SCHOOL OF
COMPUTING

Title: The 16th Overture Workshop

Names: Ken Pierce and Marcel Verheof (eds.)

TECHNICAL REPORT SERIES

No. CS-TR- 1524 October 2018

TECHNICAL REPORT SERIES

No. CS-TR- 1524 Date: October 2018

Title: The 16th Overture Workshop

Authors: Ken Pierce and Marcel Verheof (eds.)

Abstract: The 16th Overture Workshop was held on Saturday 14 July 2018 in association with the Federated Logic Conference (FLoC) 2018 and the 22nd International Symposium on Formal Methods (FM 2018). The workshop provided a forum for discussing and advancing the state of the art in formal modelling and analysis using VDM and its family of associated formalisms including extensions for describing distributed, real-time and cyber-physical systems. The workshop covered topics including tool support, applications of VDM, and perspectives of current and future directions of the community.

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building,
Newcastle upon Tyne, NE4 5TG, England.

Bibliographical Details : The 16th Overture Workshop

Title and Authors : Ken Pierce and Marcel Verheof (eds.)

NEWCASTLE UNIVERSITY
School of Computing. Technical Report Series. CS-TR- 1524

Abstract: The 16th Overture Workshop was held on Saturday 14 July 2018 in association with the Federated Logic Conference (FLoC) 2018 and the 22nd International Symposium on Formal Methods (FM 2018). The workshop provided a forum for discussing and advancing the state of the art in formal modelling and analysis using VDM and its family of associated formalisms including extensions for describing distributed, real-time and cyber-physical systems. The workshop covered topics including tool support, applications of VDM, and perspectives of current and future directions of the community.

About the authors: Ken Pierce is a Lecturer in Cyber-Physical Systems in the School of Computing at Newcastle University. His main interests lie in developing methods and tools for collaborative, model-based design and engineering of cyber-physical systems (CPSs). In particular, he is interested in helping engineers from multiple disciplines collaborate effectively through co-modelling and co-simulation to achieve better performing, more resilient designs across a range of domains. He has been convener of the VDM Language Board since 2014.

Marcel Verhoef is a Software Engineer at the European Space Agency (Flight Software Systems Section at ESTEC). He received his PhD in 2009 from Radboud University Nijmegen, NL. He has worked in commercial R&D positions in industry, with his main focus on development of complex large-scale technical computing systems, both mission-critical and embedded systems in aerospace, defence and industry. He has a keen interest in the application of formal methods in systems engineering.

Suggested keywords: VDM, Overture, formal methods

Preface

The 16th Overture Workshop was held on Saturday 14 July 2018 in association with the Federated Logic Conference (FLoC) 2018 and the 22nd International Symposium on Formal Methods (FM 2018).

The 16th Overture Workshop is the latest in a series of workshops around the Vienna Development Method (VDM), the open-source project Overture, and related tools and formalisms. VDM is one of the best established formal methods for systems development. A lively community of researchers and practitioners in academia and industry has grown around the modelling languages (VDM-SL, VDM++, VDM-RT, CML) and tools (VDMTools, Overture, Crescendo, Symphony, and the INTO-CPS chain). Together, these provide a platform for work on modelling and analysis technology that includes static and dynamic analysis, test generation, execution support, and model checking.

Current projects on model-based design for cyber-physical systems (INTO-CPS and the CPSE Labs experiments TEMPO, CPSBuDi and IPP4CPPS) are generating real results. There are also important developments in Japan with the release of VDMTools under an open source licence. It is thus timely to focus on the future of the methods and toolchain, improvements in capabilities, and potential applications. We held a structured discussion on the future of Overture, which will be published in a separate technical report.

Previous workshops have been invaluable in encouraging both new and established members of the community in their work, and helping to determine priorities and future directions. Proceedings of former workshops are available at http://www.overturetool.org/. We would like the members of the programme committee to do a great job providing in depth feedback to the authors of the submitted papers. That has not only raised the quality of the papers themselves, but moreover also provide a lot of stimulus for discussions at the workshop.

We would also like to thank the PC chairs of the F-IDE workshop, for joining forces to bring our communities a step closer together and sharing invited talks.

July 2018

Ken Pierce and Marcel Verhoef
Program Chairs
16th Overture Workshop

Organization

The 16th Overture Workshop was organised in association with the Federated Logic Conference (FLoC) 2018 and the 22nd International Symposium on Formal Methods (FM 2018).

Program Chairs

Ken Pierce (Newcastle University, United Kingdom)
Marcel Verhoef (ESTEC, ESA, The Netherlands)

Program Committee

Keijiro Araki (Kyushu University, Japan)
Victor Bandur (Aarhus University, Denmark)
Nick Battle (Newcastle University, United Kingdom)
Luis Diogo Couto (UTRC, Ireland)
John Fitzgerald (Newcastle University, United Kingdom)
Leo Freitas (Newcastle University, United Kingdom)
Fuyuki Ishikawa, National Institute of Informatics, Japan
Peter Würtz Vinther Tran-Jørgensen (Aarhus University, Denmark)
Peter Gorm Larsen (Aarhus University, Denmark)
Paolo Masci (Universidade do Minho, Portugal)
Tomohiro Oda (Software Research Associates, Inc., Japan)
Jos Nuno Oliveira (Universidade do Minho, Portugal)
Nico Plat (Thanos, The Netherlands)

Table of Contents

Author Index

Part I

Tools Session

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.

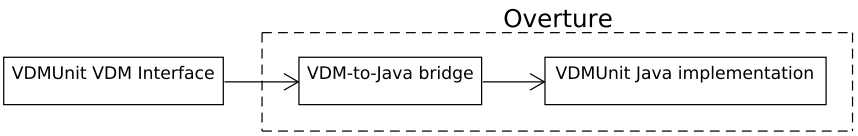


Fig. 1: VDMUnit architecture.

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

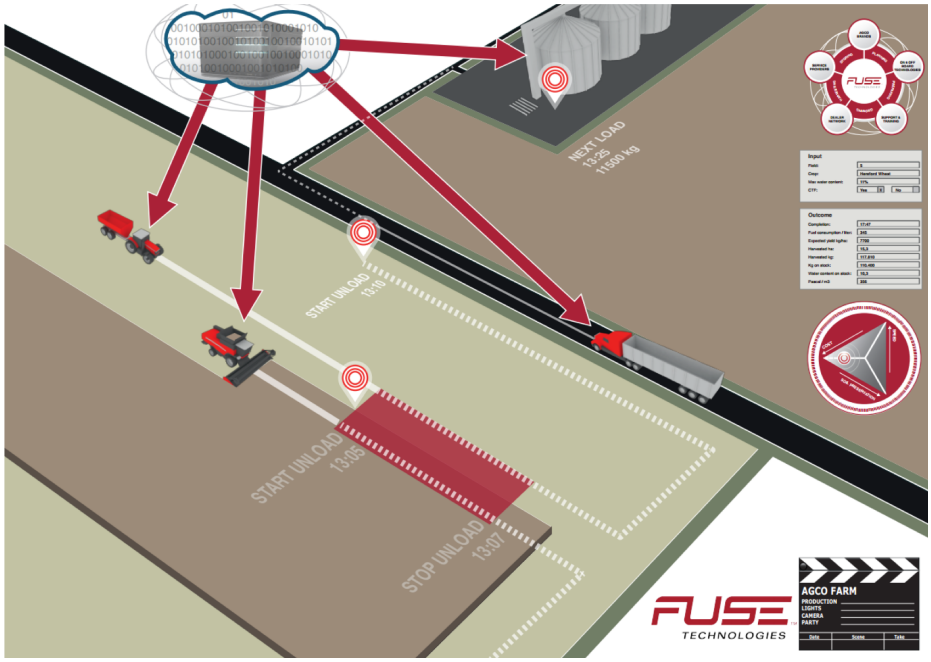


Fig. 2: Harvest logistics illustration.

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

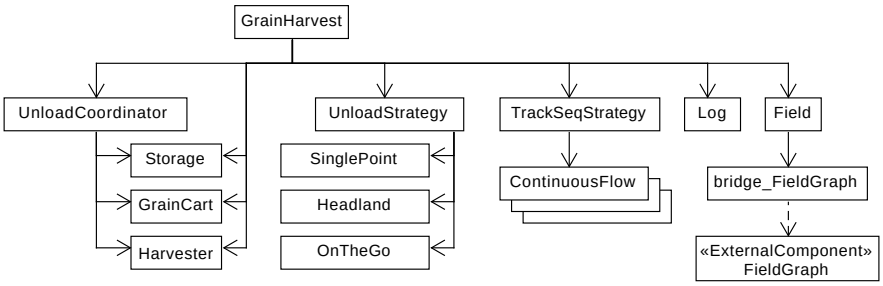


Fig. 3: Simplified VDM-SL model structure.

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)

Overture FMU: Export VDM-RT Models as Tool-Wrapper FMUs

Casper Thule[1], Kenneth Lausdahl[2], and Peter Gorm Larsen[1]

[1] Aarhus University, Department of Engineering
Finlandsgade 22, 8200 Aarhus N, Denmark
{casper.thule,pgl}@eng.au.dk
[2] Mjølner Informatics A/S
Finlandsgade 10, 8200 Aarhus N, Denmark
kgl@mjolner.dk

Abstract. The Functional Mock-up Interface is a standard for co-simulation, which both defines and describes a set of C interfaces that a simulation unit, a Functional Mock-up Unit (FMU), must adhere to in order to participate in such a co-simulation. To avoid the effort of implementing the low level details of the C interface when developing an FMU, one can use the Overture tool and the language VDM-RT. VDM-RT is a VDM dialect used for modelling real-time and potentially distributed systems. By using the Overture extension, called Overture FMU, the VDM-RT dialect can be used to develop FMUs. This raises the abstraction level of the implementation language and avoids implementation details of the FMI-interface thereby supporting rapid prototyping of FMUs. Furthermore, it enables precise time detection of changes in outputs, as every expression and statement in VDM-RT is associated with a "timing cost". The Overture FMU has been used in several industrial case studies, and this paper describes how the Overture tool-wrapper FMU engages in a co-simulation in terms of architecture, synchronisation and execution. Furthermore, a small example is presented.

Keywords: Overture, Functional Mock-up Interface, VDM-RT, Co-Simulation, Real-time, Discrete-Event

1 Introduction

In general, co-simulation enables different constituent models, which form a coupled system, to be modelled in a distributed manner and then simulated in collaboration [5, 6]. Hence, the modelling is carried out at the constituent model level without a detailed understanding of the other constituent models. A challenge in co-simulation is to synchronise the different simulating units ensuring that the timing of the overall simulation is sufficiently close to how this would work in reality. In such co-simulations it is often convenient to combine Discrete Event (DE) formalisms (typically describing cyber control aspects) with Continuous-Time (CT) formalisms (usually describing physical phenomena being controlled). Enabling such hybrid combinations generally require some kind of coordination and in this paper the focus is on the Functional Mock-up Interface (FMI) standard.

The contribution of this paper is to enable VDM-RT models to be exported as FMUs such that these models can be incorporated in a setting where some of the constituent models are made using Overture while others are made other other tools supporting FMI version 2.0. In this way the extension described here extend the places where Overture can be used in a CPS context for DE models. This provides a more abstract language for modelling and developing FMUs as opposed to implementing them in a native language, which can be beneficial in the systems engineering process [8].

In Sect. 2 we provide a short introduction to the background of this work. Next, Sect. 3 demonstrates how Overture can be used to produce FMUs by means of a small case study. Section 4 presents an overview of the architecture of this capability. Finally, Sect. 5 gives a few concluding remarks.

2 Background

The VDM-RT dialect historically started off as a notation called "VDM In Constrained Environments" (VICE) [18]. However, VICE performed poorly in the analysis of distributed systems [22]. Thus the notation was rethought, and extended with support for distribution and called VDM-RT [24]. Initial work with co-simulation using VDM-RT and 20-sim was carried out in Marcel Verhoef's PhD thesis [23]. VDM-RT was then further developed in a co-simulation context inside the DESTECS project [2]. The main result here was the Crescendo tool [14] combining DE formalism VDM-RT [10] with the CT formalism bond graphs using the 20-sim tool [9]. The co-simulation carried out here was bespoke and worked in general between these two tools. However, it was also demonstrated in DESTECS that it was possible to use Matlab/Simulink instead of 20-sim via an XML-RPC interface (revisited in Sect. 4).

Subsequently the INTO-CPS project [4] took this further using the FMI standard to achieve an open tool chain enabling any modelling and simulation tool able to produce Functional Mock-up Units using version 2.0 of the standard to be co-simulated [12]. The coordination of this co-simulation is performed by the INTO-CPS Co-simulation Orchestration Engine called Maestro [21]. FMI is a result of the MODELISAR project [7] and it is a tool independent standard for model exchange and co-simulation, where we only concern ourselves with the co-simulation part in this article. FMI defines a C interface that simulation units must implement in order to participate in a co-simulation. A simulation unit implementing the FMI interface is called a Functional Mock-up Unit (FMU). Such an FMU is packaged as a Zip archive, which contains libraries for the platforms that the executable part of the FMU has been compiled for, a model description file describing the scalar variables and their causality (input, output, parameter etc.) of the FMU, and a resources folder containing elements used internally by the libraries. The iteration carried out by a co-simulation master is roughly equivalent to getting inputs, setting outputs, and invoking the FMUs to progress for a determined step size. The process is repeated until a predetermined end time is reached.

An extension to FMI that adds an additional function to the interface called `GetMaxStepSize` has been proposed [3]. The purpose is that each FMU can be queried for the maximum step it can perform, and then the chosen step size is the minimum of all the reported step sizes, as an FMU always must be able to perform a smaller step than the

reported maximum step. This makes it possible to avoid rolling back FMUs by setting a previous retrieved state, a feature not supported by VDM-RT FMUs. Furthermore, it makes it possible to synchronise at the specific point in time where an output is changed by a DE FMU. Besides detecting the point in time to synchronise it also makes it unnecessary to execute the co-simulation with a small time step to ensure detection of the changes in outputs, as the proper step size is reported by the FMU in question. The querying for maximum step sizes by the co-simulation master would occur after setting inputs and before invoking the FMUs to progress in the iteration described above.

3 Developing an FMU with Overture

In this section the FMI additions to a VDM-RT project required to export the project as an FMU is presented. This presentation concerns the structure of a VDM-RT project in order to be exported as an FMU and the template generated by the plugin. Afterwards, an example of a co-simulation is demonstrated where one of the FMUs is generated by using the plugin. Overture can generate both tool-wrapper FMUs and source code FMUs [1]. In this paper we focus on tool-wrapper FMUs.

3.1 FMI Additions for VDM-RT using Overture FMU

The Overture FMU plugin contains functionality to automatically generate a project template that complies with the required structure. This template and thereby the required structure is the following:

- A VDM-RT system `System` containing the definition of a given system by describing how different parts are deployed to different Core Processing Units (CPUs) [13]. This is not a class, but a system. The syntax is similar to ordinary classes with some differences, for example that it cannot be instantiated.
- A conventional [11] VDM-RT class `World` that provides an entry point into the model.
- A VDM-RT class called `HardwareInterface`, which is exemplified below in Sect. 3.3. This class contains the definition of the ports of the FMU. Its structure is enforced, and a self-documenting annotation scheme[1] is used such that the `HardwareInterface` class may be hand-written. The annotation format is `-- @ interface: type = [input/output/parameter/local], name="...";` and must be located directly above a **`value`** or an **`instance variable`** of one of the subclasses of the `Port` class described below. Ports of type parameter must be **`values`**, and all other ports must be **`instance variables`** of the class `HardwareInterface`. The reason for this approach is to capture all assumptions about FMI in the VDM-RT model itself opposed to extending the VDM-RT language or providing addition configuration files. This provides a solution where the generic FMI interface can be defined in a library and any instantiation

[1] The annotation scheme is documented on the INTO-CPS website http://into-cps-association.GitHub.io under "*Constituent Model Development → Overture → FMU Import/Export*.

hereof can be type checked with the concrete specification. Annotations must be provided since the FMI causality cannot be deduced automatically. Furthermore, these annotations convey the causality of the ports and has not resulted in any changes of the VDM-RT language, since they are written in comments.

– The library file `Fmi.vdmrt` defines the hardware interface port types used in `HardwareInterface`. This file contains an inheritance structure with a top-level generic `Port` class that is subclassed by ports for each FMI type: Bool, Real, Int and String. These subclasses are constructed with an initial value and contain get and set methods. Part of the class is presented in Listing 1; it also contains `StringPort`, `RealPort` and `BoolPort` implemented in a similar fashion to `IntPort`. The `getValue` function is declared **`pure`**, which means it does not update state (and other constraints described in [13].

```
class Port

types
  public String = seq of char;
  public FmiPortType = bool | real | int | String;

operations
  public setValue : FmiPortType ==> ()
  setValue(v) == is subclass responsibility;

  public pure getValue : () ==> FmiPortType
  getValue() == is subclass responsibility;

end Port

class IntPort is subclass of Port

instance variables
  value: int:=0;

operations
  public IntPort: int ==> IntPort
  IntPort(v)==setValue(v);

  public setValue : int ==> ()
  setValue(v) ==value :=v;

  public pure getValue : () ==> int
  getValue() == return value;

end IntPort
```

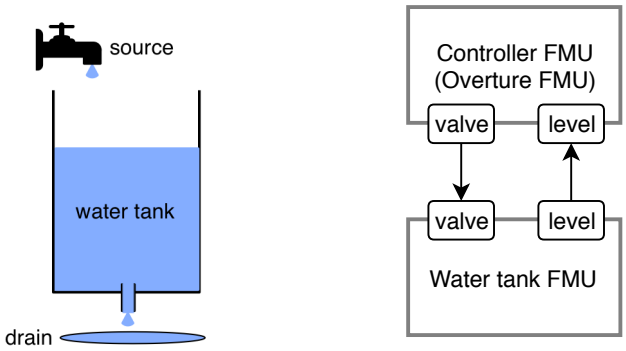
Listing 1: FMI library for VDM-RT containing `Port` class and subclasses for each FMI type.

After this required project structure is set up, the behaviour of the FMU must be implemented, which is demonstrated in the following section.

3.2 Overture FMU Example

In this section the controller of a water tank [17] shown in Fig. 1a is presented[2]. Afterwards, it is described in Sect. 3.4 how the FMI extension `GetMaxStepSize` maps to a VDM-RT model. Finally, Sect. 3.5 presents the results of a co-simulation where the VDM-RT model and Overture FMU is used.

The water tank is equipped with a source of water, two sensors, representing minimum and maximum water level, and a valve. When the valve is open, water pours out of the tank, and when the valve is closed, the water level rises, as the water is still flowing from the source. The water level is regulated by a controller expressed in a DE model using Overture and VDM-RT, which models the actuator that opens the valve when a maximum water level is reached and closes the valve when a minimum water level is reached. The draining and filling of the water tank and thereby the water level is expressed in a CT model described in [17]. The FMUs, their ports and dependencies between them are shown in Fig. 1b.



(a) Water tank example [17] (b) Water tank FMUs

Fig. 1: Overview of the water tank

3.3 VDM-RT Model

The model realisation in VDM-RT is structured as presented in Fig. 2, which matches the description of the template in Sect. 3.1. The realisation is described below.

[2] The other FMU describing the CT part of the water tank is not described here; the interested reader is referred to [17].

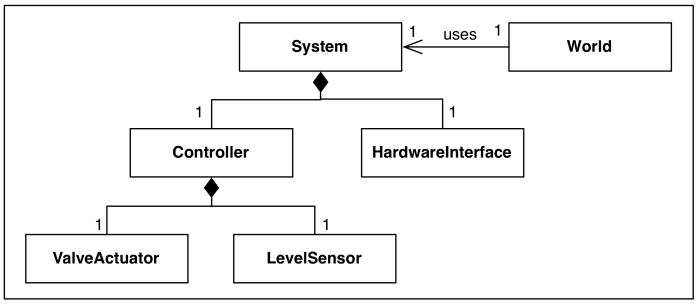


Fig. 2: Architecture of the VDM-RT model [17]

The `HardwareInterface` class is the interface of the DE model. In order to determine this interface it is necessary to consider the entire water tank system. As the state of the valve is operated by the DE model and has an impact on the calculation of the water level performed by the CT model, it must be an output from the DE model to an input on the CT model. The water level is calculated by the CT model and the DE model requires this information to determine whether to open or close the valve, and therefore it is an output from the CT model to an input on the DE model. Furthermore, it is necessary with two parameters on the DE model describing the minimum and maximum water level, as the DE model controls the state of the valve. The source of water is embedded in the CT model and therefore not considered. This leads to the interface presented in Listing 2, where the valve state has the type `BoolPort`, the water level has the type `RealPort`, and the parameters have the type `RealPort`. The value of the parameters can initially be changed by the co-simulation master based on the co-simulation configuration.

```
class HardwareInterface

values
  -- @ interface: type = parameter, name="minlevel";
  public minlevel : RealPort = new RealPort(1.0);
  -- @ interface: type = parameter, name="maxlevel";
  public maxlevel : RealPort = new RealPort(2.0);

instance variables
  -- @ interface: type = input, name="level";
  public level : RealPort := new RealPort(0.0);

  -- @ interface: type = output, name="valve";
  public valveState : BoolPort := new BoolPort(false);

end HardwareInterface
```

Listing 2: The hardware interface of the water tank controller

The `LevelSensor` class in Listing 3 encapsulates the port representing the level input. Notice that the set method is absent as level is an input, and therefore it is only possible to read a value from the port.

```
class LevelSensor
instance variables
  port : RealPort;
operations
  public LevelSensor: RealPort ==> LevelSensor
  LevelSensor(p) == port := p;

  public getLevel: () ==> real
  getLevel()== return port.getValue();
end LevelSensor
```

Listing 3: The encapsulation class for the water level sensor

The `ValveActuator` class in Listing 4 is similar in structure to the `LevelSensor` described above, but it captures the valve output instead of an input. It follows that the get method is absent, as valve is an output, and therefore it is only possible to write a value to the port.

```
class ValveActuator
instance variables
  port : BoolPort;
operations
  public ValveActuator: Port ==> ValveActuator
  ValveActuator(p) == port := p;

  public setValve: bool ==> ()
  setValve(value)== port.setValue(value);
end ValveActuator
```

Listing 4: The encapsulation class for the valve actuator

The `Controller` class in Listing 5 is the core logic of the DE model. It is instantiated with the `LevelSensor` and `ValveActuator` instances described above. The behaviour is contained in the `loop` operation, which takes 2 cycles[3] and runs every 10

[3] A cycles or duration statement at the top level of the loop operation as in this case can lead to undesired behaviour. Everything within the body of the cycles statement is executed atomically with the given cycle number, and thus prevents the scheduler from swapping out the current atomic block. As a result, periodic threads will not have the next period thread swapped before the current is completed. Therefore, no overlapping errors will be raised because the next period threads are not yet executing, even though the period has elapsed. This can be seen by setting the CPU frequency in Listing 6 to e.g. 20 Hz, thereby the cycles would take 50 milliseconds, but the period is still 10 milliseconds but no error is reported.

milliseconds until the simulation terminates. 2 cycles in this case corresponds to exactly 10 milliseconds and is calculated as:

$$\tau = cycles/freq_{CPU} = 2/200Hz = 0.01seconds$$

where τ is time, $cycles$ is the number of cycles from Listing 5, and $freq_{CPU}$ is the CPU frequency from Listing 6.

The behaviour of the loop operation is first to read the level, check whether it is above the maximum level or below the minimum level and open or close the valve respectively.

```
class Controller

instance variables
  levelSensor   : LevelSensor;
  valveActuator : ValveActuator;

values
  open : bool = true;
  close: bool = false;

operations
  public Controller : LevelSensor * ValveActuator  ==> Controller
  Controller(l,v)==
  (
    levelSensor   := l;
    valveActuator := v;
  );

  private loop : () ==>()
  loop()==
    cycles(2)
    ( let level : real = levelSensor.getLevel() in
      ( if( level >= HardwareInterface`maxlevel.getValue())
        then valveActuator.setValve(open);

        if( level <= HardwareInterface`minlevel.getValue())
      then valveActuator.setValve(close); );
    );

thread
  periodic(10E6,0,0,0)(loop);

end Controller
```

Listing 5: The `Controller` class with the core logic

The system entity `System` shown in Listing 6 is responsible for describing how the `controller` class of the water tank controller is deployed to a CPU and how it

is connected to other parts in the model. Therefore `System` instantiates the hardware interface, instantiates and initialises the hardware encapsulation classes and passes these to the `Controller`, which is also instantiated and deployed on a CPU.

```
system System

instance variables
  -- Hardware interface variable required by FMU Import/Export
  public static hwi: HardwareInterface:= new HardwareInterface();
  public levelSensor : LevelSensor;
  public valveActuator : ValveActuator;
  public static controller : [Controller] := nil;
  cpu1 : CPU := new CPU(<FP>, 200);

operations
public System : () ==> System
  System () ==
  ( levelSensor := new LevelSensor(hwi.level);
    valveActuator :=  new ValveActuator(hwi.valveState);
    controller := new Controller(levelSensor, valveActuator);
    cpu1.deploy(controller,"Controller");
  );

end System
```

Listing 6: System class of the DE model

The `World` class launches the simulation by invoking the **`start`** statement in the `Controller` class instance, which is contained in `System` described above. This leads to the thread contained within the `Controller` class described above to be started. The implementation of the `World` class is presented below in Listing 7.

```
class World
operations
  public run : () ==> ()
  run() ==
  ( start(System`controller);
    block(); );

  private block : () ==>()
  block() == skip;

sync
  per block => false;
end World
```

Listing 7: World class of the DE model

3.4 Synchronisation

Synchronisation in terms of FMI is when outputs are exchanged. From an Overture FMU perspective, the synchronisation should ideally occur just before reading a value from a port and after writing to a port. This ensures synchronisation exactly when an output has changed or allows for retrieving updated inputs just before an input is read. Listing 8 shows an implementation of the `loop` operation, where the cycles statement is removed and thereby expressions and statements takes 2 cycles. This allows synchronisation at the desired synchronisation points, which `GetMaxStepSize` will return. The VDM-RT interpreter makes use of transactions [16], in the sense that it calculates the behaviour until the next synchronisation point, but does not commit it and thereby it is not exposed until the correct point in time. Thereby it is possible to calculate the value that `GetMaxStepSize` as the minimum time of all transactions as:

$$min(\{\tau|(\Sigma,\tau)\ \in\ T\}) - \tau_{now}$$

where (Σ, τ) is a transaction pair of state Σ to expose at time τ, T is a set of all transactions across CPUs, and τ_{now} is the global current time of the co-simulation.

```
private loop : () ==>()
loop()==
   -- SYNCHRONISATION
   let level : real = levelSensor.getLevel() in
   (  if( level >= HardwareInterface`maxlevel.getValue())
      then valveActuator.setValve(open);
      -- SYNCHRONISATION if condition yields true

      if( level <= HardwareInterface`minlevel.getValue())
      then valveActuator.setValve(close);
      -- SYNCHRONISATION if condition yields true
  );
```

Listing 8: Control loop the DE model with desired synchronisations.

3.5 Co-simulation Result

The result of a co-simulation of the water tank using the VDM-RT model is presented in Fig. 3. It shows that when the water level exceeds the maximum water level of two the valve opens, represented by the value 1. It remains open until the water level is below the minimum water level of one, at which point the valve is closed represented by the value 0. The step size of the co-simulation is 0.1 seconds. The small step delay is a result of the Jacobian master algorithm [19] used by the employed co-simulation orchestration engine. Instead one could use the Gauss-Seidel [19] master algorithm. This particular issue is addressed in [20].

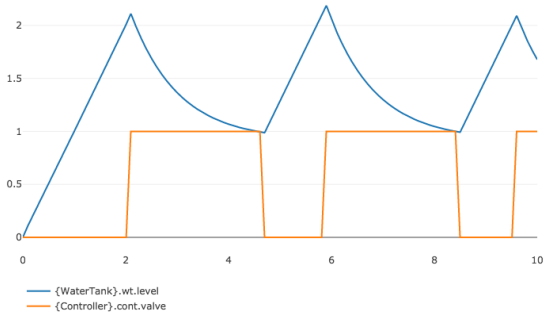


Fig. 3: Result of a co-simulation of the water tank

4 Architecture of Overture FMU

The architecture of the Overture FMU and the flow of messages is shown in Fig. 4[4]. The Overture FMU product is split into three parts that communicate via shared memory and protobuf messages. The first part, FMU, defines the FMU library that is invoked by the co-simulation master. The next part, Shared Memory (SHM), are the libraries involved in converting the data to Protobuf messages and using shared memory to pass data from the co-simulation Master to the last part, Model Execution, and back. The third and last part, Model Execution, describes the functionality that carries out the simulation of the model, where the Java application Overture-FMU[5] essentially is a Crescendo implementation [15] with a different protocol. The reason for this structuring is, that the SHM part is implemented in such a way, that it could easily be adapted to contain other messages, that are not FMI-specific. The three parts are described below along with the loading process of an Overture FMU and transferring of messages between the different parts.

In order to understand this section, some terminology must be known:

Protobuf: Protobuf[6] is the short name for Protocol Buffers, which is a language and platform-neutral extensible mechanism for serialising structured data developed by Google. It supports Java and C++ among others, which is used in the development of the Overture FMU described in Sect. 4.

Java Native Interface (JNI): JNI is a framework that makes it possible for Java applications to communicate with native libraries. This is also used in the Overture FMU.

