to check specific cases (for example that a function computes a value for some input) another approach is *property-based testing*, which allows one to test model/program properties in general using generated input. An example of a tool that supports this approach is QuickCheck for Haskell [3]. Concretely, QuickCheck enables one to execute several tests cheaply, while still allowing one to control the tests and input being generated. Inspired by QuickCheck, property-based testing is available for several popular programming languages, including Java [15], .NET [11] and C++ [25]. Property-based testing is similar to combinatorial testing [18], which is already available for VDM and supported by the Overture tool [17, 4, 24]. In this paper we seek to improve unit and integration testing for VDM, hence we focus mostly on VDMUnit [10] and extensions of this framework.

VDMUnit provides most of the features one would expect from a unit and integration testing framework (Section 2). However, as VDMUnit relies on VDM++/VDM-RT's [20] object-orientation and exception handling features, this library does not work for testing VDM-SL models. To address this, we have extended VDMUnit to support unit and integration testing of VDM-SL models (Section 3). Our extension provides two VDM-SL modules that expose the features of VDMUnit in a VDM-SL context. To further support this development process, we have extended Overture's VDM-to-Java code-generator [14] to support fully automated translation of VDM-SL unit tests to equivalent JUnit tests that can be used to validate code-generated models (Section 4). While this approach only re-uses the model tests, another way to perform validation is to compare the output computed using the model to that produced using the corresponding software implementation [8].

The new testing features have supported the development of an industrial harvest planning system [5, 6] that enables farmers to calculate harvest plans based on different optimisation strategies (Section 5). In this project, the master algorithm that computes the harvest plans are modelled in VDM-SL and implemented via Java code-generation. At the modelling level, this algorithm is validated using VDM-SL unit tests, while code-generated tests are used to check for subtle errors introduced during the code-generation process.

2 Background

In this section we highlight some of the existing features of VDMUnit for VDM++/VDM-RT, and later explain how these have been supported in a VDM-SL setting. In addition, we briefly describe Overture's code-generation infrastructure, which we have used to translate VDM-SL tests into equivalent JUnit tests.

2.1 VDMUnit architecture

The features of VDMUnit are exposed as VDM++ classes that use a Java component to automatically identify and execute tests using Java's reflection features. These VDM

classes are connected to the Java component using Overture's VDM-to-Java bridge, which enables combined execution of VDM and Java [22] in order to

- improve execution performance in a VDM setting (as Java is executed as compiled code, which generally performs better than VDM which is interpreted),
- use language/framework features that are not directly available in VDM (e.g. reflection or access to the underlying operating system), and
- share functionality between VDM dialects.

The architecture of VDMUnit is shown in Figure 1.

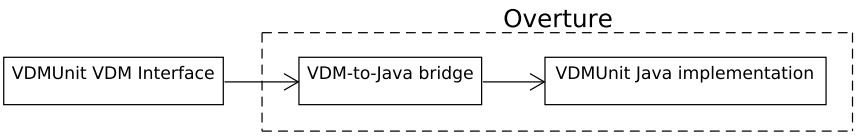

Fig. 1: VDMUnit architecture.

2.2 Testing VDM++ models using VDMUnit

In this paper, a *test class* is a modeller-defined subclass of VDMUnit's `TestCase` class, which defines *test* operations that validate functionality using *assertions*. The name of a test operation must begin with "test", and the operation itself is expected to take zero input arguments – otherwise it will be recorded as an error, once executed by VDMUnit. Listing 1.1 shows an example of a test class that contains a single test.

```
1 class MyTest is subclass of TestCase
2 operations
3 public testOne : () ==> ()
4 testOne () == Assert`assertTrue("Expected `someFeature` to
      generate an even number ", someFeature() mod 2 = 0);
5 end MyTest
```

Listing 1.1: Example of a modeller-defined `TestCase`.

The `TestCase` class defines `setUp` and `tearDown` operations to intercept and control the life-cycle of a test. The `setUp` operation is invoked by VDMUnit before any test is executed in order to initialise test data whereas `tearDown` is invoked after a test has been executed in order to free test resources.

Once a test has been executed, it is either recorded as a

- *failure* if a condition checked using an assertion is false, an

- *error* if the tests produces a runtime error, or a
- *success* otherwise.

