CSC3094: Dissertation

A literate programming and animation environment for VDM

Henry Thomas Hughes (190227241)
Supervisor: Dr Leo Freitas

0.1 Abstract

Modern approaches to software development have changed significantly since VDM (the Vienna Development Method) was first proposed. This project explores the implementation of two of these newer features: a literate programming environment (akin to the Jupyter Notebook system) as well as graphical output from and interaction with a model. Its findings show that VDM is infeasible for a Notebook environment for a number of key reasons, primarily VDM not being designed as a shell language, but also its lack of a true line-by-line interpreter. There is significant success shown in providing animation and interaction support for VDM models.

0.2 Acknowledgements

I would like to thank Dr Leo Freitas for his guidance and assistance in this project, providing key expertise and insight into all aspects of the work that needed doing. I would also like to thank my parents and especially my mother for proof reading all my work and telling me when I need a comma.

0.3 Declaration

I declare that this dissertation represents my own work except where otherwise stated.

0.4 Note on repositories

During the writing of this report, all repositories were kept private to ensure validity of the above declaration. As of the submission of this report, these repositories will be made public and their URLs will be included in the Appendix. There may be work done after submission on these so I have linked to the specific commit I submitted at.

Contents

Introduction

1.1 Motivation

VDM (the Vienna Development Method) is an established formal method, used primarily in digital systems modelling. Whilst VDM is technically the specification, it is commonly used to refer to the languages through which it is implemented. A separate category of languages is required for this purpose, as normal programming languages are not set up to generate formal proofs. VDM has been in development for at least 30 years and has had various evolution's in that time.

Formal specifications and modelling can be intimidating to learn, I know this personally from the process of writing this project. Even with a solid grounding in other programming languages, the purpose of these models can be hard to understand or grasp initially. This project aims to make them more approachable for all stakeholders in any model, some of whom may be new learners, by providing intuitive interaction with VDM models both in the development and demonstration stages.

One of the key requirements for this is a design that will work for whatever model is being developed, be it a system that lends itself to visual output , or one that needs data visualisation on its performance or outcomes etc.

The first challenge is to enable GUI interaction with VDM models, allowing live animations and interaction. The method outlined in my proposal is to write a system to allow VDM code to run within a REST API, as well as hosting web content from that REST API. This yields benefits such as:

- Allowing near-universal options for VDM frontends.
- Enabling VDM models to be inserted into web-based infrastructure.

The second component is an adaptation of the popular Python development environment, *Jupyter Notebook*[1]. This environment changes the normal process of development from plain text files containing both source code and plain text comments into a JSON-based file format, storing separate cells as JSON objects that have their language identified alongside.

Jupyter Notebook offers the following benefits for development:

- Isolating code elements and comment/documentation elements into separate cells means that comment cells can be rich text.
  - Separation allows users/stakeholders who cannot read code to gain insight from rich text
  - Rich text enables images and formulae to be presented clearly, which plain text comments cannot achieve

[1] Jupyter Notebook available at https://jupyter.org/

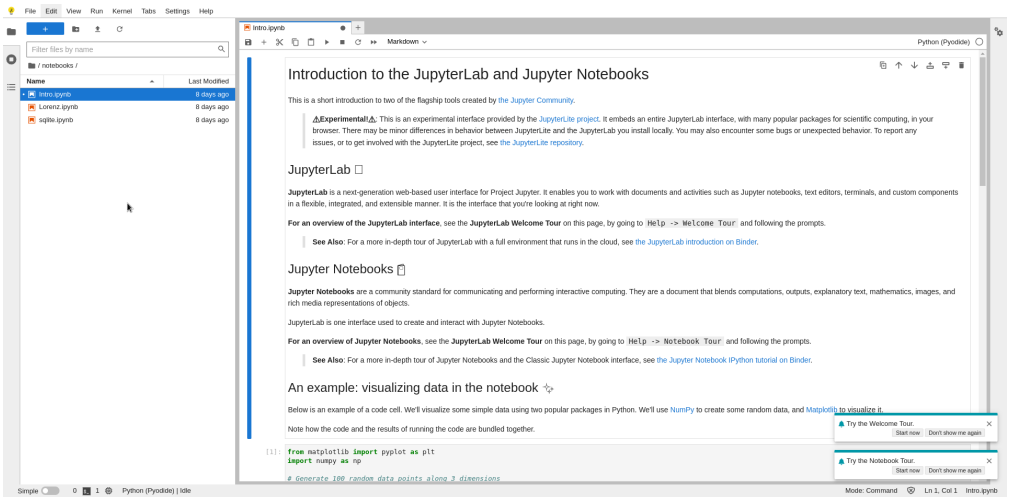

Figure 1.1: Jupyter Notebook in the Jupyter Lab web app

- Allowing complex output, not simple shell or opening a window which would require OS integration.
  - Jupyter Notebook allows output of images and animated GIFs
  - The addition of *ipywidgets*[2] user input to rerun blocks with different variables makes it possible to have user interaction
- Persisting the environment between code cells so they can be rerun.

A key inspiration for the animation environment being produced was the ProB Animation tool (Leuschel et al. 2023) which offers similar functionality to that required by an animation environment for VDM.

[2]ipywidgets available at https://github.com/jupyter-widgets/ipywidgets

1.2 Structure of project

As this project essentially consists of a frontend and backend, the backend had to be developed first in order for the frontend to have a backend to test against. Therefore I began by constructing a REST API based RPC (remote procedure call) system.

I chose to write the backend in Java and, in order to use the VS Code extension API, had to write the frontend in TypeScript.

The project proposal contains a Gantt chart outlining the time-line breakdown of each component required for the project. This can be seen in Figure 1.2,

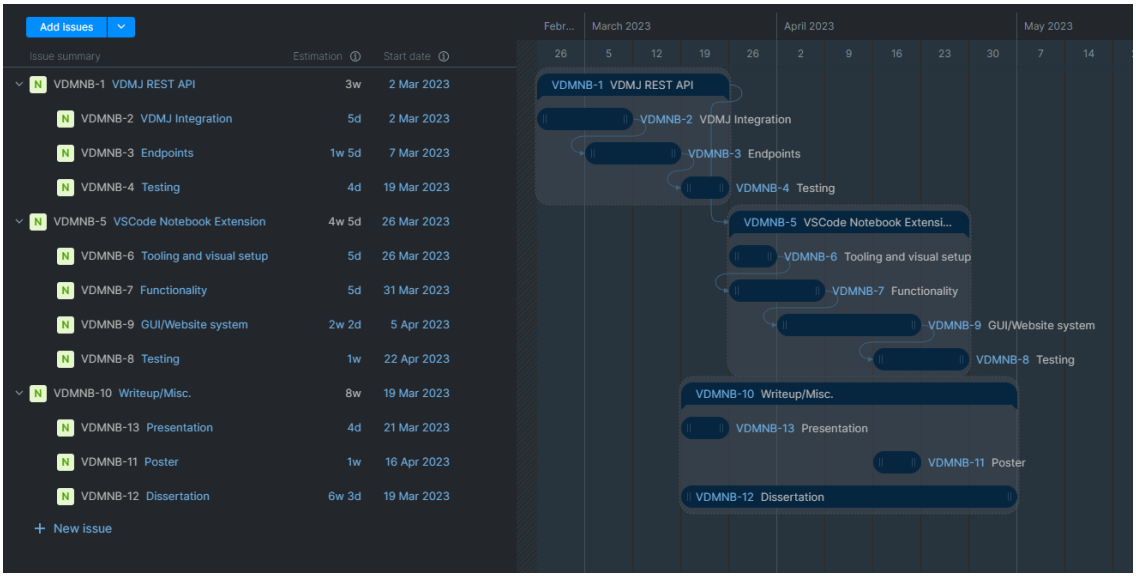

Figure 1.2: Gantt Chart

This Gantt chart is somewhat out of date as the time taken to produce the REST API was far more than that taken to produce the notebook extension. It also does not feature the time taken to produce a working example project to test against. This took approximately a week of work, although some was already completed with the prototype.