[4] The libraries mentioned in the figure is part of the contribution of the work presented in this paper, unless explicitly stated otherwise.

[5] Notice the hyphen, which differentiates the application (Overture-FMU) from the name of the product (Overture FMU).

[6] Available at https://developers.google.com/protocol-buffers/

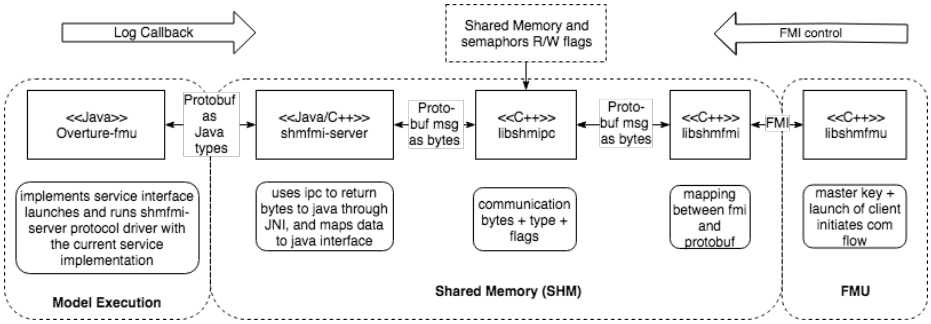


Fig. 4: Architecture of Overture FMU split in three parts parts: FMU, Shared Memory (SHM), and Model Execution.

4.1 FMU

The overall responsibility of the FMU block[7] is to provide a C implementation of the FMI interface allowing it to serve as an FMU. libshmfmu is therefore the FMU implementation adhering to the FMI interface that is loaded when Maestro loads the library inside the FMU. It initialises the communication flow, instantiates the SHM libraries libshmfmi and libshmipc, and launches Overture-FMU, which is described in Sect. 4.3. When a function of FMU is invoked, the function invocation information is passed to the libshmfmi within SHM. The master key mentioned in the figure is a session key ensuring that multiple Overture tool-wrapper FMUs can coexist without interfering with each other.

4.2 Shared Memory (SHM)

The SHM part is responsible for passing messages between Model Execution and FMU using shared memory. This involves mapping each FMI function invocation to protobuf messages and the other way around. These messages are sent through a fixed-size shared memory space with read/write access being controlled by several semaphores. This native part had to be developed almost without any frameworks, as most of the frameworks suitable for the task could not be cross compiled with a reasonable effort. It was challenging to ensure cross compilation, which is an important feature. Concretely, the implementation is split into three libraries that are described below:

libshmfmi: This library does the mapping of FMI function invocations to protobuf messages, which is stored in the shared memory. Furthermore, it maps the response from protobuf messages to FMI.

libshmipc: This library embeds the shared memory required for communication and two semaphores to control access.

shmfmi-server: The shmfmi-server contains the bridge between Java and native code required to extract bytes from the shared memory, convert them to protobuf messages

[7] The source code is available at https://github.com/overturetool/shm-fmi.

and invoke the relevant functions in Overture-FMU described below. It invokes the relevant functions by exposing a Java interface, which defines the FMI calls with protobuf data types, that is implemented by CrescendoFMU, which is part of Overture-FMU. This Java Interface defines the FMI calls with protobuf data types. Furthermore, it also performs the mapping the other way with replies. Note, that this also operates in reverse, as there are callbacks in FMI e.g. for logging. Conceptually this is simple, but the implementation at the low level is not trivial. Concretely, shmfmi-server extracts bytes from the shared memory and embeds the JNI interface, which enables access from Java. It is instantiated by Overture-FMU in the Model Execution part.

As mentioned in Sect. 2 Crescendo features an XML-RPC interface, which is an alternative to this shared memory approach. The reason for choosing to use shared memory is that XML-RPC uses XML and a socket, which is slower in terms of performance than Java or native C calls. Furthermore, Crescendo did not feature `GetMaxStepSize`, so the Crescendo protocol would have to be changed in order to achieve the same functionality. Additionally, Crescendo was co-simulation between two instances and not N instances, so the slow down per simulator would be significant. Knowing that the shared memory approach is faster, it is a better solution in this case. It was also envisioned that the SHM functionality can be reused for other projects and for the C code generator described in [1], which it was not unfortunately.

4.3 Model Execution

This represents the left-hand side of Fig. 4 and thus the actual execution of the VDM-RT model. It contains the Java application Overture-FMU[8] that is launched by libshmfmu and acts as an interface to the Crescendo implementation. As mentioned in Sect. 2 VDM-RT was used in a co-simulation context in the DESTECS project, and therefore the main functionality of participating in a co-simulation was already present. It was therefore of interest to preserve most of the main functionality, but it was necessary to make changes to the interface in order to support FMI. The extension was realised by exposing the simulation driver of Crescendo, thereby enabling overriding. Thus Overture-FMU is an application that interprets FMI messages and utilises Crescendo to execute the model, and then adapts the Crescendo response to FMI. A low level detail is, that the VDM-RT interpreter [16] is packaged inside the resources folder of an FMU, mentioned in Sect. 2. The significant development additions in order to perform this adaption consist of the elements described below:

New Entry Point (main): This receives the master key to the shared memory as an argument and connects to the existing SHM via shmfmi-server described above. Afterwards, it instantiates the CrescendoFMU described below.

Mediation Between FMI and Crescendo (CrescendoFMU): This invokes the FMISimulationManager to perform a given task based on the protobuf message converted to Java by shmfmi-server. Afterwards, it creates an FMI Protobuf reply, which is returned to shmfmi-server. It implements the interface described in shmfmi-server above.

[8] The source code is available at https://github.com/overturetool/overture-fmu.

Extension of the SimulationManager (FMISimulationManager): In Crescendo the controlling entity was inherent in the VDM-RT resource scheduler, but this is not the case when using FMI, where Maestro is the controlling entity, and therefore additional changes had to be carried out. Furthermore, the synchronisation is different. The FMISimulationManager ensures that the interpreter only calculates to a certain time. It will calculate until just before it needs to read an input and right after an output, as this is where the synchronisation should occur. This way the interpreter is "ahead" of the global co-simulation time, as described in Sect. 3.4, but capable of calculating a value for `GetMaxStepSize` presented in Sect. 2.

Handling State (StateCache): The StateCache is necessary because of the way Crescendo operates, where the execution of a step takes all inputs and returns all outputs. However, FMI operates differently where the `setinput`, `dostep`, and `getoutput` calls are separated. Therefore this cache was added in order to support FMI.

5 Concluding Remarks

An advantage of a tool-wrapper FMU as opposed to source code / native FMUs is that the model is interpreted exactly as the language describes and modifications to the VDM-RT language are available out of the box. Furthermore, it allowed reuse of the existing tooling without changes to the interpreter. A disadvantage is, that it requires prerequisites to execute a tool-wrapper FMU, e.g. that Java is installed and available, and it is most likely slower in terms of performance compared to a native FMU. Future work on the Overture FMU tool-wrapper is to test with multiple FMI engines and publish the results on the FMI tools website [9]. Additionally, it would be interesting to perform benchmarks and comparisons with other FMUs. A new version of the FMI Standard is also under development, and Overture FMU should be updated to support this.

In this article it has been shown how one can use VDM-RT and Overture to develop a tool-wrapper FMU that can participate in an FMI co-simulation. This has been exemplified by realising a DE controller of a water tank system using Overture FMU and co-simulating it. Additionally, the architecture of a tool-wrapper Overture FMU has been described in depth. It contains native libraries, as an FMU must expose a C interface, that communicates with other shared native libraries over shared memory protected by semaphores. Furthermore, this also involves launching a Java application that reuses functionality from the Crescendo tool [2]. Emphasis has also been placed on describing how calculation of step sizes and synchronisation is carried out, as Overture FMU is unique in this field. Overture FMU has been successfully used in several industrial case studies as part of the INTO-CPS project [4].

Acknowledgments The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047. We would like to thank all the participants of those projects for their efforts making this a reality. Furthermore, we would like to thank the anonymous reviewers for their comments, which helped to improve the paper.

[9] Available at http://fmi-standard.org/tools/

References

1. Bandur, V., Tran-Jørgensen, P.W.V., Hasanagic, M., Lausdahl, K.: Code-generating VDM for Embedded Devices. In: Fitzgerald, J., Tran-Jørgensen, P.W.V., Oda, T. (eds.) Proceedings of the 15th Overture Workshop. pp. 1–15. Newcastle University, Computing Science. Technical Report Series. CS-TR- 1513 (September 2017)
2. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., Wouters, F.: Design Support and Tooling for Dependable Embedded Control Software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. pp. 77–82. ACM (April 2010)
3. Broman, D., Brooks, C., Greenberg, L., Lee, E., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of FMUs for co-simulation. In: Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on. pp. 1–12 (2013)
4. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)
5. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. ACM Comput. Surv. Accepted on January 11, 2018 for publication in ACM Computing Surveys
6. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. Tech. rep. (feb 2017), http://arxiv.org/abs/1702.00686
7. ITEA Office Association: Itea 3 project, 07006 modelisar. https://itea3.org/project/modelisar.html (December 2015)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Heyward Street, Cambridge, MA02142, USA (April 2006), iSBN-10: 0-262-10114-9
9. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
10. 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 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
11. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
12. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: CPS Data Workshop. Vienna, Austria (April 2016)
13. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: The VDM-10 Language Manual. Tech. Rep. TR-2010-06, The Overture Open Source Initiative (April 2010)
14. Larsen, P.G., Lausdahl, K., Coleman, J., Wolff, S., Kleijn, C., Groen, F.: Crescendo Tool Support: User Manual. Tech. Rep. TR-001, The Crescendo Initiative, www.crescendotool.org (November 2013)
15. Lausdahl, K., Coleman, J.W., Larsen, P.G.: Towards a co-simulation semantics of VDM-RT/Overture and 20-sim. In: Plat, N., Nielsen, C.B., Riddle, S. (eds.) Proceedings of the 10th Overture Workshop. pp. 30–37. No. CS-TR-1345 in Technical Report Series, Computing Science, Newcastle University (August 2012), http://www.cs.ncl.ac.uk/publications/trs/papers/1345.pdf

16. Lausdahl, K., Coleman, J.W., Larsen, P.G.: Semantics of the VDM Real-Time Dialect. Tech. Rep. ECE-TR-13, Aarhus University (April 2013)
17. Mansfield, M., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 3. Tech. rep., INTO-CPS Deliverable, D3.6 (December 2017)
18. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In: Bicarregui, J., Fitzgerald, J. (eds.) Proceedings of the Second VDM Workshop (September 2000)
19. Petridis, K., Clauß;, C.: Test of Basic Co-Simulation Algorithms Using FMI. In: Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015. pp. 865–872. No. 118, Fraunhofer IIS EAS, Zeunerstrasse 38, 01069 Dresden, Germany, Linköping University Electronic Press, Linköpings universitet (2015)
20. Thule, C., Gomes, C., Deantoni, J., Larsen, P.G., Brauer, J., Vangheluwe, H.: Towards the Verification of Hybrid Co-simulation Algorithms (2018), accepted for publication at CoSim-CPS-18
21. Thule, C., Lausdahl, K., Larsen, P.G., Meisl, G.: Maestro: The INTO-CPS Co-Simulation Orchestration Engine (2018), submitted to Simulation Modelling Practice and Theory
22. Verhoef, M.: On the Use of VDM++ for Specifying Real-Time Systems. Proc. First Overture workshop (November 2005)
23. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
24. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

All links were last followed on May 24rd, 2018.

ViennaVM: a Virtual Machine for VDM-SL development

Tomohiro Oda[1], Keijiro Araki[2], and Peter Gorm Larsen[3]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] National Institute of Technology, Kumamoto College (araki@kyudai.jp)
[3] Aarhus University, Department of Engineering, (pgl@eng.au.dk)

Abstract. The executable subset of VDM allows code generators to automatically produce program code. A lot of research have been conducted on automated code generators. Virtual machines are common platforms of executing program code. Those virtual machines demand rigorous implementation and in return give portability among different operating systems and CPUs. This paper introduces a virtual machine called ViennaVM which is formally defined in VDM-SL and still under development. The objective of ViennaVM is to serve as a target platform of code generators from VDM specifications.

1 Introduction

Quality of software systems is important in many cases. Model-based development with automated code generation techniques is a promising approach to develop software systems with affordable quality and productivity. Many automated code generators from different VDM dialects [6] have been studied and developed as strong tools to reduce cost of the implementation phase [3,1,7,10]. Those automated code generators emit source code for general programming languages such as C++, C, Java and Smalltalk. However, there are still challenges with applying code generators. The first challenge is the availability of compiler and runtime environments for various target hardware. General programming language systems often provide rich language with build-in functions and libraries. Thus, it is often costly to port the compiler and full set of libraries to brand-new hardware platforms. The second challenge is the portability of the compiler and runtime environment. In some programming languages that provide low-level programming functionality, such as C and C++, does not provide the source level compatibility among different platforms.

This paper proposes and introduces the development of a Virtual Machine (VM) named ViennaVM that is designed as a common target platform for automated code generators from VDM dialects. A VM is an abstracted computer platform targeting efficient execution of code represented in an Intermediate Representation (IR) [9]. IR code is typically designed specific to a particular guest programming language. For example, the Java VM executes Java byte-code of which instruction set efficiently implements the language features of Java, such as primitive operations, boxing/unboxing and method invocations. ViennaVM will have an instruction set suitable to model-based developments with VDM.

One significant advantage of VMs is portability. For example, the VM for Squeak Smalltalk is auto-generated from Squeak Smalltalk itself with the exception of several platform dependent interface to the host operating system. Having a small fraction of hand-written code, the VM of Squeak Smalltalk was ported to various platforms including Windows, Unix and PDAs [2].

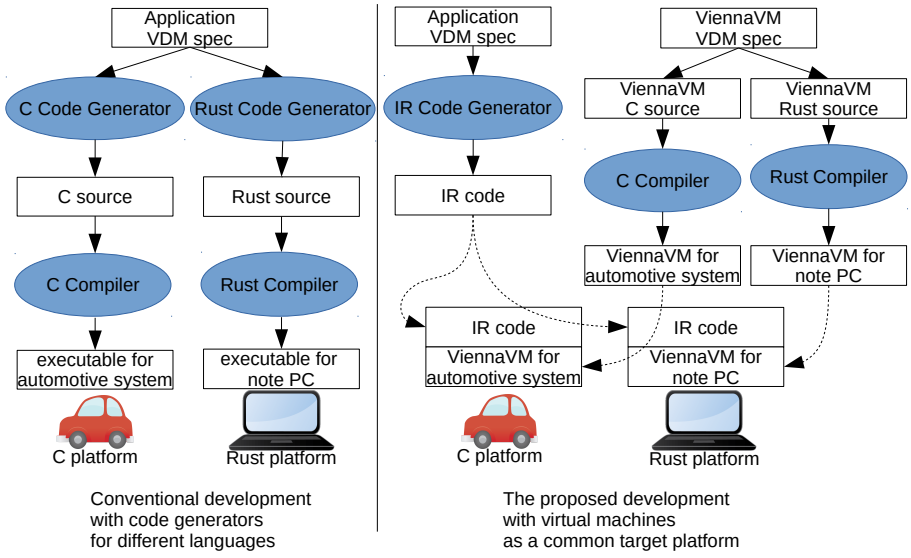


Fig. 1. Code generator-based development and VMs as a common platform

Figure 1 shows how ViennaVM will serve as a common target platform for code generators. Assuming that platform A provides only a C compiler and platform B offers Rust as its standard programming language, two source code, one for C and another for Rust would be generated. Each source would need modification to work with external libraries, such as networks and user interfaces. Using a VM as a common target platform, it is possible to implement one ViennaVM in C and another in Rust to run the same IR code. ViennaVM's IR code is binary compatible among versions of VMs in different programming languages on various target platforms. Each instance of ViennaVM can also be reused in later developments of other applications on the same device.

The rest of this paper starts of with an overview of the objective of ViennaVM in Section 2. Afterwards an overview of the formal specification of ViennaVM using VDM-SL is provided in Section 3. Then the preliminary implementation of a ViennaVM as well as initial benchmarks are provided in Section 4. This is followed by Section 5 about the planned further development. Finally, Section 6 provides a few concluding remarks about the possibilities for this work.

2 Objectives and Non-functional requirements of ViennaVM

The objective of ViennaVM is to provide a common target platform for software development with VDM dialects. To achieve the objective, it is desirable that the VM is *reliable*, *portable*, *productive* and *adaptable* to the host platform. This section explains why these non-functional qualities are required to VMs for smart devices.

In order to make ViennaVM dependable it will itself be developed using VDM-SL. We split the specification process into two phases: an exploratory specification phase that we use ViennaTalk [8] as a development platform, and rigorous specification phase that we use the Overture tool [4] to gain and ensure the quality of the specification. ViennaTalk is a Smalltalk-based IDE with live specification animations for interactive specification authoring in an exploratory manner. ViennaTalk also provides a pretty printer and automated execution of unit tests to enhance agility-related qualities of specification against frequent modifications. ViennaVM will be tested using a unit testing framework on ViennaTalk, called ViennaUnit. ViennaUnit is a simple unit testing framework for VDM-SL that automatically collect test modules whose names end with `Test` and automatically run all test operations whose name start with `test`. After developing a valid specification of ViennaVM, the Overture tool will be used as a development platform. The Overture tool is full-fledged IDE based on the Eclipse platform. The Overture tool's functionality includes combinatory testing and automated generation of proof obligations that enhance rigour qualities of specification as the final product of the specification phase. Combinatory testing will be used to rigorously test a large number of combinations of IR code [5].

ViennaVM needs to be portable. The term *portable* has two sides; one is the portability of ViennaVM, and the second is the portability of IR code. The portability of the both sides is required to software systems distributed among various platforms. The term *productivity* also has two sides; the productivity of ViennaVM and the productivity of the target software on ViennaVM. For the productivity of the VM, the combined use of ViennaTalk and Overture as a tool chain will be a significant factor. For the productivity of the target software, ViennaVM will be ported to Smalltalk so that the target system can be seamlessly developed on ViennaTalk and ViennaVM as a tool chain.

ViennaVM needs to be adaptable to different host platforms. Considering diverse constraints on hardware such as user interfaces and computational resources, ViennaVM needs to be implemented differently according to those constraints, yet satisfying its formal specification. One smart device may be equipped with voice cognition and speech synthesis while another smart device may have small touch screen in a few square centimetres and a physical push button. It is desirable that one program in IR code works on every smart device without having redundant UI code because those devices typically have limited computational resources. Conventional VMs provides low-level interface to UI devices and standard libraries written in the IR code of VMs provide UI frameworks. For example, the Java VM provides the awt and swing frameworks in Java byte-code. Application developers implements UI code for different UI devices and choose either to create different deployment file for each host platform or to provide one deployment file that has all UI code. ViennaVM is planned to provide an abstraction of interactions with the user to make its applications adaptable to various smart devices with different physical user interfaces.

3 Specification of ViennaVM

This section explains the specification of ViennaVM. Although the formal definition is not complete yet, the current snapshot of the formal definition of ViennaVM can execute a simple numeric computation.

3.1 Definition layers

The current snapshot ViennaVM is specified in a moduled form of VDM-SL. Figure 2 shows modules in layers of definition.

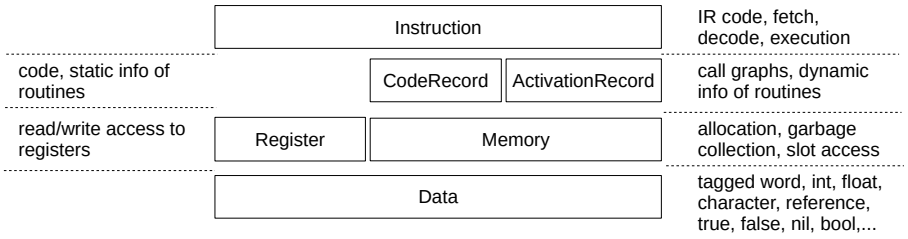


Fig. 2. Layers of the VDM-SL definition of ViennaVM

The bottom layer is the `Data` module that defines the data model of ViennaVM including data type definitions and constant values. Based upon the `Data` module, the `Register` module and the `Memory` module are defined. The `Register` module specifies the internal structure of each register, and the `Memory` module provides a memory model including data layout in a heap object and a garbage collector. ViennaVM is a register machine, which has large number of registers and passes arguments and return values via registers, while Java VM is a stack machine which handles temporary values in a data stack and passes arguments and return values via the data stack. `CodeRecord` and `ActivationRecords` are modules that defines heap objects that represents static and dynamic properties of routines. Then, the `Instruction` module defines IR code instructions and its execution mechanisms such as code fetch and a decoder. The sections below explains each module.

3.2 Data definitions

The `Data` module specifies the data types internally used in ViennaVM and also VDM's basic types. ViennaVM uses 64 bits tagged word as an atomic data entity in IR code. Tagged words are fixed sized data packed with runtime type informations so that the VM can uniformly handle values of different types. A tagged word can either be an integer, a floating point number, a character or a pointer with flags that identify its runtime type. A pointer is not a raw address of a heap object, but it is a reference which

tagged word vector	64 bits unsigned int (b0: int flag, b1: non-heap flag, b2: type flag, b3: option flag)		b63, ..., b4 b3 b2 b1 b0	
int	63bit signed int			1
pointer to value	56 bit unsigned int		0 0 0 0	0 0 0 0
float	8 bits dummy	IEEE 754 float32	0 0 0 1	0 0 1 0
unicode character	8 bits dummy	32 bits unicode	0 0 0 1	1 0 1 0
nil	56 bits dummy		0 0 1 0	0 0 1 0
true	56 bits dummy		0 0 1 0	1 0 1 0
false	56 bits dummy		0 0 1 1	0 0 1 0
unit type	56 bits dummy		0 0 0 0	0 1 1 0
bool type	56 bits dummy		0 0 0 0	0 1 1 0
:	:			
pointer to type	56 bit unsigned int		0 0 0 0	0 1 0 0
invalid word			0 0 0 0	1 0 1 0
int	1 bit dummy	63 bit signed int		
float	32bit dummy	IEEE 754 float32		
char	32bit dummy	32bit unsigned int		
pointer	8 bits dummy	56 bits unsigned int		

Fig. 3. Data format of primitive values inside ViennaVM

consists of the heap page index and offset of the heap object. The basic types and values of VDM are also defined in the `Data` module. Figure 3 lists the data definitions.

If the Least Significant Bit (LSB) of a tagged word is 1, the remaining 63 bits represents a signed integer. The second least significant bit of a tagged word is the non-heap flag that indicates whether the tagged word carries a pointer or not. If the second least significant bit is 0, the tagged word has a pointer. The third least significant bit is the type flag that indicates whether the data is a VDM's value or a VDM's type. For example, the **`bool`** type is encoded as `0x06` in the tagged word format, and the **`true`** value is encoded as `0x2a`.

The `Data` module also provides functions that converts values between a tagged word and a primitive value namely integer, float, character or a pointer. Figure 4 is an excerpt from the `Data` module. Because VDM-SL does not have bit manipulation operators, arithmetic operators on integers are used instead.

3.3 Registers

ViennaVM has 2^{16} data registers. Each data register has five statically typed fields, namely `oid` (tagged word), `i` (integer), `f` (floating point number), `c` (character) and `p` (pointer). Figure 5 is an excerpt from the definition of ViennaVM's registers in VDM-SL. The `read_int` operation accepts a 16 bits register ID and look for a cached value in the `i` field of the specified register. If the cache is invalid, it yields a 63 bits signed integer value. Because fields in a register and tagged words in memory slots are strongly typed, IR code has strongly typed semantics. Type safety at IR code level will contribute to reliability of applications.

One major overhead of tagged words is the cost of tagging and untagging. Some VMs provide explicit tagging and untagging instructions to convert values. The explicit

```
functions
    oid2int : OID -> Int
    oid2int(oid) ==
        if oid mod 2 = 1
        then
            (if oid <= 0x8000000000000000
             then oid div 2
             else oid div 2 - 0x8000000000000000)
        else
            invalidIntValue;

    int2oid : Int -> OID
    int2oid(i) ==
        if i <> invalidIntValue
        then
            (if i >= 0
             then i  * 2 + 1
             else (0x8000000000000000 + i) * 2 + smallIntegerTag)
        else
            invalidOidValue;
```

Fig. 4. Data conversion functions defined in VDM-SL

tagging/untagging naturally requires those tagging and untagging instructions in the IR code which makes the IR code larger. Also, erroneous IR code may possibly mix up values of wrong types, e.g. use an integer value as a pointer. Other VMs handles only first class objects in the form of tagged words. This approach brings an overhead of massive tagging and untagging operations. For example, when evaluating `(1 + 2) * 3`, the VM should untag to obtain the values `1` and `2`, compute `1 + 2`, tag the resulting value `3` to computer the `1+2` part. Then, the VM untag it, then untag to obtain the value `3`, compute `3 * 3` and then tag the resulting value `9`. This approach is costly in return of type safety.

The basic idea of ViennaVM's five statically typed fields per register is to cache the tagged and untagged data to avoid unnecessary tagging/untagging operations. When evaluating `(1 + 2) * 3`, ViennaVM stores the tagged value of `1` and `2` into the `oid` field of each register. Then, `ADD` instruction will read the integer values from the registers, which will implicitly untag and cache the integer values into the `i` fields, and then stores the resulting value `3` into the `i` field of another register. Then `MUL` instruction will read the integer value, which need untagging operation and stores the resulting value into the `i` field of a register. Instructions that read those registers can later read the integer values cached in the `i` fields. Wrong conversions, e.g. read a float value from a register that has integer value, can be detected at runtime. This approach using statically typed fields in a register can provide reduced cost of tagging and untagging while keeping type safety of the IR code.

```
types
     Reg ::
        oid : Data`OID
        i : Data`Int
        f : Data`Float
        c : Data`Char
        p : Data`Pointer;
     Register = nat inv r == r < 65536;
operations
    read_int : Register ==> Data`Int
    read_int(r) ==
        let reg : Reg = registers(r), i : [Data`Int] = reg.i
        in
            if i = Data`invalidIntValue
            then
                let i2 : Data`Int = Data`oid2int(reg.oid)
                in
                    (if i2 <> Data`invalidIntValue
                    then registers(r) .i := i2;
                    return i2)
            else return i;

    write_int : Register * Data`Int ==> ()
    write_int(r, i) ==
        (let p = registers(r).p
        in
            if p <> Data`invalidPointerValue
            then Memory`decrement_reference_count(p);
        registers(r)
            := mk_Reg(
                Data`invalidTag,
                i,
                Data`invalidFloatValue,
                Data`invalidCharValue,
                Data`invalidPointerValue));
```

Fig. 5. The definition of ViennaVM's registers in VDM-SL

Another benefit of this approach is that garbage collectors (GCs) can accurately count references from registers. Because the pointers in registers are stored in the `p` field, a GC can detect whether or not the value in a register is a pointer or not. Runtime type information at IR code level enables simple and reliable implementation of GC that does not rely on the correctness of the application code.

3.4 Memory model

The `Memory` module defines the memory model of ViennaVM. An object allocated in a heap space has slots that store tagged words and headers for memory management. Figure 6 shows the format of objects allocated in the heap space. Because all data other than the object headers are tagged word, a GC can retrieve type information from the binary data. Unlike most other VMs, ViennaVM's instructions and immediate values in the IR code page are also tagged words and thus subject to garbage collection.

offset	field name	type	description
0	SIZE_OFFSET	32 bits unsigned int	size of this object aligned by 64 bytes
4	FLAGS_OFFSET	32 bits unsigned int	flags for memory management
8	REFERENCE_COUNT_OFFSET	64 bits unsigned int	reference count for garbage collection
16	FORWARDER_OFFSET	64 bits unsigned int	link to the updated object
24	SLOTS_SIZE_OFFSET	64 bits unsigned int	number of slots in this object
32	SLOT1	64 bits tagged word	the first element of this object
40	SLOT2	64 bits tagged word	the second element of this object
	:		

Fig. 6. Data format of object in a heap space

The heap allocator gives 64 bytes alignment to the required size of an object. ViennaVM uses reference counters to collect unreferenced objects in the heap space. Although GCs based on reference counts have difficulty in detecting objects with cyclic references, ViennaVM assumes tree structured data from specifications in VDM-SL and therefore there will be no cyclic references. ViennaVM manages reference counters from tagged pointers not only in heap objects but also in registers. At every write access to a register or a memory slot, the contents of the old tagged word and the new tagged word are checked, and if a tagged word has a pointer, the reference counter will be increased and/or decreased.

3.5 Code Record and Activation Record

In ViennaVM, VDM functions and operations are implemented as *routines*. A code record is an object that represents a routine that consist of a series of IR code, type signature, precondition, postcondition, measure function and declaration of registers used in the IR code. An activation record, also known as a stack frame, is an object that represents an execution contexts that holds the caller activation record (the dynamic

link), the caller code record, the caller's instruction pointer, measure value, old state, register id to pass a return value and slots to save registers.

Code records and activation records are also stored as objects. Like other heap objects, code records and activation records are subject to garbage collection.

3.6 Instructions

ViennaVM's IR instruction set has basic instructions for memory allocation, data transfer, primitive operators, control structures (error, jump, conditional jump, call, recursive call, return and conditional return). Table 1 lists the basic instructions. Since tagged words in heap objects and registers have type information, data transfer instructions and primitive operator instructions perform runtime type checking by default. ViennaVM needs to be adaptive to different target platforms, so each implementation of ViennaVM may or may not perform such runtime checking to manage the balance between safety and computational costs.

