Testing Quick Reference Handbooks in Flight Simulators

Anthony Berg (200871682)
Supervisor: Leo Freitas

Word Count: 6686

22nd May 2024

Preface

Abstract

This is an abstract.

Declaration

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

Acknowledgements

I would like to thank my supervisor Leo Freitas for supporting, guiding, and providing with areas of improvement for me throughout the project.

Contents

1 Introduction **1**
1.1 Scene . . . 1
1.2 Motivation . . . 1
1.3 Aim . . . 1
1.4 Objectives . . . 1

2 Background **3**
2.1 Hypothesis . . . 3
2.2 Safety in Aviation . . . 3
2.2.1 History . . . 3
2.2.2 Checklists . . . 3
2.3 Formal Methods . . . 4
2.4 Solution Stack . . . 4
2.4.1 Formal Model . . . 5
2.4.2 Checklist Tester . . . 5
2.4.3 Flight Simulator Plugin . . . 5

3 Design/Implementation **6**
3.1 Components . . . 6
3.2 Formal Method . . . 6
3.3 Checklist Tester . . . 7
3.3.1 Designing . . . 7
3.3.2 Compose Multiplatform . . . 7
3.3.3 Storing Data . . . 9
3.3.4 VDMJ Wrapper . . . 12
3.3.5 Connecting to the Flight Simulator . . . 13
3.3.6 Testing . . . 14
3.4 Simulator Connector Plugin . . . 14
3.4.1 Creating Maven Package . . . 14
3.4.2 Submitting a Pull Request . . . 15
3.5 Scenarios . . . 15
3.6 Decisions . . . 15

4 Results **16**
4.1 Final Prototype . . . 16
4.1.1 Formal Model . . . 16
4.1.2 Checklist Tester . . . 16
4.1.3 Setting up Tests . . . 16
4.2 Problems Found . . . 17
4.3 LOC? . . . 17
4.4 Reflection . . . 17
4.4.1 Planning . . . 17
4.4.2 Implementation . . . 18
4.5 Time Spent . . . 18

5 Conclusion **20**

5.1 Changes 20
5.2 Objectives 20
5.3 What Next 21

A Formal Model **22**

B Database **30**
B.1 SQL Schemas 30

References **33**

Chapter 1

Introduction

1.1 Scene

Designing aviation checklists is difficult and requires time to test them in simulators and the real world. [1] The simulators require trained pilots to test the checklist and make sure that they work consistently [2]; testing that the steps in the checklist are concise, achieves the goal of the checklist, and will not take too long to complete to the point it could compromise the safety of the aircraft. These checklists are also carried out by the crew in high workload environments, where this workload would be elevated if an emergency were to occur. [3]

1.2 Motivation

Testing procedures in checklists is often neglected by designers. [1] This is shown in historic incidents, where the checklists to aid resolve the problem at the time was not fit for the specific scenario that crew was in.

An example of this is the checklist used on US Airways Flight 1549. This flight suffered a dual engine failure due to a bird strike at an altitude of 2818 ft (859 m). The first action by the pilot was to turn on the Auxiliary Power Unit (APU), allowing critical systems, such as the flight controls and navigational aids, to be powered as the engines were no longer able to power those systems. However, if the first call was to run through the dual engine failure checklist (the one used on the flight), it would have been the 11^{th} item on the checklist. Using the checklist from the beginning could have resulted in a worse outcome of the incident, but due to the crew's experience, they managed to execute the most successful ditching (water landing) in history. [4]

Therefore, this calls for a way to implement a way to test checklists for aspects that may have been overlooked during the development of the checklist.

1.3 Aim

The goal of this project is to test checklists in Quick Reference Handbooks (QRH) for flaws that could compromise the aircraft and making sure that the tests can be completed in a reasonable amount of time by pilots. It is also crucial to make sure that the tests are reproducible in the same flight conditions and a variety of flight conditions.

