Bridging the Requirements-Specification Gap using Behaviour-Driven Development

Kristoffer Stampe Villadsen, Malthe Dalgaard Jensen, Peter Gorm Larsen, and Hugo Daniel Macedo

DIGIT, Aarhus University, Department of Electrical and Computer Engineering,
Finlandsgade 22, 8200 Aarhus N, Denmark
{villadsen67,malthedj}@hotmail.com,{pgl,hdm}@ece.au.dk

Abstract. How is it possible to bridge the gap between requirement elicitation and formal specification? This paper proposes a solution that involves the methodology Behaviour-Driven Modelling inspired by the agile methodology Behaviour-Driven Development and Behaviour-Driven Specification. To support this methodology, we developed a tool that enables users to define behaviours as executable scenarios. A scenario is executed on a formal model through the tool to validate requirements. The tool is developed as a Visual Studio Code Extension. The tooling uses VDMJ to read and interpret formal specifications and Cucumber to discover and execute Gherkin features and scenarios. We believe that the methodology will increase readability of requirements for formal specifications, reduce the amount of misunderstandings by stakeholders, and facilitate the process of developing formal specifications in an agile process.

Keywords: Behaviour-Driven Development, VDM, Agile Methods, Formal Methods

1 Introduction

Formal modelling involves the transformation of a set of elicited requirements listed in a natural language into an executable model in the form of a specification written in a formal language [12]. There are well defined methods for writing formal models, e.g.: Vienna Development Method (VDM) [9], Z [18] and B [1]. The same happens with requirements elicitation e.g.: use-cases diagram, entity-relationship modelling, and user stories.

In most of the approaches, the practice is to transform the requirements into the specification as a jump which consists of a mental exercise including the iteration of two steps: first understand the requirements, then formulate them as a formal model. A conceptual illustration of the metaphoric jump can be seen in figure 1. This jump bridges what we will metaphorically describe as the Requirements-Specification Gap (RSG). The writing of adequate specifications becomes an art that experienced formal modellers acquire by performing several transformations (different jumps across the RSG in several projects).

The existing tool support for working with formal specification includes editing, interpreting, debugging and testing through various libraries and Integrated Development

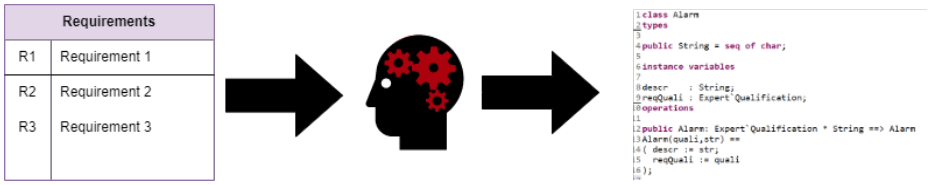

Fig. 1. An abstract list of requirements are shown on the left side. These are interpreted by the modeller, which is illustrated by the pictogram in the middle. The modeller translates the interpreted requirements into a model, shown on the right side. Thereby, performing the metaphoric jump.

Environments (IDEs) e. g., VDMTools [10], Overture [11], VDM-VSCode [16], Rodin [2], and VDMJ [3]. Testing of specifications can also be performed within VDMJ by utilising combinatorial testing or VDMJUnit. However, no domain-agnostic tool exists which supports bridging the RSG.

In this paper, we show how and propose a tool to bridge that gap and ease the translation task for modellers by assimilating and providing tool support for Behaviour-Driven Modelling (BDM), a methodology inspired by the Behaviour-Driven Development (BDD) agile approach. BDD prescribes a methodology to develop software focusing on exemplifying requirements into concrete natural language examples, increasing readability and reducing misunderstandings.

Behaviour-Driven Modelling adds a step between requirement elicitation and specification development, where requirements are translated into behaviours. The behaviours are written in a constrained subset of natural language with the goal of becoming more understandable by stakeholders. A prototype of a Visual Studio Code (VS Code) extension has been developed and is described throughout this paper. The prototype uses Cucumber[1] as a BDD test runner to discover and execute the behaviours. The tool maps behaviours to concrete operations within a formal specification. This mapping is made possible using the VDMJ interpreter and VDMJ Annotations, such that operations can be annotated within the VDM++ specification language. The extension allows for a modeller to create a BDM project and use the BDD test runner to validate behaviours against the specification. The goal of the extension is to be incorporated as part of the Overture tooling support for the VDM languages.

We believe this methodology and tooling will contribute to closing the RSG and increase readability of requirements for formal specifications, reduce the amount of misunderstandings by stakeholders, and facilitate the process of developing formal specifications in an agile process.