Figure 7 is the definition of the `SUB` instruction in VDM-SL. The `SUB` instruction takes three operands each of which specifies a register ID. This instruction computes the second operand minus the third operand and stores the result into the first operand. The VM first checks the operands are specified. If any operand is omitted, the VM issues an error. The definitions part of the let statement defines data retrieval from the registers. The VM tries to read an integer value from the register specified by the second operand. If failed, it tries to read a float value from the same register. The VM does the same to the third operand. Then, the "in" clause of the let statement defines the computation and data transfer to the register specified by the first operand. If both values are successfully retrieved, the VM computes `num1 - num2` and stores it to the integer field of the destination register if the both arguments are integer values. Otherwise, it stores the resulting value to the float field of the destination register.

4 Example IR code and Preliminary Performance Evaluation

ViennaVM is still under development. We have created a prototypical implementation from the current snapshot of its specification in VDM-SL for a preliminary performance check. Although performance is not specifically pointed as a requirement to ViennaVM in Section 2, we conducted a preliminary performance test to check feasibility of the design of ViennaVM. The C version for now implements a subset of the full instruction set of ViennaVM.

4.1 Performance of Fibonacci

Figure 8 shows the specification of the benchmark function `fib`, the fibonacci sequence function. One call to the `fib` function with an argument larger than 1 will make two recursive calls, and therefore costs exponential time.

The fibonacci series function is implemented by C and also IR code of ViennaVM along with the executable VDM-SL specification. Figure 9 shows the IR code. Informal

Table 1. Basic instruction set of ViennaVM

opcode	operand1	operand2	operand3	description
ERR				Triggers error handler
NOP				Do nothing
ALLOC	r1			Allocate an object with the slot size given by r1
RESET				Restart with the latest image file
DUMP				Dump an image file
MOVE	r1	r2		copy r2 to r1
MOVEI	r1			copy the given immediate value to r1
LOAD	r1	r2	r3	copy the r3-th slot of the object pointed by r2 to r1
LOAD1	r1	r2	r3	copy the (r3+1)-th slot of the object pointed by r2 to r1
:				
LOAD7	r1	r2	r3	copy the (r3+7)-th slot of the object pointed by r2 to r1
STORE	r1	r2	r3	copy r3 to the r2-th slot of the object pointed by r1
STORE1	r1	r2	r3	copy r3 to the (r2+1)-th slot of the object pointed by r1
:				
STORE7	r1	r2	r3	copy r3 to the (r2+2)-th slot of the object pointed by r1
ADD	r1	r2	r3	set r1 to r2 + r3
SUB	r1	r2	r3	set r1 to r2 - r3
MUL	r1	r2	r3	set r1 to r2 * r3
IDIV	r1	r2	r3	set r1 to r2 div r3
IMOD	r1	r2	r3	set r1 to r2 mod r3
: $\star 3$				
EQUAL	r1	r2	r3	set r1 to r2 $=$ r3
NOTEQ	r1	r2	r3	set r1 to r2 $<>$ r3
LESSTHAN	r1	r2	r3	set r1 to r2 $<$ r3
LESSEQ	r1	r2	r3	set r1 to r2 $<=$ r3
GREATER	r1	r2	r3	set r1 to r2 $>$ r3
GREATEREQ	r1	r2	r3	set r1 to r2 $>=$ r3
NOT	r1	r2		set r1 to not r2
JUMP	r1			set ip to r1
JUMPTRUE	r1	r2		if r1 is true, set ip to r2
JUMPFALSE	r1	r2		if r1 is false, set ip to r2
CALL	r1	r2		call the code record pointed by r2, save registers and set the return value to r1
CALLREC	r1			call the current, save registers and set the return value to r1
RET	r1			return to the caller, restore registers and pass r1 as the return value
RETTRUE	r1	r2		if r1 is true, return to the caller, restore registers and pass r2 as the return value
RETFALSE	r1	r2		if r1 is false, return to the caller, restore registers and pass r2 as the return value

$\star 1$ other VDM-SL built-in operators on numbers and booleans

```
operations
  sub : Register`Register * Register`Register
        * Register`Register ==> ()
  sub(dst, src1, src2) ==
    if (dst > 0 and src1 > 0) and src2 > 0
    then
      let
        int1 = Register`read_int(src1),
        num1 : [real] =
          if int1 <> Data`invalidIntValue
          then int1
          else
            (let float1 = Register`read_float(src1)
             in
               (if float1 <> Data`invalidFloatValue
               then Data`float2real(float1)
               else nil)),
        int2 = Register`read_int(src2),
        num2 : [real] =
          if int2 <> Data`invalidIntValue
          then int2
          else
            (let float2 = Register`read_float(src2)
            in
              (if float2 <> Data`invalidFloatValue
              then Data`float2real(float2)
              else nil))
      in
        if num1 <> nil and num2 <> nil
        then
          let num3 : real = num1 - num2
          in
            if
              int1 <> Data`invalidIntValue and
              int2 <> Data`invalidIntValue
            then
              Register`write_int(dst, num3)
            else
              Register`write_float(dst, Data`real2float(num3))
        else
          err("sub instruction error: operands not integer")
    else
      err("sub instruction error: operands not specified");
```

Fig. 7. The formal definition of the `SUB` instruction

```
functions
   fib : nat -> nat
   fib(x) == if x < 2 then x else fib(x-1) + fib(x-2);
```

Fig. 8. The specification of fibonacci sequence function

explanation of each instruction is shown as a comment led by semicolons. As ViennaVM is a register machine, arguments passed to a routine is stored in r1, r2 and so on in order. In this case, the argument `x` is stored in r1. The first instruction, `MOVEI` set the immediate value to the specified register. In this case, this instruction sets the tagged word of `1` to r2. The `LESSTHAN` instruction is a three-operanded opcode that compares the second and the third operands and stores the result to the first operand. The `RETFALSE` instruction is a conditional return. In this case, if r3 is false, then return r1. Then, two sets of `SUB` and `CALLREC` instructions follows that computes fib(x-1) and fib(x-2). The `ADD` instruction will add the two return values from the `CALLREC` instruction and stores the result to r1. Finally, the `RET` instruction returns the r1.

```
                        ; x is passed to r1
MOVEI         2, int(1) ; r2 <- 1
LESSTHAN      3, 2, 1   ; r3 <- r2 < r3
RETFALSE      3, 1      ; if r3 is false then return r1
SUB           1, 1, 2   ; r1 <- r1 - r2
CALLREC       3         ; r3 <- fib(r1)
SUB           1, 1, 2   ; r1 <- r1 - r2
CALLREC       1         ; r1 <- fib(r1)
ADD           1, 1, 3   ; r1 <- r1 + r3
RET           1         ; return r1
```

Fig. 9. The IR code for fibonacci series function

Table 2 shows the result of performance test. The IR code shown in Figure 9 was executed on ViennaVM in C. The fibonacci series function in Figure 8 was implemented in C as an ideal performance target. The fibonacci series function in VDM-SL was also animated by VDMJ as a baseline performance. VDMJ is a standalone interpreter implemented in Java. VDMJ is used as a baseline in this benchmark because its implementation is mature and has less overhead than GUI-based IDEs.

The ViennaVM was also implemented in Smalltalk using automated code generator and a hand tuning was done to five methods. The Smalltalk version also executed the IR code to check the feasibility of ViennaVM used in ViennaTalk for debugging. The specification of ViennaVM was also animated by VDMJ as a baseline performance against the Smalltalk version.

Two expressions `fib(5)` and `fib(30)` were executed. The prototypical ViennaVM in C performed better than VDMJ and slower than the C version. Although the ViennaVM in C is a naive IR code interpreter without just-in-time or runtime optimisation, it performed reasonably well. ViennaVM in Smalltalk took almost 50 minutes to compute `fib(30)`. The result mentions that the use of ViennaVM on Smalltalk will be limited to debug by step execution. One possible way to debug a computationally demanding application is to use two versions of ViennaVM, in C and in Smalltalk, and combine them by image files. The ViennaVM in C will execute the IR code and dump an image file at a break point. The ViennaVM in Smalltalk will resume the image file to interactively debug by step execution.

Table 2. Performance comparison by fibonacci numbers

	fib(5) (ms)	fib(30) (ms)
C	0.0	11.0
ViennaVM (C)	1.8	410.2
VDMJ	1.8	3,335.4 ⋆2
ViennaVM (Smalltalk ⋆1)	18.4	2,870,079.0
ViennaVM (VDM-SL)	351.6	NA ⋆3

⋆1 automated code generator + hand tuning
⋆2 measured after 10 warm-up runs
⋆3 computation did not finish in 4 hours

4.2 Discussions about perspectives

VMs are commonly used architecture of runtime systems of various programming languages. While most language VMs are designed for programming languages, ViennaVM is specially dedicated for VDM languages. To take VDM's advantage in productivity of developing embedded software, ViennaVM should support not only in the final product, but the prototypical development of smart devices. The most VMs for high level programming languages can assume that the compiler generates valid IR code. ViennaVM, on the other hand, should provide rich runtime checking mechanism for types, states, precondition, postconditions and metrics. Another interesting feature of ViennaVM is that it is specified in VDM-SL. The development of VMs requires high skills of system programming and long development time to debug. We expect the utilisation of formal specification language would improve reliability, productivity and portability of the VMs.

The preliminary performance evaluation in Section 4 shows that the prototypical C implementation of ViennaVM can perform better than the VDMJ interpreter. Although there are still a big performance gap between ViennaVM and native C code, it can be possibly narrowed down by runtime optimisations and just-in-time compilation.

5 Planned Further Development

5.1 User Interfaces

We are planning to define UI instructions in ViennaVM. Smart devices have various user interfaces. While many VMs provide low level functions and let user programs to implement UI frameworks, we will provide high level instructions to absorb the difference of physical user interfaces. Figure 10 illustrates how ViennaVM will provide portability of its application code among different smart devices. *inform a text message to the user* instruction that will play a voice on ViennaVM for automotive smart speakers and display a small notifier on ViennaVM for smartphones. UI instructions of ViennaVM provide hooks to invoke and accept interactions with the user, and each implementation of ViennaVM will supply UI functions available in the target platform.

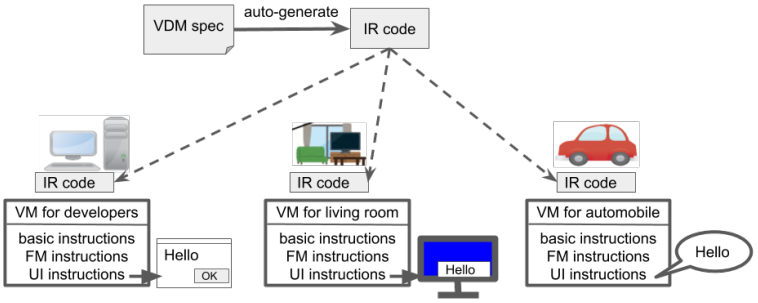


Fig. 10. The same IR code will adapt to different UI devices

5.2 Code Generators and Runtime Support for Formal Methods

We will develop an automated code generator that generates IR code of ViennaVM, and extend ViennaTalk to run an external ViennaVM implemented in C and also an internal ViennaVM implemented in Smalltalk. The internal ViennaVM will be better integrated with ViennaTalk and gives more flexible debugging feature while the external ViennaVM will execute the specification in closer environment to the target platform.

We also plan for extending ViennaVM by instructions that support assertions. A state invariant will be checked at every update to the state variable by setting it as a watch variable. Precondition and postcondition of a function or an operation will be held in a code record, and measure function of a recursive function will be also stored.

6 Conclusions

The development of ViennaVM is still at an early stage. Its objective is to provide a common target platform of automated code generators from VDM. It can also be seen

as a case project of VDM in VM developments. We will investigate how VDM can improve the development of language VMs as well as how a dedicated VM may change the process of the model-based development using VDM.

Acknowledgments

The authors would thank Christoph Reichenbach, Kumiyo Nakakoji and Yasuhiro Yamamoto for their inspiring comments on the initial idea of this research. A part of this research was supported by JSPS KAKENHI Grant Number JP 18K18033. We would also like to pay special thanks to the anonymous reviewers who have helped improving the quality of the paper.

References

1. Diswal, S.P., Tran-Jørgensen, P.W., Larsen, P.G.: Automated Generation of C# and .NET Code Contracts from VDM-SL Models. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 32–47. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
2. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future - the story of squeak, a practical smalltalk written in itself. ACM SIGPLAN Notices 32(10), 318–326 (1997)
3. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
4. 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 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
5. 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 (September 2010), http://dx.doi.org/10.1109/SEFM.2010.32, ISBN 978-0-7695-4153-2
6. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
7. Oda, T., Araki, K., Larsen, P.G.: Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 48–62. Aarhus University, Department of Engineering, Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
8. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
9. Smith, J.E., Nair, R.: The architecture of virtual machines. Computer 38(5), 32–38 (May 2005)
10. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. International Journal on Software Tools for Technology Transfer pp. 1–25 (2017), http://dx.doi.org/10.1007/s10009-017-0448-3

Part II

Applications Session

Multi-modelling of Cooperative Swarms

Georgios Zervakis[1], Ken Pierce[2], and Carl Gamble[2]

[1] Elsevier, United Kingdom
g.zervakis@elsevier.com
[2] Newcastle University, United Kingdom
{kenneth.pierce,carl.gamble}@newcastle.ac.uk

Abstract. A major challenge in multi-modelling and co-simulation of cyber-physical systems (CPSs) using distributed control, such as swarms of autonomous Unmanned Aerial Vehicles (UAVs), is the need to model distributed controller-hardware pairs where communication between controllers using complex types is required. Co-simulation standards such as the Functional Mock-up Interface (FMI) only supports simple scalar types. This makes the protocol easy to adopt for new tools, but is limiting where a richer form of data exchange is required, such as distributed controllers. This paper applies previous work on adding an explicit network VDM model, called an ether, to a multi-model by deploying it to a more complex multi-model, specifically swarm of UAVs.

Keywords: multi-modelling, swarms, FMI, co-simulation

1 Introduction

The design of cyber-physical systems (CPSs) requires engineers from across disciplines to collaborate in order build the collections of physical, control and network systems from which they are built. Swarms of Unmanned Aerial Vehicles (UAVs) represent CPSs; they require their hardware, software and elements to be closely integrated if they are to function properly. While model-based engineering approaches for CPSs are desirable, the distinct modelling formalisms used by different disciplines is a barrier to collaborative design. A promising approach to overcome this is multi-modelling, where individual models of the components, retained in their appropriate tools and formalisms, are combined into system-level models that can be analysed through co-simulation.

Multi-modelling of distributed CPSs such as UAV swarms presents a challenge to current approaches. Such swarms form sets of controller-hardware pairs within the multi-model, which must also communicate between pairs in order to model distributed control (c.f. Figure 3). Co-simulation standards such as the Functional Mock-up Interface (FMI)[3] support simple scalar types and strings, and not the rich set of data abstractions familiar to engineers modelling software components in formalisms such as Vienna Development Method (VDM) [7]. Also as the number of units in the swarm increases, direct connection between all controller models becomes unwieldy.

[3] http://fmi-standard.org/

In previous work [2], introduction of an explicit network model to a multi-model was demonstrated and applied to a building case study. This approach overcomes both the limited number of types available and the increasing number of connections required as swarms increase inside. In this paper we present the multi-modelling of a swarm of UAVs using the INTO-CPS technologies[4]. The multi-model comprises the simple network model and a scalable set of controller-hardware model pairs, modelled in VDM and 20-sim, respectively (see Section 2). We demonstrate the potential of this approach and consider weaknesses and future directions based on experiences.

The remainder of the paper is structured as follows. Section 2 describes the modelling technologies used in the paper. Section 3 introduces the case study, namely a swarm of UAVs. Section 4 describes the modelling of the case study, focusing on the controller and network model. Section 5 shows results of co-simulation of the multi-model. Finally, Section 6 draws conclusions and presents future work.

2 Background

In this section we briefly describe the modelling tools used for the multi-modelling covered in Section 4: the INTO-CPS technologies, VDM-RT and 20-sim.

2.1 INTO-CPS Technologies

The INTO-CPS technology allows the user to build and analyse *multi-models* comprising multiple constituent models [5], using both discrete-event (DE) and continuous-time (CT) formalisms. INTO-CPS adopts the Functional Mock-up Interface (FMI) standard, where constituent models are packaged as Functional Mockup Units (FMUs). Over 30 different tools can produce FMUs, with partial or upcoming support bringing the total to over 100[5]. INTO-CPS provides a co-simulation engine, Maestro, which acts as a co-simulation master algorithm, offering both fixed step size and variable step size co-simulation. The INTO-CPS technologies are supported by the INTO-CPS Association, a group of organisations working on the technologies, based around a community of industrial users. As their organisations are part of the Association, INTO-CPS has guaranteed support for FMUs produced by Overture and 20-sim, described below.

2.2 VDM-RT

The Vienna Development Method (VDM) [7] is a well-established formal method for systematic analysis of system specifications. VDM++ is an object-oriented extension of the original VDM Specification Language (VDM-SL) used for the development of computer-based systems and software. VDM-RT is an extension of VDM++, intended for the specification of real-time embedded and distributed systems [8]; it includes features required for description of real-time controllers, in particular native support for a computational time model and distribution of functionality between compute units, interconnected by a communications network.

[4] http://into-cps.org/
[5] http://fmi-standard.org/tools/

2.3 20-sim

20-sim[6] represents continuous-time (CT) models using graphs of connected blocks or icons [4]. Blocks and icons contain differential equations or code that represent physical phenomena, while the connections denote channels over which the phenomena interact. These channels may be one-way (signals) or two-way (bonds). The bi-directional bonds carry the domain-independent values of effort and flow— these map to familiar physical concepts, e.g. voltage and current. Bonds offer a powerful, compositional and domain-independent way to model physical phenomena. These CT models are solved numerically to yield high-fidelity simulations of physical components.

3 Case Study

Swarms of relatively inexpensive UAVs have the potential to save time and money and even lives across a variety of applications, including border patrol [**?**], monitoring of nuclear power plants, monitoring of forest fires [**?**] and search missions for finding missing persons [**?**]. Using automated guidance and autonomous decision making, UAVs have the potential to operate for extended periods of time without significant human supervision [**?**]. Consequently, they can be used for wilderness search and rescue operations that may require hundreds of hours of search at low altitude, and using flight profiles close to objects where piloted systems cannot be flown [**?**, **?**]. Even with only simple algorithms for generating search paths, a team of UAVs is more efficient than a single UAV [**?**], therefore the development of robust, cooperative UAV systems will lead to improved outcomes in operations such as search and rescue.

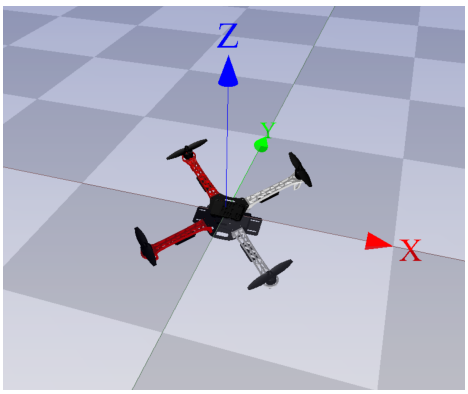


Fig. 1. A quadcopter visualised in 20-sim

In this paper we consider a case study of a UAV swarm searching a geographical location. The INTO-CPS technology was applied in the design of a UAV swarm for

[6] http://www.20sim.com/

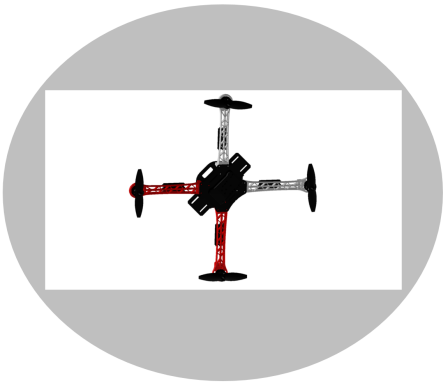


Fig. 2. Dual-camera sensor footprint (white rectangle determines waypoint spacing)

searching and surveillance. Specifically, the swarm was tasked with collaboratively and autonomously searching for human bodies within a specified area. Each UAV in the swarm is a quadcopter (Figure 1), a compact drone with four fixed-pitch propeller blades and requiring a low-level loop controller for aerial stability. Searching is carried by two downward-facing cameras (not modelled), covering the visual and infrared spectrum [**?**]. This gives a rectangular footprint (the minimum of the two cameras) as shown in Figure 2. To search an individual area, a UAV must visit a number of locations to guarantee visual coverage of its assigned area. As a swarm, the UAVs must divide up larger search areas to be searched by individual UAVs.

4 Multi-Modelling of the UAV Swarm

4.1 Multi-model Composition

The UAV swarm is realised as a set of FMUs connected together as a multi-model. The logical connections between the DE, CT and network models are shown in Figure 3. Each UAV is represented by a DE-CT model pair, representing the controller and hardware respectively. The DE model passes control outputs (for the motors) to the CT model, which in turn passes back positional information, i.e. x-, y- and z-coordinates. Each DE model is also connected to the network model, so that the controllers can pass messages for coordination. Figure 4 shows how these logical connections are actually realised in the FMI protocol, via the co-simulation engine (Maestro).

4.2 CT Model

The CT model for a single UAV is shown in Figure 5. Each block is an element of the physical system (motors, rotors, frame, battery). The arrows in the top left are inputs to the model (throttle, pitch, yaw, and roll) which allows the DE model to move the UAV. The arrows on the lower half are outputs (sensor readings of position and orientation).

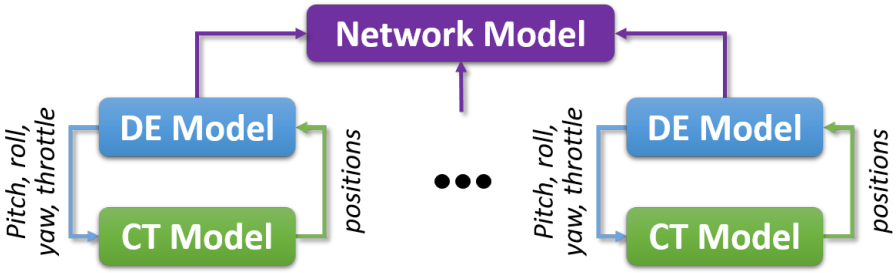


Fig. 3. Logical connections between constituent models in the multi-model

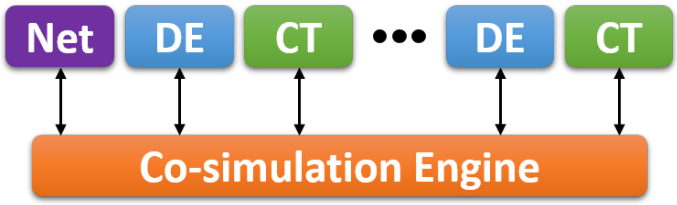


Fig. 4. Connections between FMUs and the co-simulation engine

Together these form the interface of the FMU. The physics and battery model allows the controller to be tuned accurately for minimal-energy path planning and return-to-base recharging. This model is derived from the high-fidelity model presented in [6], but incorporates loop-level control. This presents a higher-level interface to the controller and speeds up simulations as fewer synchronisation steps are needed per co-simulation.

4.3 DE Model

The DE model provides the supervisory control of the UAVs, specifically movement and distributed coordination for searching. The modes of the controller are shown in Figure 6. Each UAV searches a specific area using waypoints and visiting these in turn. The waypoints cover the entire assigned search area, including along straight line paths (see Section 5, as the UAV must pause briefly to take a clear image at each waypoint. The controller also monitors its battery usage and returns to base to recharge, resuming its visiting of the waypoints until its search is complete. The costliest maneuver for a quadcopter is a U-turn [**?**], which the path planner takes into account by minimising U-turns when dividing and generating the assigned waypoints in the search space.

The UAV has four modes: INITIALIZATION, RETURN_TO_BASE, TAKE_OFF, and FLY. The first mode that the UAV enters is the INITIALIZATION mode, which the UAV will enter only once to generate waypoints for a specific area, and plan its trajectory, as well as to store its initial coordinates. Since the UAV starts from a base,

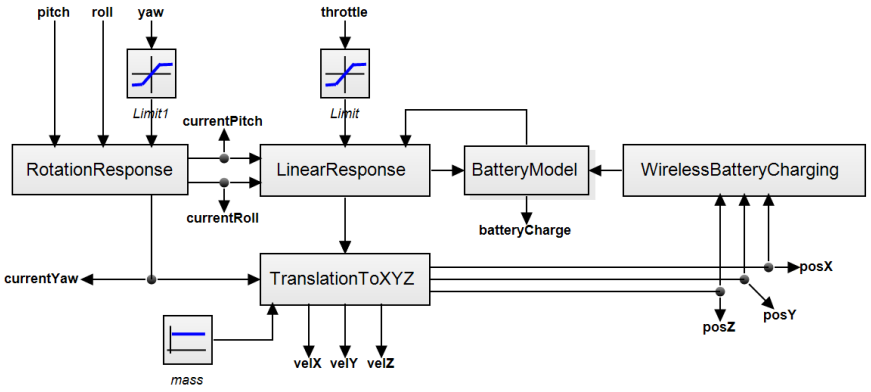


Fig. 5. Physical model of a quadcopter in 20-sim

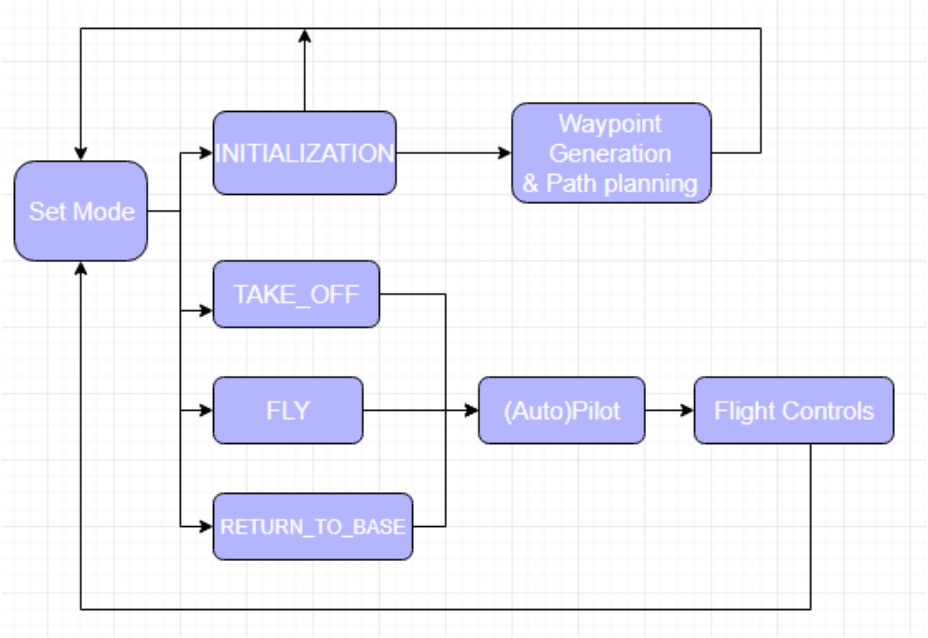


Fig. 6. Modes of the controller model

it is essential to store those coordinates to return when it has finished its task, or when there is a need for refuelling.

When the UAV has generated waypoints and trajectory and has sufficient fuels, it enters the TAKE_OFF mode to carry out the mission. As we model a quadcopter able to take-off and land vertically, the TAKE_OFF mode is used to launch the UAV vertically. When the UAV reaches a desired altitude that is considered safe to start its mission, it enters the FLY mode to visit the waypoints. These can be seen in Section 5.

In FLY mode, the UAV continually determines its target position, which is the next waypoint that exists in the sequence. It is also responsible for updating the sequence with the remaining waypoints, erasing the visited waypoints. A waypoint is considered visited only if the UAV has reached approximately the position of that waypoint (in order to take an accurate photo and ensure coverage). In this model, a waypoint was considered visited if the UAV inclines less than 0.3 distance units (metres) per each coordinate.

The RETURN_TO_BASE mode forces the UAV to return to its base. The UAV enters that mode for two reasons. Firstly, the UAV enters that mode when it finishes its mission in order to return back to its base. Secondly, it enters that mode if there is insufficient fuel to carry on the mission. In each iteration, the UAV calculates its average consumption based on the distance travelled and the battery consumption. Afterwards, it checks whether the distance from its base is greater than the distance that can be travelled, taking into account the remaining fuel. A safety buffer is included, so that a UAV will return to base before its fuel is exhausted, ensuring there is sufficient fuel to return to base.

4.4 Network Model

The network model is a single FMU that represents an abstract communications medium, called the ether [3]. We adopt this approach because connecting each controller model directly to every other controller is unwieldy, and FMI currently lacks native support for such network connections. The ether is aware of all controllers connected to it, and passes all messages received to all other UAVs. By encoding messages as strings, we can also overcome the limited types supported by FMI connections. In this multi-model, identifiers and message recipients are handled by the VDM model directly, because this would form part of the software as deployed on the real UAVs, however message handling could be added to the ether model if this was appropriate.