In case an assertion is false, VDMUnit terminates the execution of the test and records it as a test failure. Specifically, this is achieved by raising an exception inside VDMUnit's `Assert` class in order to signal a test failure to the framework.

VDMUnit offers different ways to execute tests. One way is to execute all tests in a single run by evaluating the expression **new** `TestRunner().run()`. This operation call uses *automated reflection* to execute all tests. Another approach that is more flexible is to execute tests selectively. An example of this approach is shown in Listing 1.2, which constructs and executes a `TestSuite` consisting of `TestCase1` and `TestCase2`.

```
1 let tests : set of Test = {new TestCase1(), new TestCase2()},
2     ts : TestSuite = new TestSuite(tests),
3     result : TestResult = new TestResult()
4 in
5 (
6     ts.run(result);
7     IO`print(result.toString());
8 );
```

Listing 1.2: Selective test execution using VDMUnit.

2.3 Overture's code generation platform

Translation of VDM-SL unit tests to equivalent JUnit tests is implemented as an extension of Overture's VDM-to-Java code-generator. This code-generator is developed using Overture's code generation platform [14, 26], which is a framework for building code-generators for VDM. The workflow of the code-generation platform is as follows: First, the platform constructs an intermediate representation (IR) of the generated code that initially mirrors the structure of the VDM model subject to code-generation. The IR is then subjected to a series of customised *transformations* in order to bring it to a form that is easier to translate into target language code (e.g. Java). For example, by replacing a node that is non-trivial to code-generate with other nodes that have a direct mapping into the target language. As transformations operate directly on the IR, which is independent of any target language, they can in principle be shared among code-generators. Once the IR has reached its final form it is handed over to the *backend*, which is responsible for translating the individual IR nodes into target language code. This step is enabled using the code-generation platform's code-emission framework, which uses the Apache Velocity template engine [1].

Generation of JUnit tests is achieved using a transformation that identifies VDM-SL unit tests in the IR according to the naming conventions described above and converts these tests into a form that eventually is translated to JUnit tests (Java code) using the code-generation platform's code emission framework. This process is described in more detail in Section 4.

3 Testing VDM-SL models

3.1 Defining VDM-SL tests

In VDM++/VDM-RT test cases can be created by subclassing VDMUnit's `TestCase` class. However, since VDM-SL does not support inheritance, tests must be defined in a different way. Instead we have found the naming convention used by JUnit3 (version 3 of JUnit) to be suitable for defining VDM-SL tests. Following this approach, the name of a *test module* must end with "Test", and *test operations* must begin with "test".

3.2 Framework overview

Our extension of VDMUnit consists of two VDM-SL modules named `TestRunner` and `Assert` that expose VDMUnit's testing features to the modeller. These modules are connected to a Java component that implements test execution by means of Overture's VDM-to-Javabridge (see Section 2). This is similar to how VDMUnit for VDM++/VDM-RT is designed (see Figure 1). The implementation of VDMUnit for VDM-SL as proposed in this paper is open-source and available via [27].

3.3 The VDM-SL interface

The `Assert` module is shown in Listing 1.3. This module defines four operations for validating model functionality: the `assertTrueMsg` operation takes two arguments, a message that describes the assertion (`pmessage`), and the condition to be checked (`pbool`). If the condition does not hold the framework will mark the test as a failure and store the description of the assertion. `assertTrue` is similar to `assertTrueMsg` – except that the former only receives the condition to be checked. The corresponding operations in VDM++ are defined by means of operation overloading, which is not supported by VDM-SL, hence different names must be used for these operations. `assertFalseMsg` and `assertFalse` in Listing 1.3, are similar to `assertTrueMsg` and `assertTrue` except that they will mark the test under execution as a failure if the condition being checked is true.

```
1  module Assert
2
3  imports from TestRunner all
4  exports all
5
6  definitions
7
8  operations
9  assertTrue: bool ==> ()
10 assertTrue (pbool) ==
11   if not pbool then
12     TestRunner`markFail();
13
14 assertTrueMsg: seq of char * bool ==> ()
```

```
15 assertTrueMsg (pmessage, pbool) ==
16   if not pbool then
17   (
18     TestRunner`markFail();
19     TestRunner`setMsg(pmessage);
20   );
21
22 assertFalse: bool ==> ()
23 assertFalse (pbool) ==
24   if pbool then
25     TestRunner`markFail();
26
27 assertFalseMsg: seq of char * bool ==> ()
28 assertFalseMsg (pmessage, pbool) ==
29    if pbool then
30   (
31     TestRunner`markFail();
32     TestRunner`setMsg(pmessage);
33   );
34
35 end Assert
```

Listing 1.3: Module used to validate model functionality.

