Behaviour driven specification

A case study in combining formal and agile methods

Simon Fraser, Alessandro Pezzoni

Anaplan Limited, York, UK
{simon.fraser, alessandro.pezzoni}@anaplan.com
http://www.anaplan.com

Abstract. The use of formal methods in industry may be resisted by developers who have embraced agile methods. There is often a perception that formal methods require much upfront work and are unable to adapt quickly to a changing landscape. This leads to a view that formal methods are a barrier to embracing change and are unsuitable for projects involving the iterative evolution of requirements. In this paper, we use a project at Anaplan – a company where agile methods are the norm – to illustrate how formal methods can be integrated into an agile development process. Specifically, we introduce the concept of behaviour driven specification as an agile-compatible method of constructing a formal specification and show how the acceptance criteria produced by this process not only provide an accessible form of requirements capture, but provide a means for validation of both the specification and a target implementation.

Keywords: Software Validation · Behavior Driven Development (BDD) · Agile Methods · Formal Methods

1 Introduction

Agile methods are becoming ubiquitous in modern software engineering companies. Meanwhile, formal methods are seen as having limited use, with the perception being that they can only provide value in niche industries, such as safety-critical systems or chip development. In our experience, many software developers maintain a view that formal methods are incomprehensible, are tied to linear development practices, and are incompatible with agile methods.

In this paper, we seek to show that formal methods are not mutually exclusive with agile methods, but can can be used alongside them to provide additional rigour in suitable projects. We begin by introducing Anaplan, a SaaS company delivering an enterprise planning platform using an agile development process. Having described the fundamentals of the platform, we will outline how its calculation engine is optimized for modelling the most common business scenarios. We will then describe a scenario that could be more efficiently handled using a different approach, how that has led to the need for an alternative calculation engine, and how formal methods could help us maintain consistency between the two engines.

We then introduce the concept of behaviour driven specification (BDS), an agile-compatible methodology that we have developed to drive specification from stakeholder

requirements, and which we have used to help deliver the above calculation engine. BDS is designed to enable specifiers to operate with agility, but we will also describe how it can be integrated into the wider agile process of a software project.

Finally, we present some metrics from our project that exemplify the impact that using BDS has had on our project, and conclude that it is feasible to successfully combine agile and formal methods in an industrial setting.

2 Background

Anaplan is a software company that produces an eponymous, cloud-based platform for business planning. The Anaplan platform provides a flexible model-building environment which customers use to build their own applications. These can model a simple business scenario or manage multiple, complex, planning use cases across a global enterprise, covering functions such as finance, sales, supply chain, workforce and marketing.

At the heart of Anaplan is a calculation engine that enables consistent management of OLAP (online analytical processing) cubes [6]. A cube (or hypercube) is a multi-dimensional array of data such that each cell represents a measure of the business. An example cube might represent unit sales and have dimensions: products, shops, weeks, and versions (for example, actual vs forecast). Here the value of a cell would indicate either a forecast or actual number of units sold of a given product, at a given shop, for a given week.

The dimensions of a cube are frequently hierarchical structures that model the structures inherent to the business. For instance, shops might have parents that are cities, which in turn have parents that are countries, and similarly weeks may be grouped to align with the business's accounting periods. These hierarchical structures can be used with an aggregation function to automatically generate additional cubes that summarize the business's data. From our examples above, we might have a cube where each cell represents either a forecast or actual total of units sold for all products in a country, for a quarter-year period. In Anaplan, the model builder chooses a default aggregation function, and the engine automatically calculates all possible summary cubes from the Cartesian product of hierarchical layers.

Not all of a business's data is a direct quantification of some measure, as much of it may be calculated from other data. For instance, if we had a cube as above which represents units sold, we may have another one with the same dimensions that represents price. Given these two cubes, we could calculate a third cube where each cell represents either a forecast or actual revenue from selling a product in a week, obtained by multiplying the values of the cells with the same coordinates from the given cubes. Anaplan has a rich formula language, providing many inbuilt calculation functions, enabling model builders to describe all the activities of their business. Within a single model, Anaplan immediately recalculates all necessary figures whenever the value of a cell is changed. This makes it possible, for example, for a CEO to visualize the impact on the business's strategic plan everytime a sale is made at an integrated EPOS.

2.1 A matter of scale

A customer's Anaplan estate will typically consist of a number of connected models. Each model will usually focus on a specific business problem, allowing detailed analysis, but will also export summarized data to be used in other models. As mentioned previously, when any change is made within a model the impact of that change will immediately be reflected throughout the model – akin to making a change in a spreadsheet. Furthermore, unlike spreadsheets – which are still commonly used for business planning – Anaplan models will frequently have billions of cells.