Included with this report are three project folders, the backend in `vdmj-remote`, the extension/frontend in `vdm-notebook-extension`, and the Conway's Game of Life example project in `ConwayWeb`.

1.3 Objectives and Requirements

My overall objectives in this project are as follows.

VDMJ Remote : To produce a stable REST API backend that can be used across all types of VDM model and can be run remotely on most commonplace system configurations (both server and personal).

1. Must provide access to all functionality normally accessible with VDMJ
2. Must handle known exceptions and prevent unexpected crashes
3. Should output universally recognised computer-readable formats[3]

[3]For example JSON or XML

4. Must allow future modification into a socket/IO stream-based true RPC system.

VDMJ Remote Tests : To provide a test suite for item 1, allowing test-driven development of future necessary changes to the system to maintain parity with underlying dependencies.

1. Must test all routes of REST API at minimum
2. Should be integrated into CI/CD[4] system to ensure testing before merges/releases
3. Ideally should test packages within the project in isolation.

VDM Notebook Extension : To produce a usable VS Code extension to enable the backend to execute VDM code from a notebook-file format, and to allow output of animations hosted as web content on the backend.

1. Must be able to write all of the VDM Dialects
2. Must allow web output for animations
3. Must handle known exceptions and prevent unexpected crashes
4. Must relay to user any pertinent information provided from the backend.

The objectives above are listed in the order they must be completed. In Figure 1.2, the timeline for the backend and frontend components development cycles is shown.

Additionally, the software produced needs to be installable on any system on which VDMJ and VS Code can both be installed. This should not be difficult to complete as VDMJ is written in Java which is purpose-built to be executable on any system. VDMJ-Remote is also written in Java for compatibility with VDMJ. The VDM-Notebook-Extension is written in TypeScript, which is based on Node/JavaScript - the same framework/language as VS Code is written in. However, this element of the project required some workarounds for different systems.

[4]Continuous Integration(as laid out in Booch 1994)/Continuous Delivery

Background Review

2.1 Note on references for this project

As the core of this project is the production of a set of tools, primary sources are mostly relevant for their effect on the decisions made during the tools' design. A great deal of research has been undertaken on feasibility of different design plans - references for this aspect of the project are mostly non-academic so they are listed as footnotes.

2.2 Sources and References

2.2.1 VDM

The early history is well documented by some of the archived lecture notes from conference proceedings (Lucas 1987).

Currently VDM is mostly used via the interpreter package VDMJ[1]. This tool is written in Java, it includes "a parser, a type checker, an interpreter ... a debugger, a proof obligation generator and a combinatorial test generator"[2]. It supports three dialects (P. G. Larsen, Lausdahl, and Battle 2010):

VDM-SL: The specification language dialect, the default VDM dialect. This is made up of modules containing defined values, types and free functions (Plat and P. Larsen 1994). It is single threaded.

VDM++: The object oriented dialect. VDM-SL is similar to C whereas VDM++ has objects like C++, hence its name. It also allows multi-threaded execution (Battle 2023b)

VDM-RT: The real-time dialect, built for modelling real-time, embedded and distributed systems (P. G. Larsen, Lausdahl, and Battle 2010).

Overall, the most useful source for understanding current VDM practices and implementing this project was the VDMJ User Guide (Battle 2023b). This provided much of the information necessary for implementing both the backend VDMJ interactions and the frontend user interface.

2.2.2 ProB

A key inspiration for this project was the *ProB Animator* (Leuschel et al. 2023) for *B* (Abrial 1988). The B-Method is based on B, a formal modelling language akin to VDM. The ProB Animator provides animation facilities for models written in B. This allows visualisations based on these models to be examined/explained, for example the N-Queens example available on their website[3].

[1] VDMJ, the VDM interpreter available at https://github.com/nickbattle/vdmj

[2] Taken from the VDMJ README.md available on GitHub

[3] N-Queens animation example for ProB available at https://prob.hhu.de/w/index.php?title=N-Queens

The ProB system provides a solid example of how to animate atop a formal model. However, instead of the system of annotations used in this project, the ProB requires exported `DEFINITIONS` according to predefined names. Additionally, there does not seem to be a literate programming environment available for B.

2.2.3 Jupyter Notebook/Literate Programming

Jupyter Notebook[4] is an industry tool without significant academic resources. However it is based on the initial idea of Literate Programming as laid out in Donald Knuth's seminal text on the topic (Knuth 1992).

Despite this, a large portion of the initial research I had to do for this project was focused on the feasibility of different approaches to extending Jupyter for VDM. In the end the time frame was insufficient for completion of this, but the Notebook system still provided good information for how to implement certain features.

2.2.4 Visual Studio Code and the Notebook API

The approach adopted to produce a literate environment for VDM was to develop an extension for Visual Studio Code[5] (VS Code). This initially seemed to be well documented and simple to do as the online documentation for the Notebook API[6] is well written if sparse on some details. However, as I discovered through the development of the extension using this API, the documentation is lacking key details about important aspects such as exceptions.

[4] Jupyter Notebook available at https://jupyter.org/
[5] VS Code available at https://code.visualstudio.com/
[6] VS Code Notebook API available at https://code.visualstudio.com/api/extension-guides/notebook

Methodology

3.1 Software Engineering techniques

This project is effectively the production of a tool for a client. In settling on this project it was implicitly envisage that behaviour driven development (BDD) would guide the process. However, to allow future development, tests were added for the backend project. Testing the frontend, by contrast, is a mostly manual process and thus requires purely behaviour driven development.

All code development was written in either IntelliJ IDEA[1] (VDMJ-Remote) or Visual Studio Code (VDM-Notebook-Extension). The former is a very effective Java IDE offering a great deal of insight. For example, it enabled me to analyse the memory allocation of the original Spring-based prototype. This uncovered that the cause of VDMJ-Remote's large memory footprint was the use of the Spring framework, and prompted the switch to Spark (as detailed in section 3.4.1). VS Code was necessary in order to develop the extension, both because VS Code extensions are usually written in it and because it has the Overture extension for VDM code highlighting/analysis.

I used the Maven[2] build tool for VDMJ-Remote, allowing me to download the latest VDMJ release directly as a dependency rather than manually including the required JAR. It also enabled a standardised system for testing, using JUnit and a Maven plugin to include testing in the build process.

The codebase was managed using Git, the ubiquitous version control system. This allowed use of GitHub's CI/CD as part of the testing for the backend and in order to release versions that can be used as Maven dependencies.

3.2 Planned approach

The first stage of this project was quickly producing a prototype of the desired result. The approach I took for this prototype provided the basis for the plan I outlined in my proposal.