1.4 Objectives

1. Research current checklists that may be problematic and are testable in the QRH tester being made
2. Implement a formal model that runs through checklists, with the research gathered, to produce an accurate test

   (a) Understand the relative states of the aircraft that need to be captured
   (b) Ensure that the results of the checklist procedures are consistent
3. Implement a QRH tester manager that
   (a) Runs the formal model and reacts to the output of the formal model
   (b) Connect to a flight simulator to run actions from the formal model
   (c) Implement checklist procedures to be tested, run them, and get feedback on how well the procedure ran

Chapter 2

Background

2.1 Hypothesis

- Checklists can be tested in a simulated environment to find flaws in checklist for things like
  - Can be done in an amount of time that will not endanger aircraft
  - Provides reproducible results
  - Procedures will not endanger aircraft or crew further (Crew referring to Checklist Manifesto with the cargo door blowout)
- Results in being able to see where to improve checklists

2.2 Safety in Aviation

2.2.1 History

- 70-80% of aviation accidents are attributed to human factors [5]
- The first use of a checklist was in 1935 after the crash of a prototype plane known back then as the Model 299 (known as the Boeing B-17 today), due to the complex procedures required to operate the aircraft normally and forgetting a step resulting in lack of controls during takeoff [2]
- It was found that because of the complicated procedure to operate the aircraft that the pilots would forget steps, and hence the concept of checklists was tested, and found to minimize human errors [2]

2.2.2 Checklists

Checklists are defined by the Civil Aviation Authority (CAA), the UK's aviation regulator, as: 'A set of written procedures/drills covering the operation of the aircraft by the flight crew in both normal and abnormal conditions. ... The Checklist is carried on the flight deck.' [6] These checklists as a result has shown to be a crucial tool in aviation to minimize human errors. [2]

There are multiple checklists that are designed for aircraft for the use of normal operation and potential problems that could arise during the flight. These checklists are stored in a Quick Reference Handbook (QRH) which is kept in the cockpit of each aircraft for use when needed. The definition of a QRH by CAA is:

> A handbook containing procedures which may need to be referred to quickly and/or frequently, including Emergency and Abnormal procedures. The procedures may be abbreviated for ease of reference (although they must reflect the procedures contained

in the AFM[1]). The QRH is often used as an alternative name for the Emergency and Abnormal Checklist. [6]

However, checklists themselves can have design flaws as noted by researchers at the National Aeronautics and Space Administration (NASA) where checklists can be misleading, too confusing, or too long to complete, as a result having the potential of compromising the safety of the aircraft. [1] An example of this is what happened on Swiss Air Flight 111, where an electrical fault was made worse by following the checklist, resulting in the aircraft crashing in the ocean. This was as the flight crew was unaware of the severity of the fire caused by the electrical fault. Following the steps in the checklist, one of the steps was to cut out power to 'non-essential' systems, which increased the amount of smoke in the cockpit. Simultaneously, the checklist itself was a distraction as it was found to take around 30 minutes to complete in testing during the investigation. [7] This incident shows that checklists need to be tested for these flaws, and considering the original checklist for Swiss Air Flight 111 would have taken 30 minutes to theoretically complete, this could be time-consuming for checklist designers, and this would be something to note whilst working on this project.

There are other potential problems with checklists, noted by the CAA, where the person running through the checklist could skip a step either unintentionally, by interruption, or just outright failing to complete the checklist. Or the crew may also not be alerted to performance issues within the aircraft, which would be a result of running the checklist. [6] Therefore, this would be useful to add for features when testing checklists, such as adding the ability to intentionally skip a step of a checklist or gathering statistics on how the performance of the aircraft has been affected as a result of using the checklist.

Another problem to note about checklists is the human factor where the crew may fail to use the checklist, like in the case of Northwest Airlines Flight 255, where the National Transportation Safety Board (NTSB), an investigatory board for aviation accidents in the United States, determined that 'the probable cause of the accident was the flight crew's failure to use the taxi checklist to ensure that the flaps and slats were extended for takeoff.' [8] This shows that even though checklists have shown to improve safety of the aircraft, there are other measures that aviation regulatory bodies are required implement, to avoid situations where the crew may completely ignore safety procedures and systems.

2.3 Formal Methods

Formal methods is a mathematical technique that can be used towards the verification of a system, that could either be a piece of software or hardware. Therefore, this can be used to verify correctness of all the inputs in a system. [9] Hence, as this project is dealing with safety, it would be beneficial to use formal methods for testing and verification.