To enable automated execution of tests, our library extension defines a `TestRunner` module with three operations as shown in Listing 1.4. As indicated using the **`is not yet specified`** statement all of these operations are implemented in Java using Overture's Java bridge (see Section 2). Once executed, the `run` operation executes all test operations that conform to the naming convention described in Section 3.1. The identification of VDM-SL tests operations is implemented using reflection (which is similar to how VDM++ tests are identified). In addition, the `TestRunner` module defines two operations. The `markFail` operation is used by the test framework to mark the test operation under execution as a failure. Similarly, the `setMsg` operation is used to pass a message to the testing framework that describes a test failure. This message is used in the final test report. The `markFail` and `setMsg` operations are used internally by the framework and should not be invoked directly by the modeller.

```
1  module TestRunner
2  exports all
3
4  definitions
5
6  operations
7
8  run : ()==>()
9  run()== is not yet specified;
10
11 markFail : () ==> ()
```

```
12 markFail () == is not yet specified;
13
14 setMsg : seq of char ==> ()
15 setMsg (msg) == is not yet specified;
16
17 end TestRunner
```

Listing 1.4: Module used to execute tests.

3.4 Limitations

Our extension of VDMUnit exposes all the existing framework features in a VDM-SL context, with the only exception of selective test execution, shown in Listing 1.2. In a VDM++ context selective test execution is achieved by passing a set of test cases to the framework, e.g. `{new TestCase1(), new TestCase2}`. This approach has the advantage that it provides a *type-safe* way to group tests. For example, if the class definition for `TestCase1` is removed or renamed then the type-checker will raise an error reminding the modeller to update the test selection as well. When test cases are grouped into modules, according to our approach, there is no type-safe way to select test cases like in VDM++. The reason for this is that modules (i.e. the test cases) cannot be instantiated or passed as values. One workaround is to pass the module names as string literals at the price of loosing type-safety. Concretely, execution of the tests defined in the modules `TestCase1` and `TestCase2` can then be achieved by passing `{"TestCase1", "TestCase2"}` to the framework.

3.5 VDM-SL test example

An example of a test module, defined using our extension of VDMUnit, is shown in Listing 1.5. This module defines a `setUp` operation to initialise state (before executing each test), and a `tearDown` operation to execute some appropriate cleanup procedure (e.g. removing temporary files). In addition, the test module defines three test operations, named `testOdd`, `testInverse`, and `testPos`.

```
1  module MyTest
2  imports from Assert all
3  exports all
4
5  definitions
6
7  state St of
8    x : int
9  end;
10
11 operations
12
13 setUp : () ==> ()
```

```
14 setUp () == initState();
15
16 tearDown : () ==> ()
17 tearDown () == cleanUp();
18
19 testOdd: ()==>()
20 testOdd()==
21 (
22   x := x + 1;
23   Assert`assertFalseMsg("Expected x to be odd", x mod 2 = 0);
24 );
25
26 testInverse: ()==>()
27 testInverse()==
28   Assert`assertTrueMsg("Expected 1/x to be positive", 1/x > 0);
29
30 testPos: ()==>()
31 testPos()==
32 (
33   x := x - 1;
34   Assert`assertTrueMsg("Expected x to be positive", x > 0);
35 );
36
37 initState : () ==> ()
38 initState () == x := 0;
39
40 cleanUp : () ==> ()
41 cleanUp () == skip;
42
43 end MyTest
```

Listing 1.5: Test module example.