The outline of this paper is as follows: Section 2 gives an introduction to relevant topics necessary to understand the paper. Section 3 and 4 describes the proposed approach and technical solution respectively. An example project uses the proposed solution in section 5. The paper concludes upon the findings in section 6 together with a paragraph describing the future work to deploy the tool in the VS Code marketplace.

[1] https://cucumber.io/

2 Background

In agile development the workload is broken down into smaller pieces which can be worked on simultaneously and iteratively [5]. This allows for smaller teams to work together improving the performance of development [8]. Agile methods ensure that the best practices, are applied and the correctness of the engineering processes is upheld. Agile methods help both stakeholders and software engineers in building, deploying, and maintaining complex software with its associated changing requirements [7].

BDD builds on the concepts of Test-Driven Development (TDD) with another approach to system analysis; how the different entities of a software system interact based on the domain model [17]. BDD utilises a Domain Specific Language (DSL) for specifying requirements, which are more readable for stakeholders. The specified requirements are formulated as concrete examples of system behaviour[2].

An agile development tool which supports BDD through acceptance testing is Cucumber. It provides the DSL Gherkin, which is used to specify, in a restricted subset of plain natural language, the desired behaviour of a system. A decisive advantage of formulating behaviours using Gherkin is that not only is a stakeholder more likely to understand it, but importantly a computer is able to. Cucumber aids developers in writing concrete examples of behaviours. This is accomplished by defining a requirement as a feature of a system. This feature is then exemplified through a scenario as seen in listing 1.1. By having concrete examples which defines acceptance tests allows for easier discovery of edge cases [17]. Working with Cucumber and following the BDD methodology gives developers an opportunity to get early feedback from the system.

```
Feature: feature-name
  feature-description

 Scenario: scenario-name
   Given a pre-condition
   When an action occurs
   Then post-condition is satisfied
```

Listing 1.1. An example Gherkin feature and scenario

Cucumber groups scenarios in a feature to provide tracing of requirements for the system. This tracing provides stakeholders with an overview of which requirements that are currently implemented. Each scenario consists of steps which are either *Given*, *When* or *Then*. These steps are conceptually equivalent to Hoare triples [6].

Given corresponds to a pre-condition, which in BDD also initialise the state. *When* corresponds to an action that is performed. *Then* corresponds to a post-condition, which is asserting if the new state is as expected. A table describing the differences and similarities between Hoare triples and BDD steps can be seen in table 1.

In listing 1.2 a definition of a *Given* step is provided. This is defined as a step definition, which is a function that is mapped to a step within a scenario. For a step definition to be mapped to a step, the function should be annotated with a step annotation. Cucumber runs through each scenario and finds a matching annotated function for each

[2] http://behaviour-driven.org/

	Cucumber	Vienna Development Method
Pre-condition	Cucumber utilises the *Given* annotation to identify the pre-condition of the scenario, within the pre-condition the state is initialised.	The pre-condition in VDM is utilised within the operation scope noted by the pre keyword at the end of the operation, VDM does not initialise any state. VDM checks if the input adhere to a defined constraint
Action	Cucumber utilises the *When* annotation to identify the action performed by program based on the state and the desired feature.	Within an operation of VDM, the model is acted on by the input based on the state of the model.
Post-condition	Cucumber utilises the *Then* annotation to identify the post-condition of the scenario, within the post-condition cucumber asserts on the state of the program to determine if it verifies the desired behaviour.	The post-condition in VDM is utilised within the operation scope noted by the post keyword at the end of the operation, VDM verifies that the output of the function adheres to a defined constraint

Table 1. Description of how the Hoare triples are applied in Cucumber and VDM

step. Cucumber executes this step definition and repeats this for all steps in a scenario. A scenario passes if step definitions executes correctly and all assertions are satisfied.

```
@Given("A pre-condition")
  public void A_pre_condition() {
    pending();
  }
```

Listing 1.2. An example Java step definition

Simon Fraser et al.(2021) presented an approach called Behaviour-Driven Specification (BDS) [5]. BDS applies BDD on formal specifications by validation of the formal specification against requirements formulated as Executable Acceptance Criteria (EAC). The Azuki platform[3] - a generic framework that assists in using BDS. The framework aids in analysing legacy systems which results in human readable acceptance criteria. Furthermore, the framework creates EAC which it runs against the specification to verify the system. EAC are BDS equivalent to executable scenarios from BDD. Mapping these directly to the behaviours of the desired system will result in validation of the specification against the requirements. The framework is designed and developed to solve a specific problem case for a company as stated in Simon Fraser et al.(2021) [5]. The framework has been developed constrained by a specific domain, therefore, challenges will inherently arise when applying this framework more widely and domain agnostic.