An example of where formal methods is used within aviation is by Airbus, where it was used during the development of the Airbus A380. Formal methods was used to test the A380 for proof of absence of stack overflows and analysis of the numerical precision and stability of floating-point operators to name a few. [10]

2.4 Solution Stack

- There would be around 3 main components to this tester
  - Formal Model
  - Flight Simulator plugin
  - Checklist Tester (to connect the formal model and flight simulator)
- As VDM-SL is being used, it uses VDMJ to parse the model [11]. This was a starting point for the tech stack, as VDMJ is also open source.

[1] Aircraft Flight Manual - 'The Aircraft Flight Manual produced by the manufacturer and approved by the CAA. This forms the basis for parts of the Operations Manual and checklists. The checklist procedures must reflect those detailed in the AFM.' [6]

- VDMJ is written in Java [11], therefore to simplify implementing VDMJ into the Checklist Tester, it would be logical to use a Java virtual machine (JVM) language.

2.4.1 Formal Model

- There were a few ways of implementing the formal model into another application
- Some of these methods were provided by Overture [12]
  - RemoteControl interface
  - VDMTools API [13]
- However, both of these methods did not suit what was required as most of the documentation for RemoteControl was designed for the Overture Tool IDE. VDMTools may have handled the formal model differently
- The choice was to create a VDMJ wrapper, as the modules are available on Maven

2.4.2 Checklist Tester

JVM Language

- There are multiple languages that are made for or support JVMs [14]
- Requirements for language
  - Be able to interact with Java code because of VDMJ
  - Have Graphical User Interface (GUI) libraries
  - Have good support (the more popular, the more resources available)
- The main contenders were Java and Kotlin [15]
- Kotlin [15] was the choice in the end as Google has been putting Kotlin first instead of Java. Kotlin also requires less boilerplate code (e.g. getters and setters) [16]

Graphical User Interface

- As the tester is going to include a UI, the language choice was still important
- There are a variety of GUI libraries to consider using
  - JavaFX [17]
  - Swing [18]
  - Compose Multiplatform [19]
- The decision was to use Compose Multiplatform in the end, due to time limitations and having prior experience in using Flutter [20]
- Compose Multiplatform has the ability to create a desktop application and a server, which would allow for leeway if a server would be needed

2.4.3 Flight Simulator Plugin

- There are two main choices for flight simulators that can be used for professional simulation
  - X-Plane [21]
  - Prepar3D [22]
- X-Plane was the choice due to having better documentation for the SDK, and a variety of development libraries for the simulator itself
- For the plugin itself, there was already a solution developed by NASA, X-Plane Connect [23] that is more appropriate due to the time limitations and would be more likely to be reliable as it has been developed since 2015

Chapter 3

Design/Implementation

3.1 Components

The best way to view the design and implementation of this project is by splitting up the project into multiple components. This has been useful for aiding in planning the implementation, as a result making being efficient with time and requiring less refactoring. The planning allows for delegating specific work tasks, and making the project modular. A benefit of making this project modular is improving the maintainability of the codebase, and allowing for future upgrades or changes, for example, using a different flight simulator for testing.

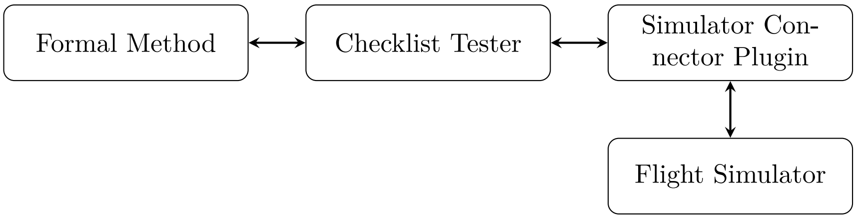

Figure 3.1: Abstract layout of components

Each of the components in Figure 3.1 will be covered in detail in this chapter.

3.2 Formal Method

Formal modelling is the heart of the logic for testing checklists in this project and is created using *VDM-SL*. The formal model is the logic behind the actions of running through a checklist and checking if the checklist has been completed in the correct manner.