Once `MyTest` is executed, by evaluating `TestRunner`run()`, the test report shown in Listing 1.6 is generated by Overture. As shown in this output, three tests are executed of which `testOdd` passes successfully, `testInverse` is recorded as an error (due to an attempt to divide by zero), and `testPos` fails due to a wrong assertion.

```
1  **
2  ** Overture Console
3  **
4  Executing test: MyTest`testOdd()
5          OK
6  Executing test: MyTest`testInverse()
7          ERROR: Error 4134: Infinite or NaN trouble in 'MyTest' (
               A.vdmsl) at line 26:56
8  Executing test: MyTest`testPos()
9          FAIL: Expected x to be positive
10 ----------------------------------------
11 |    TEST RESULTS                      |
```

```
|--------------------------------------|
| Executed: 3                          |
| Failures: 1                          |
| Errors  : 1                          |
|______________________________________|
|              FAILURE                 |
|______________________________________|


TestRunner`run() = ()
Executed in 0.055 secs.
```

Listing 1.6: Output obtained by executing the tests in Listing 1.5.

3.6 Java implementation

Automatic execution of tests involves inspection of modules in order to identify the tests that must be executed. As this is not possible to do solely using VDM-SL, the part of the framework that handles test execution is implemented in Java, which achieves this using Java's reflection feature. In this way, one can inspect the individual test modules at the abstract syntax level in order to identify and execute the individual test operations. The combined execution of VDM-SL and Java is enabled using Overture's VDM-to-Java bridge.

3.7 Jenkins integration server

In addition to generating test reports such as that shown in Listing 1.6, our extension of VDMUnit supports test report generation in a JUnit compatible XML format. Using this approach, one can, for example, inspect and visualise the test reports using the Jenkins [13] integration server. To generate XML test reports one simply has to pass the property `-Dvdm.unit.report` to the Overture interpreter when executing tests. An example of how this feature has been applied in the context of the harvest planning project is given in Section 5.

4 Code-generating VDM-SL tests

Overture's Java code-generator is exposed as a Maven plugin [21] in order to improve build and test automation in a VDM setting [19]. This Maven plugin already supports translation of VDMUnit tests, specified using VDM++, to equivalent JUnit tests.[4] Essentially, this feature enables one to reuse the model tests to validate the implementation of the model. This step helps ensure that the code-generator did not introduce subtle bugs in the translation. As part of our work we have extended the code-generator

[4] An online tutorial that demonstrates how to invoke the Java code-generator using Maven is available via [7].

to also support code-generation of VDM-SL unit tests that use the naming convention introduced in Section 3.1.

The output obtained by translating the VDM-SL tests in Listing 1.5 to JUnit4 tests is shown in Listing 1.7. This is achieved by first translating the test module, including the test operations, to Java using Overture's VDM-to-Java code-generator. Secondly, the generated test methods and life-cycle methods are annotated using appropriate JUnit annotations. This involves annotating the `setUp` and `tearDown` methods using `@Before` and `@After`, respectively, as well as annotating all tests using `@Test`. Finally, the VDM-SL assertions are translated to equivalent JUnit method calls, i.e. `assertTrue` and `assertFalse`.

```
1  package dk.au.seng.cge.codegen;
2
3  import java.util.*;
4  import org.overture.codegen.runtime.*;
5  import org.junit.*;
6
7  @SuppressWarnings("all")
8  final public class MyTest {
9    private static St St = new St(null);
10
11   @Before
12   public void setUp() {
13     initState();
14   }
15
16   @After
17   public void tearDown() {
18     cleanUp();
19   }
20
21   @Test
22   public void testOdd() {
23     St.x = St.x.longValue() + 1L;
24     Assert.assertFalse("Expected␣x␣to␣be␣odd", Utils.equals(
           Utils.mod(St.x.longValue(), 2L), 0L));
25   }
26
27   @Test
28   public void testInverse() {
29     Assert.assertTrue(
30         "Expected␣1/x␣to␣be␣positive", Utils.divide((1.0 * 1L),
               St.x.longValue()) > 0L);
31   }
32
33   @Test
34   public void testPos() {
35     St.x = St.x.longValue() - 1L;
```

```
36      Assert.assertTrue("Expected␣x␣to␣be␣positive", St.x.
            longValue() > 0L);
37    }
38
39    public void initState() {
40      St.x = 0L;
41    }
42
43    public void cleanUp() {
44      /* skip */
45    }
46
47    public String toString() {
48      return "MyTest{" + "St␣:=␣" + Utils.toString(St) + "}";
49    }
50
51    public static class St implements Record {
52      public Number x;
53
54      public St(final Number _x) {
55        x = _x;
56      }
57
58      public boolean equals(final Object obj) {
59
60        if (!(obj instanceof St)) {
61          return false;
62        }
63
64        St other = ((St) obj);
65
66        return Utils.equals(x, other.x);
67      }
68
69      public int hashCode() {
70        return Utils.hashCode(x);
71      }
72
73      public St copy() {
74        return new St(x);
75      }
76
77      public String toString() {
78        return "mk_MyTest`St" + Utils.formatFields(x);
79      }
80    }
81  }
```