3 Behaviour-Driven Modelling

Traditional formal modelling consists of a modeller performing a transformation of elicited requirements into an executable model. This is a two-step iterative process: understand the requirement, then formulate it as a formal model. Performing the transformation will map the requirements into a specification which seeks to prove the properties and be used as a Quality Assurance (QA) metric of the system. The understanding of the requirements and formulation of the specification is at the mercy of the modeller. This means that the properties and the QA is determined by the modeller. Attempts at adapting this traditional formal approach through merging of formal methods and agile

[3] https://github.com/anaplan-engineering/azuki

methodologies have been proven to be possible [13, 19]. The results of the conceptual approaches include an iterative process focusing on smaller parts of the system.

The conceptual difference between the traditional approach and the agile formal modelling approach is shown in figure 2. In this paper, the Agile formal modelling approach has been extended with tool support for the BDD approach, and the approach will be described as BDM. BDM introduces the additional step which bridges the gap

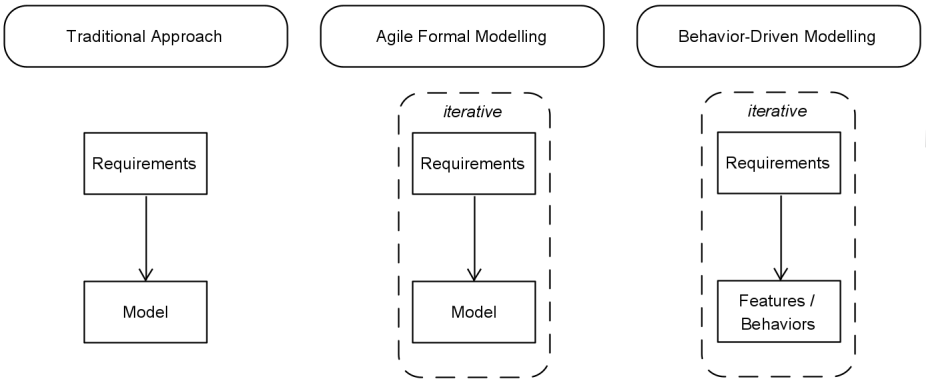

Fig. 2. This figure shows the evolution of RSG. On the left is the traditional approach of a direct transformation from requirements into a specification. In the middle is the conceptual depiction of the agile formal merging approach. On the right is the approach introduced in this paper.

from requirements to the specification within an agile approach as seen in figure 2.

The RSG is bridged by focusing on requirement analysis to define behaviours in terms of features and scenarios. The behaviours act as guidance for both developers and stakeholders to capture requirements as artifacts that can be traced as they are being implemented in the formal model. The tracing allows for quality assurance teams to be part of the early development process. It allows for modellers and developers to be part of requirement elicitation in a way that can result directly in step definitions, which can run against both the model as well as the production code.

The tooling for BDM focuses on requirement elicitation through stakeholder communication to define behaviours of the desired system. The BDM approach is illustrated in figure 3, firstly, the requirements elicitation is performed. After the requirements have been elicited and analysed, the behaviours are defined. These behaviours are then mapped to step definitions. The BDM tool allows for mapping of the properties of the model to the step definitions. Lastly, the specification is validated against the requirements through these step definitions using a BDD test runner. The BDD test runner performs the mapping and validation of behaviours against the specification. This approach is then repeated for each requirement that is defined and can be applied directly in an agile process.

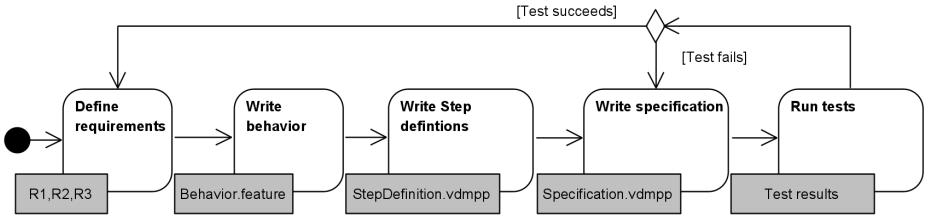

Fig. 3. This diagram shows an overview of the proposed BDM methodology. The process repeats the steps illustrated until all requirements have been satisfied.

A screenshot of the developed tooling can be seen in figure 4. The figure shows the tooling from the user's perspective. At the current stage of the tooling, three areas are of interest, these are noted with the three numbered red boxes on the figure:

- **Box #1** - Shows the project file structure depicting the location of the specification files, features, and step definitions.
- **Box #2** - Shows the file that is currently being edited, in this case, it is a feature file describing a scenario with different input values.
- **Box #3** - Shows the output of running the executable scenario described in the feature. Here, it shows that three tests have been executed successfully.