To efficiently maintain the consistency of the Anaplan model, all cube data is held in memory to enable rapid retrieval and calculation. Larger instances of models can consume more than 700GB of memory to store the cube data alone. When the memory required to process the data, operate an API, and run the OS is added, it can take a server with 1TB of memory to operate a model. Although such amounts of memory can be installed in your own data centres, it is at or near the limit of instances offered by cloud hosting solutions. This is a concern as it can limit options for disaster failover.

Despite the scale of model offered by Anaplan, we are sometimes asked to provide for cubes with trillions of addressable cells. Often this is due to practices carried over from less flexible, legacy OLAP tools, but sometimes they are due to inherent properties of the scenario being modelled and, upon closer inspection, we find that in the vast majority of these scenarios not all of these addressable cells are 'valid'. For example, an aviation company may wish to measure the time taken to fly between airports and compare it with their competitors. According to IATA [1], there are approximately 9100 airports and 1100 airlines. To track the duration of flights between any two airports by day and airline for the period of a year would require 3.3×10^{13} cells. IATA also reports [2] that there are less than 40 million flights made a year[1]. In practice, therefore, less than a thousandth of one percent of the addressable cells will ever be populated.

2.2 An alternative calculation engine

The Anaplan calculation engine was optimised for densely populated models. Using large chunks of contiguous memory for cube data enables efficient calculation, aggregation, and dependency management through mechanical sympathy with the CPU caches [9]. Although memory may not always be assigned for every addressable cell, the requirement to use contiguous memory would be inefficient for cubes like those in our example.

We decided to begin work on an alternative calculation engine that was optimised for the sparse population of a model. Here we would only assign memory for populated cells – enabling larger models – and make use of sparse computation strategies to retain efficiency. This calculation engine will be offered alongside the existing engine, enabling a user to choose appropriately for the business case they are modelling. We would generally expect the user to be able to create the same models with either engine, and for both engines to give the same calculation results.

The calculation engine is an integral component of Anaplan, and in order to introduce another one we needed to refine the interface between the engine and the rest of

[1] In fact, many of these flights will be between the same two airports on the same day.

the platform. This was a task with risk, as it was critical to neither impact the functionality nor performance of the existing calculation engine. The approach taken was to completely fork the platform, quickly make changes to the fork to determine best practice and, once established, apply production-quality changes to the mainline. The new sparse calculation engine would then be developed in parallel and continuously integrated. This led to having four alternative platforms available for the duration of the project: the fork and mainline platforms, each paired with either the dense or sparse engine.

2.3 The need for formal methods

The aim of the project was to have all four platforms eventually exhibit the same behaviour. However, there would naturally be some variation throughout the lifetime of the project. We would be working using the agile methodology [18] and the features available in all four alternatives would vary over time and would not be equivalent at most points.

Furthermore, in some instances we wanted to alter the existing behaviour since, like most organically evolving software products, Anaplan is the culmination of design decisions made over many years. If we were to revisit some of these decisions today, we might choose other options, as now we have more or better data that has invalidated our previous assumptions. This is particularly true for design decisions that were influenced by the implementation method at the time or which would be made differently for a sparse paradigm. Typically, these decisions impact performance or efficiency.

For example, consider the value of 0^0: in Mathematics this is either left undefined or set by convention to $0^0 = 1$, and indeed the original definition of the POWER function in Anaplan follows this convention, giving $\mathrm{POWER}(0, 0) = 1$. This makes sense in a dense context, where we can expect that the vast majority of points will have a meaningful user-defined value. On the other hand, in a sparse context most points will not have a user-defined value and will instead have a default value, which for numbers is 0. As a result, the memory consumption would be dramatically increased if $\mathrm{POWER}(0, 0)$ returned 1, leading to performance degradation, so for our sparse engine we chose a different convention: $\mathrm{POWER}(0, 0) = 0$.

We needed the means to not only ensure that each system exhibited the same base behaviour, but also to rigorously capture any differences between them. We had a difficulty here as the platform has been developed using agile principles, and one of the tenets of this methodology is to prefer 'working software over comprehensive documentation' [5]. Like most teams [20], we still possess and value documentation, but the working software is the 'source of truth', and we did not have an independent means of benchmarking its correctness.

The decision was made that a specification was needed. It was necessary that we were able to capture both the existing behaviour and any differences between the alternative platforms. It was essential that we could do so rigorously and without ambiguity, making the use of formal methods an obvious choice. However, we also needed the specification to fit into our agile way or working; it would not be possible to complete the specification before starting work on the new calculation engine, and the platforms would change as the specification was written. Furthermore, we needed to make use of

the specification as soon as possible to enable quick identification and resolution of any discrepancy between the platforms' behaviour. In summary, we needed to combine the rigour of formal methods and the flexibility of agile methods.