The network FMU used in this case study is an initial, abstract network model built in VDM as a proof of concept for modelling communications within the limitations of FMI. The model does not currently cover advanced features such as specific protocols, error handling, or data rates. Work on improving the network model is ongoing. The openness of FMI means that this FMU could be replaced by a sophisticated, dedicated network model such as OpNet[7] and NS2[8] for improved predictability of real behaviors.

[7] `https://www.riverbed.com/gb/products/steelcentral/opnet.html`
[8] `https://www.isi.edu/nsnam/ns/`

4.5 Distributed Coordination with Communication

During INITIALIZATION, each UAV announces itself and a LEADER is elected, with the others becoming WORKER UAVs. In the model the election scheme is deterministic, as no randomness is modelled, however Bryans et al. [1] demonstrate a robust, distributed election scheme that could be implemented in the final system. The LEADER then divides the search space between the available UAVs and assigns them a sub-area. Each UAV then searches its sub-area, managing its waypoints and battery levels.

Using the ether FMU, each UAV is able to broadcast its id number, position, battery life, a tag id, four real numbers, an acknowledgment, a map consisting of the id numbers of the UAVs undertaking a task and the coordinates of their task, and a map consisting of the UAVs that have finished their tasks and the coordinates of their tasks. This is shown in the listing below:

```
public uavPosition :: x : real
                      y : real
                      z : real
                    bat : real;
...

Step() == cycles(2)
(
  -- broadcast own position
  etherOut.setValue(
    VDMUtil`val2seq_of_char[nat*real*real*real*real](
      mk_(id, posX.GetValue(), posY.GetValue(),
          posZ.GetValue(), battery.GetValue())
    )
  );

  -- receive messages
  if len etherIn.getValue() >= 2 then (
    let mk_(list,l) =
     VDMUtilDebug`seq_of_char2val[
     seq of(nat*real*real*real*real)](etherIn.getValue())
     in if list then (
       for all z in set inds l do (
         let x = l(z) in (uavPositions:= uavPositions ++
           {x.#1 |-> mk_uavPosition(x.#2,x.#3,x.#4,x.#5)};
         )
       )
     )
  )
)
```

Listing 1.1. "Message processing in the UAV controller VDM model "

The id number and position are sent, allowing every member of the swarm to determine the other UAVs that are taking part in the mission and their position. In the message tuple, the first (natural) number is the tag id, and the remaining four (real)

numbers are used from the LEADER when it sends a task to a WORKER UAV. The tag id indicates which UAV should undertake the task, and the remaining four numbers represent coordinates indicating the sub-area that the worker UAV should cover. The worker UAV knows when a task is intended for itself using its id number.

The acknowledgment is used from the worker UAVs to send an acknowledgment to the LEADER that they received their task. After the completion of their task, the workers send the LEADER another acknowledgment indicating that they finished it. If the LEADER receives such an acknowledgment, it erases the worker UAV from the list of the UAVs undertaking a task, and stores the UAV and its task in the map consisting of the UAVs that have finished their tasks. The LEADER does the same upon finishing.

The two maps are being sent from the LEADER UAV to the other members of the team, allowing them to know which UAVs have been assigned a task, and which UAVs have finished their tasks and the areas covered so far. At this stage, these maps are not used for anything, but have potential for future work.

If a UAV is not heard from in a certain amount of time, its sub-area is reassigned to the first available UAV to finish its initial sub-area. If communication with the LEADER is lost, a new leader election occurs. In this way, the swarm is resilient to lost UAVs.

5 Results

The multi-model was successfully able to simulate a range of scenarios and demonstrate the possibility of studying distributed communications and swarm behaviours with multi-modelling techniques. A live plot of a single UAV searching an area is shown in Figure 7. Here the vertical take-off is shown as `{uav}.uav.posZ` (green), with a zig-zag searching pattern shown by `{uav}.uav.posX` and `{uav}.uav.posY` (blue and orange respectively). At 400 seconds the UAV returns to base for refuelling before completing its search.

A 3D visualisation of three UAVs is shown in Figure 8. Such visualisations give intuitive feedback to software engineers about the effects of their design choices, however to better demonstrate the behaviour of the swarms, the co-simulation outputs (in CSV format) were post-processed in Matlab. A plan view of the data shown in Figure 7 is given in Figure 9. Here the waypoints are marked as triangles, with the path of the UAV as a black line, including its return to base to recharge. The UAV controller divided the space along the long axis to minimise U-turns.

Figure 10 shows a UAV swarm dividing and searching an area cooperatively. As in Figure 9 this is a plan view of the x- and y-position of the UAVs. In this co-simulation, there are three UAVs that can potentially join the swarm, however as the area is smaller than in Figure 9, negotiations result in two of the three UAVs dividing up the area and performing the search (represented by the black and green lines). Note that they fly in a latitude-first direction to minimise U-turns. Figure 10 shows the same three-UAV swarm searching another area. The second UAV (green line) is lost, as can be seen by the line ending at around point (17,28). After a timeout, the swarm decides that this UAV is missing as no further communications were received. The second search area is then reassigned to the first UAV, which completes searching this area after completing its own sub-area.

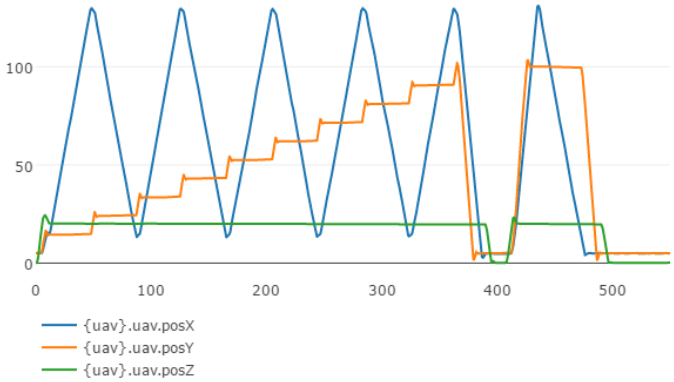


Fig. 7. Live plot of the position of a single UAV during a search

Fig. 8. 3D visualisation of the UAV swarm multi-model

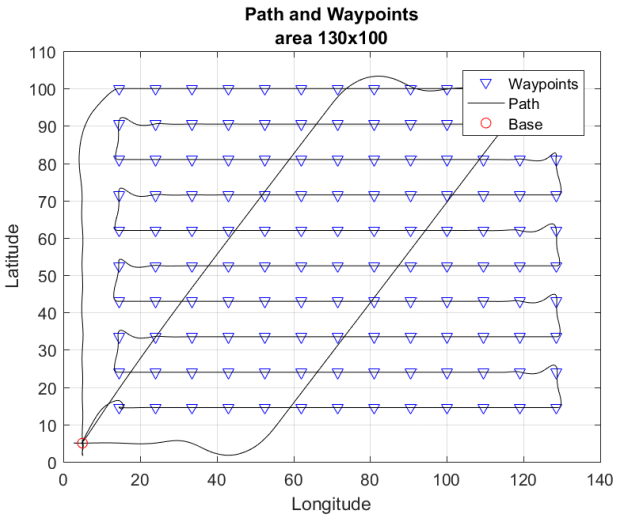


Fig. 9. Plan view of a UAV searching an area

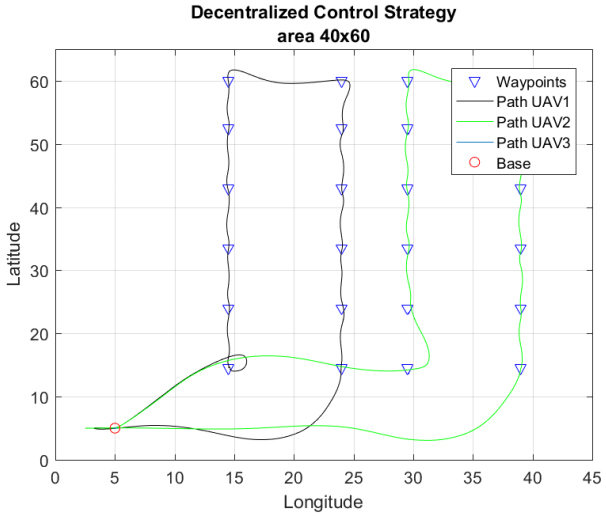


Fig. 10. Two UAVs searching an area cooperatively

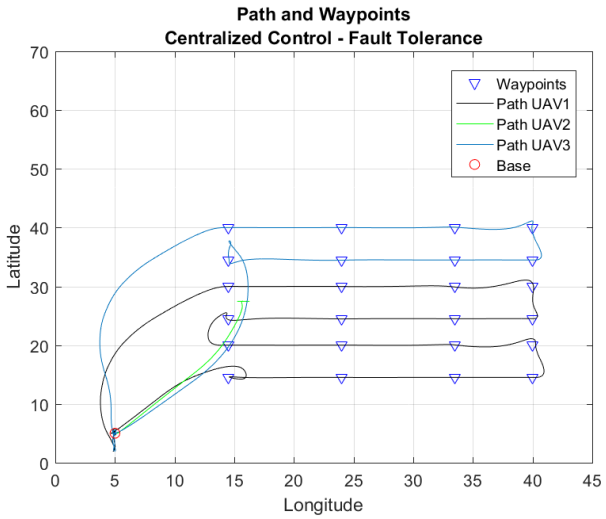


Fig. 11. Three UAVs searching an area; UAV2 fails and UAV1 takes over this sub-area after completing its own sub-area

6 Conclusions and Future Work

In this paper we applied a multi-modelling approach to a swarm of UAVs. The multi-model uses a simple network model [2] to allow a set of UAV controllers to communicate using complex data types in order to model realistic communication protocols. Each controller model is paired with a high-fidelity physics model. The results presented demonstrate the potential of this paradigm for multi-modelling of swarms and other CPSs with distributed control within the constraints of the FMI standard.

The network FMU used in this case study is an initial, abstract network model built in VDM as a proof of concept for modelling communications within the limitations of FMI. As highlighted in previous work [2], there are some drawbacks to that specific network FMU approach. Firstly, messages require a number of co-simulation cycles to reach recipients, so careful consideration of co-simulation synchronisation timing is required[9]. Secondly, the model does not currently cover advanced features such as specific protocols, error handling, or data rates. The openness of FMI means that improved network models can be swapped in to improve modelling fidelity, either improving the existing FMU or replacing it with a dedicated network model such as OpNet or NS2.

Improvement of the network model is a key next step. Another intriguing next step forward is to observe in Figure 3 the potential for an analogue of the network model to link together CT models on the bottom of the diagram. This would suggest an "environmental ether" model representing physical interactions between components. This could include, for example, physical obstacles, occlusion of line-of-site for communications, and even collisions. Physical interactions such as contact and collision models are particularly challenging as they require tightly-coupled interactions.

Acknowledgements

The work reported here is supported by the EU Horizon 2020 Projects "Integrated Tool Chain for Model-based Design of Cyber-Physical Systems" (INTO-CPS, Grant Agreement 644047) and "CPS Engineering Labs - expediting and accelerating the realization of cyber-physical systems" (CPSELabs, Grant Agreement 644400).

References

1. Bryans, J., Fitzgerald, J., Payne, R., Kristensen, K.: Maintaining Emergence in Systems of Systems Integration: a Contractual Approach using SysML. In: INCOSE International Symposium on System Engineering. INCOSE (2014)
2. Couto, L.D., Pierce, K.: Modelling Network Connections in FMI with an Explicit Network Model. In: J. S. Fitzgerald, P.W.V.T.J., Oda, T. (eds.) The 15th Overture Workshop. pp. 31–43. Newcastle University, Computing Science. Technical Report Series. CS-TR- 1513, Newcastle, England (September 2017)
3. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), http://link.springer.com/book/10.1007/978-3-642-54118-6

[9] The Maestro co-simulation engine supports minimum frequency constraints which could alleviate this problem somewhat

4. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
5. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In: 2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data). IEEE, Vienna, Austria (April 2016), http://ieeexplore.ieee.org/document/7496424/
6. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards semantically integrated models and tools for cyber-physical systems design. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Proc 7th Intl. Symp. Lecture Notes in Computer Science, vol. 9953, pp. 171–186. Springer International Publishing (2016)
7. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (April 2013)
8. Verhoef, M., Larsen, P.G.: Enhancing VDM++ for Modeling Distributed Embedded Real-time Systems. Tech. Rep. (to appear), Radboud University Nijmegen (March 2006), a preliminary version of this report is available on-line at `http://www.cs.ru.nl/~marcelv/vdm/`

Design Space Exploration for Secure Building Control

Martin Mansfield, Charles Morisset, Carl Gamble, John C. Mace, Ken Pierce, and John Fitzgerald

School of Computing, Newcastle University, UK
martin.mansfield@ncl.ac.uk

Abstract. By automation of their critical systems, modern buildings are becoming increasingly intelligent, but also increasingly vulnerable to both cyber and physical attacks. We propose that multi-models can be used not only to assess the security weaknesses of smart buildings, but also to optimise their control to be resilient to malicious use. The proposed approach makes use of the INTO-CPS toolchain to model both building systems and the behaviour of adversaries, and utilises design space exploration to analyse the impact of security on usability. By separation of standard control and security monitoring, the approach is suitable for both the design of new controllers and the improvement of legacy systems. A case study of a fan coil unit demonstrates how a controller can be augmented to be more secure, and how the trade-off between security and usability can be explored to find an optimal design. We propose that the suggested use of multi-models can aid building managers and security engineers to build systems which are both secure and user friendly.

Keywords: Security · Smart Buildings · Multi-Modelling · Design Space Exploration · Optimisation

1 Introduction

The critical services required to operate many existing buildings, such as Heating, Ventilation, Air-Conditioning (HVAC), lighting, access control and fire detection are automatically managed by building control systems. Reduced energy usage, improved efficiency, maintenance, comfort and safety are just a few of the many potential benefits automated building control can bring. In today's information age, these standalone building control systems are being connected to the Internet and other data networks to further improve operational efficiency and occupant safety by offering smart interactions, centralised and remote control, monitoring and maintenance. In doing so, smart buildings are becoming vulnerable to disruptive, damaging and life-threatening cyber-attacks (e.g. [1–3]). Furthermore, the increasing rate of connection to realise the potential operational benefits has exposed a lack of practical processes and mechanisms for securing existing building control systems.

Likely adversaries behind smart building cyber-attacks are varied: corporations, cyber-criminals, traditional criminals, occupants, nation states, hacktivists, and terrorists [4]. The different motivations, capabilities, resources and objectives these adversaries bring highlights the potential for building control systems to be attacked in many

different ways, some of which may not yet be evident. The best one can do is to identify a set of known feasible threats and defend against them by designing and implementing effective security mechanisms. A starting point is to consider building control systems as Cyber-Physical Systems (CPSs) due to their constituent cyber elements (e.g. controllers, adversaries) and physical elements (e.g. sensors, actuators, environment, users). Traditionally, cyber and physical security are treated as separate concerns, however the need to understand attacks and their impacts requires a holistic view incorporating both the cyber and the physical domains. Importantly, once secured, building control systems must still provide a required level of service which makes it necessary to also understand the impact security mechanisms would have on building operations before implementation (i.e. the security/usability trade-off).

In this paper we build on our multi-modelling approach introduced in [5], and show how practical security monitors for building control systems can be designed. We make use of the INTO-CPS open toolchain [6] to model building control systems and the attacking behaviour of potential adversaries towards those systems. We then utilise the Design Space Exploration (DSE) functionality of INTO-CPS to design practical security monitors by establishing the security/usability trade-off. The contributions of this paper are therefore as follows:

1. a security monitor model for a building control system cyber controller.
2. guidance for using DSE to explore the monitor's security/usability trade-off.
3. the extension of an existing case-study by including the security monitor.

To illustrate our modelling approach we consider the design of a security monitor for a single Fan Coil Unit (FCU), a common building control system found in buildings for managing room temperature. We explore just one aspect of the FCU's security by analysing a specific Man-in-the-Middle style attack launched against it. The attack illegitimately modifies sensor readings to maximise fan usage while minimising the deviation of temperature from the required set-point. An attack of this type could cause overheating and even fire, increase energy usage, wear and tear, routine maintenance and other financial costs. Building control systems including lighting accounts for 70% of a building's energy usage. If operated incorrectly (e.g. as a result of attack) a 20% increase in energy use can be expected in general [7].

Section 2 provides an overview of smart buildings and the INTO-CPS toolchain, as well as a summary of the existing use of multi-models to describe building control systems and security concepts. Section 3 introduces how controllers can be more resilient to attack by adding security monitors, and how DSE can be employed to establish a balance between secure and usable control. Section 4 demonstrates our approach by modification of a previous FCU case study, and Section 5 describes possible future directions for this work. Finally, concluding remarks are included in Section 6.

2 Background

This section introduces typical characteristics of smart buildings alongside their vulnerabilities. Additionally, it includes an introduction to multi-modelling and associated technologies, and how these technologies have been employed in the description of smart buildings and their potential attackers.

2.1 Smart Building Security

Building Control Systems Today's buildings take advantage of automated systems to control crucial operational services such as HVAC, lighting, water supplies, mobility, access control, and security, among others. The motivations for automating building control systems are improved operational efficiency, productivity, environmental sustainability, occupant health and safety, and reduced energy consumption. To achieve these potential advantages, building control systems integrate sensors to measure and collect environmental data from within a building (e.g. air temperature, humidity and occupancy). This data is transmitted to digital controllers which process it and calculate control instructions, which are then transmitted to actuators capable of altering the state of a building's environment. Lights may be turned on or off, air-vents opened or closed, a room's temperature raised or lowered, doors locked or unlocked, and so on. The integration of software-based cyber controllers and physical sensing and actuating devices means that most building control systems can be considered CPSs.

Security Concerns Traditionally, building control systems were standalone entities with no external connectivity. The security of these systems was provided largely by obscurity. Now, these once siloed systems are being retro-fitted with technologies connecting them to the Internet and other data networks to facilitate greater operational efficiency and safety. For instance, centralised monitoring and control, historical data storage, and remote maintenance (e.g. uploading software updates) can be supported. Consequently, the rapid hyper-connectivity of building control systems has opened up a large and complex cyber-attack surface and exposed the security provision for building control systems as being way behind the times.

One key reason for this is existing building control systems were built solely to work. Their designs typically came purely from civil engineering fields resulting in specifications that rarely stated the need for security. As a result, many of today's building control networks incorporate pre-existing legacy systems to which current security mechanisms cannot simply be 'bolted on' [2]. Also, memory, power and processing constraints of communicating devices (e.g. sensors) means security mechanisms such as data encryption are often too costly to implement. The use of common open protocols (e.g. BACnet [8] and KNX [9]) by disparate devices to facilitate communication exposes data traversing a network to varied cyber-attacks such as injecting fake sensor readings [3]. Furthermore, a lack of authentication processes enables attackers to use electronic devices (e.g. laptops, tablets and smart phones) to easily infiltrate and take control of systems on the network without detection.

The inherently insecure nature of building control systems exposes them to novel cyber-attacks such as disabling critical systems until a ransom is paid, or controlling systems to cause damage and disruption to the physical environment [1]. In the latter case, destructive commands could be transmitted over the building control network to place a system into a dangerous state for which it has not been designed. For instance, heating, air or water supplies can be disabled; rooms caused to overheat and damage sensitive data stores (e.g. patient records) or material (e.g. forensic evidence); systems overloaded to cause fires or floods while locking fire doors to trap occupants; or electronic doors unlocked to rooms holding sensitive information. Attackers have already

demonstrated they can launch highly damaging attacks on industrial control systems (e.g. explosions at a steel works [10] and nuclear power plant [11]). Attackers are now beginning to demonstrate they can exploit similar vulnerabilities in building control systems and gain control over them remotely [12, 13]. The inadequacies in building control network security means smart buildings are becoming an attractive target as they enable disruptive, damaging and life-threatening attacks with minimal effort.

2.2 Multi-modelling and Co-Simulation

Here, we give a brief introduction to the techniques used for modelling the case study, specifically a foundation of heterogeneous multi-modelling and design space exploration (DSE) techniques built on top of this.

INTO-CPS The INTO-CPS technologies[1] comprise a tool chain and supporting methods for model-based engineering of cyber-physical systems (CPSs) [14]. The core of INTO-CPS is support for definition and analysis heterogeneous system models, called *multi-models*, which combine individual models of the CPS' components and a description of their connections. The primary analysis technique for such multi-models is co-simulation, in which the individual component models are simulated together. The Co-simulation Orchestration E[ngine (COE) of INTO-CPS is called Maestro[2], which fully implements the emerging Functional Mock-up Interface (FMI) standard[3] for co-simulation. In FMI, component models are packaged into a standard format called a as Function Mock-up Unit (FMU). Maestro acts as a so-called master algorithm orchestrates the co-simulation, managing the passage of time and data exchange between FMUs. The INTO-CPS COE implements both a standard, fixed time-step algorithm and a variable time-step algorithm, which can speed up co-simulations and improve the fidelity of results for certain classes of FMUs. An INTO-CPS Association has been formed to continue work on the INTO-CPS technologies based around a community of industrial users. In addition to co-simulation, INTO-CPS provides support for automated testing, model checking, code generation, and hardware-in-loop (HiL) testing. Of relevance to the work in this paper are design space exploration (DSE), as described in the next section; and configuration of multi-model architectures through SysML, as described in Section 4.

The use of the FMI standard provides for an open tool chain that lowers the barriers to entry for model-based engineering, and allows for different paradigms of model to be connected together. This means that the most appropriate modelling formalism can be selected for each component of the system. Over 30 tools can produce FMUs, with more than 100 having partial or upcoming support[4]. At the time of writing (Q2 2018), Maestro is the most widely supported FMI engine, available on Windows, Linux, MacOS and ARM (Raspberry Pi). The case study described in Section 4 uses continuous-time (CT) model of the physical phenomena of the building system, combined with a

[1] http://into-cps.org/
[2] https://github.com/INTO-CPS-Association/maestro
[3] http://fmi-standard.org/
[4] http://fmi-standard.org/tools/

discrete-event (DE) model of the controller and security components. CT models represent systems as a set of differential equations which are solved numerically to provide high-fidelity simulations of physical phenomena, whereas DE models primarily represent data, state and events which alter these, and are best-suited for describing computing components. In this paper we use 20-sim for CT modelling, which describes models using graphs of connected blocks or icons [15], and VDM-RT for DE modelling. VDM-RT is an extension of the well-established notation VDM (Vienna Development Method) [16] that includes features required for description of real-time controllers including as classes, object orientation and native support for a model of computation time and distribution of functionality between compute units [17].

Design Space Exploration It is likely that there are many choices to be made when designing a CPS and these choices will affect the resulting performance of the CPS. Choices could include physical properties of the CPS, such as the thickness of walls in a building or the number or placement of sensors within a room, or they could regard cyber properties such as the choice of algorithm controlling heating or the frequency at which sensors are sampled. These choices along with the options for each define the design space for the CPS. One use for a multi-model then is to allow the engineer to explore the design space to find design options that are optimised with respect to one or more performance measures. Many of the design choices can be left open by the domain experts that produced the original models by exposing them as parameters of the resulting FMUs, in which case the user has the option to make use of the DSE facilities included in INTO-CPS to automatically explore the design space [18].

As a minimum, a DSE requires the definition of three aspects. The first aspect, parameters, is where we describe which parameters the DSE search may change and also gives a list of values each parameter may take. These parameters define the design space that is to be searched.

The second and third aspects relate to how we measure performance of a system and how we compare different designs using those measures. INTO-CPS simulations produce results in three forms, live graph plots of variables during a simulation, logs of monitored variables in CSV format and also 3D visualisations of the models if the user has created one. Since DSE is likely to run a great many simulations it is not practical for a user to observe all the live plots or 3D visualisations and so DSE makes use of Objective scripts that process the CSV simulation logs to produce objective values that characterise the performance of a CPS during simulation. Such objective functions might compute the total energy consumed by a system or the maximum deviation of a variable from an acceptable value. Once the objective values are computed for each design, they may then be used to compare different designs. If there is only a single performance measure then results may simply be placed in an list, ordered by that measure, to find the best, however, if there are multiple measures then a different means for comparison must be used. In the latter case, INTO-CPS makes uses the Pareto method to present the user with a non-dominated set of best designs [19].

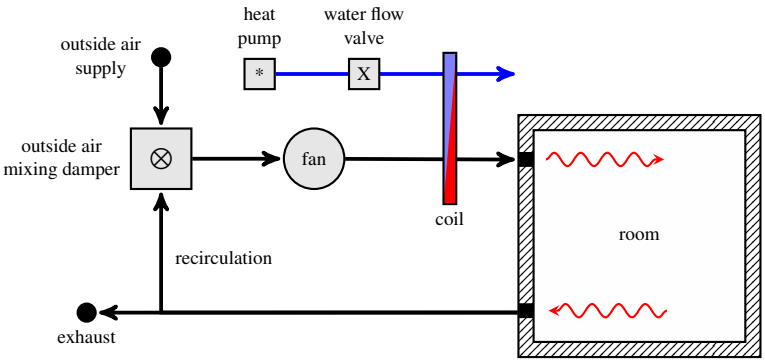


Fig. 1. Overview of the fan coil unit (FCU) example.

2.3 Modelling Building Control Systems

One cyber-physical building control system common to smart buildings is the Heating, Ventilation, and Air-Conditioning (HVAC) system. An HVAC system is responsible for controlling the air temperature and quality of a building, and does so using one or more Fan Coil Units (FCUs). Each FCU is comprised of several (physical) components for sensing and controlling temperature, and a (cyber) controller, implemented in software and responsible for the coordination of actuators based on sensed data. With a typical FCU able to service up to 150m^2, it is typical for a single building to include many FCUs.

An abstracted overview of an FCU is given in [20], and illustrated in Figure 1. The FCU uses a *fan*, to intake air and pass it across a cooling/heating *coil* and into a *room*. A bidirectional *heat pump* uses water to control the temperature of the coil, where the rate of temperature change is determined by the rate of water flow from the heat pump to the coil, controlled by a *water flow valve*. Both the fan speed and valve position are set by a digital controller. An *outside air supply* ensures adequate ventilation, and is mixed with *recycled air* recirculated from the room by a *outside air mixing damper*, with any excess leaving the system via an *exhaust*.

An initial FCU model proposed in [20] is transformed into a multi-model in [21]. The FCU multi-model contains 3 constituent models:

External A continuous-time model implemented in 20-Sim which specifies external stimuli, including a room air temperature set-point and the outside air temperature.

RoomHeating A continuous-time model implemented in 20-Sim comprised of two sub-models: `Room` and `Wall`. The `Room` model calculates the current room air temperature based on the fan speed, the water flow rate, and the wall surface temperature. The temperature of the wall surface is calculated by the `Wall`C model and is based on the outside air temperature and the room air temperature.

Controller A discrete-event model implemented in VDM-RT, this model calculates the fan speed and the aperture of the water flow valve based on the room air temperature and room air temperature set-point.

The model presented in this paper builds upon that presented in [21] by adding an explicit cyber-attacker model, an updated controller model that aims to address this attack, and modified objective scripts to capture the usage of the fan.

Modelling Adversaries The FCU multi-model is extended in [5] to demonstrate the use of multi-models in assessing the security of building control systems. The multi-model is extended by introducing an `Adversary` model which intercepts and potentially modifies the room air temperature set-point being communicated between the `Environment` and the `Controller`. The `Adversary` model is a discrete-event model implemented in VDM-RT.

By modifying the room air temperature set-point, the adversary executes a basic attack on the system by manipulating the controller in order to expedite wear and tear on the FCU by maximising fan usage. The implementation of this Man-in-the-Middle type attack continually increases and decreases the room air temperature set-point at a given frequency, causing the FCU fan speed to oscillate. Listing 1.1 outlines the control loop of the adversary model, which includes parameters for the frequency of room air temperature set-point modification (`attackFrequency`), and the upper (`upperModificationLimit`) and lower (`lowerModificationLimit`) limits of the modified room air temperature set-point.

```
instance variables
ACSP : real := 0.0 -- Attack Current Set-Point

operations
public setAttack: () ==> ()
setAttack()==
(
  let SP = RATSP_IN.getReading() in
  if SP > 0.0 then
    if ACSP < SP then ACSP = SP + upperModificationLimit
    else ACSP = SP - lowerModificationLimit;
  )
  else ACSP = SP;


  RATSP_OUT.setState(ACSP)
);

thread periodic(attackFrequency)(setAttack);
```

Listing 1.1. FCU Man-in-the-Middle attack where `RATSP_IN` is the intercepted room air temperature set-point and `RATSP_OUT` is the (potentially) modified room air temperature set-point sent to the controller.

3 Secure Controller Design

In this section we propose an approach to using INTO-CPS multi-models for the design of controller augmentations intended to make the control of smart building systems more secure. Furthermore, we introduce how DSE can be used to explore the trade-off between security and usability inherent in the design of such a controller, and inform the design of an optimised solution.

3.1 Security Conscious Control

In an attempt to negate the impact of harmful or malicious use of a smart building and its constituent systems, we propose the addition of a *security monitor*, which observes any input parameters and intercepts usage patterns which might cause damage. In monitoring inputs, the system might utilise a range of metrics to assess system use, such as the frequency of instructions sent to the controller, the rate of change of inputs, or the calculated ware on equipment.

It is important that a security monitor can be included in a diverse set of control systems, so its design should be modular and independent of any particular controller. By designing the controller to be added to a generic input stream, it can be made suitable for inclusion in a range of applications, including integration with legacy systems and those with which limited information about their operation is available.

By employing a multi-model based approach in the design of such a monitor, we can constrain its implementation to an independent model to effectively demonstrate how security monitoring can be added to existing systems without their modification, and that any necessary computation can be executed on separate hardware.