Listing 1.7: Output obtained by translating the VDM-SL tests to Java.

Since Overture's Java code-generator is exposed as a Maven plugin it can be invoked using the Maven build system in order to code-generate VDM specifications and model tests, as well as running the generated JUnit tests automatically. Once the Maven plugin is configured [7], all of this can be achieved by invoking a single Maven command such as `mvn install`. The output obtained by running the code-generated versions of the VDM-SL tests in Listing 1.5 is shown in Listing 1.8. As expected, the output in this listing shows that the test results are equivalent to those obtained by running the VDM-SL tests.

```
1  Running MyTest
2  Tests run: 3, Failures: 1, Errors: 1, Skipped: 0, Time elapsed:
       0.066 sec <<< FAILURE!
3  testPos(MyTest)  Time elapsed: 0.005 sec  <<< FAILURE!
4  java.lang.AssertionError: Expected x to be positive
5   ...
6  testInverse(MyTest)  Time elapsed: 0.002 sec  <<< ERROR!
7  java.lang.ArithmeticException: Division by zero is undefined
8   ...
9  Results :
10 Failed tests:   testPos(MyTest): Expected x to be positive
11 Tests in error:
12   testInverse(dk.au.seng.cge.codegen.MyTest): Division by zero
         is undefined
13 Tests run: 3, Failures: 1, Errors: 1, Skipped: 0
```

Listing 1.8: Output obtained by running the generated JUnit tests.

5 Assessment

The new VDM-SL testing features have supported the development of an industrial harvest planning system, which enables farmers to optimise the logistics of harvest operations. A typical harvest workflow starts with a combine harvester harvesting the crop. The collected yield, which is contained in the harvester, is then unloaded into an in-field grain cart that transports and unloads the yield into a larger on-road truck that finally delivers the yield to a drying or storage facility. This concept is illustrated in Figure 2, where parts of the optimised route plans for each vehicle are shown. A centralised master algorithm in the cloud initially generates route plans for each vehicle based on a system configuration, including the field shape, number of vehicles, yield estimates, and optimisation strategies. Once the harvest operation starts, the master algorithm monitors the state of all vehicles as well as the overall harvest progress, and if necessary, modifies the route plans for the individual vehicles to address potential deviations (e.g. yield discrepancies).

The master algorithm is modelled in VDM-SL and implemented using Overture's VDM-to-Java code-generator. Figure 3 shows the structure of the model, including only the most important modules. The model captures the state and behaviour of the different vehicles, the overall harvest progress of the field, the route optimisation strategies, and handling of deviations and unload coordination between vehicles. Totalling to 4200