2.4 Existing work

In the same way that none of the leading relational database systems satisfies Codd's twelve rules [7], commercial OLAP systems are inconsistent in their mathematical foundations. Anaplan encourages the use of many small, dense, and distinct cubes that are connected through formulae to form a model. Some of the cubes explicitly represent inputs to the model and others its outputs, such that dashboards, charts and reports are constructed automatically from those outputs. Thus, a user does not use the typical OLAP operations, such as roll-up, drill-down, or slice and dice to *interact with the cube*. Instead once a model has been constructed, the user simply makes a change and observes its immediate impact on the output artifacts.

Formalisms have been proposed for a 'pure' OLAP approach [15,14], but these have focused on the use of operations – particularly roll-up – within a single cube; our specification needed to guarantee consistency across a directed graph of cubes. Furthermore, Anaplan's connected planning methodology enables the use of multiple models – where each typically corresponds to an area of a business – and our specification was also required to define the asynchronous relationships between models. These differences, and a desire to use the same methods throughout, discouraged us from pursuing the use of an existing OLAP specification.

3 Behaviour driven specification

When using agile methods, work is broken into small chunks that each provide some value to a stakeholder in the system. These chunks are often framed as user stories [8] which describe the work to be done from the perspective of the stakeholder. Part of the user story is a set of acceptance criteria (AC) that give a checklist of what must be achieved in order to consider the work complete.

Test driven development (TDD) [4] is a process frequently used by teams employing agile methods. Whenever the developer wishes to change the code, failing tests are created before writing just enough code to make those tests pass. The developer then repeats the process until they have made all the changes required. TDD does not align neatly with user stories as the granularity of the changes tested is usually much finer than that of the user story.

Behaviour driven development (BDD) [21] is a process that can be used in addition to TDD and which operates at the level of a user story. BDD involves the creation of scenarios written from the point of view of the stakeholder, and which exemplify their requirements. Each scenario is a form of Hoare Triple [11] where we capture that:

- *given* an initial context (precondition)
- *when* the stakeholder performs an action (command)
- *then* the outcome is as expected (postcondition)

These scenarios are written using a domain-specific language (DSL) for which the developer creates an adapter that enables the scenario to be executed and verified.

We believed that we could adapt BDD to form a behaviour driven specification process which would enable us to create a formal specification following agile principles. Our principal aim was to be able to comprehensively specify small chunks of the system on a regular cadence, which would enable developers to leverage the specification to validate the corresponding chunks of implementation, providing constant feedback on its correctness and quickly identifying where differences had arisen – whether intentional or not. We also wished to provide the developers with a set of requirements for the system in a form with which they were familiar – acceptance criteria. By doing this, we also aimed to significantly reduce the need for natural language commentary on the specification, replacing it with scenarios written in a DSL which would be both less ambiguous and more maintainable.

3.1 What does BDS involve?

The BDS process is much like BDD, and it too requires a DSL in which to write scenarios. However, unlike BDD, where the inputs are ACs and the output is code, with BDS we look to establish ACs and use those to drive the creation of the specification, with both being valuable artifacts of the process.

For each behaviour, the first step is to write a natural language description of the AC. A scenario is then created to exemplify the AC; this may involve the extension of the DSL if new language is needed to express the criterion. This scenario now forms an executable acceptance criterion (EAC) – we shall discuss how these are run in section 3.2 – that we can use to validate the behaviour of both the specification and the target implementation. When first run this EAC should fail, otherwise it is likely that the AC is redundant or that the scenario does not correctly capture the unicity of the AC. The specifier then makes just enough changes to the specification to satisfy the EAC without invalidating any others. This process is repeated until all of the ACs of the behaviour have been established and agreed with the stakeholder. At this point the EACs can be handed to a developer to implement the behaviour. The developer can now start their work with a set of ACs, including examples scenarios that fail against the target implementation — an ideal place to start BDD.

We can rapidly create scenarios to exemplify behaviour questions that arise during implementation, and we can run them against the specification to determine the expected outcome. If the developer is not satisfied, we can take that example and the specified results, and present to the stakeholder. If the stakeholder believes this scenario is distinct from our existing EACs and wants a change of behaviour, we make the scenario a new EAC and change just enough of the specification to satisfy it. At this point, the specifier may discover that it is not possible to make a change that is consistent with the existing EACs. If so, the conflicting criteria are presented to the stakeholder and a resolution agreed. The use of the DSL allows us to do this in the language of the stakeholder, but without the ambiguity of natural language.

Similarly, the developer may discover efficiency or performance issues with implementing a particular EAC, or in fact, any member of the team may spot some behaviour

they disagree with. Once again, these can be discussed with the stakeholder, the outcome of the EAC changed, and the requirements of the behaviour, the specification, and the implementation iteratively refined.