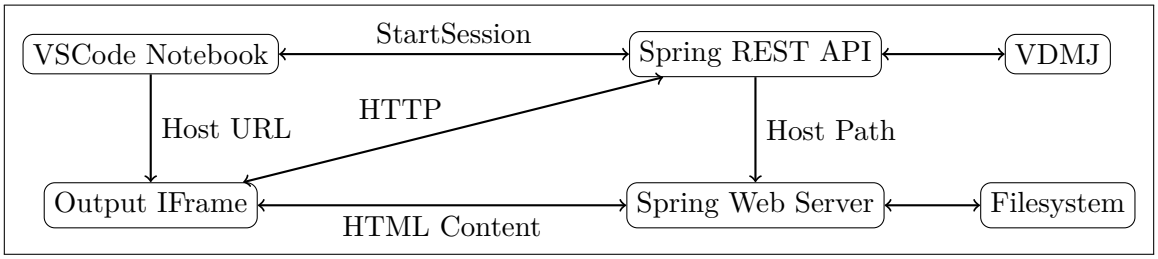

Figure 3.1: Systems diagram, as included in my project proposal

[1] IntelliJ IDEA available from JetBrains at https://www.jetbrains.com/idea/
[2] Apache Maven available at https://maven.apache.org/

3.3 Actual approach overview

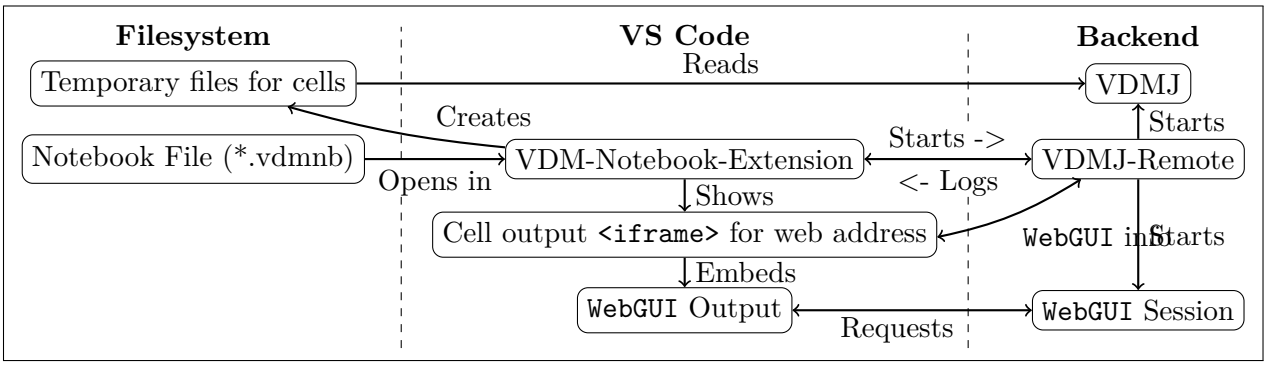

Figure 3.2: Broad systems diagram

Figure 3.2 shows a broad systems diagram for the whole project. Complex communication between components inside the same system (for example VDMJ-Remote and `WebGUI` Session both communicating with the VDMJ instance via the `VDMJHandler` class instance) has been omitted for clarity and simplicity.

The file extension I chose was `.vdmnb` standing for VDM Notebook. VS Code should automatically detect the association with the extension provided. However, the operating system may need informing of the association with VS Code.

3.4 VDMJ-Remote development process

3.4.1 First steps

The first step on my Gantt chart (Figure 1.2) was to develop the REST API for VDMJ. This started with further work on the prototype.

The first deviation from the plan in Figure 3.1 was to switch from the Spring Framework[3] to Spark[4]. Both of these frameworks are used to provide REST API routes and helpers, however Spark has a lower memory overhead[5]. I made the choice to change after discovering that launching multiple versions of the original prototype quickly allocated an unfeasible proportion of average memory availability. For example, using my personal computer as a base line, with the 8GB of memory available, each Spring-based implementation would take up approximately 4% of the total. With operating system, IDE and other demands taken into account this becomes more like 10% of the operating memory per instance.

This change required me to learn some new features and search for Spark equivalents for the Spring code. For example, Spark does not automatically serialize Java objects to JSON if returned, they need to be manually 'stringified' or managed through a Jackson[6] `ObjectMapper` instance.

There are a few other key differences between Spring and Spark, but many of these differences actually helped with the design decisions. For example, where Spring uses the `Controller` annotation interface to define classes containing routes that can be used as endpoints, Spark uses closures provided to either a static or provided class instance (`Service`). This made moving to an inherited

[3]Spring framework available at https://spring.io/
[4]Spark Java can be found at https://sparkjava.com/
[5]Comparison from https://craftsmen.nl/memory-usage-6-popular-rest-server-frameworks-compared/
[6]The Jackson JSON serializer/deserializer available at https://github.com/FasterXML/jackson

service class system (detailed later) far easier as each class can call its `super` function to populate the `Service` with the required endpoints/configuration.

During this time I also noted the implicit security concern of this RPC system. Essentially any security flaws in VDMJ (which is not tested for security vulnerabilities) could be carried to the network interface. I therefore set Spark to allow requests only from the local system by binding the server to `127.0.0.1` in configuration. This can be changed by modifying the code and recompiling, but without necessary security measures (for example user/password authentication or a token system) I would strongly discourage this.

Finding an example model for testing was also key at this stage, an example of an existing system to emulate. So I opted to adapt the VDM Toolkit ConwayNB example as it already had a simple animation made for it.

3.4.2 Using annotations

Animation outputs are communicated to the VDM backend by including an annotation. In order for this to work, I had to extend VDMJ with an annotation as laid out in the annotation guide (Battle 2023a). The annotation for an output is `-@WebGUI("<nickname>", "<path>")` where `<path>` is replaced with a path on the system (relative to the current working directory or absolute) to a folder with static web content. By default this folder should have an `index.html` but can also have JavaScript and CSS. The `<nickname>` parameter can be used to give a meaningful label to the tab for this output.

To do this I had to write two classes to enable VDMJ to pick up the annotation when used, these are called `ASTWebGUIAnnotation` and `TCWebGUIAnnotation`. This caused an issue as these are mapped with a `ast-tc.mappings` file which already exists in VDMJ: when compressed into a JAR only one of the two is included. The only solution I could find to this problem was to use a shaded JAR, a system by which files can be merged or appended in certain ways into the JAR to prevent conflicts. To address the issue, I simply appended my mappings onto the VDMJ mappings file at compile time.

In order for the REST API to pick up all of the registered outputs, I created the `RemoteOutputRegistry` class to statically store all of the information provided with each annotation.

This is another possible area for future work, the ProB (Leuschel et al. 2023) system uses variables similarly, but also to define the function being called each step for animation. It would be possible to do this with annotations for this system, although it may not be useful given that the whole range of functions/commands is available for execution by the animator.

3.4.3 Improving VDMJ integration

One of the initial issues I encountered with the interactions with VDMJ was that the `RemoteInterpreter` class would require recreation every time code was added. This caused issues with the notebook system, and also meant that I would have to regularly update whenever the VDMJ interface was changed. For example, during development of this project, VDMJ was modified to switch its main function to the new plugins-based approach.

Usefully, VDMJ allows replacement of its IO streams, which allowed me to replace them with my custom `PairedPipedIOStream` instances to simulate console IO. This emulates user interaction with VDMJ via a terminal.

The method by which the application communicates with VDMJ is not perfect by any means: it has to parse the printed characters until the `>` prompt is output. There could be many situations in

which this sequence of characters is printed erroneously causing a truncated output to be given to the client, but I could not come up with a quick solution to this.

Parallel Execution

I took several steps to make the VDMJ integration effective for parallel requests, using a system to queue requests to prevent race conditions. I anticipated this as a problem: REST APIs are designed to handle a certain number of requests simultaneously and this could obviously cause issues.