To be able to check that the checklist has been properly completed, the formal model keeps track of aircraft states, such as what state each switch in the aircraft is in; and the state of the checklist, such as what steps in the checklist has been completed.

As there are invariants, pre-, and post-conditions, which are used for setting well-formedness conditions for types or functions, provide type and input safety, which will result in an error when broken. This is useful to make sure that the actions taken when completing the checklist is done correctly, such as making sure that a switch that may have 3 possible states is moved in properly, such as moving from off, middle, to on in order, rather than skipping from off to on. The cases where errors would occur is when these well-formed conditions are broken, which can be a sign that the checklist has been completed incorrectly, such as when the checklist is not completed in order, could signify that a step in the checklist failed, which could mean that the step in the checklist is problematic.

Testing

Making sure that the formal model does not have well-formed conditions that can be broken by the formal model itself is important, as the goal of the formal model is to have a rigorous specification that is verifiable.

Since *VDMJ* version 4.5.0, the VDM interpreter has included the *QuickCheck* tool, [24] which is an automated testing tool to prove and find counter examples to specifications. [25]

There were multiple counter examples that was produced by *QuickCheck* that aided the development of the formal model, as the `qc`[1] command in *VDMJ* every time a new function was created to find potential counter examples and fix them. Checking every time when creating a new function was useful as it would avoid having to refactor more of the model.

3.3 Checklist Tester

The Checklist Tester is what provides a Graphical User Interface (GUI) for defining checklists to be tested, and to run the tests on the checklist. It is also responsible for connecting the Formal Method and the Simulator Connector Plugin together.

3.3.1 Designing

Creating an interface design before creating the GUI is useful as it is a form of requirements for the code.

Figma was used to create the design for the GUI as there is support for plugins and having a marketplace for components. This saved a lot of time in designing as Google provides components for *Material 3*[2] and a plugin for creating a colour scheme for *Material 3*.

Having this design was useful as it aided in understanding what parts of the GUI could be modular and reused, kept the feel of the design consistent, and helped memorize what parts of the GUI needed to be implemented.

The final design for the interface can be seen in Figure 3.2, where the components at the top are reusable modules, and the rest below are sections of the application that the user can navigate through.

Limitations of Figma

There were some limitations when working with *Figma*, one of them being that the components created for *Material 3* did not include all the features that are available in the *Compose Multiplatform* Framework.

This can be seen in the 'Simulator Test' screen at the bottom of Figure 3.2, where there is not an option for leading icons [26] in each of the list items, and therefore had to be replaced with a trailing checkbox instead. However, *Figma* allows for comments to be placed on the parts of the design, which was used as a reminder to use leading icons in the implementation of the design.

Another limitation of *Figma* is that in Figure 3.2, the title of the screen in the top app bar [27] is not centred, this is because the auto layout feature in *Figma* works by having equal spacing between each object, rather than having each object in a set position. However, this is not detrimental to the design, it is just obvious that the title is not centred in the window.

3.3.2 Compose Multiplatform

Setup

To set up *Compose Multiplatform*, the *Kotlin Multiplatform Wizard* was used to create the project as it allows for the runtime environments to be specified (at the time of creation, Desktop and Server), automatically generating the *Gradle* build configurations and modules for each runtime environment, for the specific setup.

[1] The command to run *QuickCheck* on the formal model in *VDMJ*.
[2] Material 3 is a design system which is used in Compose Multiplatform UI Framework.