The use of EACs not only encourages a more agile approach to specification, but enables its continuous validation through DSL-based requirements which are accessible to everyone involved in the project.

3.2 Running executable acceptance criteria

Running scenarios requires a DSL in which to express the actions and checks of the system in the stakeholder's language, and means by which to translate that language into commands and queries that the implementation can execute. Cucumber [21] is synonymous with BDD, but we found it unsuitable for use with BDS. Although Cucumber accepts scenarios in the given-when-then format, the distinction between the blocks is lost at runtime and the scenario is reduced to a single sequence of steps. When executing a program it is perhaps unimportant that certain steps are declarative and others are imperative[2], but when animating a specification we wanted to transform the declarative steps into a single declaration rather than calling a series of functions. After some time trying to adapt Cucumber to our needs, we found that we were better able to make progress with a custom Kotlin DSL [19].

In practice, we found the majority of our effort in this area was in creating the DSL and adapting to the specification and the various implementations, rather than in the mechanics of execution. We briefly describe our approach to these mechanics for completeness, and it is likely that another team employing BDS would benefit from replicating the approach rather than using our tooling directly.

Example of some trivial EACs are given in listings 1.1 and 1.2[3].

```
@Eac("A created list is empty")
fun create() {
  whenever {
    createAList("list")
  }
  then {
    listContains("list")
  }
}
```

Listing 1.1. EAC for creating a list

```
@Eac("When an entity is added,
  it is contained by the list")
fun addToAList() {
  given {
    thereIsAList("list", "a")
  }
  whenever {
    addEntityToList("list", "b")
  }
  then {
    listContains("list", "a", "b")
  }
}
```

Listing 1.2. EAC for adding to a list

A single EAC is run using a custom JUnit runner; this means that all the usual JUnit tools are available for running from an IDE, a build tool, or a continuous integration environment. To run an EAC, the runner first identifies what implementations are available on the classpath and whether any constraints have been placed on the EAC. For

[2] We found it useful to make the distinction with the implementation too. Anaplan is a transactional system and it was useful for us to be able to create the declarative, initial context in a single transaction.

[3] Note that the keyword `whenever` is used in the scenario as `when` is a Kotlin keyword.

example, we might want to skip a particular implementation due to a known bug and use an annotation to indicate this. Once the list of available implementations has been determined, the runner will attempt to run the EAC against each of them.

The DSL is backed by an API that is implemented by an adapter for each implementation. The JUnit runner will use this API to convert the DSL into a series of implementation specific commands. There is not a one-to-one correspondence here as, particularly with the declaration, an adapter may choose to maintain a state and create a single command corresponding to several calls to the API.

The adapter is also allowed to return an 'unsupported command' from any API method – indicating that there is piece of functionality that has not yet been implemented, or perhaps which may never be implemented. For this reason, during conversion, the runner does not instantiate an implementation and none of the commands are executed. Only if all commands are supported does the runner actually execute them against the implementation, otherwise it can quickly skip the EAC without having to partially execute the scenario. This is extremely valuable for BDS where – unlike BDD – we are usually working with more than one adapter and development may not immediately follow specification. When the specification of a behaviour is complete, we can mark that behaviour as unsupported in the implementation's adapter until development can begin.

This capability is even more useful when working with multiple implementations — as in our Anaplan project. Consider the example in 2.3: we want to be able to specify both the dense and sparse behaviour of calculating 0^0, but clearly each implementation will only support one outcome. We can extend our DSL to be able to disambiguate the two methodologies[4] and write two EACs as shown in listings 1.3 and 1.4. Our two sparse platforms would return an 'unsupported command' for the EAC requiring dense arithmetic and would be skipped, but would be able to run the EAC requiring sparse arithmetic (and vice versa for the two dense platforms). Our specification would be able to support both arithmetic methodologies and evaluate both EACs. From a documentation perspective, this is ideal as we can place the EACs side by side, to compare and contrast the two behaviours.

```
given {
  thereIsAFormula("f",
    "POWER(0,0)",
    arithmetic = DENSE)
}
then {
  formulaEvaluatesTo("f", 1)
}
```

Listing 1.3. Dense EAC

```
given {
  thereIsAFormula("f",
    "POWER(0,0)",
    arithmetic = SPARSE)
}
then {
  formulaEvaluatesTo("f", 0)
}
```

Listing 1.4. Sparse EAC

3.3 Adapting to a specification

A requirement of BDS is the capability to create an adapter from the DSL to the specification, which requires a specification framework where animation is feasible. We considered a number of formal specification systems with suitable tooling, but – to reaffirm

[4] To prevent cluttering, we would only specify the arithmetic used in the EAC when it mattered and we would expect identical behaviour otherwise.

the arguments presented in section 3 of [13] – as engineers in industry we preferred the flexibility and familiarity of VDM [12] to more alien, proof-oriented methods such as Event-B [3].