3.2 Controller Optimisation

As described previously, there are multiple system metrics that may be evaluated when modelling the FCU CPS, though perhaps not all are critical for the design of the controller and security modules. It is important then that the stakeholders of the system are consulted and the appropriate metrics are highlighted. As Avizienis *et al.*[22] tell us, security is

> *"...a composite of the attributes of confidentiality, integrity and availability..."*

thus we should pick metrics that speak to these and in this way we can observe the trade-off of these antagonistic concerns.

We consider the trade-off between integrity (security) and availability (usability) of a system. In designing the security monitor, a more restrictive approach to security is likely to restrict the usability of the system in some way. By employing DSE in the design in such a monitor, we are able to explore the impact of each design on both of these considerations, enabling the selection of a design which falls within a desired threshold for some metrics of both security and usability.

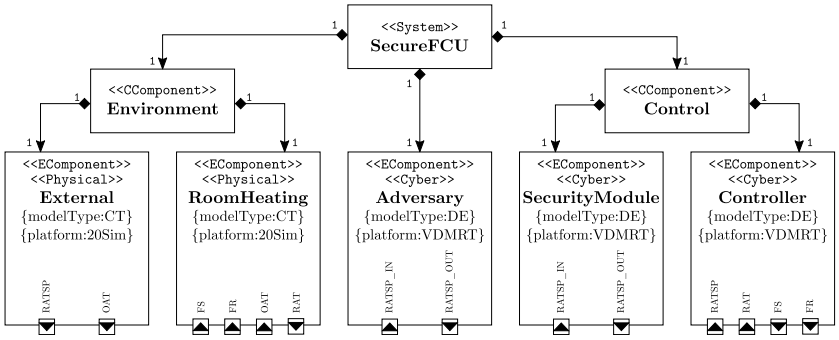


Fig. 2. Architectural Structure Diagram of secure FCU with adversary example.

4 Fan Coil Unit Case Study

In this section we illustrate our approach using a case study of an FCU. We describe an augmentation of the FCU multi-model provided in [5], which includes an additional model to undertake security monitoring. The multi-model is created using INTO-CPS technologies, and its simulation is used to determine fan usage in both the presence and absence of an attack. DSE is used to explore the trade-off between security and usability which results from varying the severity of response by the security monitor.

4.1 Security Controller Specification

We extend the FCU multi-model described in [5] by the addition of an additional `SecurityModule` model. The INTO-CPS SysML profile [23] is used to define the multi-model and its constituents in an Architectural Structure Diagram (ASD), illustrated in Figure 2.

The multi-model comprises 5 FMUs: `External`, `RoomHeating`, `Controller` and `Adversary` models are included from [5], and a complementary `SecurityModule` implements the behaviour of the security monitor. The `SecurityModule` model is a discrete-event model implemented in VDM-RT.

Each model is encapsulated in an independent FMU, and each FMU is defined using the *Encapsulating Component* (`<<EComponent>>`) stereotype. Additional parameters specify the model type (continuous/discrete) and a platform (in this case, VDM-RT and 20-Sim) for each FMU. The INTO-CPS SysML profile facilitates logical grouping of independent models by use of the *Collections Component* (`<<CComponent>>`) stereotype. This mechanism is used to indicate a relationship between `External` and `RoomHeating`, described as `Environment`, and similarly a relationship between `SecurityModule` and `Controller` described as `Control`.

The ASD is also used to define an interface for each FMU, specified as a series of ports which either input or output data to or from the model. Each port is labelled to describe the nature of the data it communicates, and an arrow specifies the direction of data flow. Data exchanged in the model includes the temperature of air both inside and outside of the room (*Room Air Temperature* (`RAT`), and *Outside Air Temperature*

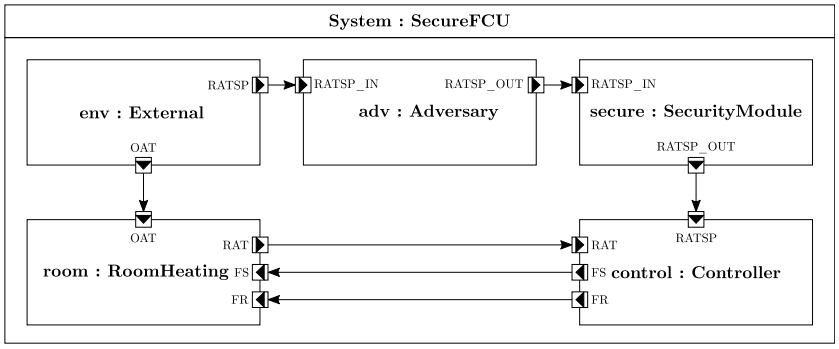


Fig. 3. Connections Diagram of secure FCU with adversary example.

(`OAT`)), the current desired temperature (*Room Air Temperature Set Point* (`RATSP`)), as well instructions for actuating temperature change (*Fan Speen* (`FS`) and *Flow Rate* (`FR`)). Data transfer between constituent models is defined using Connections Diagram (CD), illustrated in Figure 3, which makes connections between ports explicit.

The attack executed by the `Adversary` model accelerates wear on the FCU fan by instructing frequent temperature set-point changes to the system. To provide a countermeasure to this attack, the proposed security monitor filters fluctuating inputs using a moving average. By taking the average input over a sample of some period, the impact of input fluctuations can be significantly dampened, however this can introduce some delay in the FCU actuating the desired change in temperature. Listing 1.2 outlines the control loop of the security monitor model, which includes a parameter for specifying the length of the sample period (`sample_period`).

4.2 Controller Optimisation

To perform a DSE for optimisation, we need to detail both the range of the search and how results are to be computed. In the case of the FCU example we start by defining a range of values for the security module sampling period, here the range has a lower bound of one sample and an upper bound of 500 samples. The sample rate is one per minute and so the upper bound essentially averages the temperature set-points over an entire working day.

The metrics selected reflect two of the three security properties described by Avizienis *et al.*[22]. Specifically we evaluate the usage of the fan as an indicator of integrity, here the designer of the FCU has specified an acceptable fan usage limit of 25 and an attacker may attempt to push the usage beyond this limit. The second metric, temperature deviation, relates to the availability of the FCU for proper operation, i.e. achieving the desired room temperature. The acceptable range for temperature deviation is 3°c The best designs would minimise both of these metrics.

The DSE was performed under two sets of conditions (scenarios), the first is a normal working day, starting at 08:30 and finishing at 17:00, with the heating being off before 08:30, set to 21°c during the working hours and then off again after 17:00, this

```
instance variables
samples : seq of real;

operations
private monitorInput:()==>()
monitorInput()==
(
  if len samples = sample_period then samples := tl samples;
  samples := samples ^ [RATSP_IN.getReading()];
  RATSP_OUT.setState(sum(samples) / len samples);
);

functions
sum: seq of real -> real
sum(s) == if len s = 1 then hd s else hd s + sum(tl s);

thread periodic(monitorFrequency)(monitorInput);
```

Listing 1.2. Security monitor where RATSP_IN is the desired room air temperature set-point and RATSP_OUT is the filtered value sent to the controller.

is named 'without attacker' in the results. The second scenario uses the same working times and temperature set-points, but this time the cyber attacker is active and is able to intercept and change the room set-point as described earlier in Section 2.3.

The results of the search, in terms of the range of fan usage and temperature deviations for each controller frequency tested, are shown in Figures 4 & 5 respectively. On these graphs the colouration is used to indicate those controller frequencies that passed the constraint for that measure (coloured green) or that failed to meet the constraint (coloured grey). This is a departure from the normal INTO-CPS DSE result, where a Pareto analysis is used and colour represents the relative rank of a simulation result. Here the graphs indicate that there is a tension, with the best results for each measure being found at opposite ends of the controller frequency range.

Figure 6 presents a view of the results where the two objective measures are represented on the axis and the green colouration is only applied to designs that meet the constraints for both fan usage and temperature deviation, both with and without the attacker being active. Each '.' and '+' connected by a line represent a single design (controller frequency), with the location of the '.' and '+' indicating the fan usage and temperature deviations for that design. This view supports the stakeholders understanding what options are available in terms of the two measures both with and without the attacker, providing the stakeholders with information that allows them to explore the range of acceptable results to trade off between the two measures.

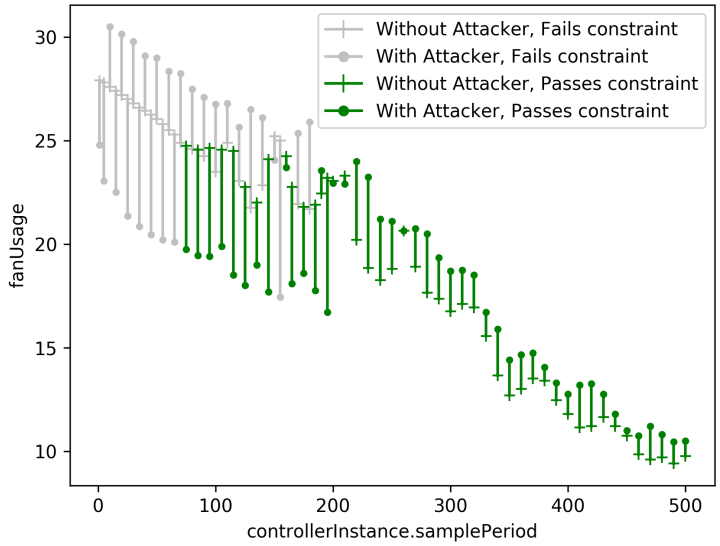


Fig. 4. Results showing the range of fan usage values for each controller frequency simulated.

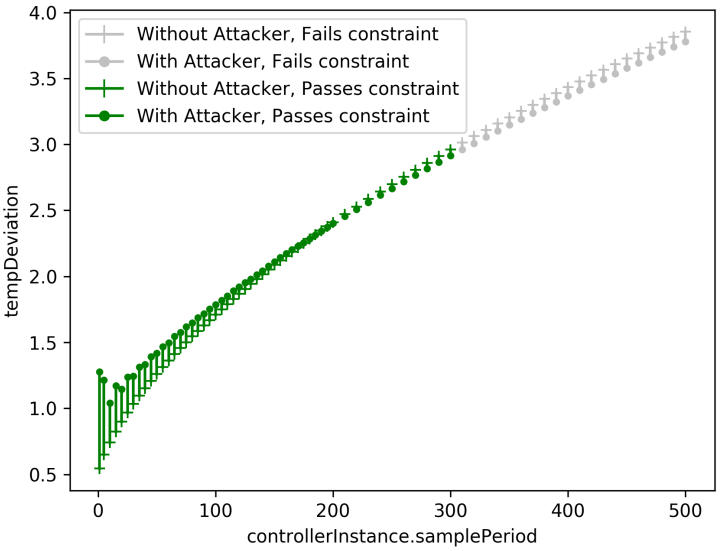


Fig. 5. Results showing the range of temperature deviation values for each controller frequency simulated.

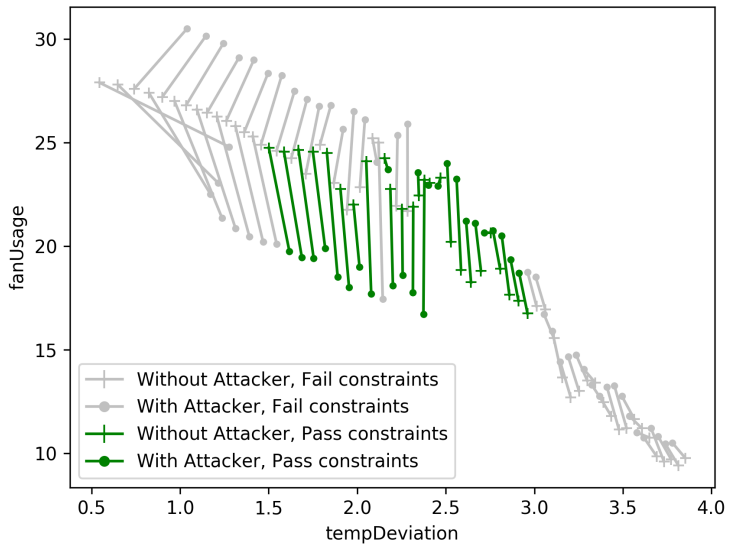


Fig. 6. Feasible security monitor designs within the bounds of maximum fan use of 25 and maximum temperature deviation of 3.

5 Future Work

The results presented in the previous section illustrate how DSE can be used to design a security controller, while assessing the security/usability tradeoff. It is particularly useful to be able to compare the effects on fan usage and temperature deviation with and without an attacker, since in practice, we cannot be certain whether an attacker is actually present or not.

The approach we presented in this paper is a stepping stone towards a generalised security controller design method, where it should be possible to sweep through multiple scenarios. Indeed, different users might have different usability requirements or ways of behaving within the system, and the security controller should be designed accordingly. However, this is likely to require the usage of probabilistic or statistical models, which are not yet supported in INTO-CPS.

The graphs in Figures 4 & 6 show that for fan usage, the range of controller frequencies that pass the constraint is not continuous. This is indicated by the presence of grey, non-compliant results, within the block of green, compliant, results. This emergent behaviour is believed to be due to the relative frequencies of both the controller and the attacker and indicates that the optimal controller frequency could differ with changing attacker frequencies. Thus we will expand upon the DSE reported in this paper to alter both controller and attacker parameters in the same search, to better understand the relationship between them.

Similarly, we would like to explore the DSE of the attacker and of the security controller using game theory, as it is likely that, in practice, the attacker will adapt its behaviour to that of the security controller, and conversely. Ideally, the tool support should help looking for a Nash equilibrium (i.e., for a configuration where neither the attacker nor the security controller has any incentive in changing their strategy).

Finally, we would also like to explore the design of more complex attackers, for instance multiple attackers synchronising their attacks, or attackers learning the behaviour from the users to increase their impact.

6 Conclusions

In this paper, we present an approach to the utilisation of multi-models in the design of a security controller for cyber-physical systems, particularly for the control of smart building systems. We have demonstrated how DSE can be particularly helpful in exploring the inherent trade-off between security and usability considerations, and have illustrated our approach on a simple case study, by the design a security monitor for a FCU, where the attacker aims at over-using the fan by tempering with the temperature set-points, and the security monitor provides a counter-measure by taking a moving average over input values.

We believe our approach is a stepping-stone towards a more integrated method to assess and design security mechanisms for the control of CPSs, and opens the door for modelling experts to include more complex defensive and offensive mechanisms in the discrete models of both controllers and potential attackers, respectively.

References

1. H. Boyes, "Security, privacy, and the built environment," *IT Professional*, vol. 17, no. 3, pp. 25–31, 2015.
2. S. Mansfield-Devine, "The dangers lurking in smart buildings," *Computer Fraud & Security*, vol. 2015, no. 11, pp. 15 – 18, 2015.
3. T. Mundt and P. Wickboldt, "Security in building automation systems - a first analysis," in *Proc. of the International Conference on Cyber Security And Protection Of Digital Services*, Cyber Security, pp. 1–8, 2016.
4. ENISA, "Threat landscape for smart home and media convergence," 2015.
5. J. Mace, C. Morisset, K. Pierce, C. Gamble, C. Maple, and J. Fitzgerald, "A multi-modelling based approach to assessing the security of smart buildings," in *Proc. of the PETRAS, IoTUK & IET 1st Int. Conf. on Living in the Internet of Things*, 2018. In press.
6. P. G. Larsen *et al.*, "Integrated tool chain for model-based design of cyber-physical systems: The INTO-CPS project," in *Proc. of the 2nd Int. Workshop on Modelling, Analysis, and Control of Complex CPS*, pp. 1–6, 2016.
7. K. W. Roth, D. Westphalen, J. Dieckmann, S. D. Hamilton, and W. Goetzler, "Energy consumption characteristics of commercial building hvac systems volume iii: Energy savings potential," 2002.
8. S. T. Bushby and H. M. Newman, "BACnet Today: Significant new features and future enhancements," *ASHRAE Journal*, vol. 44, no. 10, pp. 10–17, 2002.

9. M. Ruta, F. Scioscia, E. D. Sciascio, and G. Loseto, "Semantic-based enhancement of ISO/IEC 14543-3 EIB/KNX standard for building automation," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 731–739, 2011.
10. "Hack attack causes 'massive damage' at steel works." http://www.bbc.co.uk/news/technology-30575104.
11. R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.
12. "Let's get cyberphysical: Internet attack shuts off the heat in Finland." https://securityledger.com/2016/11/lets-get-cyberphysical-ddos-attack-halts-heating-in-finland/. Accessed: 12-03-2018.
13. "Lock out: The Austrian hotel that was hacked four times." http://www.bbc.co.uk/news/business-42352326. Accessed: 12-03-2018.
14. P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh, "Integrated tool chain for model-based design of cyber-physical systems: The into-cps project," in *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, (Vienna, Austria), IEEE, April 2016. http://ieeexplore.ieee.org/document/7496424/.
15. C. Kleijn, "Modelling and Simulation of Fluid Power Systems with 20-sim," *Intl. Journal of Fluid Power*, vol. 7, November 2006.
16. P. G. Larsen, K. Lausdahl, N. Battle, J. Fitzgerald, S. Wolff, S. Sahara, M. Verhoef, P. W. V. Tran-Jørgensen, and T. Oda, "VDM-10 Language Manual," Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org, April 2013.
17. M. Verhoef and P. G. Larsen, "Enhancing VDM++ for Modeling Distributed Embedded Real-time Systems," Tech. Rep. (to appear), Radboud University Nijmegen, March 2006. A preliminary version of this report is available on-line at http://www.cs.ru.nl/ marcelv/vdm/.
18. C. Gamble, "Comprehensive DSE Support," tech. rep., INTO-CPS Deliverable, D5.3e, December 2017.
19. J. Fitzgerald, C. Gamble, R. Payne, and B. Lam, "Exploring the cyber-physical design space," in *INCOSE Int. Symp.*, vol. 27, pp. 371–385, 2017.
20. J. Fitzgerald, C. Gamble, R. Payne, P. G. Larsen, S. Basagiannis, and A. E.-D. Mady, "Collaborative model-based systems engineering for cyber-physical systems, with a building automation case study," *INCOSE Int. Symp.*, vol. 26, no. 1, pp. 817–832, 2016.
21. M. Mansfield, C. Gamble, K. Pierce, J. Fitzgerald, S. Foster, C. Thule, and R. Nilsson, "Examples Compendium 3," tech. rep., INTO-CPS Deliverable, D3.6, December 2017.
22. A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.
23. E. Brosse, "SysML and FMI in INTO-CPS," tech. rep., INTO-CPS Deliverable, D4.3c, December 2017.

Part III

Perspectives and Methods Session

VDM at large: Modelling the EMV®2^{nd} Generation Kernel*

Leo Freitas

School of Computing, Newcastle University, UK
leo.freitas@ncl.ac.uk

Abstract. The EMV®[1] organisation specify payment protocols to facilitate worldwide interoperability of secure electronic payments. This paper is about the application and scalability of formal methods to a current and complex industry application. We describe the use of VDM to model EMV®2^{nd} Generation Kernel. VDM is useful for both formal specification, as well as simulation, test coverage, and proof obligation generation for functional correctness.

1 Introduction

EMVCo is a technical body that publishes and manages the EMV®specifications to facilitate worldwide interoperability and acceptance of secure payment transactions. Their protocols have been around since the late nineties and are used by major payment services providers (*i.e.* American Express, Discover, JCB, MasterCard, UnionPay, and Visa). As of 2018, there are over 7 billion EMV®payment cards, up from 4.8 billion in 2016, representing 55% of all payment cards issued globally. Their cards cover 64% of all worldwide transactions, and 99% of transactions in Europe. They are responsible for major payment technologies, such as contact (Chip&Pin), contactless (Wave&Pay), 3D-Secure online payment validation, and so on. In practice, relevant attacks on EMV1 were discovered [2, 4, 6, 5], with financial fraud related to payment systems rising in the last few years both in volume and type: for example, in the UK, there has been a 80% percent increase in value between 2011-16, when the fraud losses were £M618 [10].

In this paper we describe our experience in using VDMSL as a tool for understanding a complex (1900 pages) requirements specification of the upcoming EMV®2^{nd} Generation Acceptance System Specifications (EMV2) [8]. They include the familiar Chip&Pin and contactless protocols, as well as a number of new operational modes and security verification types (including biometric). We assume the reader is familiar with VDM [15]. Unfortunately, due to NDA restrictions, detailed information about the model, the choice of VDM, what the model exercise achieved, and how its application might have an impact in industry cannot be given. We hope to talk about this in future publications once EMV2 becomes public. Nevertheless, the underlying methodology we are following has recently been published [13].

---

* This is a preliminary version of the paper. Please cite the published version: L. Freitas. "VDM at large: Modelling the EMV®2nd Generation Kernel," in *Formal Methods: Foundations and Applications — 21st Brazilian Symposium, SBMF 2018*, Salvador, Brazil. Springer, 2018.

[1] EMV® is a registered trademark or trademark of EMVCo, LLC in the US and other countries.

We use VDMSL to formally specify the EMV 2^{nd} Generation kernel to enable specific protocol runs. The results have been productive and substantial. To date, we have modelled about 80% of the EMV2 kernel, and hope to complete it before public release. It comprises 135 VDM-SL modules and about 50 KLOC in VDMSL, some (20%) of which is automatically generated from an XSD/XML data dictionary, which describe the data structures used by the kernel APIs.

Elegance, which academics have in high-regard and can be useful for clarity and maintainability, often needs to be compromised in order to ensure stakeholders notice/accept the formal results: they ought to see the formal model built as something they recognise, as their artefact, rather than a nicer (more elegant) abstraction. Moreover, the complexities involved are quite substantial. The system needs to run seamlessly for long periods of time in different countries, currencies, financial services, policies and banking institutions; quite a task.

The work unravelled many technical issues in VDM, the identification of VDM tool bugs, as well as the limits of Overture as a tool. We hope these issues are interesting and that the VDM community finds tool suggestions useful.

2 EMV protocols and EMV2

The most common payment protocols are Chip&Pin and contactless, but a number of variations is technically possible. Analysis of EMV protocols is non-trivial due to the complexity of its requirements [7, 9]. They have to incorporate competing (and conflicting) interests from multiple user needs, banks and financial regulators worldwide.

A key differentiating feature of EMV2 is the fact its many (14) modules are completely distributed and may run concurrently (see Figure 1), as opposed to the monolithic sequential world of EMV1.

EMV protocols have many similarities. They constitute a series of steps encompassing a number of players, stages and features. The most common players are the so-called "point of interaction" (POI) terminals used by merchants (*e.g.* ATM machines, supermarket fuel pumps, card payment machines, *etc.*) and a card-profile used by customers (*e.g.* plastic cards, electronic tokens like smartphones/watches, *etc.*), as well as the issuer (*e.g.* banks and payment clearing systems). The main stages are:

1. **Application Selection** establishes the functionality of interest (*e.g.* credit card payment, or cash withdrawal from a specific account, *etc.*) with any additional extras (*e.g.* loyalty points, air miles, cash back, *etc.*), as well as the kind of transaction to engage with (*e.g.* acceptable challenge mechanisms, risk levels, information required by all parties, *etc.*);
2. **Transaction Processing** performs the necessary checks around agreed challenge mechanisms (*e.g.* pin-number, signature, biometric readers, *etc.*) and information required to make a decision about whether payment is to be approved, which may involve the issuer's approval;
3. **Other (kernel administrative)** stages exist for transaction restarting, rescuing, configuring, *etc.*

The core functionality comprises most kernel modules managing various protocol stages and features. The acceptance system comprises the POI and card communication layers.

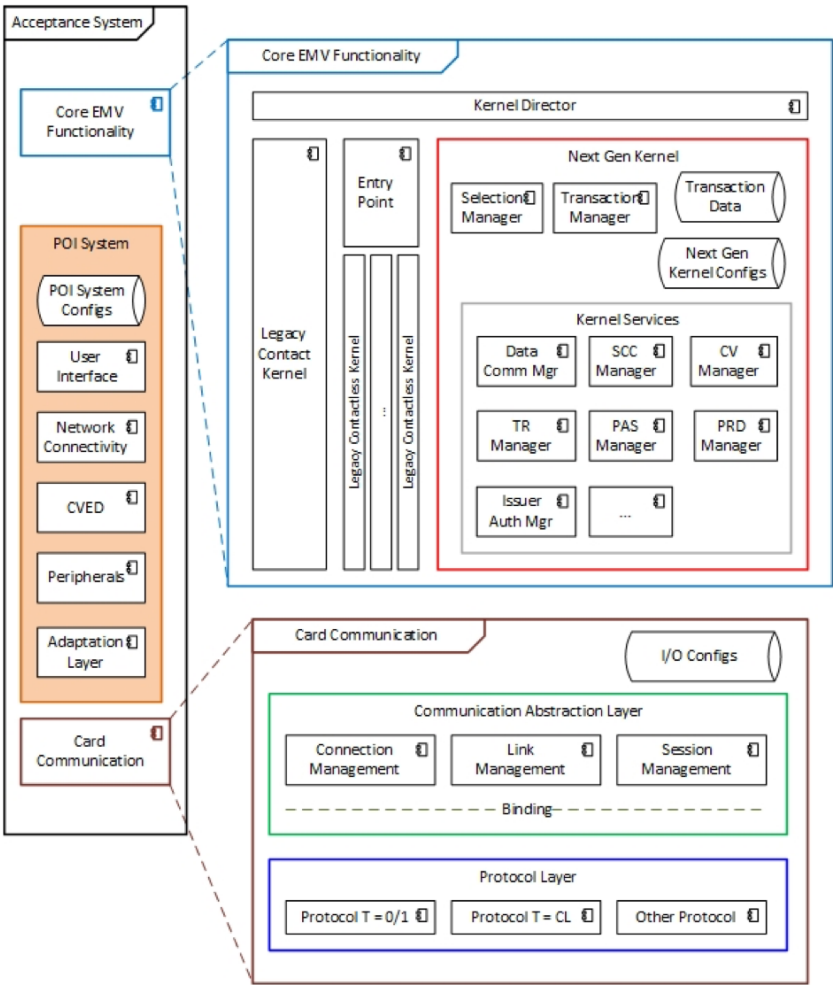


Fig. 1. EMV2 modules architecture

The former includes terminal management and card holder verification entry devices (*e.g.* pin pad, biometric scanners, *etc.*); whereas the latter implements a communication abstraction layer with the card-profile. This enables varying communication protocols to be instantiated outside the kernels core functionality.

3 Socio-technical challenges in modelling EMV2

We have prior experience with formal verification of protocols in Mondex [14], and in discovering attacks in EMV1 [5]. We are developing a methodology for the modelling and analysis of payment protocols. It involves a number of specification languages and tools used to capture different aspects of the process (see Figure 2). Crucially, these languages serve to shield payment-system engineers from formalism details, as well as to increase levels of automation as much as possible.

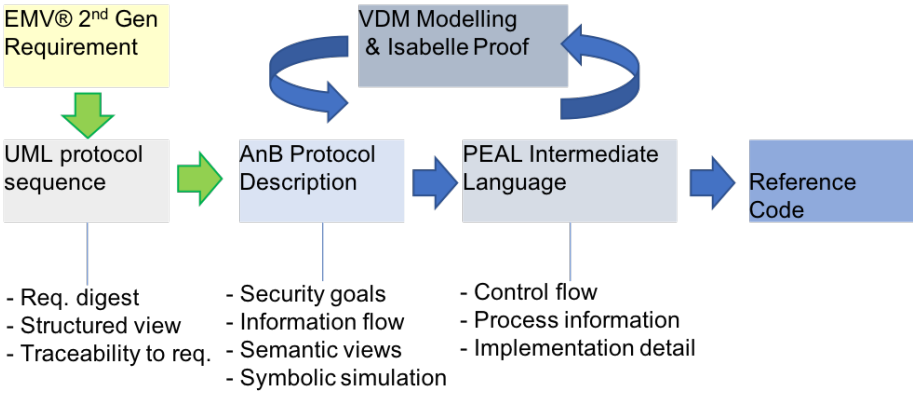


Fig. 2. EMV2 modelling methodology

Green arrows indicate informal (if rigorous) steps where we encode/capture requirements in a semi-formal notation akin to UML sequence diagrams. Blue arrows indicate formal steps through formal simulators (*e.g.* VDM Overture/VDMJ), new domain specific language semantics (*e.g.* PEAL) and compilers, and proof tools (*e.g.* Isabelle/HOL). For instance, PEAL is part of a current Newcastle PhD, and the work with AnB, an IBM protocol description language, is being developed with collaborators [19, 18]. This is ongoing work and more details are beyond the scope of this paper.

Given the size of EMV2 (*i.e.* 17 books in $1,900$ pages of requirements), we followed a set of principles advocated by Praxis: clear separation of concerns, consistent and well-defined "modelling hygiene" (*e.g.* naming conventions, indentation/documentation practices, dependency management, *etc.*).

4 VDM and its tools

We used Z (CZT and Z/Eves, see czt.sourceforge.net) to analyse EMV1 [12] and uncover some relevant attacks [5]. This approach worked well because we had a combination of empirical knowledge of EMV1 through understanding of its specification in implemented simulators (thanks to our collaborator Dr. Martin Emms). This meant our abstractions for how to represent the relevant parts of EMV1 were suitable to the practical realities of its protocols. Proof (or rather their failures) was used as the mechanism to identify and prevent theoretical threats, which we next tested for in practice with simulators using mobile-devices to perform attacks. This enabled us to practically demonstrate the severity, as well as the ease/difficulty, in enacting such attacks.