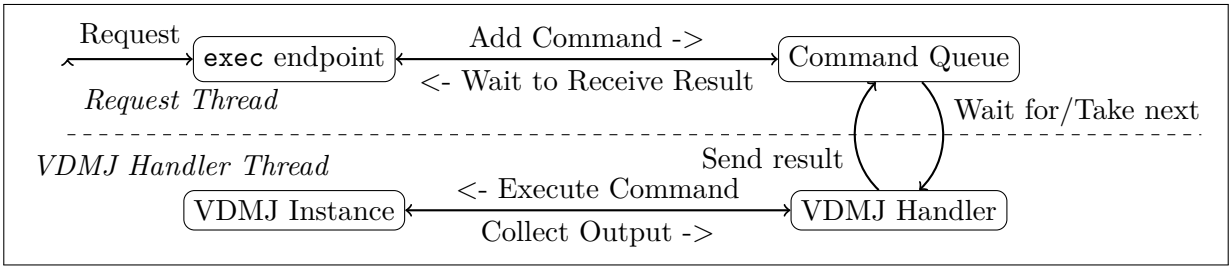

Figure 3.3: Parallel system for command execution, preventing race conditions

Figure 3.3 shows the parallel system and is divided between the thread created by Spark for each request and the single thread managing interaction with VDMJ. Interactions are completed via a `LinkedBlockingQueue` from the Java standard library. This prevents issues with parallel requests in all the cases I have simulated, but I have not modelled it or stress tested it so it may require modification in the future for edge cases or larger demand.

An issue is that this approach does not allow for interruption. If a command is sent that either does not terminate or should be terminated by user decision, then the process should discard the progress and the instruction. However, in my design the system simply continues until the overall process is killed. This was an oversight in the systems design and is addressed in *Future Work*.

Due to the multi-threaded requirements of the multiple REST API instances and this multi-threaded approach, the application will spawn a lot of threads that could all receive activity in a complex use case. I have therefore used every possible care to implement blocked logic instead of busy waiting, so threads should sleep until they need to process some request or command.

3.4.4 Developing the REST API

Essentially, the development of a REST API is akin to designing a library or interface. Endpoints are like functions/methods in a library, although typically they are purely used to get data from a database or similar. Therefore this section of the project mostly involved choosing what endpoints to create.

The first and most important endpoint was the `exec` endpoint. I designed this to receive commands as if they are typed rather than any complex data model, due to the primary purpose being terminal-like input. My initial approach was to give a simple text output as printed from VDMJ, however this was quickly proven to be an ineffective approach as the backend needs to provide information in the event of an error. Therefore I adopted an output data model as exemplified in Figure 3.4

If an error is discovered then the key `error` will be included as a true boolean so that a client can easily tell if the server encountered issues processing the request.

During this time I discovered a few key errors were being thrown. Their messages should be carried to the client as an error response. So I wrote a custom `Exception` called `SessionException` to be caught as a non-fatal error, passing it's message as a response to the client without the server

```
{
  "id": "d75b31fc-fa12-4034-a409-2641ae641eac",
  "command": "p tests()",
  "response": "= [true, true, true, true, true, true]\nExecuted in 0.316 secs. \n",
  "requested": "2023-05-10T14:30:38.556785850Z",
  "queued": "2023-05-10T14:30:38.873680629Z",
  "executionStart": "2023-05-10T14:30:38.873739973Z",
  "executionFinished": "2023-05-10T14:30:38.873964888Z"
}
```


Figure 3.4: Output format from the `exec` endpoint for command `p tests()`

needing to shut down. This is automatically handled by Spark to provide meaningful output and the correct HTTP error code.

The next endpoints to develop were ones for handling `WebGUI` output starting and stopping, these are:

`outputs`: To list the outputs in a JSON format with information about the module they are from: these JSON objects are what the next two requests require for operation.

`startOutput`: For starting a session, taking one of the JSON objects from `outputs` and starting that as a new session if it isn't already running, returning the hosting info either way.

`stopOutput`: To stop a running output, returns a `SessionException` on issues. This is necessary so that a client could gracefully shutdown its started sessions but leave the server running.

It was at this point that I began to test the REST API with the prototype frontend that I had developed. This threw up an immediate but simple problem to fix, that of CORS. CORS (Cross-Origin Resource Sharing) is a system to allow requests from other domain names to webservers either conditionally or universally. It is used primarily to prevent Cross-Origin requests from pages using a client's browser. In this case, I disabled it universally as requests need to be allowed from whatever domain name VS Code uses for each page. I did this by including a filter on all requests to send back the CORS policy allowing all hostnames.

The other endpoints that needed to be created were:

`reload`: Causes the VDMJ instance to be reloaded, picking up changes to scripts and deleting the current environment.

`startup`: Gets the string printed by VDMJ on startup, so that the 'terminal' shown in the extension would show this information as normal.

The `reload` endpoint had to be created to pick up script changes, however it causes any state modifications in the running session to be cleared. This is another issue that cannot be solved without significant modifications to VDMJ.

Creating an extensible session system

Due to the requirement for dynamically hosting content on this backend, Spark sessions need to be created on different ports to host the static web content as specified in the VDM annotation. My initial approach had a significant amount of code duplication, so I moved it to a system of object inheritance starting with the minimum requirements for all `OutputSessions`.

The `VDMJOutputSession` class (inheriting from `OutputSession`) represents all sessions that require access to the `exec` endpoint to run commands on VDMJ. The `MainOutputSession` class (inheriting from `VDMJOutputSession`) represents the main session, outputting the web CLI and managing other sessions.

This system allows future possibilities for other types of sessions. However, as will be discussed in the *Future Work* section, I have a different proposal for enabling output from VDM.

3.4.5 Finalising initial VDMJ-Remote version

There were a few further steps I took to produce a usable version of VDMJ-Remote. These primarily involved issues discovered by testing with the prototype VDM-Notebook-Extension. These issues often arose from the specific eccentricities of the VS Code Notebook API. For example the first system for handling files was required due to the Notebook not identifying individual cells uniquely.

IPC

In the initial system I designed, the VDM Notebook Extension communicated with the child VDMJ Remote instances purely via reading its command line log. This is a bad system as it requires constant parsing and checking against *RegEx* patterns to ensure the backend hasn't encountered internal errors. After reading more into how Jupyter Notebook handles its kernels[7], I discovered that it uses sockets operating with the ZeroMQ[8] messaging library.

This prompted me to see how adaptable my design would be to using socket-based IPC for communication with the notebook. I managed to implement a somewhat effective IPC system over sockets, but ran out of time before I could make it comparable to the complexity required by for a Jupyter kernel. However, it was not a difficult system to implement and further work on this may be possible as detailed in *Future Work*.

Files

One of the primary issues with my implementation is that VDMJ doesn't allow more VDM code to be added dynamically. This causes significant incompatibilities with the notebook system as by default code cells must be runnable and re-runnable or not run at all.

Initially, when using the Spring version of VDMJ-Remote, I approached this problem by using a file upload over HTTP. The server would then manage code files as it saw fit, loading them and replacing them as requested by the upload facility. I came across an immediate issue with temporary files not being deleted on the process exiting. This is a problem for the use case as it could be that the VDMJ-Remote instance is restarted causing more files to be created, slowly encroaching on more and more disk space. Additionally, this approach produced significant bloat to the frontend. It also necessitated a whole set of endpoints that allow the backend to receive instructions to clear the files it has loaded, or to list what is currently existing.

After careful consideration, I decided to have the frontend managing the files and the backend simply being restarted to pick up new files. It is not simple to manage this and the process of doing so adds significant complexity to the extension, but it currently works effectively. This system is further detailed in the development process for the extension (detailed in section 3.6.1).