VDM's Overture ecosystem provides a choice of IDEs that can animate a specification from the UI, while the published JARs enabled us to perform animation programmatically. Specifically, the capability to animate any function or operation from the specification was a significant boon as it enabled us to construct focused EACs rather than requiring a full system definition in every instance. This flexibility, and the maturity of the tooling, enabled us to build the specification adapter iteratively, doing 'just enough' to run new EACs. Having this capability enabled us to be fully agile in our process, and we believe that it made VDM an ideal choice for our project, and for BDS generally.

In [16], Oda illustrates how animation in VDM can be used to bridge the language gap between that of a project's stakeholders and that of a formal specification. The EACs used in BDS provide an extra layer of abstraction to provide a single language that can also be used by developers. Sections 4, 5, and 6 of [16] demonstrate how animation can be driven interactively through various interfaces, but for our EACs we took an even simpler approach. Our VDM adapter creates a module that represents the EAC, animates it, and then checks the final returned value to determine success. The module consists of a single operation that begins with all the declarations made in the *given* section of the EAC, followed by the commands corresponding to the *whenever* section. Each check in the *then* section is translated to a block that returns `false` if the check is failed. If the animation proceeds through all checks successfully then the EAC has passed and `true` is returned.

```
AnimateCheck: () ==> bool
AnimateCheck() ==
(
  (
    dcl formulaA: Formula`Formula := mk_Formula`UnsafeFormula([
      DataTypeLiterals`CreateNumberLiteral(0.0),
      DataTypeLiterals`CreateNumberLiteral(0.0),
      AnaplanFormulaFunctions`POWER_SPARSE_2
    ], {1 |-> [], 2 |-> [], 3 |-> [1, 2]});
    (
      dcl f: Formula`Formula := formulaA;
      let
        expected = ValueOption`Create[Number`Number](0)
      in
        if not TestEquality`Equals(FormulaModelling`Evaluate(f, { |-> }), expected)
        then (
          IO`println("* Check 'FormulaEvaluationCheck' failed (tolerant = true)");
          IO`print("    Expected: ");
          IO`println(expected);
          IO`print("    Actual: ");
          IO`println(FormulaModelling`Evaluate(f, { |-> }));
          return false;
        )
        else IO`println("* Check 'FormulaEvaluationCheck' passed");
    );
  );
  return true
)
```

Listing 1.5. A translated EAC specification

For example, the EAC in listing 1.4 generates the VDM operation shown in listing 1.5.

If an EAC fails, it is trivial to open the workspace in Overture and debug the animation of the specification using its rich tooling.

4 Integrating into the Agile Process

An agile project typically starts with the creation of user stories, which are then prioritized and placed into an initial backlog. This is usually led by a product owner (PO) who collaborates with customers to get a general feel for what a new feature should do and to gauge the importance of each of its aspects. As work progresses, the team then gets feedback from the customer and new user stories are created to refine and extend the behaviours of the features.

For our project we had a good idea of what the calculation engine should do — generally the same as the existing engine. The initialization of the backlog was therefore a little different as the PO was able to quickly identify the features required and rank them according to customer need[5]. Specification and development teams worked with the same prioritized backlog, but independently.

There was no need for specification to complete before implementation, if the specification team found bugs or missing behaviours they would simply feed into the backlog of the development teams. Similarly, the specification was not untouchable, if different choices made sense for the implementation, they could be discussed with the PO, and if approved the required change would be added to the specification team's backlog. In either case the first step would be to create an EAC that exemplified the expected behaviour, which would be referenced in the ticket created in the backlog and could be immediately committed with an appropriate annotation indicating that it was not supported by relevant implementations. The fluidity of the specification and the implementations meant that the EACs generated during BDS were considered to be the 'source of truth'. The specification was 'complete' when it satisfied all EACs in the system, and an implementation was 'complete' when it satisfied the subset of EACs that described its full behaviour[6].

As a project progresses, the PO and team groom the backlog on a regular basis and refine the features at the top of the backlog. For development work this usually involves establishing the set of AC required for the PO to mark the story as done. This was a little different for specification work as the ACs were an *output* of the BDS process. Instead, grooming involved identifying the functional elements required for the feature and establishing a broad set of behaviours for those elements. A trivial example, might be a list functional element – a data structure which can be used as a dimension of a cube – with behaviours such as creating a list, adding an item to a list, or moving an item within a list.

[5] Note that, although our intent was to provide feature parity, like all agile projects we wished to ship when we had a minimum viable product and incrementally broaden the feature set with regular releases thereafter.