For EMV2, on the other hand, we believed a similar approach might not get us far. That is mostly because EMV2's considerably higher complexity and distinction in comparison to EMV1, in our view. We decided to use VDM and its formal simulation capabilities with Overture and VDMJ (see overturetool.org and github.com/nickbattle/vdmj) instead. This enabled a quicker prototyping of the kernel from its requirement specifications in order to provide us with the necessary knowledge about EMV2, and to start the discussion about its design decisions, as well as to enable the discovery of potential issues. We also envisaged the use of VDM's combinatorial testing [16] in order to exercise the number of protocol scenarios of most interest.

Moreover, we needed libraries for binary blobs and matrix manipulations. For the former, we used the nice VDM "DLL" style link with natively implemented libraries following the examples of `IO` and `VDMUtil`; whereas for the latter we used a combination of available and our own libraries from years of working with VDM's mathematical toolkit. For the most crucial libraries of binary blob transformers used for interfacing EMV1 legacy transactions within EMV2, and for matrix calculations used for transaction processing decision making, we used Isabelle/HOL (see isabelle.in.tum.de) to formally verify that proof obligations generated by the library definitions were correct. For example, in the binary library with varied word-size precision:

```
bv2nat: BitVector -> nat          nat2bv: nat -> BitVector

byte2bin: Byte -> BinByte         bin2byte: BinByte -> Byte
byte2bin(n) == binPad(nat2bv(n))  bin2byte(bv) == bv2nat(bv);
```

It transforms a bit vector into a `nat` and vice-versa, as well as their word-bounded variations with adequate padding. Beyond proving satisfiability of proof obligations in Isabelle/HOL, we also proved by induction interesting theorems like

$$bin2byte(byte2bin(bp-1), bp-1)$$

where $bp = 2^{wsize}$. This library also contains various operations over bit vectors like `and, or, not, xor`, *etc.*

4.1 VDM language issues

Our strategy to tackle design decisions led to interesting choices within the VDM language. In the process, a number of corner cases and interesting situations about language semantics arose. This led to a number of fruitful discussions with the VDM community, as well as tool extensions and corrections. Abstracting away all involved details in order to get to a minimal example was often time consuming and hard to tolerate. For example, lambda-expressions mistakenly allowed access to mutable state, which when used as a function-parameter or on-the-fly, led to unexpected behaviours.

```
op(x: nat) r: nat == is not yet specified ext wr c
post (lambda v: nat & v > c~)(c~+1);
```

Here Overture would not give an error, whereas VDMJ complains. In practice these lambda expressions were part of parameters to functions participating in the postcondition. The other example mistakenly allowed access to explicit-operation definition's return values within the operation's body or its precondition.

```
f(n: nat) r: nat == is not yet specified
pre some_condition_over(n, r);
```

Another interesting example had to do with type-invariant cascading/checks that were not quite right, and despite being very common, have not been uncovered before. All three print statements ought to flag a type invariant violation.

```
types
Dot = nat inv d == d < 4;
Bag = set of Dot inv b == card b > 2;

functions
test: Bag -> Bag
test(b) == b;

> print inv_Dot(-1)   > print inv_Bag({2,3,4})
true                  true

>print test({2,3,4})
Error 4060: Type invariant violated for Bag
```

VDM does not seem to handle cross product type parameters uniformly. For instance, it treats `(T * T) -> T` as a single tuple-input, whereas `T * T -> T` as a two parameter input. In itself, this is okay. Yet when developing libraries that involved polymorphic parameters and high-level functions (*i.e.* lambda-expressions as parameters) this distinction creates unnecessary confusion and difficult-to-debug/understand situations in the development of our generic libraries for binary numbers and matrixes. It

took a number of iterations (and a lot of time) to get to the bottom of it with this minimal example of the larger-scale scenario involving matrix calculations.

```
functions
f[@T]: @T -> @T * @T
f(a) == g[@T * @T](lambda x : @T * @T & x, mk_(a, a));

g[@T]: (@T -> @T) * @T -> @T
g(a, b) == a(b);

> p f[nat](1)
Error 4087: Cannot convert 1 (nat1) to (nat * nat) ...
6:      g(a, b) == a(b);
```

Calls to `f` fail with a cryptic error message, which did not help to figure out the underlying problem. It was about the same name of parameter `@T` was being used by `f` and `g`, but with `@T` referring to a different type in each case. When more than one polymorphic function is used in a call chain, the type parameter `@T` was not being uniformly passed. This, in combination with the function call non-uniformity, explained the reason why error messages were cryptic, and figuring out what was happening was difficult.

In Overture, export and import clauses are not treated properly. In imports, one can mix names of operation and functions without error or warnings, whereas VDMJ complaints. More seriously, `struct` export in Overture is not properly implemented at all, and works partially in what is quite confusing. Again VDMJ's stricter choices means if it is happy, so will Overture be. In a large specification where `exports all` is not adequate, the mishandling of (struct-)exports by Overture was a surprise with some cost as it was only discovered late. Finally, another quirk is VDM's "possible semantics", which in complex scenarios again led to a considerable amount of time to figure out what was going on. For example:

```
Type2 = bool | int;
Type3 = <A> | <B> |<C>;
Type4 = Type3 inv t4 == t4 in set {<A>, <B>};

functions
f1: Type2 -> bool                 g1: Type4 -> bool
f1(y) == is not yet specified;    g1(y) == f1(y);

f2: [Type2] -> bool               g2: [Type4] -> bool
f2(y) == is not yet specified;    g2(y) == f2(y);
```

As expected, the definition of `g1` gives an error about an inappropriate type for the argument. Nevertheless, `g2` only gives a type-error at run time thanks to the possibility of a `nil` input. A nicer/stronger warning in such cases would be welcome.

```
> print g2(<A>)
Error 4087: Cannot convert <A> (<A>) to (bool | int) ...
35: g2(y) == f2(y)

> print g2(0)
Error 3061: Inappropriate type for argument 1 ...
Expect: [Type4] Actual: nat
```

In all, all these scenarios served to highlight relatively simple issues that have been fixed in recent versions of both Overture and VDMJ. This brings up some interesting question to ask. How much do we trust our own tools to do work in safety and reliability? What evidence do we have that the tools themselves are sound? Having two independent tools can be practically useful, yet theoretically also increase this soundness concern. Interesting examples of how this has been mitigated for some tools exist, such as the *Circus* model checker [11], the CakeML compiler (https://cakeml.org), and the seL4 verified system (https://sel4.systems).

4.2 VDM language patterns

We had to come up with a number of ingenious VDM constructs in order to capture specific design decisions. In some cases, an alternative (more elegant) solution would be possible in theory, but we could not afford to take it in practice. Yet, in other cases, we could not think of a nicer solution at all.

Payment protocols by nature involve a considerable amounts of data from both the kernel to the card and vice versa. For example, different cards/terminals might require different information in order to setup a transaction and enable variability. Effectively, they entail a sort of reflective request over internal kernel/card state. For example, if the kernel might need the card for its long number and its expiry date in some transactions, or its public signature keys in others.

One solution to this kind of query would be to have the kernel state defined as a map from a somewhat structured string into whatever the target type was. Unfortunately, there are hundreds of type (and invariant) definitions, some of which are grouped as records with invariants between constituent fields. That means a kernel module state map would require an extraordinarily complex (and pretty much unreadable) invariant. Thus, we kept (often simple) invariants very close to where they were defined, and the various composition needs imposed the overall compound invariant of interest.

Our solution to enable reflective access was to take the XSD-schemas used to define EMV's type dictionary in order to automatically generate ($6,528$ LOC over 52 VDMSL modules) various data types of interest, as well as map transformers needed for reflective access. Maps were defined from `ID` (structured strings) into the so-called VDM wildcard ("?") type[2]. This enabled both reflective access and update, where we transformed records into maps and vice-versa.

[2] An example of its use can be seen in the `VDMUtil` library definition.

```
types
  emv_[P]_[M]_map = map ID to ?
  inv map_ == dom map_ = { ''IDs of interest'' };

functions
  default_emv_[P]_[M]: () -> emv_[P]_[M]
  [P]_[M]2map: emv_[P]_[M] -> emv_[P]_[M]_map
  [P]_[M]_map2rec: emv_[P]_[M]_map -> emv_[P]_[M]
  [P]_[M]_map_update[@T]: emv_[P]_[M]_map * ID * @T ->
                          emv_[P]_[M]_map
```

For this we wrote a Java program ($4,058$ LOC) that processed XSD and XML files and transformed them into VDMSL based on a template VDMSL file of about 90 LOC. This is used to produce the actual VDMSL files populated with XML/XSD information, where XSD files has a corresponding VDMSL files representing data types and constraints. These VDMSL files varied from 80 to 500 LOC depending on the underlying record type size and complexity. The VDMSL template key functionality is defined by four exported functions.

The `''IDs of interest''`, as well as the actual implementation of these functions, are populated through the data structures defined in the XSDs. The `[P]` stands for different packages, whereas `[M]` stand for different modules. Not all modules have packages and some packages have no modules API signatures for each kernel module and a few global configuration options are also defined by XSDs, which again we used to automatically generate top-level and internal operation signatures.

The use of "?" effectively enables a unbounded union type, something that arguably could have serious semantical consequences: that is why we do not `struct` export the map type. Even though this is arguably semantically dangerous, we are left with no alternative choice we could think of for specifying reflective (string based) access/update for records. The functions provide a default initialiser for the underlying record type (*i.e.* `emv_[P]_[M]`), conversion from the record to the corresponding map type (*i.e.* `emv_[P]_[M]_map`), conversion from the map type back to the record type, and finally a map update function for a given `ID`. The use of polymorphic type `@T` is important in order to ensure external users of the function satisfy the record type invariant not imposed within its corresponding map type. The update function has a precondition about the `ID` belonging to the domain of the map type as the invariant requires: this ensures only fields known within the record type can be "reflected" over the map type. With this setup, it is possible to write expressions like

```
x := x_map2rec(x_map_update[Type](x2map(x), id, value))
```

which projects a record (`x`) of corresponding record type (`emv_[P]_[M]`) into its map type (`emv_[P]_[M]_map`), updates it at the specific `id` name with a specific `value` of the right `Type`, and then transforms it back to the original record type. Thus, this provide

reflective access/update to records represented as maps with type invariants implicitly guaranteed.

This way, we managed to have both strong type invariants across multiple fields and records, and yet still allow for reflective (string-based) access to kernel state via our automatically generated map transformers. In practice, this worked quite well: we anticipate changes to the kernel over time will most often come from data structure variations, rather than new API or protocol stages. When such changes happen, all we have to do is regenerate the mapping specification, and mostly all is automatically up to date with internal version updates prior to release. This process has proved invaluable for productivity: that is because considerable amount of (tedious and error prone) work is completely (and correctly/safely) automated.

Another pattern of interest was in the use of data structures for data exchange between the kernel and the external world (of the card and the POI). As with reflective state access, the data structures have to fetch kernel state data for a given list of IDs requested. That entails different (if often predictable under various conditions) data structures with invariants depending on the ids requested. In order to define such dynamic invariants, which are often only knowable once the specific request has been assembled, we defined records with structures like

```
types
  IDList = seq of ID
  ContainerData = seq of ?;
  Container ::
    dil   : IDList
    data  : ContainerData
    invariant : IDList * ContainerData +> bool
  inv mk_Container(list,data,invariant) ==
    --@doc all IDs have corresponding data
    len list = len data
    and
    ---@doc IDList within container must respect ID allowances
    respect_boundaries(list)
    and
    --@doc extra invariant to ensure specific needs
    invariant(list,data);
```

A concrete example of such dynamic invariants could be given by a `mk_Container` expression with a lambda-expression or invariant function with known conditions; or when in less unpredictable circumstances, an extended record where the actual known invariant per module can be checked:

```
  KernelDirectorContainer = Container
  inv mk_Container(list,data,invariant) ==
    invariant(list,data) <=> some_inv_function(<kd>,list,data);
```

4.3 Pushing language boundaries

Use of VDM wildcard type.
In discussion with the VDM community about these issues, a number of language extensions were discussed. For instance, what roles (if any) should the wildcard ("`?`") type play? I first came across it within `VDMUtil` library, and when I looked at the VDMSL manual, it was not in the Lexer's token vocabulary! Yet, once I understood what it achieved, I started playing with its possibilities. Perhaps types involving "`?`" cannot ever be struct-exported. It was incredibly useful in a few places, yet it can also be incredibly dangerous to have around untamed. Without it, however, we would struggle to provide a sensible/scalable solution to state reflective-access and other problems. It can be quite dangerous to use, though: for the reflective maps above, our initial type was `map ID to [?]`, given some values could be `nil`. The combination of wildcard that can be nil leads to a situation where both Overture and VDMJ get in quite some trouble without any sensible error message. The solution was to realise that `nil` was already part of "`?`", hence no need to define it twice!

Another interesting example was inspired by SPARK/Ada modules. We defined a `Stack` type as an abstract data type, which in SPARK terms means an opaque type with public APIs to manipulate it (*e.g.* `push, pop, peek`, *etc.*). In VDM, that would mean having a non-struct exported `Stack` type so that its internal implementation can never be exploited by its users, and only its public APIs are usable. This worked well in VDM with VDMJ checking for struct exports properly, but up to a point. Like with C++ template-class or Java Generics, what if we wanted a stack (however it is implemented), but of a specific type (*e.g.* `Stack<nat>`)? Because VDM does not allow polymorphic type declarations to participate in type definitions, this was quite hard/convoluted to impose/define. Again, we used the wildcard type for such module parametric types. Perhaps allowing polymorphic type variables in type definitions, or even have module type parameters might be an interesting language extension.

Framing conditions.
Coming from the Z world, I found the lack of linguistic support for complex state framing conditions an issue. VDM explicit-extended operation definitions allow the specifier to define framing conditions in terms of what can be read/written, which also define access conditions in pre/post specifications. This is quite useful, yet also quite limited. Assuming complex state, say with a number of fields, each as records with other fields, totalling 10 to 50 fields. What happens when an operation touches only one or two fields of one of these records? For example:

```
operations
  Some_API() ==
      (... x.a := y.b + 1;  ...)
  ext   rd y  wr x
  post  some_api_frame(x~,x) and ...;
```

It writes on one of the state fields (`x`), and uses information from another (`y`). The only state update involved is to change one field, and everything else remains constant. The

VDM frame condition does not allow changes to `y`, but it does allow changes to any of the other fields in `x` that must remain constant. That entails the definition of a framing postcondition as:

```
functions
  some_api_frame: X_Type * X_Type -> bool
  some_api_frame(bx,ax) ==
    --bx.a == ax.a and
    bx.b == ax.b and bx.c == ax.c and ...;
```

These framing postconditions may also be conditional on possible paths taken within the API. The lack of a linguistic mechanism in VDM to tackle such complex-state simple framing-conditions became quite a drag within the many API implementations. In Z, this is easily done with a combination of schema calculus (*e.g.* Ξ and hiding) operators. An anecdotal summary was given by a collaborator within the VDM community: "you've touched on a couple of really interesting points: one about how the tools work, and one about the best way to write a specification!".

4.4 Overture and VDMJ

As a tool, Overture offers all the modern-day IDE "bells and whistles" most users expect, such as asynchronous specification checking (*i.e.* type check as you type), various useful dialogs and keyboard shortcuts for common tasks, integrated execution/building/debugging, and so on. VDMJ, on the other hand, works like a Linux command-line killer app, which includes all the functionalities Overture provides, as well as debugging and other facilities. So, users may wonder: why have both? Well, they are independent implementations from different sources. Yet, given their different interpretation of the language semantics, they effectively "speak" different VDM "dialects".

As far as I know, internally they are quite different in the sense that VDM ASTs were reengineered for various reasons [1]. In practice, the experience was that Overture cannot cope with the scale of a model of this size. From very early on, Overture started to lag considerably, and parsing/typechecking would take too long (5 to 20 sec.) to be productive. The debugger also stopped working without any warning/error: it simply freezes for reasons yet unknown. It was often more lax with language construct issues/errors, which entailed hours wasted chasing complicated red herrings of no interest.

VDMJ, on the other hand, has always been quite reliable. Most important, it is fast. All debugging and simulation since at least half way through the project has been done through it. Debugging in VDMJ is not as smooth as in the Overture Eclipse-like environment, but it works quite well and is more stable. Complex breakpoint conditions in Overture often led to connection errors and tool freezes, whereas in VDMJ they work reliably and were invaluable.

In practice, I work with a combination of both, where I use Overture for typesetting and project management chasing top-level (quick to parse and feedback) errors, and VDMJ for guaranteeing all is well, and for simulation, testing and debugging. It

Kernel Module	Book	APIs	%	LOC
Kernel Director	2	40	100	3,950
Selection Manager	3	23	95	3,647
Transaction Manager	4	20	95	3,423
Cardholder Verification Manager	5	18	100	3,613
Terminal Risk Manager	6	16	100	1,691
Additional Services Manager	7	9	10	1,081
Payment Related Data	8	?	0	0
Issuer Authorisation Manager	9	16	100	2,272
Data Communication Manager	10	31	95	2,633
Secure Channel Manager	11	?	0	0
Communication Abstraction Layer	12	11	20	1,006
Point of Interaction Terminal	13	35	90	1,944
Card profile and Detection Service	14	9	20	690
Data Dictionary XSDs	15	0	100	9,537
Support				
Module Data Store	-	0	100	8,591
EMV Database Link	-	0	80	593
VDM support libraries	-	0	100	1,242
Total		228	80	45913

Table 1. EMV2 VDM specification

often happens that Overture will say a specification is okay, when VDMJ will throw you a number of residual errors; this is yet to happen the other way round. In early 2018, we experienced a quite severe lag in Overture, which led to some profiling with VisualVM (https://visualvm.github.io), and provided evidence there is a serious (deterministic) memory leak somewhere. In an industrial setting, this kind of complication, alongside the already alien nature of formal reasoning, can sadly become the excuse for non-adoption.

5 Evaluation and Discussion

Overall, the exercise has been quite worthwhile, and VDM and its tools have worked well. To give a clearer sense of scale, Table 1 gives a specification breakdown per module, where we point to the relative completion of each module, and its size in VDM lines[3]. Some modules have a large number of public APIs like the "Point of Interaction", but they are simple; whereas others like the "Cardholder Verification Manager" is small in API numbers but is way more complex. We are yet to work on the payment data and secure channel modules, as they are not crucial for the overall kernel functionality, but rather its additional services and secure communication features. The module data store comprises reflective access to state information via string-based named, data container types used for exchange between modules and external entities. We have an

[3] Numbers were calculated with a mixture of string search, and Linux tools like `find` and `wc`.

initial Java emulator implementation that is also development and is informed by our VDM model. At first, we considered using the Overture automatic code generator for Java with its translation of VDMSL into Java specifications in JML. Unfortunately, it quickly became clear the code generator's breath over VDMSL was not good enough for our needs. Simple constructs like constant value declarations could not be translated, even though constant functions with no input parameters (*i.e.* an alternative way to define constant values) did work. It would be a valuable exercise to extend the code generator for the future.

The lack of a mechanism for a VDM mathematical library/repository is a problem, and entails many people using VDM have to reinvent the wheel with respect to commonly used specification constructs. Perhaps something like the Maven-central (https://search.maven.org) repository in style, where all the necessary due diligence can be done by the repository manager would encourage more people to submit libraries themselves.

The distinction in error handling between Overture and VDMJ can be quite dangerous: it might completely knock of the confidence of users leaving them uncertain whether their choice for VDM was the right one. This, together with more targeted error messages to help developers fix the problems is quite important. The inability to debug/run EMV2 in Overture due to its freezing caused quite some concern at the time because it created a sense of time wasted and wrong choice of language made; this issue is still pending.

5.1 Tools wish list

Throughout this work, there were a number of tool extension ideas. Some of them are EMV2-specific, yet a number can be of wider use, and we list them here as a suggestion to the VDM community in our perceived order of relevance.

1. **VDM profiler**. For larger modules, identifying where resources (time/memory) are being consumed is quite important in order to fine-tune modelling decisions. In CZT, users can instrument the tools to have counters for various specification constructs (*i.e.* number of names, or predicate parts, *etc.*), as well as detailed information about load times at what stages (*e.g.* parsing, typechecking).
2. **Direct separation between dialects**. In the current exercise, we had to cope with asynchronous API calls and a concurrent programming paradigm that we wanted to specify. VDM-RT already has `asynch` and thread concepts, but it imposes on the user the extras of VDM++. That is, the VDM dialects (SL, ++, RT) are somewhat (unnecessarily) interwoven. In CZT, developers can pick and choose (sub-)dialects (*e.g.* Circus composes Z and CSP; TCOZ composes Object Z, CSP and Time; OhCircus comprises Z, CSP and an object oriented extension) and compose them according to need without having to have unwanted (sub-)dialects. It would be nice to have say SL with real time constructs and/or asynchronous calls and threads without VDM++ extensions.
3. **Dependencies-graph generator for imports xand operation call-graphs**. One way we found that minimised load time (from 57s to 20s in VDMJ) was to minimise dependencies, particularly circular dependencies. I presume Eclipse-based plugins already exists (in C and Java) for such dependency and call graph management.

4. **Heap images / serialisation to speed processing/execution**. For large specifications, reload time is costly. Once a specification stabilises and simulation time will be spent running/adjusting minor issues, avoiding total (lengthy) reloads would be useful. In CZT, that is viable through the use of ZML: an XML-encoding of ISO-Z that is lightening fast to process. Isabelle/HOL use Poly-ML heap images to store "compiled" proofs of larger libraries. Both these solutions serve to improve scalability and usability of tools and would greatly benefit users of larger VDM specifications.
5. **Test case generation from specification based on something like Eisenbach patterns**. Combinatorial testing in VDM is great, and enables productive specification validation. Yet, identifying what to test is sometimes difficult. A tool can create test cases of interest based on the shape of specifications involved and user instrumentation. For example, a disjunctive precondition `A or B` might need specific tests per disjunct; earlier work on VDM for this exist [3]. Predicate pattern-languages like Isabelle/HOL's Eisbach [17] could be used to determine what shapes tests should come from.
6. **Quickcheck/nitpick style test case generator for simulations**. QuickCheck is a test case generator written for Haskell programs[4], and now ported to a number of different situations. Nitpick identify counter examples to conjectures in Isabelle/HOL. A combination of these tools in Isabelle/HOL considerably improves proof effort. Something similar in VDM could help create test cases of interest and root out specification errors quickly.
7. **Record form filler for long record initialisers**. When modelling complex records, `mk_` expressions can be quite awkward. Having an automatically generated GUI to construct such values from declared type information would be quite useful.
8. **XSD/XML/Swagger VDM integration**. XSD/XML and JSON-based languages like Swagger are used in industry to provide a semi-formal/rigorous type/API-signature definitions. A translator to/from VDM would be valuable: translating to VDM increases productivity, whereas translating from VDM keeps requirement specification documents accurate and up to date.
9. **"Fix imports" + "quick-format" refactoring on imports and indentation**. Imports in VDM are a bit awkward. Explicit module imports (*i.e.* `Module‘Type`) make it difficult to identify module dependencies, whereas complete import listing with renaming is quite tedious to do. Something like Eclipse's fix-imports and quick-formatting for indentation would be quite useful.

6 Conclusions

The work presented in this paper demonstrates it is possible to use VDM for a large scale (50 KLOC VDMSL) specification, despite various tool problems and practical challenges involved. The availability of a formal simulator for EMV2 and the careful documentation of its assumptions/commitments will hopefully pave the way for the influence of formal modelling within payment systems industry.

[4] See https://hackage.haskell.org/package/QuickCheck

Acknowledgements. This work is associated with a long term collaboration with Dr. Martin Emms, an expert in EMV protocols, simulators, and hardware; and Dr. Paolo Modesti, an expert in security protocols design. We are part of a team developing the underlying methodology applied to EMV®2^{nd} Generation [13]. We are also grateful for EMV®'s support and technical discussions, specifically by Mike Ward and John Beric from the EMV®Security Working Group, and by Carlos Silvestre from the EMV®2^{nd} Generation Task Force. Finally, I am grateful for my department support, and Nick Battle and the VDM community for many interesting discussions, and patience in handling a number of issues.

References

1. N. Battle. Analysis separation without visitors. In *15th Overture Workshop*. Newcastle University, 2017.
2. M. Bond, O. Choudary, S. J. Murdoch, S. Skorobogatov, and R. Anderson. Chip and skim: cloning emv cards with the pre-play attack. In *S&P*, pages 49–64. IEEE, 2014.
3. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, Odense, Denmark, April 19-23, 1993, Proceedings*, pages 268–284, 1993.
4. S. Drimer, S. J. Murdoch, et al. Keep your enemies close: Distance bounding against smartcard relay attacks. In *USENIX security symposium*, volume 312, 2007.
5. M. Emms, B. Arief, L. Freitas, J. Hannon, and A. van Moorsel. Harvesting high value foreign currency transactions from emv contactless credit cards without the pin. In *CCS*, pages 716–726. ACM, 2014.
6. M. Emms, B. Arief, N. Little, and A. Van Moorsel. Risks of offline verify pin on contactless cards. In *Financial Cryptography and Data Security*, pages 313–321. Springer, 2013.
7. EMVCo. Emv integrated circuit card specifications for payment systems [books 1 to 4], November 2011. https://www.emvco.com/emv-technologies/contact/.
8. EMVCo. Next generation kernel system architecture overview. Technical report, EMVCo, 2014.
9. EMVCo. Emv contactless specifications for payment systems [books a,b,c-1,c-2,c-3,c-4,c-5,c- 6,c-7 and d], February 2016. https://www.emvco.com/emv-technologies/contactless/.
10. Financial Fraud Action. Fraud the fact. the definitive overview of payment industry fraud and measures to prevent it, 2017. https://www.financialfraudaction.org.uk/fraudfacts17/.
11. L. Freitas, A. Cavalcanti, and J. Woodcock. Taking our own medicine: Applying the refinement calculus to state-rich refinement model checking. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, pages 697–716, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
12. L. Freitas and M. Emms. Formal specification of emv protocol. Technical report, Newcastle University, 2014.
13. L. Freitas, P. Modesti, and M. Emms. A Methodology for Protocol Verification Applied to EMV(R) 1. In *Formal Methods: Foundations and Applications - 21th Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 28-30, 2018, Proceedings*, volume 11254 of *Lecture Notes in Computer Science*. Springer, 2018.
14. L. Freitas and J. Woodcock. Mechanising mondex with z/eves. *Formal Aspects of Computing*, 20(1):117, Jan. 2008.
15. C. B. Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990.

16. P. G. Larsen, K. Lausdahl, and N. Battle. Combinatorial testing for vdm. In *SEFM*, pages 278–285. IEEE, 2010.
17. D. Matichuck, M. Wenzel, and T. Murray. *Eisbach User Manual*. Technical University of Munich, Oct. 2017.
18. P. Modesti. Efficient Java code generation of security protocols specified in AnB/AnBx. In *Security and Trust Management STM, Proceedings*, pages 204–208, 2014.
19. P. Modesti. AnBx: Automatic generation and verification of security protocols implementations. In *Foundations & Practice of Security*, LNCS 9482. Springer, 2015.

Transforming an industrial case study from VDM++ to VDM-SL

René S. Nilsson[1,2], Kenneth Lausdahl[3], Hugo D. Macedo[1], and Peter G. Larsen[1]

[1] Department of Engineering, Aarhus University, 8200 Aarhus N, Denmark
[2] AGCO A/S, Dronningborg Allé 2, 8930 Randers NØ, Denmark
[3] Mjølner Informatics A/S, Finlandsgade 10, 8200 Aarhus N, Denmark

Abstract. Normally transitions between different VDM dialects go from VDM-SL towards VDM++ or VDM-RT. In this paper we would like to demonstrate that it actually can make sense to move in the opposite direction. We present a case study where a requirement change late in the project deemed the need for distribution and concurrency aspects unnecessary. Consequently, the developed VDM-RT model was transformed to VDM++ and later to VDM-SL. The advantage of this transformation is to reduce complexity and prepare the model for a combined commercial and research setting.

Keywords: VDM, industrial application, model transformations

1 Introduction

The Vienna Development Method (VDM) is one of the most mature formal methods [2]. The method have been extended with multiple dialects over time, including the ISO standardised VDM Specification Language (VDM-SL) [3], VDM for object-oriented modelling (VDM++) [4] and VDM Real Time (VDM-RT) [12]. The choice of dialect highly depends upon the type of system or behaviour that must be modelled.

In this paper we present an industrial project involving optimization of the logistics in harvest operations. Such an operation is inherently distributed, as it involves a number of independent vehicles that need to coordinate and interact. Development guidelines for distributed real-time systems [7] were initially followed, resulting in a rather complex VDM-RT model [1].

A significant requirement change was introduced late in the project. Concretely, a specific communication protocol between vehicles as well as a specific hardware platform on each vehicle were imposed. Consequently the system architecture was changed to comprise a single centralised control algorithm and thin data-acquisition applications on each vehicle. The change of architecture diminished the need for distribution in the model, as the core functionality now only consisted of a single control algorithm and not a distributed control. In order to keep the model consistent with the modelled system and to reduce model complexity, distribution was removed from the model and thereby transformed to VDM++.

In this paper we will focus on a further transformation to VDM-SL, which was conducted to reduce model complexity and prepare the model for a combined commercial and research setting.