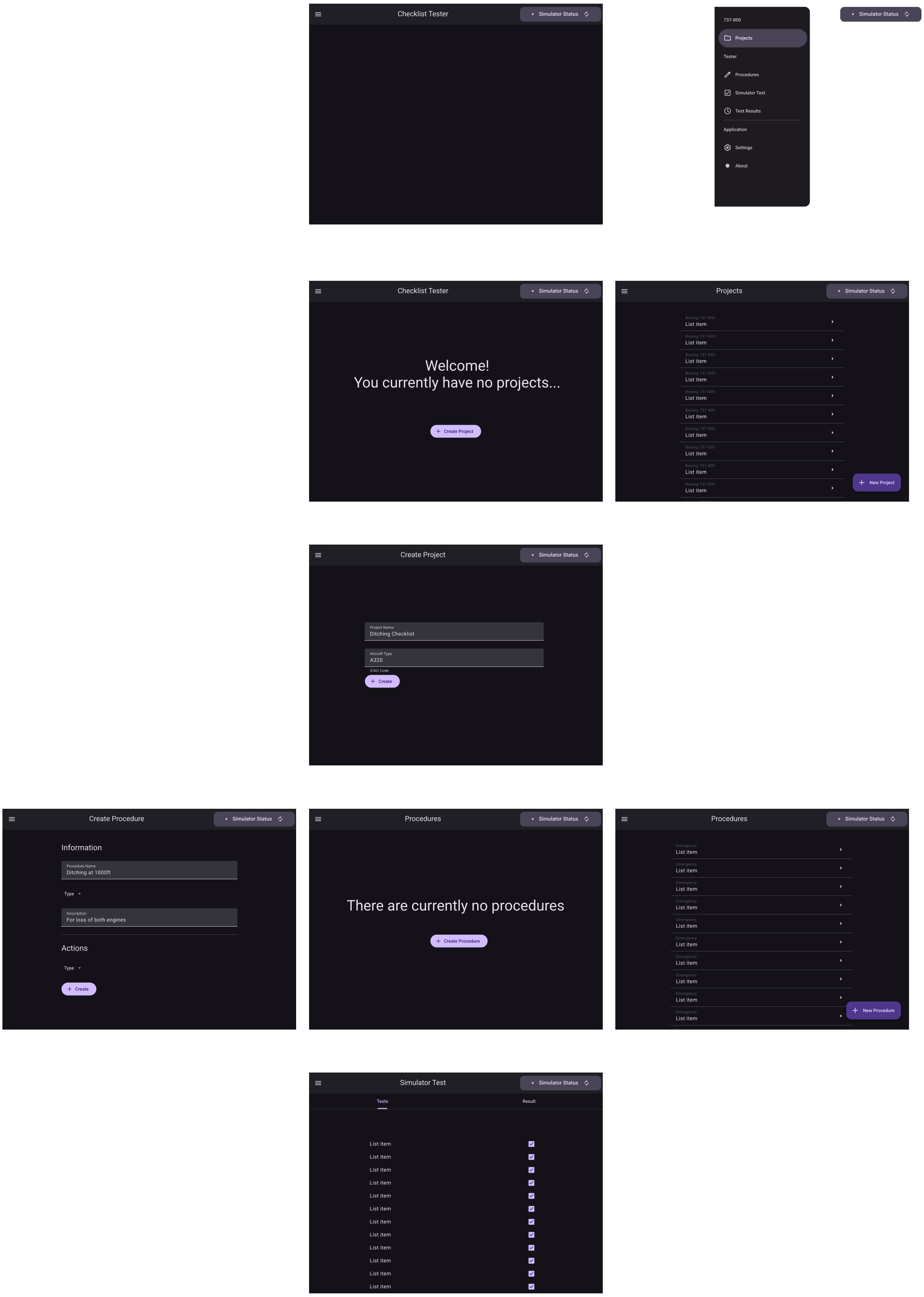

Figure 3.2: Design for the Checklist Connector GUI in Figma

Implementation

Planning was important when implementing as *Compose* is designed to use modular components, otherwise a nested mess would occur as *Compose* is designed to have *Composable*[3] objects passed into another *Composable* object. Therefore, due to how *Kotlin* is designed with functions, there will be function nesting occurring naturally. To aid in readability of code due to the nesting functions, the *Composable* objects are split into separate *Composable* functions. An example of this is in Listing 3.1, where instead of 10 *Composable* functions being nested in the `Content()` function, the items in the list (`LazyColumn` is used for creating lists) is split to a separate function, `ActionItem()`, as a result making the maximum amount of nested functions to 5 for all functions. Another benefit is that it allows for the `ActionItem` to be reused if desired, making the code modular.

Voyager [28] was used to handle the navigation of the application as it handles replacing previous navigation screens, and allows for inserting data into the navigation screens. This is as Voyager has integration with Koin [29][30], which is a library that specifically handles dependency injection. Using Koin allowed for data to be fetched from the database and to handle asynchronous functions, such as running VDMJ and sending instructions to the flight simulator.