[6] For this reason it was important to distinguish an action that was not yet supported from one that would never be supported

The initial set of behaviours would be established from a cursory inspection of the existing system and was created to give an idea of the work involved but, as normal in an agile environment, we would expect change as more detailed analysis and specification were completed.

It was not considered feasible to operate a methodology where sizing was required, as these size estimates are usually guided by the ACs. Instead, a lean methodology [17] was used, with an intent that a team member complete the specification of a single behaviour without interruption. We would imagine that this would be a choice made by all agile specification teams.

4.1 Specifying a behaviour

The general procedure of BDS has been detailed in section 3.1, while here we elaborate a little more on the process followed in our project.

The first step of BDS is to identify an AC. For example, the behaviour that allows adding an entity to the end of a list could be described as 'when an entity is added, it is contained by the list'. We would then write an EAC for this – as previously seen in listing 1.2. Here, we had the advantage that we could immediately run the EAC against the dense platforms to validate that we had captured the behaviour correctly.

We would then try to run the EAC against the sparse platforms. If the required functionality had not yet been implemented, we could ensure that the adapter returned the *unsupported* command. If the EAC ran and succeeded we could conclude that it has been implemented correctly, but if it ran and failed our next step would be to determine whether the difference was believed to be correct or incorrect. If the difference was acknowledged to be incorrect, we could raise a bug referencing the EAC and feed it into the appropriate team's backlog. On the other hand, if the development team believed that they had implemented the correct behaviour, we would raise a ticket in a formal difference backlog. At this point we would mark the EAC as 'under review' until the difference was discussed by a working group at a fortnightly meeting, where all the stakeholders would discuss differences and make an informed decision on whether the proposals should be accepted or rejected[7]. Once the difference was resolved, it would be necessary to return to the EAC and ensure if reflected the decision made.

The specifier then runs the EAC against the specification to ensure it fails, the EAC is marked as unsupported by the specification, and the author pushes the work to a branch and raises a pull request (PR) for review by another team member. Once the review of the PR is complete – and all necessary builds have passed – the EAC is merged to the mainline, and specification can proceed.

There is not too much to say about writing the specification itself, other than to reinforce that the only changes should be those required to make the EAC pass, and that the presence of EACs does not divest the author of the responsibility to write units tests within the VDM, which should focus on testing the correctness of each individual function, as well as its pre and post conditions. Having the EACs in place should discourage

[7] Note that, as this was an agile process, there was always a possibility that a future discovery could invalidate the decision, and members of the team were free to reopen the discussion and present new evidence to the working group before a new decision was made.

the specifier from writing end-to-end tests directly in the VDM, and instead ensure the focus is on good coverage of the specification.

4.2 Continuous validation

An important part of agility is continually integrating the pieces of work from all parts of the team to ensure that changes do not cause problems elsewhere. The Gradle plugin introduced in [10] enables us to build the specification, run all the VDM unit tests in Jenkins, and publish the specification to Artifactory.

Change to the DSL and its associated adapters are managed like any other agile development work – with frequent commits, PRs, and adoption upstream. The implementation adapters leverage Java's service loading mechanism to use whichever version of the implementation they find on their classpath, while the VDM adapter was necessarily tied to a specific version of the specification, since part of its function was to package the text-based specification into a Java library that could be 'run'.

The EACs are all placed into a single validation repository, and whenever the mainline of this project builds successfully, it also checks for newer versions of the DSL, the adapters and the implementations. If a new version is found, the system automatically raises a PR to update the version numbers of the dependencies, and if this PR builds successfully, it is considered a team priority to merge it quickly. On occasion, a version update PR will fail, typically due to either a regression of behaviour in an implementation, or because some previously unsupported action has been implemented, causing an EAC that was previously being skipped to fail. At this point we follow the same process that we use when a new EAC fails, as described previously. Again, resolving this difference is considered a team priority. As with any agile development team, team members will work on a failing mainline or version update build – rather than regular assigned work – until it is fixed.

5 Results

Behaviour driven specification was not a fully-formed concept when we began, but – like all agile processes – has evolved as the project progressed. We believe that the use of BDS not only has enabled us to deliver more efficiently, but has produced a more maintainable specification where behaviours are encapsulated and decoupled. In total, we have used BDS to create more than 60,000 lines of VDM-SL, of which 26,000 lines are the 231 modules that form the specification. There are also 115 modules of VDM-based unit tests, containing more than 3,000 tests and comprising 34,000 lines.