The evolution of the project is further described in Section 2. Afterwards Section 3 continues with concrete transformation examples and guidelines on transforming a VDM++ model to VDM-SL. Next, Section 4 presents an overview of the structural changes between the VDM++ and the VDM-SL models from the case study, while Section 5 provides an evaluation on the results obtained herein. Finally, Section 6 concludes on the findings of this work.

2 Case Study Project Evolution

The industrial case study presented in this paper originates from a research project named *Off-line and on-line logistics planning of harvesting processes*, involving Aarhus University and AGCO A/S.[4] Logistics optimisation in this setting includes both static and dynamic route planning of all involved vehicles. Specifically two tools were developed, 1) an *off-line simulation tool*, where a harvest operation can be optimized and simulated under the assumptions that no deviations occur, and 2) a *real-time guidance system*, that provides guidance to all drivers in the different vehicles involved and continuously monitors and reacts to possible deviations.

A common workflow in a harvest starts with a combine harvester harvesting the crop. The collected yield is unloaded into an in-field grain cart, which again unloads to an on-road truck, which delivers the yield to a drying or storage facility. This is illustrated in Figure 1, where parts of the optimized routes for each vehicle are shown.

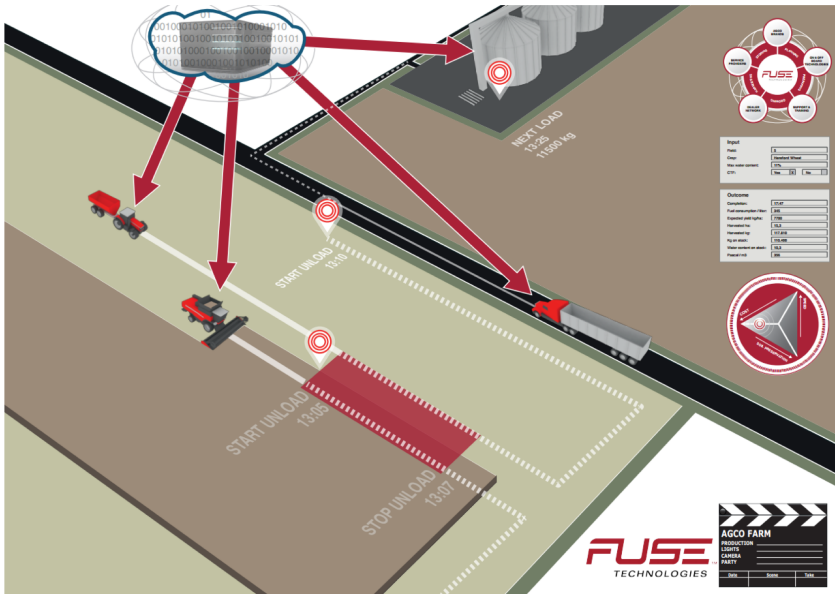


Fig. 1: Harvest logistics illustration.

[4] https://goo.gl/6tT8tK

During the four year span of the research project, the underlying VDM model has evolved and changed dialect a number of times, as depicted in Figure 2. The arrows and numbering illustrate how the model evolved over time. The arrows *2* and *4* highlights that substantial modifications were made to the VDM++ and VDM-RT model during the research project. These modifications are one of the main reason why the final VDM-SL model is different from the initial SL model. Note that the initial SL model was not kept up to date with the changes introduced into the VDM++ nor the VDM-RT model. Additionally, change in personnel had the side effect that knowledge of the initial SL model was lost. The following subsections further describe the motivation and reasoning behind each change of dialect.

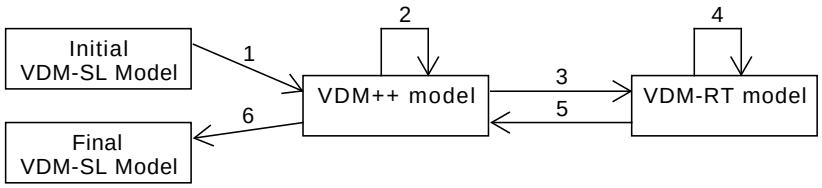


Fig. 2: Model evolution over time

2.1 Initial VDM modelling

The initial requirements for the project were defined based on the domain knowledge about the problem at hand. The system was considered a complex distributed system with embedded devices deployed in each vehicle. Therefore the development guideline for distributed real-time systems proposed by [7] was mostly followed. It involves a step-wise transition starting with a VDM-SL model, which is transformed to VDM++ and finally to VDM-RT, where more details of the system is included in each transition. The initial VDM-SL model captures the specification or the core functionality of the system. The transition to VDM++ adds concurrency and object-orientation, and the final transition to VDM-RT adds real time and deployment aspects.

For performance reasons, the Java bridge technology, offered by the Overture Tool, was leveraged [9]. This allowed the VDM model to invoke external Java components, such as existing Java graph libraries and proprietary performance optimized Java code [10, section 3.4]. Code-generation of the VDM model to Java further improved the performance [6], while easing the deployment process. Figure 3 shows how the VDM model and tests were connected to external components through a `Bridge` and how the same external components were integrated with the code-generated system through a corresponding `Delegate` class.

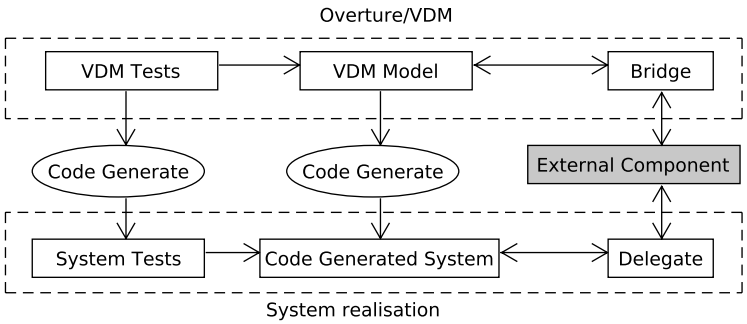


Fig. 3: Code generation and external components integration from [1].

2.2 Requirement changes

Late in the project a major requirement change was introduced. Initially the communication between vehicles had not been restricted in any way, and the hardware platforms were not constrained either. This allowed for an easy implementation of a distributed control algorithm. The requirement change meant that the system should work with existing hardware platforms and backend systems. All communication should now use a Publish/Subscribe (P/S) service, and the possibility to deploy software to the vehicles were highly constrained, as it should fit within an existing platform. As a consequence, it was decided to implement a centralized control algorithm in the cloud, and each vehicle should now only be responsible for real-time data acquisition and providing a user interface to the driver. This change towards centralized control diminished the purpose of including distribution in the model. Therefore, distribution was removed and replaced with a P/S interface and thereby transforming the VDM-RT model to a VDM++ model again.

2.3 Generalisation and commercialisation

At the end of the research project the model architecture was revised with the purpose of preparing the model for commercialisation. Additionally, a generalisation of the system towards other farming operations was envisioned. During the revision, focus was on the core functionality. One conclusion was that communication should be separated from the core model, hence removing the need for concurrency. This is enabled by the use of the Java bridge as described above.

Ideally the core optimisation and control algorithm could be achieved in a purely functional manner, taking the current state and incoming P/S events as input, and outputting a new state, including route plans for all vehicles.

In order to reduce complexity and best model the system, VDM-SL was chosen as the most appropriate dialect. From a research standpoint, the use of VDM-SL and a functional style should also allow new students to easily work on delimited parts of the

model. Additionally it should enable formal proofs of certain model properties, which is not easily done in a VDM++ model. This led to the final transformation from VDM++ to VDM-SL, which is further described in the following sections.

3 Transformation guidelines

Transformation of a VDM-RT model to VDM++ is relatively simple since both dialects are object-oriented and we had limited use of VDM-RT specific constructs. However, transformation from VDM++ to VDM-SL is not straight forward, because the two languages does not share the same feature set, nor semantics [8]. Some of the main transformation challenges include concurrency, objects, hierarchy, operation overloading, and visibility.

In an attempt to create a systematic approach to transforming VDM++ models to VDM-SL a number of guidelines and concrete transformation rules are defined as described below.

3.1 Guideline 1: Concurrency

The VDM-SL dialect is a single threaded model and thus does not have any support for multiple threads nor coordination thereof. Therefore models cannot in general be converted into the VDM-SL dialect unless the nature of the problem is such that the multi-threaded behaviour can be moved out into an external Java component. This is in particular the case if the multi-threaded behaviour is present in order to facilitate communication where the data stream instead can be converted into a sequence of events, which can be consumed by the VDM-SL model sequentially.

3.2 Guideline 2: Visibility

In VDM++ the visibility of operations is declared using the access modifiers **`public`**, **`private`**, **`protected`** and **`static`**. In VDM-SL all definitions are static and the visibility is declared using the **`export`** and **`import`** constructs. Hence, if an operation should be visible in another module, the declaring module should export the operation including any internal referenced types, and the other module should import it.

State in VDM-SL is only visible within the module it is declared in. If the visibility needs to be extended, getters and setters can be implemented, which follows best OO practices.

3.3 Guideline 3: Operation overloading

Operation overloading is not supported in VDM-SL. All operations must be unique based on their name and any calls that relied on the overload behaviour must be guarded by an if statement to determine which operation to call.

3.4 Guideline 4: Objects and state

All object instances must be transformed into **`state`** components. This is described in the following transformation rule, and the accompanying example in Figure 3.

```
class A

types
Data : seq of char;

instance variables
data : Data;

operations
opX : () ==> ()
opX () == data := "ok";
```

```
class B

instance variables
objA : A;
data : real;

operations
opY : () ==> ()
opY () == (
  objA.opX();
);
```

(a) VDM++ classes

```
module A

types
ID = token;
S = seq of char;
InstanceMap = map ID to S;

state AST of
  a_m : InstanceMap
  init s ==  s = mk_AST({|->})
end

operations
opX : ID ==> ()
opX (id) == a_m(id) := "ok";



...
```

```
module B

types
ID = token;
S ::
  objA : [A`ID]
  data : real;
InstanceMap = map ID to S;

state BST of
  b_m : InstanceMap
  init s ==  s = mk_BST({|->})
end

operations
opY : ID ==> ()
opY (id) == (
  A`opX(b_m(id).objA);
);
```

(b) VDM-SL modules

Fig. 4: Transformation example: VDM++ classes translated to VDM-SL using guideline 4: *Objects and state.*

Transformation rule 1: Object instances map to state A class is transformed into a module that exports all functions and operations according to guideline 2. The class instance variables are translated into a module **`state`**. This is done by first defining a type that encapsulates all class instance variables e.g. `S`, where all objects are represented with an object id rather than an object reference. Secondly, a type `ID` for the object instances of the class itself is defined. The module state shall then comprise a mapping from instance id to state: **`map`** `ID` **`to`** `S`, where `S` could be a record. Finally, define an operation `newId : () ==> ID`, which creates a unique id and a new instance record of type `S` and add it to the instance map, while returning the id. In this way, duplication of `ID`s is avoided.

Once the transformation is completed, all modules now reference state "objects" by an `ID`, rather than having a direct object reference, which is the case in an OO setting. Rather than invoking operations directly on the object, modules now need to invoke operations on the "objects" module, passing the `ID` along.

3.5 Guideline 5: Inheritance

Inheritance in a VDM++ setting includes a number of features, such as inheriting instance variables and operations, overriding operations, and extending a class with new instance variables and operations. All of which are features that are not directly supported in VDM-SL. Possibly, a complex transformation might be able to support all inheritance features, but the resulting VDM-SL model will be very complex and not easily understood or maintained. Our general guideline is therefore to avoid constructs that mimic inheritance in VDM-SL if possible. However, if only a few features are used for a specific purpose, simpler transformations can be used, with reasonable results. Specifically, we present two transformations related to inheritance, which have been used in the case study.

Strategy design pattern: A strategy design pattern consist of two types of classes, a strategy interface class and concrete strategy classes [5]. In an OO setting, a strategy pattern will be used, by having an object reference defined by the interface class and invoking operations on the object. The strategy can be changed easily, by replacing the object reference with another object reference that implements the same interface. In VDM++ the strategy interface class would be implemented as a base class, and the concrete strategies would be subclasses hereof. This is not possible in VDM-SL, but a strategy design pattern can be transformed to VDM-SL using union types and cases expressions, as described in *Transformation rule 2* and the accompanying example in Figure 5.

Transformation rule 2: Strategy patterns map to a union type and cases expressions Each concrete strategy class is transformed to VDM-SL, following all necessary previously defined guidelines. The strategy interface class must define a union type `Type` with a type for each concrete strategy module. A parameter of type `Type` must be added to each operation defined in the interface class. Additionally, a **`cases`** expression on the type parameter must be added, with an entry for each concrete strategy type, which delegates the call to the concrete strategy module.

```
class X

operations
opX : nat ==> nat
opX (x) ==
  is subclass responsibility;
end X
```

```
class A is subclass of X

operations
opX : nat ==> nat
opX (x) == return x + 1;
...
end A
```

(a) VDM++ classes

```
module X

types
Type = <A> | <B>;

operations
opX : Type * nat ==> nat
opX (t, x) ==
  cases t:
    <A> -> return A`opX(x),
    <B> -> return B`opX(x),
    ...
    others -> exit "Unknown Type"
  end;
end X
```

```
module A

operations
opX : nat ==> nat
opX (x) == return x + 1;
end A



module B

operations
opX : nat ==> nat
opX (x) == return x * 2;
end B
```

(b) VDM-SL modules

Fig. 5: Transformation example: VDM++ classes translated to VDM-SL using guideline 5: *Inheritance and strategy pattern.*

Basic inheritance: *Transformation rule 3* along with the example in Figure 6 describes how some of the basic inheritance features can be transformed in a simple manner, if type information is available whenever operations are invoked on the instances. In an OO setting this information is known through the object reference, but in VDM-SL it must be handled explicitly. Embedding this information in the inheritance modules in VDM-SL greatly complicates the model, which is not desirable. Therefore, we suggest this reduced transformation, which can be used to extend a module with a relatively small effort. Specifically, this reduced transformation will mimic the following inheritance features: Extending a class with more instance variable and operations, operation overriding, and allow calls to a super class.

Transformation rule 3: Basic inheritance used for extendibility/reusability In each module define a type `S` that encapsulates all the instance variables of that class and its superclass. Additionally, in the base module, add a union type `S_UNION` that holds the `S` type from all the modules and add an instance map from a type `ID` to `S_UNION`, similar to Guideline 4. Note that all "object instances" of all the subclasses will be kept as state in the base module. Next, add getters and setters for state in the base module, such that all sub-modules can access the necessary state. Finally, add a `newId` operation as described in Guideline 4.

4 Structural changes to the VDM models

4.1 Existing VDM++ model

The VDM++ model supports both off-line simulation and real-time guidance of harvest operations, but the model presented here is simplified to ease the understanding and only includes the core functionality. As VDM++ is an OO language, the model contains common OO constructs, such as design patterns including the strategy and the template pattern [5]. This also means that many of the individual instances of classes have state information about themselves so references to these are frequently passed around. Figure 7 shows a simplified class diagram of the model, where `Harvi` is the top-level class of the control algorithm. The strategy pattern is used both for `UnloadStrategy` and `TrackSeqStrategy`, whereas the template pattern is used for the `Resource` class and its subclasses.

The core of the model is single-threaded, but the integration with the P/S framework introduces more threads and asynchronous callbacks. In the presence of concurrency this means that permission predicates on certain instance variables and operations are included in the model, and it seems that this can have a negative impact on the performance, since every time an operation is called the permission predicates must be analysed.

4.2 VDM-SL model

Given the transformations defined in Section 3, the VDM++ model was transformed to VDM-SL. In addition, the VDM-SL model was made more general in the sense that

```
class A

instance variables
x : nat;

operations
opX : () ==> nat
opX () == return x + 1;
end A
```

```
class B is subclass of A

instance variables
y : real;

operations
opY : () ==> real
opY () == return y + 1.5;
end B
```

(a) VDM++ classes

```
module A

types
ID = token;
S :: x : nat;
S_UNION = S | B`S;
InstanceMap = map ID to S_UNION;

state AST of
  a_m : InstanceMap
  init s ==  s = mk_AST({|->})
end

operations
opX : ID ==> nat
opX (id) ==
  return a_m(id).x + 1;

getState : ID ==> S_UNION
getState (id) ==
  return a_m(id);
end A
```

```
module B

types
S :: x : nat
     y : real;

operations

opY : A`ID ==> real
opY (id) ==
  return A`getState(id).y + 1.5;
end B

...
```

(b) VDM-SL modules

Fig. 6: Transformation example: VDM++ classes translated to VDM-SL using guideline 5: *Basic inheritance.*

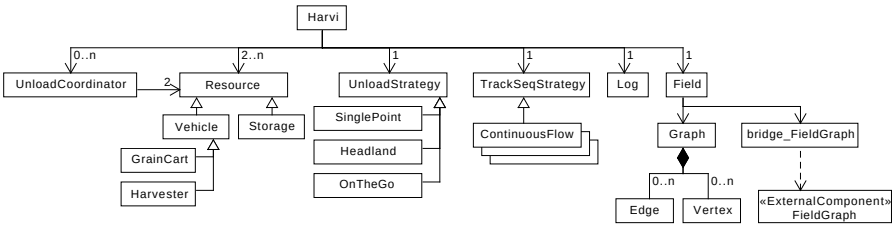


Fig. 7: Simplified class diagram of VDM++ model.

the VDM++ model was only able to cope with one plan, whereas the VDM-SL model has been prepared to be able co cope with multiple plans. A simplified overview of the most important modules in the VDM-SL model can be found at Figure 8. Note how the level of plans simply is added as a layer above the other modules. The `GrainHarvest` module is similar to the `Harvi` class from the VDM++ model. The four modules below that, all include state information organised as mappings from identifiers to data about them. In this way the state information is centered at specific places and the `MQTT` module represent all the P/S communication with the centralised cloud service. This is realised in Java using the bridge technology explained in Section 2. This has its own thread of control but since this is outside the actual VDM model the model complexity is significantly reduced.

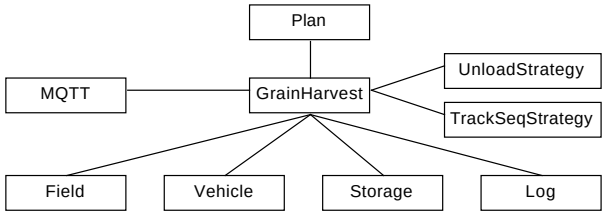


Fig. 8: Simplified overview of the VDM-SL modules.

5 Evaluation

Although most of the listings presented above indicate that the VDM-SL version is larger than the corresponsing VDM++ model, the transformation from VDM-RT to VDM-SL resulted in a new smaller model as shown in Table 1. However, the transformation process led to the discovery of cases where the responsibilities of modules were mixed. This discovery was made because of the explicit imports added in the process. The many dependencies between the modules is likely an artefact of the initial

distributed system where each vehicle had more control over its own behaviour oppose to the current approach where a more "functional" approach is taken. The new model aims to provide an operation which can perform all required computation to consume events received from the vehicles and in turn produce new or updated routes. All communication have been removed from inside the planning operation and converted into a sequence of events that is then consumed as part of the plan generation. The ideal function would have looked like illustrated in Listing 1.1, but due to the caching of the large graphs, used to represent the field, it had to be modelled as an operation and thus keeping state in many modules.

	VDM++	VDM-SL
Lines Of Model	3701	3041

Table 1: Lines of model in VDM++ and VDM-SL implementations, excluding libraries and tests.

```
planRoutes :  FieldPartition *
              seq of Route *
              FieldProgress *
              seq of Event -> seq of Route
```

Listing 1.1: Ideal route planning function.

The model transformation was carried out manually partly using the transformation principles from Section 3 that in itself can be quite error prone due to human factors. To make this even worse the testing framework *VDMUnit* only provided support for OO based models and thus could not be used for the new SL model. To overcome this issue and provide some validation against the source model a new extension to *VDMUnit* was developed to mimic the unit test behaviour for SL models as described in [11]. The test validation did provide the required basis for comparison with the source model but also showed a limitation of a VDM module based approach where all modules have mutable state. The primary issue is that all operations are directly imported and since VDM does not have *operation values* there is no way to provide true module based testing of each module in isolation using stubs that respect the pre-, post-conditions of the imported operations.

To assess if the newly created VDM-SL model performs similarly to the VDM-RT model the execution time of the full test suite for both models are compared in Table 3 and for one of the test scenarios in Table 2[5]. It shows that some of the improvements done during the transformations likely had a positive impact on the performance. During the transformation multiple places in the model were identified that constructed

[5] All experiments were performed on a server hosting a 64 bit VM configured with 6 x Intel Core Processor @ 2.0 GHz and 15 GB RAM.

route sequences in a way that grew exponentially in time in relation to the route length. One example was looping over a route to check relevant sequence elements, by using **`hd`** and **`tl`** expressions, rather than an index. By using the **`tl`** expression, the route was internally cloned for every loop iteration, causing poor performance both in the interpreter and in the generated code.

The comparison in Table 2 clearly shows that the original model had scalability issues.[6] As a result it was not possible to determine the full execution time of the full test suite for the interpreted model as shown in Table 3, where the experiment was turned off after 7 days execution running at 100% CPU load.

	VDM++	VDM-SL	Difference
Interpreted	2246.65 s	358.16 s	-84%
Code generated	40.36 s	19.46 s	-52%

Table 2: Performance comparison between VDM++ and VDM-SL implementations for one big scenario test.

	VDM++	VDM-SL	Difference
Interpreted	> 7 days	150 min	> -98%
Code generated	91 min	7 min	-92%

Table 3: Performance comparison between VDM++ and VDM-SL implementations for all tests.

It should be noted that the generated code does not perform any pre-, post-condition, or invariant checks. This is one of the primary reasons to why the test framework was upgraded to support VDM-SL. The usage of the *VDMUnit* for the OO model have revealed especially pre-condition errors that were not revealed by the tests at the generated code level.

6 Concluding remarks

In the new VDM-SL model the main focus is on the calculation of routes for the different vehicles. Compared to the VDM++ model there is also a cleaner separation between the planning aspects and the event-based communication carried out via the Publish/-Subscribe server connection to the cloud. Actually it was a positive surprise for us that transforming the model from an imperative style to a more functional style had the side effect of increasing the performance. However, it also turned out that the transition

[6] The difference is calculated as: $\text{Difference} = \frac{a-b}{b} * 100\%$, where $a =$ VDM-SL performance and $b =$ VDM++ performance

rules suggested were not sufficient to take care of all refactorings. In particular in relation to inheritance there is a tendency that there is an overhead of what needs to be written in a VDM-SL setting. Therefore, we do not see a possibility for automating the transformation from a VDM++ model to VDM-SL.

Initial work targeting the use of this VDM-SL model in a combined commercial and research context have started and this looks promising, but no conclusion can be drawn from this yet. Finally, it is worth noting that it is possible to use traditional unit testing in a VDM-SL model with the use of a newly extended VDMUnit testing framework.

Future Plans Transitions from VDM++ to VDM-SL are not something one would embed in a methodology like the one proposed by [7], as it is not a desired transition. Ideally one would not end up in a situation where such a transformation is necessary. However, requirement changes are a common phenomena and might lead one into such a situation. We see the transformation guidelines proposed in this papers as a help or inspiration for others who might face the same needs as we did.

Theoretically it might be possible to include all features of VDM++ into similar guidelines, but it is our belief that this would be at the expense of model complexity and readability. Therefore we do not propose to go this way, and we do not suggest automating the transformation either, as especially the inheritance transformations might be somewhat use case specific.

Acknowledgments We would like to thank the Danish Innovation Foundation for funding this project and to our colleagues that have worked with us in this project. We would also like to pay special thanks to the anonymous reviewers who have helped improving the quality of the paper.

References

1. Couto, L.D., Tran-Jørgensen, P.W.V., Larsen, P.G.: Enabling continuous integration in a formal methods setting. In: Submitted for publication (2018)
2. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc.
3. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
4. 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/`
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
6. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)

7. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
8. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: The VDM-10 Language Manual. Tech. Rep. TR-2010-06, The Overture Open Source Initiative (April 2010)
9. 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), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
10. Tran-Jørgensen, P.W.V.: Enhancing System Realisation in Formal Model Development. Ph.D. thesis, Aarhus University (Sep 2016)
11. Tran-Jørgensen, P.W.V., Nilsson, R.S., Lausdahl, K.: Enhancing Testing of VDM-SL models. In: Proceedings of the 16th Overture Workshop (July 2018)
12. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

Integrating VDM-SL into the continuous delivery pipelines of cloud-based software

Simon Fraser

Anaplan, York, UK
simon.fraser@anaplan.com
http://www.anaplan.com

Abstract. The cloud is quickly becoming the principle means by which software is delivered into the hands of users. This has not only changed the shipping mechanism, but the whole process by which software is developed. The application of lean manufacturing principles to software engineering, and the growth of continuous integration and delivery, have contributed to the end-to-end automation of the development lifecycle. Gone are the days of quarterly releases of monolithic systems; the cloud-based, software as a service is formed of hundred or even thousands of microservices with new versions available to the end user on a daily basis. If formal methods are to be relevant in the world of cloud computing, we must be able to apply the same principles; enabling easy componentization of specifications and the integration of the processes around those specifications into the fully mechanized process. In this paper we present tools that enable VDM-SL specifications to be constructed, tested and documented in the same way as their implementation through the use of a VDM Gradle plugin. By taking advantage of existing binary repository systems we will show that known dependency resolution instruments can be used to facilitate the breakdown of specifications and enable the easy re-use of foundational components. We also suggest that the deployment of those components to central repositories could reduce the learning curve of formal methods and concentrate efforts on the innovative. Furthermore, we propose a number of additional tools and integrations that we believe could increase the use of VDM-SL in the development of cloud software.

Keywords: Continuous delivery · Software as a Service · Vienna Development Method (VDM) · Overture

1 Introduction

The ubiquity of fast internet access has led to a move from shrink-wrapped, on-premise products to the adoption of the software as a service (SaaS) model [2,10]. Both start-ups and large, established enterprises are embracing a cloud delivery model, and are benefiting from this in the form of increased revenues [16].

Shipping SaaS is significantly different from other forms of software as it is possible to completely automate the delivery process by following the principles of continuous integration (CI) and delivery (CD) [3,4] and thus significantly reduce the cost of releasing. This leads to more frequent releases where each release contains smaller – and thus less risky – changes to the system. This shift has coincided with the growth of both

lean software development [15] and the microservice architecture [13], which not only encourage the ruthless mechanisation of all aspects of the development lifecycle, but promote the breakdown of the system into compact, independently-releasable components.

The move to smaller, encapsulated components should make the use of formal methods more inviting, yet in our experience companies producing SaaS are unlikely to use them. One reason for this is a lack of suitable tooling. Integrated development environments (IDE) such as Overture [5], Rodin [1] and CZT [8] have significantly improved the ability of individuals to engage with their respective methodologies, but to get any traction within a company delivering SaaS, it is essential that all aspects of the process can be automated and integrated into existing CD pipelines.

A very basic CD pipeline incorporating a formal specification is presented in fig. 1. Whenever a change is made to either the service's specification or its implementation, all verification and validations actions are executed on a centralized CI/CD system [11,18] and if all succeed, the new version of the service is immediately deployed to the production environment. There should be no manual process required, so all steps must be automatable and we must have complete confidence in our process to ensure the correctness of the service.

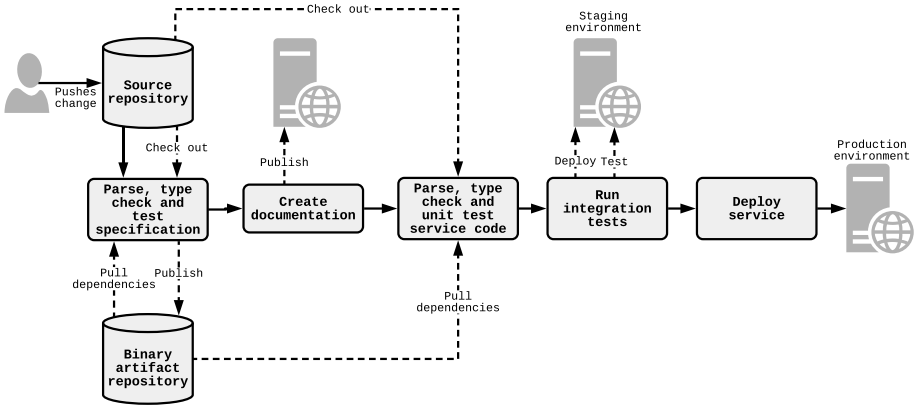


Fig. 1. A basic CD pipeline

This paper introduces a VDM plugin to the Gradle [12] build-system [9] that uses the core components of Overture to integrate VDM-SL into a CD pipeline. Gradle is widely used and plugins are available for many development languages including Java, Scala and C++. These plugins are used to automate commonly performed tasks including building, verifying and deploying code; the VDM plugin automates similar tasks on specifications for VDM-SL users. All configuration, including which plugins to use, is declared in a *build.gradle* file that is usually found in a project's root directory. Using Gradle is simply a matter of invoking the tool in the appropriate directory with the list of tasks to execute, for example: `gradle build`. A user can do this locally from the

command line or an IDE plugin; CI/CD systems are designed to execute Gradle tasks. As all configuration is defined in a file there is no dynamic set-up and a CD pipeline is thus defined as a sequence of tasks to execute on one or more projects.