3.3.3 Storing Data

SQLDelight was used to handle the database as it creates typesafe *Kotlin* application programming interfaces (APIs) to communicate to the database. It was specifically chosen as it provides support for *Compose Multiplatform* [31], making implementing *SQLDelight* into the project easier.

A benefit of using *SQLDelight* is that it only allows for database queries to be written in SQL, allowing for more complex, and more control of SQL queries. It also provides 100% test coverage [32] which is necessary to ensure that the database will not cause artefacts to the results.

The choice of relational database management system (RDBMS) to complement *SQLDelight* was *SQLite* as it allows for the database to run within the application, rather than running on a separate server, either remotely or through a containerized instance using something like Docker [33]. As a result, this avoided spending extra time implementing the server and adding extra complexity due to requiring additional dependencies, which would also add extra maintenance overhead to the project.

Designing the Database

The database could be looked at as having 2 sections, with relationships in mind between the two sections, to fulfil of the objectives, as it will allow tracking of the checklist tests that will be run, as a result being able to provide detailed statistics of the test. These relationships can be seen in the entity relationship diagram in Figure 3.3.

One of the sections is for user inputs to control the tests. The *Project* table handles creating separate aircraft, or it could be used for separate iterations of Quick Reference Handbooks (QRHs). Then the *Procedure* and *Action* table handles defining steps/actions in a checklist/procedure.

The other section of the database would be providing test results for each of the checklists, which are stored in the *Test* and *ActionResult* tables.

Expanding on the relationships between each table in Figure 3.3, the reasons for these relationships is to allow for segregation of data and the ability to associate test data with what checklist was tested.

Linking into Compose Multiplatform

Compose Multiplatform has support for different runtime environments which should be taken into account when adding *SQLDelight* to *Compose Multiplatform*. However, as this project is only being developed for Desktop, the *JVM SQLite* driver is the only one necessary to implement.

However, to improve maintainability of the code, the functions of the database was written in the `shared/commonMain` module (a shared module that is accessible to multiple runtime environments).

[3] A *Composable* is a description of the UI that will be built by *Compose Multiplatform*

```kotlin
@Composable
override fun Content() {
    // Content variables...

    Scaffold(
        topBar = {/* Composable content... */},
    ) {
        Column(/* Column option parameters... */) {
            Box(/* Box option parameters... */) {
                LazyColumn(/* LazyColumn option parameters... */) {

                    item {
                        Header()
                    }

                    items(
                        items = inputs,
                        key = { input -> input.id }
                    ) { item ->
                        ActionItem(item)
                    }
                }
            }
        }
    }
}

@Composable
private fun Header() {
    Text(text = "Edit Actions")
}

@Composable
private fun ActionItem(item: Action) {
    Column (/* Column option parameters... */) {
        Row(/* Row option parameters... */) {
            Text(text = "Action ${item.step + 1}")

            IconButton(/* IconButton definition parameters... */) {
                Icon(
                    Icons.Outlined.Delete,
                    // Rest of Icon options...
                )
            }
        }

        Row(/* Row option parameters... */) {
            OutlinedTextField(/* TextField definition parameters... */)

            OutlinedTextField(/* TextField definition parameters... */)
        }

        HorizontalDivider()
    }
}
```