Our specification is split into multiple projects and we use the Gradle plugin presented in [10] to manage dependencies between them. This enables us to apply a greater degree of componentization than VDM-SL provides out-of-the-box and prevents specifiers accidentally using information that they *should not* know about. For example, we have a 'hierarchical data' project that has a dependency declared on an 'ordered data' project. When working in the hierarchical data project, the specifier can use modules from the ordered data project, but when working in the ordered data project the specifier cannot use modules from the hierarchy data project. The underlying dependency

mechanism is the same one Gradle uses for Java; dependencies are acquired transitively without redeclaration and cyclic dependencies are forbidden.

Our specification is split into 54 VDM-SL projects that are all depended upon – directly or transitively – by a 'system' project, which is published to the company's Artifactory instance and used by the VDM adapter for running the EACs.

We additionally have seven library projects which provide Java implementations for VDM modules. For the most part, these provide speedy implementations for generic functionality that is not specific to our problem. For instance, we have libraries for IEEE-754 floating point, ISO-8601 dates, and for standard graph algorithms. We also used JNI to create a Rust-based library to overcome Java's limitations with Unicode characters outside the Basic Multilingual Plane.

In hindsight, we should have used libraries more heavily to provide the execution mechanisms for our functions, and focussed on solely writing VDM-SL pre and post-conditions. As we added more and more EACs, and particularly as we generated larger scenarios, the speed of animating the specification became restrictive, and we had to make compromises in our approach; this was particularly true when a scenario had a large memory footprint and garbage collection became a significant proportion of the runtime.

One reason that we did not completely embrace library modules earlier in the project was that we already had fully-tested VDM from a previous project, which we were not keen to throw away. What we would like to have seen is the capability to use a library with a module where function definitions were provided with a runtime flag used to select whether to use the VDM definition or the library definition. If that were available, we would likely use the VDM definition in local work and the library definitions in the CI environment.

At time of writing we have approximately 3,000 EACs that are run against each of the four alternative platforms whenever a commit is made to the validation repository, giving us confidence that each continually meets the defined system requirements. Once EACs are published they are also run against each of the platform implementations when commits are made to their respective repositories. As alluded to above, running the same set of EACs against the specification on every commit has proved to be too slow. However, we run these in a perpetual loop on a dedicated cloud instance and get immediate feedback if and when an EAC fails. We have found that regressions are very rare in the specification, but this process is useful for capturing unintended consequences when requirements change in a related area. We have also tended to tighten preconditions, postconditions and invariants as the project has progressed, and this continuous validation process has enabled us to quickly spot issues that low-level VDM unit tests did not catch.

As with behaviour driven development, creating a DSL and relevant adapters is a necessary part of BDS. Although we had five adapters to maintain, it was not as onerous as it might seem: much of the complexity in Anaplan is in the point transformations that are possible within the geometry of the cubes. These transformations are expressed in Anaplan's formula syntax, and early on in the project we established conventions within the adapters for automatically mapping keywords into functions within the specification. We introduced a special module – `AnaplanFormulaFunctions`

– into our specification which served as an 'interface' for these keywords. For instance, in listing 1.5, we can see that the formula keyword from listing 1.4 has been converted by convention to `POWER_SPARSE_2`[8]. The specifier creates a value in the `AnaplanFormulaFunctions` module that is an alias to the actual definition that they can specify in the most appropriate location. Similarly, the VDM adapter runs a pre-build process to generate stubs for all exported parts of the specification, so that we can ensure compile-time consistency between the specification and adapter. The four implementation adapaters were all based on the same abstract adapter and only varied where they supported different capabilities or where the code of the alternative platforms was not consistent.

One of the most important outputs of the specification process has been the identification of differences between the sparse and dense calculation engines. Nearly 200 differences have been captured by the specification team, and it is highly unlikely that they would have been discovered by other means. Each of these has been fully discussed and an informed, justified decision has been made. These differences can now be fully documented *before* a customer encounters them, so when they do they will not only be able to see that the difference is intentional, but they will be able to view the reasoning behind it.

For the members of the specification team, BDS has had another benefit. We have grown the size of the team from two to seven during the course of the project, with new members having varied backgrounds. BDS enforces the breakdown of work into small, distinct chunks, which we have found to enable new members of the team to make meaningful contributions early in their tenure. It is reasonably easy for anyone to learn to analyse a behaviour and write EACs quickly, and doing so gives new members confidence to alter the specification – they really understand the change needed even if they do not understand the whole specification. Similarly, it allows a more experienced specifier to move more easily to an area of the specification they have not worked with before – they can select some simple EACs and run through how they are handled by the specification, then add some basic EACs for their new behaviour. The EAC-driven workflow really helps comprehension of the specification, even for those who already have VDM expertise.

6 Future work

We have begun to use the tools developed for BDS to assist with verifying our implementations. We are constructing a verification framework that generates semi-random scenarios from the instructions available in the DSL and runs them against target systems. These scenarios are slightly more expressive than those used for validation. For example, it is possible to use a query to determine the valid points in a cube and then check that the value at each point in the target system satisfies the specification.