The execution of tests utilises VDMUnit to read and interpret the specification to validate it against the features.

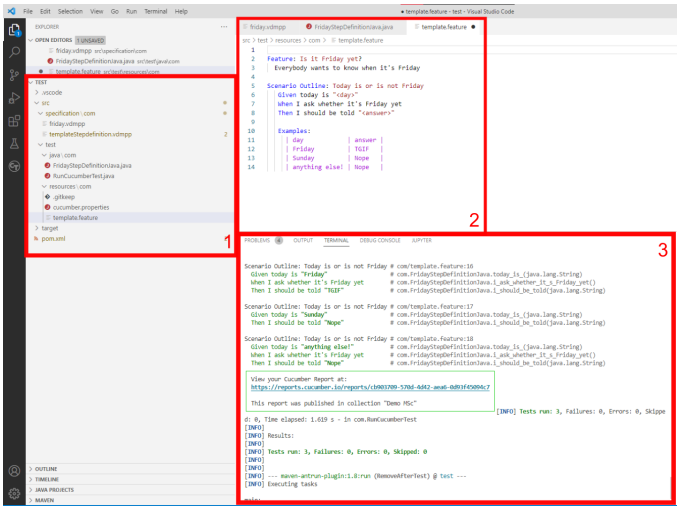

Fig. 4. Screenshot of the VS Code extension.

4 Technical Solution

This section describes the tool developed to enable users to follow the approach described in section 3. An overview of the proposed solution can be seen in figure 5 illustrating how the Cucumber test engine finds and executes scenarios through generation of Java step definitions.

The user of the tool defines scenarios and VDM++ specifications. The VDMJ interpreter performs type checking of the defined VDM source. During this operation, the tool looks for annotations for VDM definitions and generates Java step definitions based on the annotations detected. The generated Java step definitions are provided to the cucumber test engine together with user defined scenarios. When the Cucumber test engine executes a scenario with the corresponding generated java step definitions, it will return either passed, failed, or pending. A pending test result means that the scenario is not yet implemented.

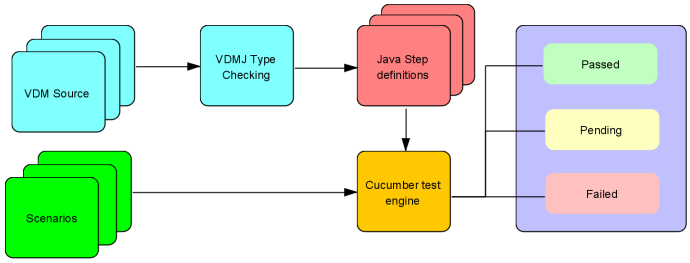

Fig. 5. An overview of how the Cucumber test engine finds and executes scenarios and step definitions created through the BDM tool.

To effectively apply the methodology, the tool provides support for the following Cucumber annotations in VDM: *Given*, *When* and *Then*. Additionally, the tool also supports an annotation called *StepDefinition*, which is used to annotate the VDM++ class containing the operations annotated with the Cucumber specific annotations. These four annotations provide the means to effectively define step definitions in the VDM++ language according to the format that the Cucumber test engine can understand and execute.

A BDM project is required to conform to a defined directory structure. The project directory structure can be seen in figure 6 and shows how the different types of files should be located. The root project folder contains two directories, src and target. The target sub-directory contains generated files from the BDM project. The src folder contains the VDM source files within the specification sub-directory. The VDM source files is both the specification and step definitions. Java source files is located within the Java sub-directory inside the Test directory. The only Java file for the project is *RunCucumberTest.java*, which is required to run the Cucumber test engine. Scenario files is located within the Resources sub-directory inside the Test directory.