The eventual goal of the VDM Gradle plugin would be to automatically perform every task that can be performed manually within Overture and thus incorporate all those tasks into a CD pipeline. In this paper, we describe what we believe is the minimum viable feature set required to enable the use of VDM within a pipeline; covering the automation of essential tasks and also the breakdown of specifications into components that can be published to and consumed from a binary artifact repository. We will outline some opportunities for immediate extension to the plugin's capabilities, but the flexibility of Gradle's orchestration mechanisms means that there are few constraints on what can be achieved. For instance, it would be possible to integrate other tools, such as theorem provers, to mechanically perform tasks that are not currently possible in Overture.

Section 2 introduces the plugin and shows how it can be used to parse and type check a set of modules and section 3 describes how that plugin is extended to automate specification testing. Section 4 proposes the use of central binary repositories for sharing specifications and explains how this is facilitated by the plugin. The tool also enables a user to integrate a natural language specification with a formal one and this is described in section 5. In section 6, we discuss a number of additional areas in which tooling could be of use and we remark upon the work thus far in section 7.

2 An automated build

Overture provides an IDE for VDM, however it does not give us the one-step 'compile, test and assemble' process that is required in the modern software engineering pipeline. Engineers expect to be able to invoke a build from a single command that can also be used by a centralized CI/CD system.

Tools such as Make have existed for many years and are capable of performing the basic tasks, but they lack the sophistication and easy integration of more modern tools. Maven and Gradle are tools that began as ways to build Java code, but are now considered general purpose, feature-rich and extendable build toolsets that are designed to form part of a CD pipeline. Both Maven and Gradle are widely used in the creation of SaaS. We chose to plug into Gradle as its task-based approach offers more flexibility than Maven's lifecycle. Given the relative similarities between modern build systems, it would be trivial to later generalize the concepts of this plugin to provide a similar plugin for Maven (and others).

Gradle defines tasks than can be standalone or which can define dependencies on other tasks. When building, an engineer will specify the task that they wish to run and the system will run that task and all of its dependencies in an order that satisfies all relationships. Gradle's base plugin defines three tasks: *clean*, *build* and *assemble*. Running the *clean* task will remove all files generated by any previous runs, the *build* task will typically perform compilation, type checking and testing (see section 3), and the *assemble* task will generate any output files required.

Our VDM plugin utilizes the base plugin and adds hooks to its tasks. Specifically, we define *parse*, *typeCheck* and *package* tasks, such that *package* depends upon *typeCheck* which depends upon *parse*. Additionally, we declare that *build* depends upon *typeCheck* and *assemble* depends upon *package*. Thus a user invoking *assemble* will expect all tasks to be run (barring build issues).

The *parse* task will find files in a specified directory (*src/main/vdm* by default) which have an extension derived from the configurable dialect (VDM-SL by default). The files are then parsed using VDMJ from the Overture core, such that the resultant AST is serialised to the Gradle build directory. This binary serialization means that we do not need to re-parse from scratch in subsequent tasks. The final task, *package* create a zip file from the original specification files parsed and places it in the build directory. Plain text files are packaged rather than binaries to improve readability downstream — see section 4.2. If at any point a task cannot be completed due to, for instance, a parse or type checking error the build fails and subsequent tasks will not run.

Even a Gradle plugin supporting this very limited set of tasks provides a great deal of value as we can ensure that our specification is correctly typed on every commit by creating a job in any Gradle-aware CI/CD tool. This immediately gives us the benefits that such tools provide; for example: a history of builds, notification of failures, build dashboards/alarms and allow us to be more confident that our head of specification is always in a 'correct' state.

3 Acceptance testing a specification

When creating SaaS, a test-driven approach to development is frequently used alongside a CI/CD system to ensure that all code is tested and that all tests pass after every commit. We believe that the same approach should be taken to the creation and maintenance of specifications; this approach to 'unit-testing' formal specifications has been explored elsewhere [22], so will not justify it further here.

The plugin is thus extended to define a new task *test*. We have followed the standard Gradle pattern of holding 'main' and 'test' files in separate directories and processed by separate tasks, thus we also introduce *parseTests* and *typeCheckTests* tasks at this point. *parseTests* will locate and parse files in a specified directory (*src/test/vdm* by default) in the context of the parsed 'main' specification (introducing a dependency on *parse*) producing a second binary output in the build directory. Similarly, *typeCheckTests* has a dependency on *typeCheck* and *parseTests* but we cannot avoid type checking all loaded binary specification files. The *test* task depends upon *typeCheckTests* and can be configured to execute different testing strategies. There are existing schemes that can be used to apply acceptance tests to VDM specifications, but in this instance our primary focus was automatability so we devised the simple strategy described below.

This strategy identifies all modules originating in the test directory that have a name beginning with 'Test'. Each of these modules is considered a test suite. In each module, the list of operations is examined and those that have a name beginning with 'Test' are considered test cases. We expect the post condition of each test operation to hold the test's assertions about the result. Each test case is then evaluated. If the evaluation completes without error the test is considered to have passed. If the evaluation leads to

a post condition error then the test is considered to have failed. If the evaluation leads to another type of error – such as precondition or invariant failure – then the test is considered to have errors.

For example, in the following block we have one test suite: *TestArithmetic* and three test cases *TestAdd*, *TestMultiply* and *TestDivide*. *CheckSubtract* is not a test case as its name does not start with 'Test'. *TestAdd* will evaluate without error and is recorded as a pass. *TestMultiply* will be recorded as a failure as the post-condition will evaluate to false. *TestDivide* will be recorded as an error as we can assume that there will be a precondition that prevents division by zero.

```
module TestArithmetic
imports from Arithmetic
definitions
operations

  TestAdd:() ==> real
  TestAdd() == Arithmetic`Add(3, 4)
  post RESULT = 3;

  TestMultiply:() ==> real
  TestMultiply() == Arithmetic`Multiply(3, 4)
  post RESULT = 14;

  TestDivide:() ==> real
  TestDivide() == Arithmetic`Divide(3, 0)
  post RESULT = 0;

  CheckSubtract:() ==> real
  CheckSubtract() == Arithmetic`Add(6, 4)
  post RESULT = 2;

end TestArithmetic
```

In order that test results can be reported in CI/CD tools, the task records the results in JUnit format (the most common test result, interchange format), producing one file per test suite in the build directory. As well as a simple result, any failure messages are listed and the evaluation duration recorded. This integration ensures that the build – and any subsequent steps in the CD pipeline – will fail when a change is made that causes a test to fail. A CI/CD system can notify all stakeholders of success or failure and it also provides time-based reporting. This not only allows us to visualize trends over time, but will highlight commonly failing tests which can often indicate poorly written specifications or tests.

4 Dependency management

Although the primary purpose of a build system is to provide a simple mechanism to reliably build and test artifacts, much of what we have discussed to this point could be reproduced using Make or Ant. The true value of tools like Gradle is their ability to go beyond this basic process and enable large systems to be broken down into small components and then declaratively pieced together through the mechanism of dependency management. Not only has this encouraged the decoupling of concerns and encapsulation within a single system, but it has allowed small, tightly-focused components to be shared globally. We would argue that Maven's primary contribution to the art is not its toolchain, but the Maven Central repository.

4.1 Componentizing a specification

When producing SaaS the tendency has been away from the monolithic and towards microservice architectures. With this approach we still have a whole system, but it is deliberately broken down, with hard boundaries introduced, so that one team can be wholly responsible for all aspects of a small subset of components. There is likely still common code, but this should be managed through the creation of library components which are owned by one team and consumed by others (often using an internal open-source model). In an environment like this, we want to break down our 'system' specification in the same way.

VDM-SL provides a module mechanism, but its level of granularity is not what is needed here; we would expect a component to be formed of a number of 'main' modules and a number of 'test' modules that correspond to a particular block of system functionality. Each component should have its own Gradle build and typically its own repository in a VCS. It should be possible to automatically publish the components for use by others and to specify the use of those components in downstream projects.

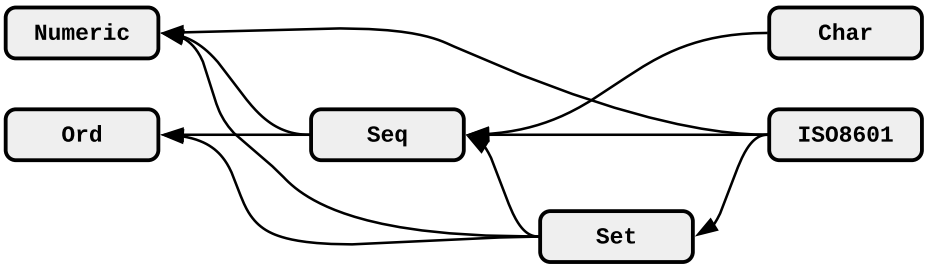


Fig. 2. Dependencies of components extracted from ISO-8601 specification

Consider, as a simple example, the ISO-8601 specification that is distributed with Overture. It contains not only the specification of the standard itself, but a number of other modules that provide utility types, values and functions in a number of areas. We

are certain that those utilities modules benefit specifications other than that for this particular standard as we have found them to be useful in many other instances. In this example, each of these modules would benefit from being split into its own component so that its specific behaviour could be packaged and shared. With some minor refactoring to prevent circular dependencies, we could create components with a dependency tree illustrated in Fig. 2. Doing this here would enable downstream specifiers to depend, for example, upon the *Seq* component without having to make any reference to the ISO-8601 specification.

4.2 Sharing via a binary repository

Gradle already has mechanisms for publishing to and consuming components from binary repositories with a number of formats including Maven and Ivy. For our implementation we have chosen initially to support Maven formatted repositories given the relative importance of Maven Central. Extending the plugin to support other repository types should not prevent a significant challenge and existing implementation choices will not prevent this.

Maven repositories assign every component group/artifact/version (GAV) co-ordinates. The group is typically the reversed domain of the owning organisation, the artifact is the component's name and the version has a value typically assigned by the maintainer using the semantic versioning system[1]. These values are normally declared in the Gradle configuration file (although the name defaults to that of the containing directory).

The first step to sharing is to publish a component without dependencies – such as the *Ord* module in Fig. 2 – to the binary repository. We do not assume that all users of the VDM Gradle plugin will wish to publish their artifacts and there will always be some specific configuration required to indicate the specific binary repository to use, so our plugin will only try to publish when the *maven-publish* plugin is applied to the build. When this has been applied a 'vdm' publication will be created automatically and a hook added to the *publish* tasks (defined by the *maven-publish* plugin) to deploy the zip file of specifications generated by the *assemble* task. The snippet below demonstrates all the salient aspects of the *build.gradle* file required to build, test and publish the *Ord* component.

```
group = 'org.overture'
version = '1.0.0'
apply plugin: 'vdm'
apply plugin: 'maven-publish'
```

The next step is to consume the published component; if we take for example *Seq*, we need to add dependencies on *Ord* and *Numeric* components. Here the plugin hooks directly into Gradle's existing dependency resolution mechanism – which is described in depth in chapter 5 of [12] – so there is little custom work required. We add a new 'vdm' configuration to distinguish dependencies from those required by Java or other

[1] http://semver.org

languages (making it possible to build specification and code in the same component — see section 6.3) and these are processed by a new *dependencyUnpack* task that extracts the specification from the zip files into a GAV based directory structure. A dependency on this task is added to *parse* and the behaviour of this task is altered slightly so that it now parses the dependent specifications as well as those introduced in the current component. An Overture user can view the dependent specifications within their project and that tool is able to parse and type check using these dependencies seamlessly. From the user's perspective it is trivial to add dependencies, as the following example for the *Seq* component demonstrates.

```
group = 'org.overture'
version = '1.0.0'
apply plugin: 'vdm'
apply plugin: 'maven-publish'
dependencies {
  vdm group: 'org.overture', name: 'Numeric', version: '1.0.0'
  vdm group: 'org.overture', name: 'Ord', version: '1.0.0'
}
```

4.3 Transitive dependency management

The previous section does not provide the whole story. This naïve approach has not taken into account the intricacies of transitive dependencies. For instance, the creator of the *ISO-8601* module knows that they depend upon *Numeric*, *Set* and *Seq*, but they also have an implicit dependency on *Ord* as *Seq* depends upon it. Fortunately, it is relatively simple to ensure that the *dependencyUnpack* task processes all dependencies – whether direct or not, but how are downstream components able to determine what this component's dependencies are? The publication mechanism described thus far has not considered this.

Again, the process required depends upon the repository type; as we are targeting a Maven repository we need to produce a project object model (POM) on publication that not only gives details of the component we are publishing, but also of its dependencies. A basic POM is produced by the *maven-publish* plugin automatically, but this just gives the details of the artifact being published. A new task *addVdmDependenciesToPom* is created to add our dependencies to the POM in the publish phase of the build[2]. This task depends upon the *maven-publish* plugin's POM generation task and a dependency is added to any publish tasks to ensure it is performed before the deploy of the component.

When dealing with a large complicated system with many components which we would expect to be typical in a microservice architecture, this form of automated dependency management quickly becomes essential. However, while automated transitive dependency management removes a large burden from the user, there is one scenario in which the unwary can be caught out. Consider the situation where version 1.0.0 of *Seq*

[2] Gradle's incubating Software Model was not considered to be mature enough for use at this time.

has declared a dependency on version 1.0.0 of *Ord*, but version 2.0.0 of *Seq* uses version 2.0.0 of *Ord*. If *Set* declares dependencies upon version 2.0.0 of *Seq* and version 1.0.0 of *Ord*, it will effectively depend upon both versions 1.0.0 and 2.0.0 of *Ord*. Gradle's default resolution mechanism is to take the newest version, but in our plugin we have enforced a strict no-conflict strategy. That is, if the same component is acquired more than once, it must have exactly the same version or the build will be failed. This forces the user to make an explicit choice of the version they require.

The user can resolve this failure by either updating dependencies so that all versions agree or by explicitly excluding specific transitive dependencies as in the example below. Note that, using version 1.0.0 of *Ord* with version 2.0.0 of *Seq* may have unexpected consequences, including but not limited to the inability to parse and type check *Seq* when building *Set*.

```
group = 'org.overture'
version = '1.0.0'
apply plugin: 'vdm'
apply plugin: 'maven-publish'
dependencies {
  vdm (group: 'org.overture', name: 'Seq', version: '2.0.0') {
    exclude group: 'org.overture', module: 'Ord'
  }
  vdm group: 'org.overture', name: 'Ord', version: '1.0.0'
}
```

Conventionally the number of declared dependencies is minimized to reduce this form of conflict. In the previous example, removing the direct dependency on *Ord* would produce the same result as updating the direct dependency to version 2.0.0.

Given this convention, the configuration for *ISO-8601* would contain only a single dependency – *Set* – as all other required dependencies would be acquired transitively.

4.4 Using central repositories for common components

We would argue that Java does not owe its success to the language, nor even the JVM, but to the sheer breadth of trusted open-source library components. For example, custom code is rarely written to implement a data structure; in almost every case there is at least one implementation that has been battle-hardened in countless production systems that we are free to employ seamlessly. Java engineers are left to worry about the core aspects of their system rather than how to re-invent the foundations. However, it was only when repositories like Maven Central and the accompanying tools made those components easy to find and use that the consumption of those really components took off.

There is no doubt that we have found the ability to componentize specifications useful, but when defining some components it has felt that we have been re-inventing the wheel. We have already alluded to the fact that we have found some of the components in the ISO-8601 specification useful beyond their stated purpose and additionally when

we write specifications for trees, graphs, multisets and multimaps we are sure that we cannot be the first to do so. Finding and using those existing specifications is possible, but not easy, and even when we do find them it is unlikely that we are able to place the same amount of trust in them as we would like.

We suggest that if VDM 'library' specifications were prevalent to the same extent in central repositories, then the uptake of VDM would be considerably greater. Specifiers would have the same ability to put together well-trusted, verified building blocks to form the foundation of their systems and would be able to concentrate on the aspects unique to their systems. Without tool support we lack a consistent way to publish to and consume from central repositories. We hope that the introduction of a Gradle plugin such that we describe in this paper would encourage specifiers to begin publishing library components of specification that others could use and that by doing so we could facilitate wider use of VDM.

5 Producing a natural language specification

Although Overture is capable of processing specifications that are embedded within a LaTeX document, it does not drive the user towards this approach, with the UI directing users to create modules and classes rather than documents. Whilst this choice enables to better tool support, it leads to a separation of the formal specification and the informal requirements (whereas Z encourages a single document, but suffers from poor tool support[3]). Additionally, whilst LaTeX can be a fantastic tool for document preparation, it does not typically form part of the standard toolkit of a cloud software engineer; nor are postscript documents or PDFs the standard instruments for sharing documents within a cloud-based company.

Agile methodologies are typically used for SaaS and end-to-end team ownership of components is encouraged, therefore all members of a team will need access to the specification, but they will use it for different purposes. The customer proxy will need to verify that the specification correctly captures their acceptance criteria, the engineers will need the specification to guide their implementation, the QA will need to verify that the implementation matches the specification and writers will use the specification as a starting point for their user documentation. Each of these individuals will have different levels of understanding of the formal aspects and all will need informal instruments to record their interpretations, even if only with which to validate a shared understanding with other team members. It is essential, therefore, that there is some mechanism for annotating the formal with the informal. Furthermore, all the members of the team need consistent access to the 'latest' version of the specification; there should be no confusion over which is the latest version of a document, nor should it require the use of any particular tool. At a cloud-based company this inevitably means that the specification itself should be in the cloud, either as a website on an intranet or in a collaboration tool, such as a wiki; an HTML representation would enable both.

[3] Note that, the sheer expressivity of Z rather than the LaTeX format is the primary reason for its lack of mechanisation.

5.1 An HTML compatible VDM pretty printer

Consider first, the necessity to view the formal specification in a browser; simply checking out raw VDM from a VCS repository and displaying it as text is unlikely to be the most useful practice. Thus a generic VDM pretty printer was created that was capable of rendering an ASCII specification into HTML and other formats. This was done by implementing Overture's 'Question/Answer' interface which facilitates interaction with the specification's AST; as such, the pretty printer could be applied to any node from a module to a simple expression.

Although we specifically required a mechanism to render the specification to HTML, we kept the implementation generic. The pretty printer was designed to accept different render strategies depending on the user's requirements. The render strategies that we have implemented presently are:

- A plain ASCII strategy — intended to format a specification in place (in the future we would like to integrate this into Overture in the same way as Eclipse's standard **Source** > **Format**).
- A mathematical unicode text strategy — uses mathematical symbols in favour of ASCII keywords where possible to produce a UTF-8 text representation.
- A mathematical unicode HTML strategy — as the previous, but produces a UTF-8 HTML representation (for example using heading, bold and italic tags where appropriate).

As well as defining how to render the tokens found in the specification, the strategy also determines how vertical and horizontal whitespace is inserted into the rendering; for example: ` ` rather than a space. Additionally, the pretty printer will insert navigational markers into the rendering and the stategy describes how they are rendered; for example an empty *div* with the identifier of the object of interest in the HTML strategy. Fig. 3 illustrates the HTML rendering of the pretty printer when applied to the classic Alarm example.

We have not yet implemented a mathematical LaTeX rendering strategy, but it should be relatively straightforward to do so if needed.

The pretty printer is not entirely naïve, it will for example keep track of renamed imports in a module and only use the fully-qualified name of an imported object where it is necessary to do so. There are also some configuration options; for example it is possible to specify a length for a node list over which each entry is rendered on a new line. We have found that the renderings produced provide a more readable version of the specification from which we can easily include snippets in wiki or emails as well as producing full HTML documents. There is additional work that could improve the output, for instance the current approach to precedence is simplistic and our rendering relies too much on the use of parentheses, but we feel that even in its current state it adds a useful addition to the practitioner's toolbox.

5.2 Integrating informal Markdown specifications

The usual language for producing documentation in cloud companies is Markdown [7], it is the markup language of choice for many SaaS solutions including GitHub and

functions

```
ChangeExpert: (Plant × Expert × Expert × Period → Plant)
ChangeExpert(mk_Plant(plan, alarms), ex1, ex2, peri) △
 mk_Plant((plan † {peri ↦ ((plan(peri) \ {ex1}) ∪ {ex2})}), alarms);

NumberOfExperts: (Period × Plant → ℕ)
NumberOfExperts(peri, plant) △
 (card (plant.schedule)(peri))
pre (peri ∈ (dom (plant.schedule)));

ExpertIsOnDuty: (Expert × Plant → Period-set)
ExpertIsOnDuty(ex, mk_Plant(sch, -)) △
 {peri | peri ∈ (dom sch) • (ex ∈ sch(peri))};

ExpertToPage(a: Alarm, peri: Period, plant: Plant) r: Expert
pre ((peri ∈ (dom (plant.schedule))) ∧ (a ∈ (plant.alarms)))
post ((r ∈ (plant.schedule)(peri)) ∧ ((a.quali) ∈ (r.quali)));

QualificationOK: (Expert-set × Qualification → 𝔹)
QualificationOK(exs, reqquali) △
 (∃ ex ∈ exs • (reqquali ∈ (ex.quali)))
```

Fig. 3. HTML rendering of the functions in the Alarm specification

Confluence. Like LaTeX, Markdown enables text to be annotated with rendering instructions, unlike LaTeX the syntax is simple and intuitive, and most IDEs provide real-time previews of the final document. Markdown is easily transformed to HTML and can be treated like code in the development process.

The approach we took was that the Markdown documents would drive the content of the rendered specification document. That is, the user would write their informal text and then place references using custom Markdown directives to include or refer to parts of a VDM-SL specification. The VDM Gradle plugin was extended to add a *docGen* task that depended upon *typeCheck*. When run, this task finds all Markdown files in a specified directory (*src/main/md* by default) and would then transform those files into HTML files placed within the project's build directory. Additionally, the task would use the pretty printer introduced previously to render all main modules into one subdirectory and all test modules into another. In order to create an integrated specification, the Markdown processor was extended to define new directives that would enable the writer to use VDM within their informal text. In our experience thus far, only a few directives have been required and they are summarized in Table 1.

The output of the process can be viewed in Overture by opening the generated HTML files, but a more useful next step would be to use the CI/CD tooling to automatically publish this documentation to a versioned location. We do not propose incor-

Table 1. Summary of VDM Markdown directives

Directive	Description	Example
{@link:*definition*}	Creates a link to the definition in the modules appendix	{@link:ISO8601‘subtract}
{@ref:*definition*}	Includes the pretty printed definition as a quoted element in the rendering	{@ref:ISO8601‘subtract}
{@mainModuleList}	Includes an unordered list of main modules in the rendering	{@mainModuleList}
{@testModuleList}	Includes an unordered list of test modules in the rendering	{@testModuleList}
{*expression*}	Parses and pretty prints any VDM expression and includes in the rendering	{forall d in set nat & d >= 0}

porating aspects of this into the plugin as existing tools already provide the means to achieve this.

6 Further requirements

We have made a start in the integration of tools supporting VDM-SL into a CD pipeline. However, there are additional aspects of the development process that must be tackled. In this section, we describe some of the most pressing.

6.1 Integration of coverage reports

Just as code coverage is an integral part of ensuring that all the paths through a piece of code have been tested, ensuring that all parts of a specification have a corresponding and testable system requirement is vital in validating that our specification matches our requirements. A plethora of code coverage tools exist [17], but there has been a convergence in the format of the files produced.

Overture has valuable support for showing coverage when running within the IDE, but we have not yet integrated that into our test framework. It is essential to do so, but for it to have genuine value we must translate the *.covtbl* files produced by Overture into a format understandable by CI/CD tools. The Cobertura XML schema is widely supported, and translation into this format would enabling visualisation of coverage over time and the capability to fail builds if coverage extent drops.

6.2 Automated test case generation

Beyond the basic acceptance testing of a specification there is a need to verify the consistency of our specification and that our implementation matches said specification.

One method of checking our specification is to use combinatorial testing [6]. Overture already provides an excellent mechanism to perform combinatorial testing through VDM traces; it would be trivial to execute these from the Gradle plugin, but thought

must be given to the selection of traces to run – executing every trace defined on every commit may prove unwieldy.

Test generation can also verify our implementation by ensuring that, within reasonable bounds, the specification and implementation give the same results when evaluating instructions. There are tools [14,19] that take different approaches to the generation and it is necessary to evaluate to what extent they could be integrated into a CD pipeline. There are a number of different forms of test case generation, including:

1. Generating a file containing a list of test steps with expected results after each step.
2. Generating code that executes test steps and asserts that the expected results are achieved.
3. Generating a file containing a list of test steps and code that can verify results achieved satisfy post-conditions at each step.

For a SaaS system the second is unlikely to be particularly useful as a microservice architecture can often promote the use of many programming languages and it would be difficult for a single tool to support the different paradigms of multiple languages. However, the first is useful to verify that the explicit acceptance tests give the correct results in the implementation and the third provides a mechanism for combinatorial testing of the code. In both these cases we would expect that the artifacts are produced with the Gradle plugin when building the specification, but are consumed independently by other means when building and testing the code.

6.3 Code generation

In some cases it may be advantageous to generate code directly from a specification. Overture provides mechanisms [20,21] for the automated generation of Java and C++. However, in a CD pipeline it is not sufficient to simply generate the code, but to build it, package it and deploy it to a binary repository. The VDM Gradle plugin should be extended to support different code generation strategies and integrated with Gradle's existing Java and C++ plugins to automatically produce and deploy the appropriate binary artifacts.

6.4 Integration with other IDEs

Eclipse is a tool in rapid decline – 64% of Java developers used Eclipse in 2012, but survey results [23] show a consistent contraction in every year surveys have been conducted since, with only 33% of respondents selecting it in the 2017 survey (twenty percentage points behind new market leader IntelliJ IDEA).

In the authors' experience Eclipse is unlikely to be used in the development of SaaS and it would certainly aid in the adoption of Overture if plugins were available for other IDEs.

7 Concluding remarks

Integration of VDM-SL into a fully automated CD pipeline is essential if VDM-SL is to be used in enterprises delivering SaaS. This paper has shown that it is possible to use an existing build system to build, test and share such specifications and that the use of these tools facilitates interaction with standard CI/CD infrastructure. The described VDM Gradle plugin not only enables these basic tasks, but integrates with Gradle's dependency resolution mechanism so that complex dependency structures can be maintained with little manual involvement and thus large systems can be componentized without undue overhead.

By using this plugin it is possible to not only share specifications within an enterprise, but globally using centralised binary repositories. We believe that this mechanism will make it easier to focus on the interesting aspects of specifications and ultimately facilitate the adoption of VDM-SL in the development of SaaS.

Tools for merging formal specifications with informal Markdown documents were also introduced in this paper alongside a mechanism for rendering the resultant document into HTML. The capability to publish integrated specifications has perhaps provided significant benefit to the authors as it has enabled all members of our team to actively engage with the specification despite a lack of previous exposure to formal methods.

We have also identified a number of additional tools that could further improve the uptake of VDM within cloud companies. However, the VDM Gradle plugin described herein enables the immediate integration of VDM-SL into a fully automated CD pipeline and will enable us, and others, to actively use formal methods in our cloud-oriented development process.

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International journal on software tools for technology transfer **12**(6), 447–466 (2010)
2. Dempsey, D., Kelliher, F.: Cloud Computing: The Emergence of the 5th Utility. In: Industry Trends in Cloud Computing, pp. 29–43. Springer (2018)
3. Duvall, P.M., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk. Pearson Education (2007)
4. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education (2010)
5. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture initiative integrating tools for VDM. ACM SIGSOFT Software Engineering Notes **35**(1), 1–6 (2010)
6. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial testing for vdm. In: Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on. pp. 278–285. IEEE (2010)
7. Leonard, S.: Guidance on Markdown: Design philosophies, stability strategies, and select registrations (RFC-7764) (2016)
8. Malik, P., Utting, M.: CZT: A framework for Z tools. In: International Conference of B and Z Users. pp. 65–84. Springer (2005)

9. Maudoux, G., Mens, K.: Correct, efficient, and tailored: The future of build systems. IEEE Software **35**(2), 32–37 (2018)
10. Mell, P., Grance, T., et al.: The NIST definition of cloud computing (2011)
11. Meyer, M.: Continuous integration and its tools. IEEE software **31**(3), 14–16 (2014)
12. Muschko, B.: Gradle in Action. Manning (2014)
13. Newman, S.: Building microservices: designing fine-grained systems. O'Reilly Media, Inc. (2015)
14. Oriat, C.: Jartege: A tool for random generation of unit tests for Java classes. In: Quality of Software Architectures and Software Quality, pp. 242–256. Springer (2005)
15. Poppendieck, M., Poppendieck, T.: Lean software development: an agile toolkit. Addison-Wesley (2003)
16. Price Waterhouse Coopers: Global 100 software leaders: Key players & market trends. New York: PWC CIL (2016)
17. Shahid, M., Ibrahim, S.: An evaluation of test coverage tools in software testing. In: 2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT. vol. 5 (2011)
18. Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access **5**, 3909–3943 (2017)
19. Tomoyuki Myojin, F.I.: Automated test procedure generation from formal specifications. In: 15th Overture Workshop on VDM (2017)
20. Tran-Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A code generation platform for VDM. In: 12th Overture Workshop on VDM (2015)
21. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. International Journal on Software Tools for Technology Transfer pp. 1–25 (2017)
22. Utting, M., Malik, P.: Unit testing of Z specifications. In: International Conference on Abstract State Machines, B and Z. pp. 309–322. Springer (2008)
23. ZeroTurnaround: RebelLabs developer productivity report. Tech. rep. (2017), `https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/`