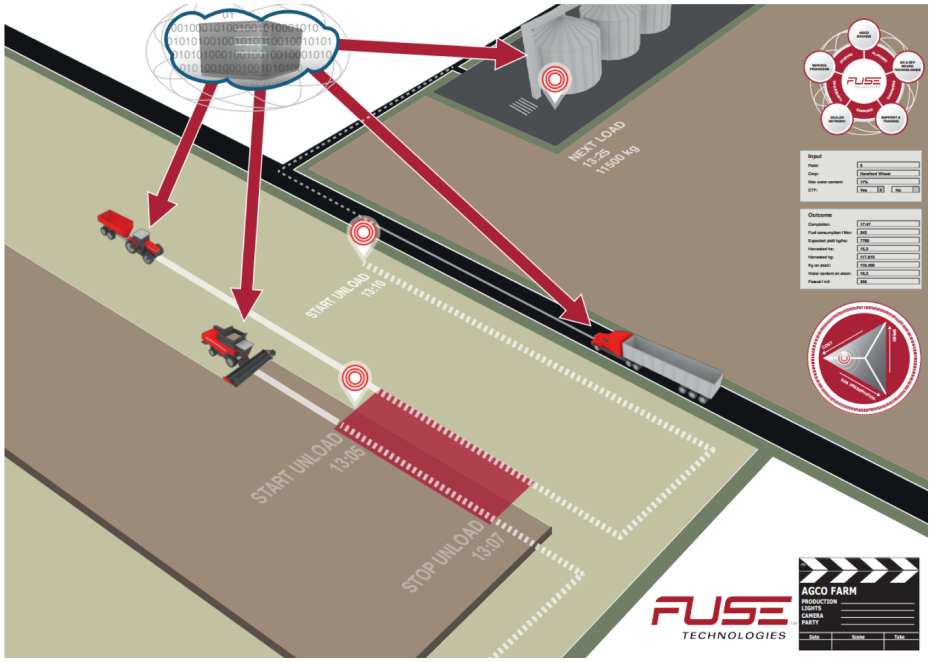

Fig. 2: Harvest logistics illustration.

lines of code[5], whereof 1100 lines implement 134 tests. Running all tests through the Overture interpreter takes approximately 7 hours, whereas the code-generated tests take approximately 30 minutes. Essentially, the time difference is first of all caused by VDM performing worse than compiled Java code. Secondly, the generated Java code does not include pre- and postcondition, and invariant checks.

As mentioned in Section 3.7, the XML-based test results (obtained from both the model tests and code-generated tests) can be inspected and visualised using the Jenkins integration server . An example of how these results can be visualised using Jenkins is shown in Figure 4. This figure provides an overview of the tests results, as well more detailed information about the changes since last test run. Additionally, a complete list of all failing tests is provided, and upon further inspection, detailed error messages and stack traces.

6 Conclusion and future plans

Prior to starting this work, VDMUnit did not support unit testing of VDM-SL models as it relied on language features only available in VDM++/VDM-RT. To address this, we developed an extension of VDMUnit that exposes the features of this framework using the `TestRunner` and `Assert` modules, which use a Java component to handle

[5] Excluding documentation, comments and empty lines

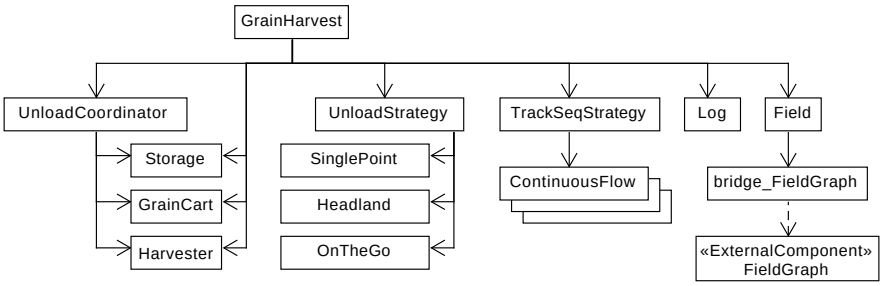

Fig. 3: Simplified VDM-SL model structure.

Test Result : (root)

5 failures (+5)

134 tests (±0)
Took 2 hr 25 min.
add description

All Failed Tests

Test Name	Duration	Age
FoulumTest.test_field_test_1_Headland_Bee	1.2 sec	1
FoulumTest.test_field_test_1_OnTheGo_Bee	1.8 sec	1
HobroLandevejTest.test_BCO_HeadlandUnload	29 sec	1
HobroLandevejTest.test_BCO_OnTheGo	38 sec	1
HobroLandevejTest.test_BCO_SP	30 sec	1

All Tests

Class	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total (diff)
BeeTest	7.3 sec	0	0	3	3
BridgeTest	0.11 sec	0	0	4	4
FieldTest	2.2 sec	0	0	2	2
FoulumTest	1 min 34 sec	2 +2	0	18 -2	20
HeadlandUnloadTest	20 sec	0	0	6	6
HobroLandevejTest	2 min 54 sec	3 +3	0	3 -3	6
LoggerTest	1 ms	0	0	1	1

Fig. 4: Visualization of VDM-SL test result in Jenkins.

test execution. This Java component is connected to these modules using Overtue's VDM-to-Java bridge which is helpful when implementing VDM libraries as described in Section 2.1. To further support this development process, we extended Overture's VDM-to-Java code-generator to support translation of VDM-SL unit tests into equivalent JUnit tests. Our work has supported the development of a harvest planning system for optimising the logistics of harvest operations. Currently, our work supports all of the testing features available in VDM++/VDM-RT, except for selective test execution as described in Section 3.4.