[7] Jupyter Notebook kernel guide at https://jupyter-client.readthedocs.io/en/stable/kernels.html
[8] ZeroMQ available at https://zeromq.org/

Flags/Options

As this tool is built to be run from the command line, it has multiple flags/options to change how it works. These are detailed in the README in Appendix .1. Adding this command line support was a relatively simple process. I detail it this late in the process as it changed multiple times. Initially, I required the source code to be provided as a string command line argument, but this caused issues with some terminal interfaces due to differences in how certain characters are treated.

3.4.6 `cli`

The `cli` package is a React[9]/NodeJS based frontend that is, by default, the main page for the VDMJ-Remote instance. It is set up to show a command line connected to the `exec` function in order to simulate the normal VDMJ command line.

I realised the need to develop this when testing with the prototype. Users would wish to interact as usual with the VDMJ environment, in order to change variables etc. The process of developing the package was not simple as I discovered quickly that there is no straightforward method to truly replicate a console correctly within a web browser. I used a Node package called react-terminal-ui[10] to provide the visual part of this. It allows rerouting of all commands to a single function, which I used to simply pass the command to the backend.

One of the key issues I did not solve with this was that of interrupting a running command mentioned previously when discussing parallel processing (in section 3.4.3). There is of course the option of gracefully inserting a button somewhere on the terminal UI to do this. It is not possible simply to use the industry standard shortcut, as this is overridden by the browser for copy/paste. This was difficult to accomplish due to the way that React components work. React components function as HTML elements, and complex ones such as in this case are difficult to add to without forking the original project and releasing my own version which wasn't feasible.

The display will show all other outputs specified in the code the instance is running as tabs along the top bar. These can be clicked on and those outputs will be automatically started. This can be seen in Figure 3.5.

3.5 Testing the REST API

In order to test the backend I took two automated approaches alongside ad-hoc user simulation testing. While TDD (Test Driven Development) is key to the future-proofing of a project, this project had to be mostly BDD (Behaviour) as previously mentioned due to the production being essentially for a client. The ad-hoc testing mostly involved me either testing normal functionality or introducing expected errors and testing for a correct result.

For example, an important ad-hoc test I conducted was to check what would happen in the situation of calling `quit` from the remote command line. This was something I had expected to cause issues but I needed to know the behaviour of VDMJ in this situation. Unfortunately, rather than returning from the main function, VDMJ simply calls `System.exit` causing the entire process to exit. Had it instead returned I could have used the `quit` call as some kind of reset command. Instead I had to filter it out with `RegEx` from the `help` call and also filter it from being executed as a command.

For the automated testing, I wrote JUnit[11] tests within the Java `test` package. These tests are divided into the following:

[9] The React framework for NodeJS available at https://react.dev/

[10] react-terminal-ui NodeJS module available at https://www.npmjs.com/package/react-terminal-ui

[11] JUnit is a Java unit testing library availble at https://junit.org/junit5/

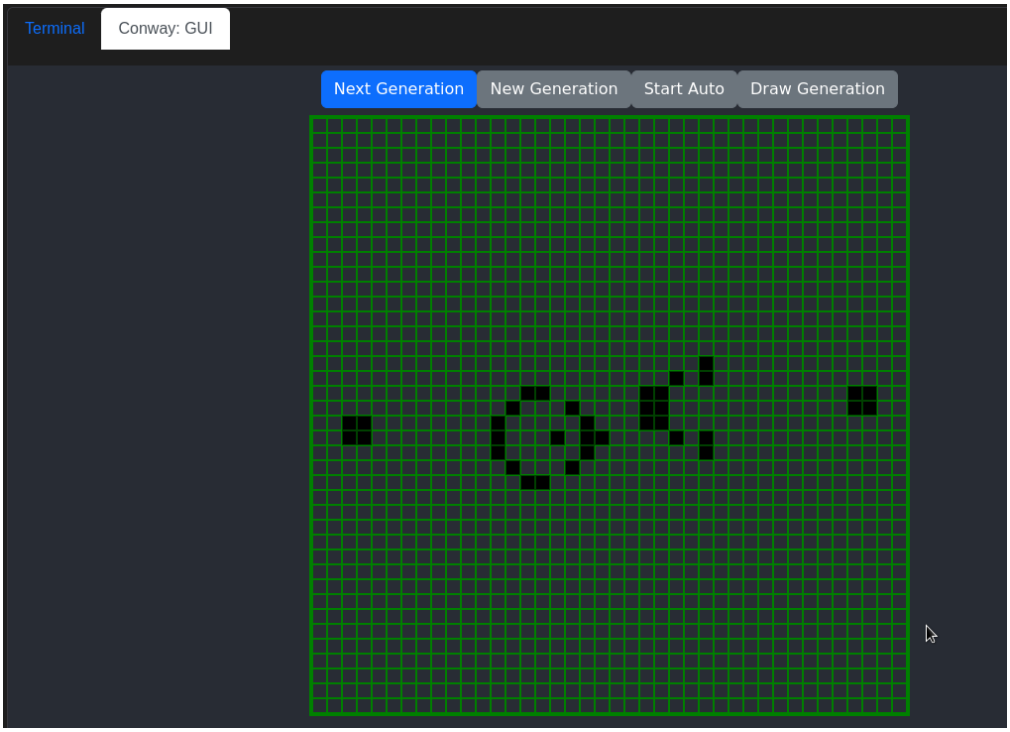

Figure 3.5: Example of the `cli` output for Conway's Game of Life

`HandlerTests :` This class tests the `VDMJHandler`, firstly by checking that the startup string is correct and then testing the responses for the commands `help` and `env`.

`MainOutputSessionTests :` This tests the `MainOutputSession`: it operates similarly to the `HandlerTests` but by sending HTTP requests to test the same functions as HTTP endpoints.

Additional test classes could be added for more granular testing. While all primary functionality is currently tested, these classes don't yet test individual parts since many of the underlying classes in VDMJ-Remote are still changing.

I also produced a file called `TestRequests.http`, this uses the relatively new and, as far as I can find, non-standardised format for HTTP requests. There are various ways to run this including *dot-http*[12], the VS Code extension *Dothttp*[13] or the built-in executor for JetBrains IDEs. It tests the routes as if it were a client, using a JavaScript environment to store session variables and sending further HTTP requests based on them. It helped me to find issues from a client perspective without having to write a website to send the requests.

3.6 VDM Notebook extension development

The extension for VS Code involves two key parts; the client and the extension provider. These separate parts are in two separate directories.

As I had no experience in writing any kind of VS Code extension let alone a notebook environment I decided to base the structure and code from an example on Microsoft's GitHub repository[14]. These

[12] dot-http, a CLI tool available at https://github.com/bayne/dot-http

[13] VS Code extension equivalent to dot-http available at https://marketplace.visualstudio.com/items?itemName=ShivaPrasanth.dothttp-code

[14] Notebook examples found at https://github.com/microsoft/notebook-extension-samples

were significantly out of date and required many changes to be compatible with my REST API backend system as opposed to the typical kernel design.

3.6.1 Extension Provider and `backend.ts`

The extension provider runs the 'kernel' for the notebook. The instance is shared between all open notebooks. This adds various complications, primarily that the extension must hold a map of documents to VDMJ-Remote instances in order to maintain the code already run.

In developing this, I used many unfortunate workarounds that have caused instability. For example, the UUID metadata added to cells for file management (covered later in this section) is a system I would prefer to be unnecessary. However, due to the Notebook API not issuing proper signals when cells are modified, deleted or reordered, this system must remain.