Listing 3.1: Example of modular code in Compose

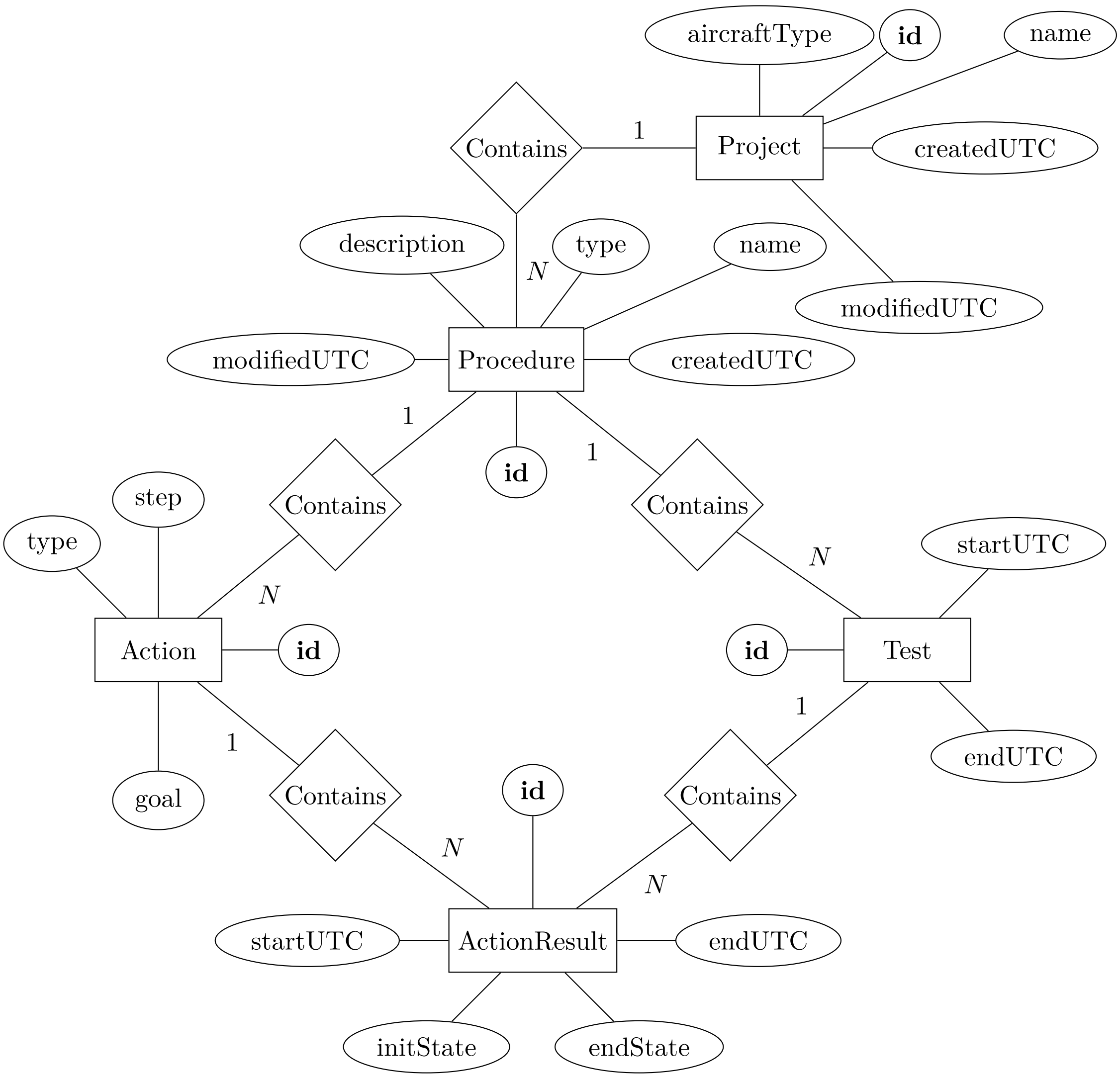

Figure 3.3: Entity Relationship Diagram for the database in Checklist Connector

This would be useful if there was a need for adding Android and/or iOS support for this project as some designers may want to run the tests on a tablet.

Handling the database was done by implementing two modules. One module is the `io.anthonyberg.connector.shared.database` module, used to handle SQLDelight API calls only; meaning no conversion of types, functions are only accessible internally within the `io.anthonyberg.connector.shared` module.

The other module is the Software Development Kit (SDK) that handle type conversions, such as `Int` to `Long`, and can handle multiple tables, such as *TestTransaction* SDK that handles calls to multiple tables when a test is run in the flight simulator.

The separation of these modules was also done to have unit testing in mind because it will make it easier to debug if a problem is due to how SQLDelight transactions are handled, or if there are type conversions errors occuring.

3.3.4 VDMJ Wrapper

VDMJ is written in Java, and it is free open source software that is accessible on GitHub. This means that VDMJ can be used within any projects as long as the licence is followed. It is important to follow for ethical and legal reasons, as not following the licensing would result in breaking copyright law. However, it may not specifically break Newcastle University's ethics, it would break the ethos behind GPLv3 and free open source software.

