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.

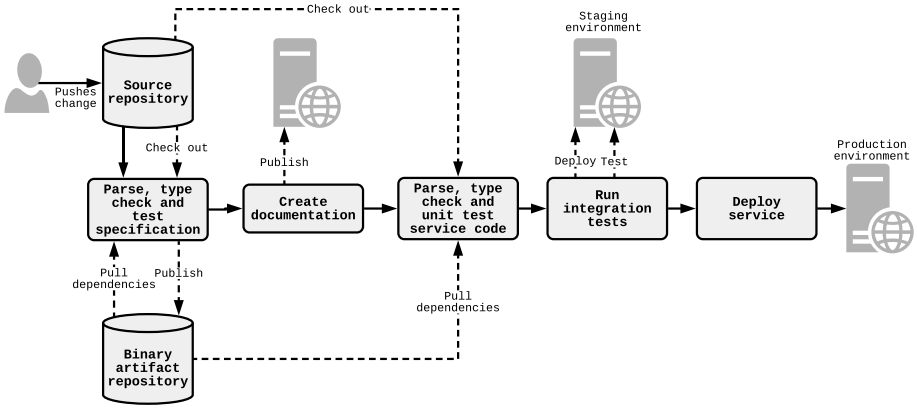

Fig. 1. A basic CD pipeline

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.

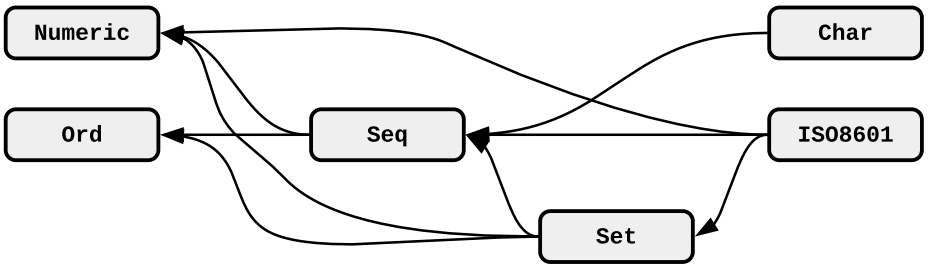

Fig. 2. Dependencies of components extracted from ISO-8601 specification

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

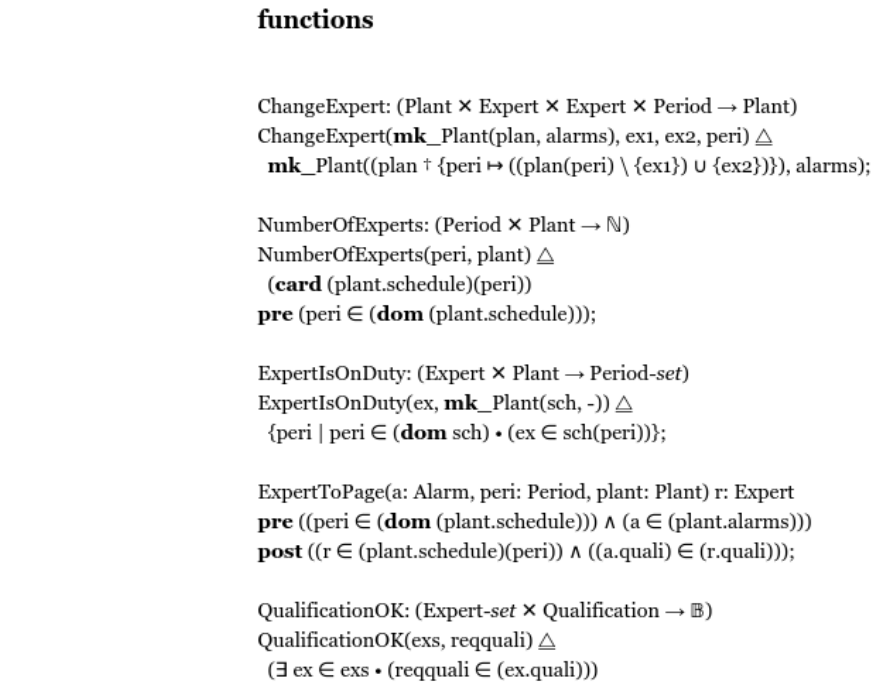

Fig. 3. HTML rendering of the functions in the Alarm specification

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/