The Overture Language Board [2] has recently developed a workflow for library submissions that enables any community member to submit library proposals. Acceptance of a library submission is expected to lead to the inclusion of the library in one or more VDM tools. Looking ahead, we plan to submit our work as a library proposal to hopefully get it included in future releases of Overture, thus making our work available to others.

References

1. The Apache Maven Project website. `https://maven.apache.org` (2018)
2. Battle, N., Haxthausen, A., Hiroshi, S., Jørgensen, P.W.V., Plat, N., Sahara, S., Verhoef, M.: The Overture Approach to VDM Language Evolution. In: Proceedings of the 11th Overture workshop (Aug 2013)
3. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00, ACM, New York, NY, USA (2000), `http://doi.acm.org/10.1145/351240.351266`
4. Couto, L.D., Larsen, P.G., Hasanagic, M., Kanakis, G., Lausdahl, K., Tran-Jørgensen, P.W.V.: Towards Enabling Overture as a Platform for Formal Notation IDEs. In: Proceedings of the 2nd Workshop on Formal-IDE (F-IDE) (Jun 2015)
5. Couto, L.D., Tran-Jørgensen, P.W.V., Edwards, G.T.C.: Combining Harvesting Operations Optimisation using Strategy-based Simulation. In: Proceedings of the 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH) (Jul 2016)
6. Couto, L.D., Tran-Jørgensen, P.W.V., Edwards, G.T.C.: Model-Based Development of a Multi-algorithm Harvest Planning System. In: Simulation and Modeling Methodologies, Technologies and Applications: International Conference, SIMULTECH 2016 Lisbon, Portugal, July 29-31, 2016, Revised Selected Papers. Springer International Publishing (2018), `https://doi.org/10.1007/978-3-319-69832-8_2`
7. Delegate Tutorial. `https://github.com/ldcouto/delegate-tutorial` (2018)
8. Dutle, A.M., Muñoz, C.A., Narkawicz, A.J., Butler, R.W.: Software Validation via Model Animation. In: Blanchette, J.C., Kosmatov, N. (eds.) Tests and Proofs. pp. 92–108. Springer International Publishing, Cham (2015)
9. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008)
10. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://overturetool.org/publications/books/vdoos/`
11. FsCheck website. `https://github.com/fscheck/FsCheck` (2018)
12. Google Test website. `https://github.com/google/googletest` (2018)

13. Jenkins website. https://jenkins.io (2018)
14. Jørgensen, P.W.V., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: Proceedings of the 12th Overture workshop (Jun 2014)
15. junit-quickcheck website. https://github.com/pholser/junit-quickcheck (2018)
16. JUnit website. http://www.junit.org (2018)
17. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (Jan 2010), http://doi.acm.org/10.1145/1668862.1668864
18. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (Sep 2010), http://dx.doi.org/10.1109/SEFM.2010.32, ISBN 978-0-7695-4153-2
19. Larsen, P.G., Lausdahl, K., Tran-Jørgensen, P.W.V., Coleman, J., Wolff, S., Couto, L.D., Bandur, V., Battle, N.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative (May 2010)
20. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (Oct 2011), ISBN 978-3-642-24558-9
21. The Maven Project website. https://maven.org (2018)
22. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012)
23. NUnit website. http://nunit.org/ (2018)
24. Overture tool website. http://overturetool.org/ (2018)
25. RapidCheck website. https://github.com/emil-e/rapidcheck (2018)
26. Tran-Jørgensen, P.W.V.: Enhancing System Realisation in Formal Model Development. Ph.D. thesis, Aarhus University (Sep 2016)
27. VDMUnit for VDM-SL. https://github.com/overturetool/overture/pull/671 (2018)