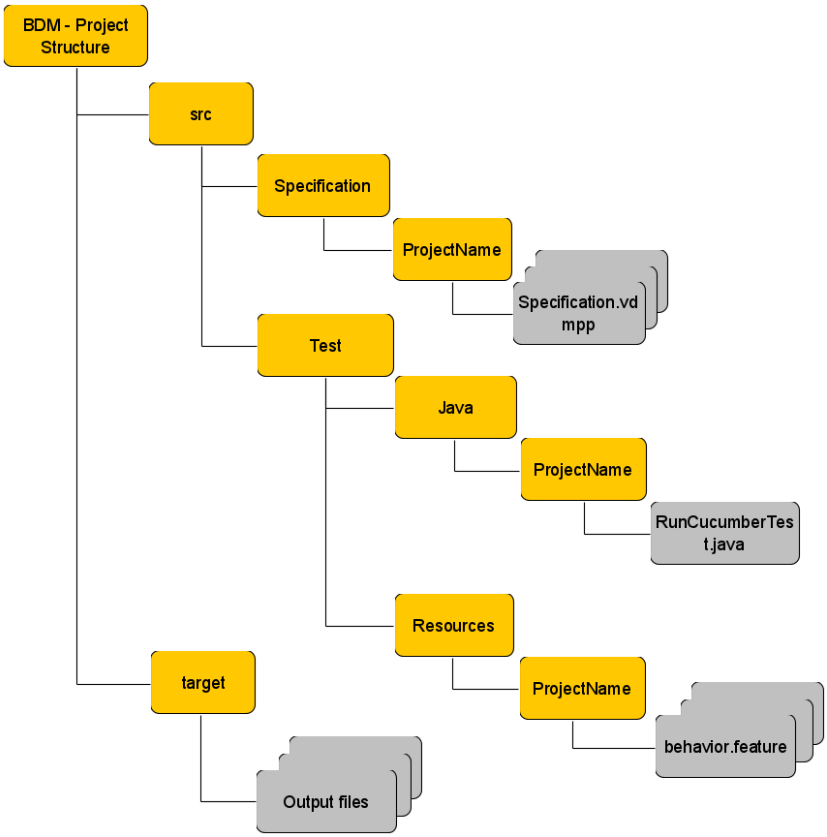

Fig. 6. An example of a BDM project structure.

A conceptual package diagram depicting the implementation and dependencies of the tool developed can be seen in figure 7. The package diagram displays the BDMAnnotations package containing the four different annotations: `Given`, `When`, `Then`, and `Step Definition` and their dependency to *BDM.Lib*. This library contains utility functionality through the VDMUtility class and functionality to build Java step definitions through the StepDefinitionsBuilder. *BDM.lib* has a dependency to VDMJ and VDMJUnit in order to read and interpret VDM++ specifications. Furthermore, the package has a dependency to Javassist to provide functionality that generates java classes at runtime.

The tool extends upon the VDMJ annotations library to enable support for defining step definitions in the VDM++ language that can be understood and executed with the Cucumber test engine as the BDD test runner. The Cucumber test engine handles discovery, selection, and execution of Cucumber scenarios. The extension of the VDMJ Annotations library was implemented with respect to VDMJ Annotations documentation [3]. When an annotation is found the tool generates a corresponding Cucumber Java

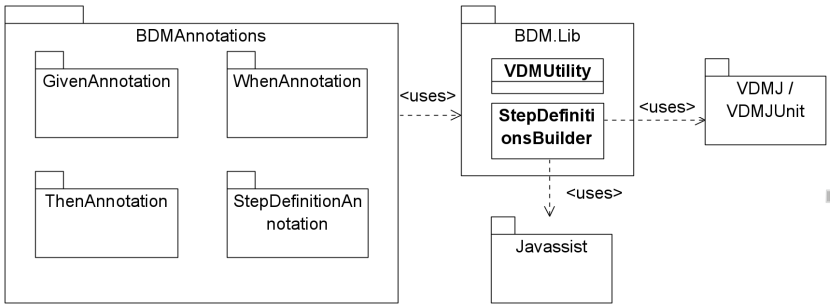

Fig. 7. Package diagram depicting implementation and dependencies of the BDM tool.

step definition that uses the VDMJUnit library to read specifications and execute the corresponding operations in the VDM++ step definitions [3].

The VDMJ Annotations library provides the ability to affect different aspects of the VDMJ operation: the parsing, the type checking, the interpreting and the proof obligation generation. For the annotations described above, the VDMJ type checking operation is affected to generate step definitions in the Java language. When implementing an extension to the VDMJ Annotations, it is required to know which VDM constructs that should be annotated and whether the functionality should be executed before or after the VDMJ operation. The developed tool performs the generation of classes before the VDMJ type checker performs its operation. To illustrate the steps performed by the tool, a sequence diagram can be seen in figure 8 which displays the steps performed when a *Given* annotation is found.

To build step definitions in the Java language, a third-party library is used called Javassist. This enables bytecode manipulation, meaning that Javassist can perform alterations and creations of programs or constructs within a program such as a class [4]. The created Java step definition is generated through the Javassist library and implements the Cucumber step definition functions that is executed by the Cucumber test engine. To effectively read specifications and execute functions or operations defined in a VDM++ model, the library VDMJUnit is used. The created Cucumber Java step definitions inherits from the *VDMJUnitTestPP* Java class and implements the *Given*, *When*, and *Then* step definition functions. The functions are annotated as specified in the corresponding VDM++ annotations.