`backend.ts` interactions with VDMJ-Remote

The extension uses the class `Backend` from `backend.ts` to interact with VDMJ Remote. This class manages an instance of VDMJ Remote running as a child process to be terminated. It provides access to all of the functions necessary for the kernel to be operated by the extension. However, it is not a universal library as I originally intended. Since TypeScript requires that all variables have a type (unlike JavaScript), producing this class started by creating the necessary types, including those for data parsed from a JSON message. The purpose of this requirement in TypeScript is to reduce errors at runtime by adding more complexity during development time therefore creating a more stable end product. This is useful for normal web development but created issues for me as I did not have a clear enough plan to be able to say exactly what data would be needed by the frontend when I was designing the backend. Therefore this caused a number of repeated changes to keep parity in the data model between frontend and backend. An example of these types is given in Figure 3.6.

```
type IPCLog = {
    type: string,
    message: string,
    errorLevel: string | undefined,
    properties: {[key: string]: string} | undefined
}
```

Figure 3.6: The TypeScript type equivalent for the backend's `IPCLog` class

During development of VDMJ-Remote I added the IPC system as previously noted. This helped significantly in allowing the `Backend` class to run a simple socket server to receive important communication from the backend. This was done using the net module from NodeJS, which provides an API for networked IPC connections. The server is started on a random port and that is passed to the `-ipcAddress` command line flag for VDMJ-Remote. I wrote a startup message for the VDMJ-Remote `ipc` package created by `IPCLog.start(...)`. This message is what the frontend was designed to wait for upon the backend being launched to signal it being ready to receive requests.

Managing the temporary files

As mentioned in the process for the development of VDMJ-Remote, the extension was eventually required to manage temporary files to store the cell contents as source code for the VDMJ instance. This was due to a difficulty managing the associations between these temporary files and the cells from the backend's perspective. However, had the Notebook API provided signals for cell deletion and change in order etc. then none of this would have been necessary.

Implementing this in the frontend introduced a further issue: cell identification was fairly difficult because the only unique identifier provided by the API for each cell was its index. The cell index changes with reordering or the insertion of a new cell, thus making it unsuitable for the purpose of unique identification. Therefore I added a UUID to each cell as metadata and stored a map of the cell indexes to the UUID associated with that cell, as well as mapping the cell's UUID to the file in which the cell's is stored. If the index does not match the UUID then the entry must be swapped, then if the UUID does not have a matching file then one must be created for it. Otherwise it simply overwrites the existing file and sends the `reload` command to VDMJ Remote.

The complexity of the above paragraph demonstrates the complexity of the system employed to solve this issue, essentially becoming an issue itself due to its potential flaws and future instability. This is one of the key issues that will be discussed in both the *Results* section and in *Future Work*.

3.6.2 Client

The client manages the rendering of the custom mime format for the web backend `x-application/vdm-web-output`, as well as the cleanup when the rendered content is no longer needed. This primarily involves rendering `iframe`s with the correct settings to allow them to resize dynamically. This was an issue and to an extent still is, as the only solution I could find does not work fluidly.

I did not have much time for improving the client system, and most of this time was spent on making the `iframe` element output functional, as well as improving the visual styling of the output to make it consistent with VS Code's style.

Configuring the `iframe` for correct functionality was not a simple task. There are many options for flags to pass to the element for security purposes in a normal context. Many of these are by default enabled and require trawling through lists of configuration specifications to discover their purpose, and to decide whether they need to be disabled. In the end I settled on a combination that works, but were this tool to move towards general use I would encourage going back over these options and more rigorously testing their effects.

There is also the matter of the dynamic content within the frame. Since users will be designing it, the size requirements are unpredictable. This raised the not insignificant problem of allowing automatic resizing to fit the content. Due to the security concerns affecting the design of `iframe` tags, they cannot easily be interacted with. I tried many solutions to this issue but eventually settled on a fairly effective one using message posting.

Essentially, if the web content inside the `iframe` resizes itself, it posts a message to the parent document requesting it be resized. This message is received by an `EventListener` in the parent page and the item is manually set to that size. A better system would be to allow content to be resized by clicking and dragging, but despite my efforts to discover a way to do this I have been unable to implement it.

3.7 Conway example

This section is describing the code included for the Conway frontend I built as an example to test features. I initially had plans to produce several different examples including the N-Queens example used by ProB, however due to time constraints I had to limit myself to the Conway example alone.

This was written as a React-based, single-page web app. I chose this method primarily due to my own experience with it and my confidence that I could produce something visually appealing. The NodeJS approach also allows many other tools to be used to speed up development. For example,

the tabs in Figure 3.5 are provided by Bootstrap[15], a simple GUI feature toolbox that automatically adapts to different screen sizes.

Initially, I simply copied the functionality of the original VDM Toolkit ConwayNB example, adapting it from the Java AWT/Swing animation, which runs the animation for a few frames then closes with no option for UI based modification of the parameters. I modified this so that my system uses buttons to allow the user to go to the next generation by clicking, and reset to the starting generation if necessary.

I attempted to replicate the automatic iteration of the ConwayNB example. However, because React's state system does not have a simple method to prevent certain functions being called on a state update, scheduling these refresh events caused the browser/display to bottleneck. This is due to to the large number of duplicate requests generated every time new data was obtained. I have kept the button assigned for this purpose, "Start Auto", in the final version as it is easier for this issue to be demonstrated than explained.

While several improvements could be made to this package, it's purpose is to serve as a proof of concept, not a perfect example.

[15]Bootstrap from https://getbootstrap.com/

Results

4.1 Final products

The final product of this project is two installable systems and the test-suite for the REST API. The results for these are listed under the corresponding project objectives.

4.1.1 VDMJ-Remote

This is the Java-based system to wrap around VDMJ and provide access to it via REST API. Additionally, it can dynamically host other web content as 'tabs' within the default page which is a command line similar to the normal CLI (command line interface) for VDMJ.

Must provide access to all functionality normally accessible with VDMJ

The `exec` endpoint provides direct access to the VDMJ terminal for all commands apart from the `quit` command (due to its incompatibility with the system logic flow). Additionally, the command line options passed to the VDMJ instance aren't as flexible as I hoped. However, all functionality I could test and for which I could confirm support is available to the user.

Various command line arguments (as shown in the *README for VDMJ Remote* in the Appendix) are used to specify settings for how the server will run. The command line interface system seems user friendly but could use some more detailed information about the configurations being made.

The application itself can host as many static web folders as necessary. It chooses a new port each time and responds to the request that started the web output with the new address that it is hosted on. Unfortunately I have not implemented a system to list details about currently running outputs: this should definitely be included for management of the application.

Must handle known exceptions and prevent unexpected crashes

The stability of the application is high: in the vast majority of situations, errors (besides `RuntimeException`) are caught and handled correctly (rather than being thrown in the main thread terminating execution).

Unfortunately, I did not have time to implement a detailed reporting/logging system for the extension to log issues to the user or provide saved log files for bug reporting.

Should output universally recognised computer-readable formats

All interaction with REST endpoints return either a JSON document or simple text in some cases.

When a user is viewing the console output (if `-i/-ipcAddress` is not specified) the IPC system will print JSON objects straight to the user. These are useful as they easily distinguish themselves from

SLF4J logs from Spark and other classes, but are not easily human readable potentially producing confusing logs. While this wasn't specified as a requirement of the project, it would be desirable to have more consistency in the logging output format.