Briefly, the verification process proceeds as follows:

[8] As it has two arguments and uses the sparse arithmetic. The number of arguments is not important in this instance, but is needed as Anaplan formula syntax allows overloading, but VDM-SL does not.

1. Generate a scenario.
2. Check that the *given* and *whenever* blocks satisfy the specification – that is, that the scenario satisfies the precondition and that the command make sense.
3. Run the scenario against the implementation in a mode that queries rather than checks.
4. Use the results of the query to generate an equivalent scenario containing a *then* block, and run that scenario against the specification.

At the time of writing, we are continuing to develop the verification framework, and we are also scaling up its use in checking the correctness of the alternative calculation engine. Currently, we have only preliminary results, which are not yet ready for publication.

7 Concluding remarks

We started using VDM at Anaplan before beginning the project described in this paper, but the focus had been on investigating new modelling concepts rather than the delivery of working software. With the switch to a delivery-focus, we knew we needed a process that could align with the workflow of the development teams assigned to the project. Behaviour driven specification proved to be such a process, enabling us to create a formal specification *with* the development teams rather than *for* them.

In the initial phases of the project, there was a suspicion that formal methods were intrinsically linked to waterfall development. This worried developers who believed they would lose the ability to influence the system's requirements. There was a belief that the specification team would form a barrier between the product owner and the developers, and at early meetings we needed to remind the wider group that the role of the specification team was to record decisions, not to make them. We believe that the use of BDS helped to overcome this: not only because it has enabled the specification process to be more flexible, but also because it has made the specifiers feel like a collaborative partner in the implementation effort rather than a siloed team issuing diktats to developers. The use of executable acceptance criteria has reinforced this; having a 'source of truth' expressed in a language that requires no special understanding enables all members of the project team to engage on an equal footing.

As we approach our first customer-facing release, the feeling across the whole project team is that the integration of formal specification into the agile process has been a success, with specifiers, developers, and product owners all highlighting the value it has provided.

References

1. Airline and location code search. `https://www.iata.org/en/publications/directories/code-search/`, accessed: 2020-06-11
2. Economic performance of the airline industry. `https://www.iata.org/en/iata-repository/publications/economic-reports/airline-industry-economic-performance-june-2020-report`, accessed: 2020-06-11

3. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
4. Beck, K.: Test-driven development: By example. Addison-Wesley Professional (2003)
5. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: Manifesto for agile software development (2001), http://www.agilemanifesto.org/
6. Codd, E.F., Codd, S.B., Salley, C.T.: Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Tech. rep. (1993)
7. Codd, E.F.: The relational model for database management: version 2. Addison-Wesley Longman Publishing Co., Inc. (1990)
8. Cohn, M.: User stories applied: For agile software development. Addison-Wesley Professional (2004)
9. Drepper, U.: What every programmer should know about memory. Red Hat, Inc **11**, 2007 (2007)
10. Fraser, S.: Integrating VDM-SL into the continuous delivery pipelines of cloud-based software. In: The 16th Overture Workshop. pp. 123–138 (2018)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
12. 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)
13. Larsen, P.G., Fitzgerald, J.S., Wolff, S.: Are formal methods ready for agility? A reality check. In: FM+AM 2010: Second International Workshop on Formal Methods and Agile Methods,. Newcastle University (2011)
14. Lenz, H.J., Thalheim, B.: A formal framework of aggregation for the OLAP-OLTP model. J. Univers. Comput. Sci. **15**(1), 273–303 (2009)
15. Macedo, H.D., Oliveira, J.N.: A linear algebra approach to OLAP. Formal Aspects of Computing **27**(2), 283–307 (2015)
16. Oda, T., Yamomoto, Y., Nakakoji, K., Araki, K., Larsen, P.G.: VDM animation for a wider range of stakeholders. In: Proceedings of the 13th Overture Workshop. pp. 18–32 (2015)
17. Shalloway, A., Beaver, G., Trott, J.R.: Lean-agile software development: achieving enterprise agility. Pearson Education (2009)
18. Shore, J., et al.: The Art of Agile Development: Pragmatic guide to agile software development. O'Reilly Media, Inc. (2007)
19. Subramaniam, V.: Programming DSLs in Kotlin. Pragmatic Bookshelf (2021)
20. Wagenaar, G., Overbeek, S., Lucassen, G., Brinkkemper, S., Schneider, K.: Working software over comprehensive documentation–Rationales of agile teams for artefacts usage. Journal of Software Engineering Research and Development **6**(1), 1–23 (2018)
21. Wynne, M., Hellesoy, A., Tooke, S.: The Cucumber book: Behaviour-driven development for testers and developers. Pragmatic Bookshelf (2017)