When a *Given* annotation is found by VDMJ, it calls the *PreCheck* function from the *StepDefinitionBuilder* inside the *BDM.Lib* library. The function performs validation of the input values to ensure that the operation name and parameters are defined as expected by the *Given* annotation. The input values are:

- **def** - The definition from the VDMJ type checker.
- **name** - The name of the Cucumber behaviour, used to map requirements and step definitions together.
- **fName** - The name of the function annotated.
- **Type** - The type of Cucumber annotation.

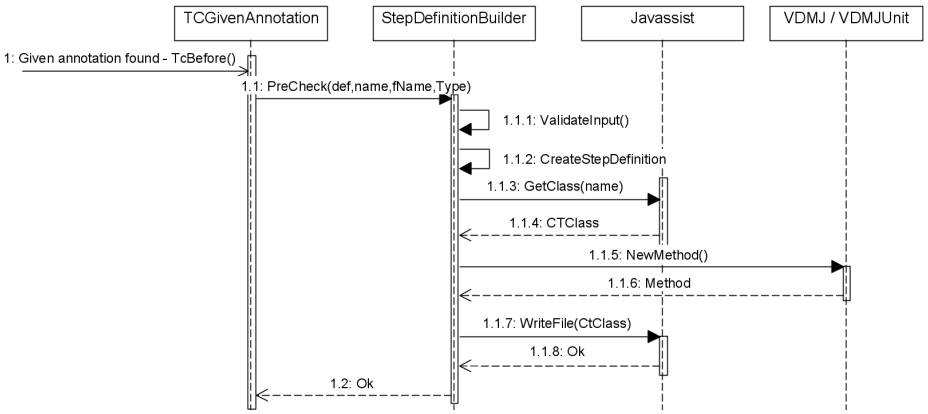

Fig. 8. Sequence diagram depicting the steps performed by the tool for the *Given* annotation.

When the input values have been validated, the builder then generates the Java step definition function within the step definition class. The builder makes sure that the generated function is annotated correctly according to the standards given by the Cucumber test engine.

The prototype for the tool is developed as a VS Code extension which handles creation of BDM projects and execution of cucumber tests. The architecture for the extension can be seen in figure 9. A detailed description of the architecture can be seen in section 6.

5 Demonstration

To demonstrate the developed tool and approach described throughout this paper, an example project is used. The example is a very simple project inspired by the Cucumber "10 Minute Tutorial"[4]. The requirements for the project are defined as the following:

- **R1** - The system must output "TGIF" if today is Friday.
- **R2** - The system must output "Nope" if today is not Friday.

The first step in the proposed approach is to define behaviours based upon these requirements. These behaviours are defined according to the traditional BDD approach, by defining Cucumber scenarios within a feature file, and can be seen in listing 1.3.

[4] https://cucumber.io/docs/guides/10-minute-tutorial/

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday
    Scenario Outline: Today is or is not Friday
    Given today is "<day>"
    When I ask whether it's Friday yet
    Then I should be told "<answer>"
    Examples:
      | day             | answer |
      | Friday          | TGIF   |
      | Sunday          | Nope   |
      | anything else!  | Nope   |
```

Listing 1.3. The Cucumber behaviour definition for the case project.

The next step involves defining step definition operations in VDM++, that should be annotated with one of the step definition operations from the defined behaviour, such as *Given*, *When*, or *Then*. The annotation takes two parameters; the first is the name of the temporary variable that is used to maintain the state of the object, and the second is a string containing the text defined for the step definition operation within the feature file. On listing 1.4, the step definition operations can be seen for the defined behaviour. An annotation is written according to the VDMJ documentation, within a comment and with an *"@"* as a prefix. The assertion of the behaviour happens in the operation annotated with *Then*, where the tool will assert that the post condition is satisfied. If the post condition is satisfied the test will succeed, if not, it will fail.

```
-- @Given("f1","today is {string}")
public Today_is: seq of char ==> ()
Today_is(str) ==
(
  friday := new Friday(str);
);
-- @When("f1","I ask whether it's Friday yet")
public I_ask_whether_it_is_friday: () ==> ()
I_ask_whether_it_is_friday() ==
(
    actualResult := friday.IsItFriday();
);
-- @Then("f1","I should be told {string}")
public ResultingThen: seq of char ==> bool
ResultingThen(expected) ==
 return expected = actualResult
post RESULT = true;
```

Listing 1.4. Step definition operations for the defined behaviours written in VDM++

When the Cucumber test engine execute the tests generated based on the behaviour descriptions, it will find and execute these VDM++ operations through a generated Java step definition class that utilises VDMJUnit to access the VDM++ step definition operations. The generated Java functions can be seen on listing 1.5. Each of the functions contains a call to a helper function called *checkLocalVariable*, this function checks

whether the local variable that are given as parameter actually exists. If the variable does not exist, the function creates it.