Must allow future modification into a socket/IO stream-based true RPC system

The IPC system as it stands is a functional example of this, allowing a parent process to properly monitor the application's status and deal with errors or restart the application. This was added late into development to improve the stability of the backend-frontend relationship. If a serious exception is raised then a log is sent via the IPC system (console IO or socket connection) notifying the parent process of what has occurred with a human-readable error explanation message. It is not truly comparable to the Jupyter socket specification as it does not use ZeroMQ, nor implement any of the key functionality described by that specification. I originally intended to include some similar functions to demonstrate how this would be approached, however due to time constraints these were omitted.

Essentially, this demonstrates how the socket interactions would work for this project, therefore the objective is met.

The *Future Work* section covers in more detail how this objective could be expanded on, and ultimately how to produce a Jupyter kernel for VDM.

4.1.2 Testing of VDMJ-Remote

Must test all routes of REST API at minimum

The test-suite does accomplish this requirement fully, primarily due to the `HTTP` requests file. This is my preferred system for testing REST APIs as it gives a simple method of simulating a client in JavaScript. However, with the lack of underlying granular testing, some edge cases could have escaped my notice.

Should be integrated into CI/CD system to ensure testing before merges/releases

Using GitHub's jobs system, the JUnit tests are run on both of these actions. If the repository were made public then I would be able to enable it on all commits, preventing the discovery of obscure errors when merging or releasing.

Ideally should test packages within the project in isolation

The testing of VDMJ-Remote has succeeded somewhat based on the requirements. However further granularity is necessary if future development is to be done on this topic. The non-standardised HTTP request tests are not a good system due to lack of universality: future tests should be written to test individual classes against their purpose.

4.1.3 VDM-Notebook-Extension

Must be able to write all of the VDM Dialects

All VDM dialects are usable, even in the same notebook (though in different instances of VDMJ). This is incredibly inefficient due to memory overhead and makes notebooks more confusing to understand.

Must allow web output for animations

I have succeeded in accomplishing this aim with a few caveats. The output is embedded correctly on the machines I've tested with, however it could easily be that on some system configurations this might throw up some errors. The system by which the embedding is created is by no means perfect and, as will be noted in *Future Work*, in implementing it I uncovered a far simpler approach that would be superior in almost all capacities.

Must handle known exceptions and prevent unexpected crashes

Known exceptions are few in the case of the Notebook API, primarily due to the lack of clear documentation. Any exceptions that arose in my limited user-simulation testing have been resolved, but most serious issues would have to be found by testing with a variety of users and use cases.

Must relay to user any pertinent information provided from the backend

The system can handle most exceptions from the backend and reports a VDMJ-Remote crash to the user via a notification in VS Code. In the event of a full crash, the document has to be reopened.

4.1.4 Additional notes

The Conway's Game of Life example is a useful one, but it does not at all exemplify all the possible use cases of what I have produced. A better example would have been to take a model for a complex system with more than one module to write a frontend for. This would have allowed me to test and discuss the implementation of other features. For example, it would have been useful to test the use of an initial script to be run before the model is loaded and available (as can easily be done with VDMJ) so that some data can have been already produced and be ready for display.

The long term maintainability of the products of this project varies. For the VDMJ-Remote project, systems are well in place for future development: most of the important interface components are well commented with JavaDocs and could be easily understood by other programmers. In the case of the VDM-Notebook-Extension, due to time constraints this is nowhere near as maintainable as I would have liked. A TypeScript project should first and foremost come with `eslint` style requirements to guide future development. I had no time to design these and was uncertain about the implications of using those from another project.

4.2 Benefits of design

4.2.1 VDMJ as an RPC system

This was an unexpected benefit that arose only because of the design approach I took. Using the VDMJ-Remote system, developers can write a model in VDM and host it as a webpage with custom output. This could be useful in a variety of scenarios, especially with automated model verification and testing.

4.2.2 Universality

The use of websites as the basis for the animation environment allows for near infinite options when designing UI for a model. With the REST API system, other applications could easily be built using different UI systems or to simply produce data.

4.2.3 Future extensibility

The VDMJ-Remote system has been built with future modification in mind. This will allow it to be repurposed to other use-cases, either by removing functionality like preventing the static web hostings, or by adding functionality such as a new type of session.

The frontend would need work to support proper extensibility, but there seems little purpose given the outcome of this project.

4.3 Issues with design

4.3.1 `<iframe>`s

The system, as I initially designed it, requires web output in VS Code and within that web output any dynamically created web content for animations. It is generally recommended to avoid using `iframe`s and especially to have them stacked within each other. However, given the time constraint on this project, there was no other system I could quickly produce.

4.3.2 REST API as a backend for Notebooks

Notebook systems, both Jupyter and the VS Code Notebook API, are designed for use with a standardised system via IPC (inter-process communication) usually via sockets. This system allows less room for mistakes in communication, since with a standardised method of data transfer errors/exceptions will have a known format.

Additionally, timings can cause strange errors: this is because most platforms seem by default to use asynchronous calls for HTTP requests in order to populate outputs as responses are retrieved.

4.3.3 VDMJ

VDMJ has caused a few issues due to its design as a purely command line tool. Due to the time constraints of this project I did not have time to attempt to modify this for better compatibility.

The major issue of being restricted to command line interface for interacting with VDMJ could have been solved by using the `RemoteInterpreter` provided by VDMJ. However this would mean any time VDMJ updated there would be a lot more to change in my system. This did in fact occur during my project as that was my original design, thus prompting the move to a simulated command line.

Simply adding some kind of socket for interfacing with VDMJ would solve this immediately, as then the REST API could simply be a wrapper for IPC. Additionally this would enable VDMJ to be run as a part of almost any web infrastructure without the need for a REST API. This design would be what I refer to throughout this document as a true RPC system.

Conclusions

5.0.1 Overview

This project was started with the aim of producing the tools as specified, but it could be better interpreted as a feasibility study. The complexity of the tools to be produced has been a significant factor in the successes/failures in this project.

In the case of VDMJ-Remote, I have a fair amount of experience writing REST APIs and specifically in Java. By contrast my understanding of VS Code's extension interface has limited my progress at every step.

The VDMJ-Remote is currently working quite well as a system to use VDM either remotely or as a website. This is the primary success of this project. Should further work be done, I would recommend building upon this.

The testing of VDMJ-Remote is mostly valid but given more time would be more comprehensive than it currently is. The tests do cover everything by sending requests simulating user routes. However improvements could be made to test each Java class/package independently.

VDM-Notebook-Extension is not a stable tool: VDM dialects are not suitable for notebook environments, nor would it be simple to implement a scripting language to run elements of VDM code from other files. The best solution I can conceive of is to write a Python wrapper library for VDMJ and use that in a Jupyter Notebook.

Overall, I think this was too ambitious a project for the time available. Had I needed to provide one of the two requirements rather than both I may have found more success. For example, if this were simply the matter of producing an animation environment for VDM, I would have had a great deal more time to improve the VDMJ-Remote system and provide more examples and features, whilst still including the bonus feature of allowing VDM to be used in a web environment.

5.0.2 Project management

My self management for this project was effective initially, but due to the significant software development I had to accomplish I left little time for the writing of this report. If I were to approach this project again, I would certainly frame it as more of a feasibility study rather than the production of stable tools. Quite simply, producing a stable, user-friendly, and cross-platform piece of software is not possible in the time I had.

Additionally, I would have chosen one of the two objectives, likely the animation environment, to complete. This would have produced a higher quality tool and more specific research rather than a process divided between two completely separate objectives. This would have allowed me to decompose the development further during the planning stage and propose precise interface specifications.

5.0.3 Future Work

In attempting what turned out to be a very ambitious project, I have made many valuable steps forward - indeed some elements of the project have been successfully completed. However, in many cases these 'steps' are in the form of realisations that different approaches and solutions would work better than the ones I chose.