```
@Given("today is {string}")
  public void Today_is(String paramString) {
    checkLocalVariable();
    run("f1.Today_is(\"" + paramString + "\"" + ") ");
  }
@When("I ask whether it's Friday yet")
  public void I_ask_whether_it_is_friday() {
    checkLocalVariable();
    run("f1.I_ask_whether_it_is_friday() ");
  }
@Then("I should be told {string}")
  public void ResultingThen(String paramString) {
    checkLocalVariable();
    run("f1.ResultingThen(\"" + paramString + "\"" + ") ");
  }
```

Listing 1.5. Step definition functions for the defined behaviours written in Java

The result of utilising this approach is the ability to define requirements in a more stakeholder-readable language, which are converted to executable scenarios using step definition annotations. These executable scenarios are executed by the Cucumber for Java test engine. This tool and approach make it possible to perform BDM of VDM++ specifications.

6 Concluding Remarks and Future Work

This paper introduces the BDM approach to bridge the RSG by extending the traditional two steps with an extra step. The additional step includes definition of behaviours based on requirements. To ease the translation task, a tool is introduced that supports BDM.

The tool described in section 4 is still under development but, at its current state, it is possible to use it to perform BDM as described in section 5. The current state of the tool is the ability to define step definition operations within a VDM++ class and use the Cucumber test runner to execute these operations. The tool requires a defined directory structure to locate specification files and feature files. A minimal VS Code [14, 15] extension have been developed to generate a new BDM project that conforms to the requirements of the tool. The extension also supports execution of tests through the Cucumber test engine.

BDM supports formal validation through testing of operations in the defined specification. BDM extends this by supporting validation of the requirements through the tool support described in section 4. Given that the testing is handled through the Cucumber test engine allows for validation of production code corresponding to the formal specification - with respect to constraints. The testing of the formal specification cannot run directly on production code, however, the behaviours can be mapped to step definitions whereas both the production code and the formal specification satisfies the requirements.

It is noteworthy to mention that this approach does not eliminate the need for verification through unit testing of individual elements of a formal specification.

Feature support	Traditional approach	Agile Merging Methodology. e.g. Scrums goes formal, FormAgi..	Behaviour-Driven Modelling
Formal specification	Supported	Supported	Supported
Iterative process		Supported	Supported
Early feedback		Supported	Supported
Tool support for specification validation			Supported
Tool support for implementation validation			

Table 2. Feature supported by different approaches.

The approach presented in this paper differs from previous attempts at bridging the gap, in a few different ways. BDM obtains the same advantages as the approaches which merge the agile methods and formal methods. BDM utilises the iterative nature from BDD, which other approaches have gained from e. g. SCRUM [19]. The early feedback and validation of BDD is still maintained with BDM. An overview of the features supported by BDM is shown in table 2. However, BDS applies a similar approach as BDM. Both methodologies applies BDD for development of formal specifications. The main difference stems from their associated tool, which have been developed for different intended users and problem domains.

A valuable extension of the developed tooling, would include the functionality of validating formal specification and the corresponding implementation against the same requirements. The extension would aid in the process of discovering differences between them. The consequences of a change in requirement would be accentuated in the validation process for formal specification and implementation. This would provide the stakeholders with insight about the sometimes costly consequences of changing requirements.

The proposed solution described by this paper is not the only solution for the problem definition. An alternative solution would be to implement native support for the VDM dialects directly within Gherkin and Cucumber. This would eliminate the dependencies for generation of Java classes that can be understood by the Cucumber Java tool. This solution was considered when developing the proposed solution but were deemed too comprehensive. This paper have introduced the methodology of BDM and shown that it can be applied to a simple case project. More research is needed in order to determine the feasibility of the methodology and associated tool for bridging the RSG. This research should include more case studies on more complex projects in order to gain insight in the scope of applicability and the limitations of adapting BDM. However, the presented methodology attempts at defining a formal process for translation of requirements into formal specifications.

The tool itself is under development with several features in the pipeline. As mentioned above, a VS Code extension is being developed to provide project creation and

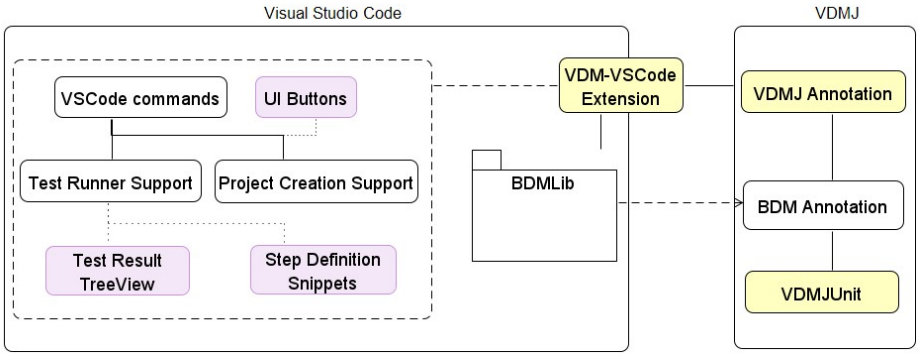

Fig. 9. Overview of the VS Code extension architecture.

testing capabilities within VS Code. The extension utilises a modified Maven Archetype[5], which has been modified to create a new project that has the necessary structure and files to use BDM. This project structure can be seen in figure 6. The testing capabilities are handled in this extension through a modified maven test command. An overview of the architecture of this extension is shown in figure 9. The current version of the extension supports the white elements on the figure, the package *BDM.lib* and the element named *BDM Annotations* are described in section 4. The yellow boxes are third party systems. The extension is not yet ready for release while writing this submission but we expect it to be complete by the time of the workshop. The extension should be extended with user interface buttons and a testing view, which are depicted in pink in figure 9, since the extension only provides its functionality through commands executed in the terminal and through the VS Code command line interface. The extension should also provide the user with snippets to step definition since this is standard practice within the Cucumber tooling. Additionally, the tool should be extended to support the two other dialects of the VDM; VDM-SL and VDM-RT, the tool only supports the VDM++ dialect.

Acknowledgments We would like to thank Simon Fraser and Alessandro Pezzoni for inspiring us to approach agile formal methods with BDD and for providing information about the Azuki framework. We would also like to thank the Overture community for providing information about extension development in VS Code and the VDMJ library.

[5] https://maven.apache.org/guides/introduction/introduction-to-archetypes.html

References

1. Abrial, J.R., Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press (2005)
2. 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. STTT 12(6), 447–466 (2010)
3. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
4. Chiba, S.: Load-time structural reflection in java. In: European Conference on Object-Oriented Programming. pp. 313–336. Springer (2000)
5. Fraser, S., Pezzoni, A.: Behaviour driven specification. In: Macedo, H.D., Thule, C., Pierce, K. (eds.) Proceedings of the 19th International Overture Workshop. Overture (10 2021)
6. Hoare, C.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–581 (October 1969)
7. Hoda, R., Salleh, N., Grundy, J.: The rise and evolution of agile software development. IEEE Software 35(5), 58–63 (2018)
8. Hoegl, M.: Smaller teams–better teamwork: How to keep project teams small. Business Horizons 48(3), 209–214 (2005), https://www.sciencedirect.com/science/article/pii/S0007681304001120
9. Jones, C.B.: Systematic software development using VDM, vol. 2. Prentice Hall Englewood Cliffs (1990)
10. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
11. 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
12. Macedo, H.D., Larsen, P.G., Fitzgerald, J.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods, 15th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 5014, pp. 181–197. Springer-Verlag (2008)
13. Olszewska, M., Ostroumov, S., Waldén, M.: Using scrum to develop a formal model–an experience report. In: International Conference on Product-Focused Software Process Improvement. pp. 621–626. Springer (2016)
14. Rask, J.K., Madsen, F.P.: Decoupling of Core Analysis Support for Specification Languages from User Interfaces in Integrated Development Environments. Master's thesis, Aarhus University, Department of Engineering (December 2020)
15. Rask, J.K., Madsen, F.P., Battle, N., Macedo, H.D., Larsen, P.G.: The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions. In: Proença, J., Paskevich, A. (eds.) Proceedings of the 6th Workshop on Formal Integrated Development Environment, Held online, 24-25th May 2021. Electronic Proceedings in Theoretical Computer Science, vol. 338, pp. 3–18. Open Publishing Association
16. Rask, J.K., Madsen, F.P., Battle, N., Macedo, H.D., Larsen, P.G.: Visual Studio Code VDM Support. In: Fitzgerald, J.S., Oda, T. (eds.) Proceedings of the 18th International Overture Workshop. pp. 35–49. Overture (December 2020), https://arxiv.org/abs/2101.07261
17. Rose, S., Wynne, M., Hellesoy, A.: The cucumber for Java book: Behaviour-driven development for testers and developers. Pragmatic Bookshelf (2015)
18. Spivey, M.: The Z Notation – A Reference Manual (Second Edition). Prentice-Hall International (1992)
19. Wolff, S.: Scrum Goes Formal: Agile Methods for Safety-Critical Systems. In: ICSE 2012: Proceedings of the 34th International Conference on Software Engineering. pp. 23–29 (June 2012), Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA 2012