In some cases, more experience might have enabled better prediction of the issues that would arise from my approach. However, some of the challenges are so complex that a more 'trial and error' approach was perhaps the only one available.

I believe that I have arrived at some valuable insights that enable me now to suggest a variety of more promising approaches.

The following sections, as well as detailing how the sound aspects of my work can be built upon, propose rather different approaches to other aspects of the design. I have presented the future possibilities starting with those that can reuse the most of my work and ending with those that would require wholly new effort.

Adapting VDMJ-Remote to work as a Jupyter Kernel

Using ZeroMQ to provide socket-based IPC would be the first step in developing this project. The REST API system can be disabled as this would only require the VDMJ integration provided in the `engine` package and some of the other bootstrap code.

The Jupyter interface will start the process with a configuration file describing the ports that each part of the process should be hosted on. These need to provide a number of services, some of which are simple such as the 'heartbeat' system (to check server status) but others more complex in this particular use case. The system by which code is communicated to the backend is not quite how VDMJ is expecting code, Jupyter may send it line by line which works for shell-like languages but not languages like VDM. This issue ties into the next suggestion.

Animation could be provided by the aforementioned ipywidgets tool for Jupyter. However, this would require a Python environment with access to the VDM environment.

Design a shell-like VDM terminal language

The command line interface for VDMJ does not provide the features necessary to serve this purpose: functions cannot be defined easily and multiline statements are impossible. If this were improved to be a true shell language then it could be used as the language of a notebook environment.

This would need to allow loading/unloading VDM content from other files in the project. I began to explore simply adding these as commands to VDMJ but due to various design elements (primarily variables being private access with no getters/setters) this was not possible.

Concept for Python VDMJ Bindings

This would be my choice if I were to approach this project again, as it provides all of the advantages of the proposed concept without the drawbacks of my chosen system.

Python is the language used by Jupyter Notebook. It works cross platform for many purposes. If bindings for either converting Python objects to VDM, a difficult prospect, or purely using Python as a shell script for VDMJ (essentially replacing the terminal commands for VDMJ), then this would allow both literate programming and animation without the need for any REST API or similar. A library in Python could simply be imported and calls could be made that return some kind of

Python readable-data: Python and ipywidgets could then be used to accomplish both animation and user interaction.

Here is an example process for reproducing the Conway Game of Life model/animation in this system:

1. Write a system using either PIL (Python Imaging Library) or Pygame (Python Games Graphics library) to render a grid of black/white or transparent squares from a list of coordinates.
2. Call a command provided by these proposed bindings to load the VDM code for the `conway.vdmsl` script (for example this command could be called `loadVDM` or be separate by dialect).
3. Write a function in Python to call the `generation` command from the script, taking a list of coordinates (alive cells) and using the bindings to make calls in VDMJ.
4. Get the value of `GOSPER_GLIDER_GUN` from the VDMJ environment as a list of coordinates.
5. If using Pygame, allow a user to press a button to go to the next generation using the function written, or if using PIL generate a certain number of frames outputting them as a GIF (this will output in Jupyter whereas Pygame will not).

One could add a step to use the function to call `generations` instead (generating n generations at once) with a widget from ipywidgets to select the number of generations to create.

This would be an effective example of how animation would work in this system. It offers the following advantages over the system proposed in this project.

- No HTML/JS/CSS: having just Python to write is a lot simpler than having several languages
- Immediate Jupyter compatibility: writing this as a Python library would allow users to simply install it in their Jupyter environment and run the code
- Less complex to maintain: this concept would require integration into the VDMJ project as an additional product of that effor. While this would be one more thing to maintain, it should be a simpler effort than managing VDMJ externally as I had to.

Create a new implementation of VDM in the style of Python/modern scripting languages

This is obviously a very large commitment, and would likely require a whole new language proposal document. I could see this being beneficial however as not only could compatibility with modern development techniques like Jupyter Notebook be enabled, but the language itself could be modernised to include new features.

For example, the data output could be brought inline with newer data storage/communication standards like JSON. I have considered writing a set of plugins for VDMJ to enable socket communication and JSON IO communication as a future project myself given my experience with these features, both as parts of this project and possible options I explored.

Bibliography

Abrial, J. R. (1988). "The B tool (Abstract)". In: *VDM '88 VDM — The Way Ahead*. Ed. by Robin E. Bloomfield, Lynn S. Marshall, and Roger B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 86–87. ISBN: 978-3-540-45955-2.

Battle, Nick (2023a). *VDMJ Annotation Guide*. URL: https://github.com/nickbattle/vdmj/blob/master/annotations/documentation/AnnotationGuide.pdf (visited on 03/22/2023).

— (2023b). *VDMJ User Guide*. URL: https://github.com/nickbattle/vdmj/raw/master/vdmj/documentation/UserGuide.pdf (visited on 05/01/2023).

Booch, Grady (1994). *Object-oriented analysis and design with applications*. eng. 2nd ed.. Benjamin/Cummings series in object-oriented software engineering. Redwood City, Calif.: Benjamin/Cummings Pub. Co. ISBN: 0805353402.

Knuth, Donald Ervin (1992). *Literate programming*. eng. CSLI lecture notes; no. 27. Stanford, Calif.]: Center for the Study of Language and Information. ISBN: 0937073806.

Larsen, Peter Gorm, Kenneth Lausdahl, and Nick Battle (2010). "VDM-10 Language Manual". In.

Leuschel, Michael et al. (2023). *The prob animator and model Checker*. URL: https://prob.hhu.de/w/index.php?title=Main_Page (visited on 04/05/2023).

Lucas, Peter (1987). "VDM: Origins, hopes, and achievements". In: *VDM '87 VDM — A Formal Method at Work*. Ed. by Dines Bjørner et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–18. ISBN: 978-3-540-47740-2.

Plat, Nico and Peter Larsen (Sept. 1994). "An overview of the ISO/VDM-SL standard". In: *ACM SIGPLAN Notices* 27. DOI: 10.1145/142137.142153.

Appendix

6.1 README for VDMJ Remote

````
# VDMJ Remote

## An RPC system/REST API for [VDMJ](https://github.com/nickbattle/vdmj)

> This is currently WIP, commits may include errors

To compile run:

```commandline
mvn clean install
mvn package
```

Use the JAR named:

`vdmj-extended-<version>-shaded.jar`

The CLI args are as follows:

```
    --help, -h
      Print this help dialogue
    --ipcAddress, -i
      Address to connect to for JSON communication with parent process, must
      match '<hostname>:<port>'
    --port, -p
      Port to bind on, if 0 random port will be used
      Default: 0
  * --sourcePath, -s
      Source as a path, dir/files
  * --type, -t
      Source type ([vdmrt, vdmsl, vdmpp])
```

For example:

```commandline
````
````

```
java -jar vdmj-extended-1.0-SNAPSHOT-shaded.jar -p 8080 -t vdmsl --sourcePath
Conway.vdmsl
```

6.2 VDMJ-Remote Repository

VDMJ-Remote available at https://github.com/pointerless/vdmj-remote/tree/1c4cb1b5edb0bb68c72f79e8d52ac1bcfc52fdbd

6.3 VDM-Notebook-Extension Repository

VDM-Notebook-Extension available at https://github.com/pointerless/vdm-notebook-extension/tree94d501b075b05df8b47211bf4b7f16401de96e16

6.4 ConwayWeb Repository

ConwayWeb example project available at https://github.com/pointerless/ConwayWeb/tree/77d204d0c26b33a7356918b9a357275524d0149f