The licence VDMJ uses is the GNU General Public License v3 (GPLv3) [34][35]. This means that as VDMJ is being used as a library, the code for this project has to be licensed with GPLv3 or any GPLv3 compatible licence [36].

Implementing VDMJ

VDMJ has packages available on Maven Central[4] making adding it as a dependency simple as it would require to be specified within the *Gradle* build configurations. The package used was `dk.au.ece.vdmj:vdmj` with version `4.5.0`, however, initially when implementing VDMJ, `4.5.0-P` was used accidentally, and it led to the rabbit hole of debugging why imports were not working, and it was found that the `-P` versions of VDMJ is not suitable to be used when being implemented intentionally a project.

The initial method of implementation was to use a Ktor server that would run alongside the desktop application, where communication between the desktop application and the server would be handled through Representational State Transfer (REST) API calls. However, this was unnecessary as the *interactive* mode of *VDMJ* was able to run on the desktop application itself. But using Ktor was useful for debugging and testing using as *VDMJ* commands could be run through an API route.

The major hurdle within implementing *VDMJ* as a wrapper was fetching the outputs that *VDMJ* sends to the console. This was implemented by creating a new *VDMJ* console handler `ConsolePrintWriter`, that handles writing to *stdout*, which is from the `com.fujitsu.vdmj.messages` package. This then gets used to replace the `Console.out` and `Console.err`, from the same *VDMJ* package, which will store the outputs to the console into a variable instead.

Parsing commands into the *VDMJ* interface was more difficult as it required using Java functions[5] to act as if the program wrote something directly into the *VDMJ* interactive console. Figure 3.4 shows a simplified flowchart of how inputs are handled. A `PipedInputStream` object was created, that gets connected to a `PipedOutputStream` object by passing the latter object in as a parameter. The `PipedOutputStream` is then used to pass inputs into `PipedInputStream`. The `PipedInputStream` handles sending inputs to the *VDMJ* console. However, to be able to write to these streams, a `BufferedWriter`, which is used to send inputs, is created by passing the `PipedOutputStream` with a bridge `OutputStreamWriter` that encodes characters into bytes. For *VDMJ* to be able to read the input streams, the `PipedInputStream` gets parsed through a bridge,

[4] Maven Central is a repository that stores dependencies required to build projects

[5] The objects created here are provided by the `java.io` package.

`InputStreamReader` that converts bytes to characters, and then allows *VDMJ* read these characters through a `BufferedReader`.

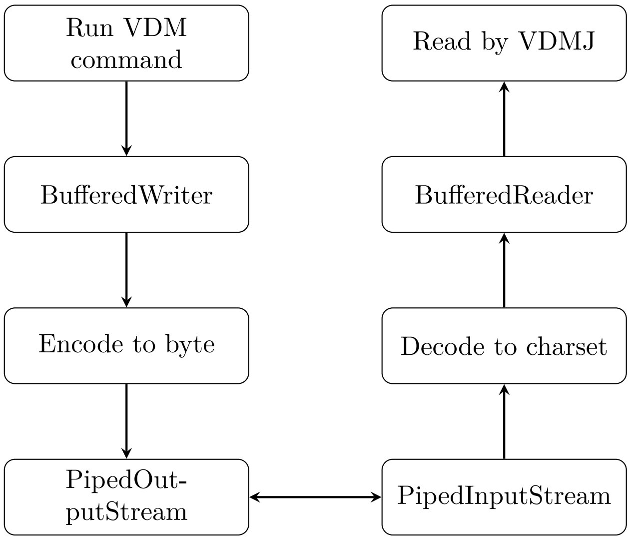

Figure 3.4: Flowchart of VDMJ Input/Output Stream handling

Handling VDMJ Command Outputs

When running a command in the *VDMJ*, it will produce an output as a string for the returned variable in the function that was executed.

To handle these strings, *Kotlin* string manipulation was used, similar concept to *Regex*, to decode the string and convert the string into correct types and store them in specific types in the formal specification, recreated in *Kotlin*.

The types recreated from the formal specification were the records types. This was done by using *Kotlin* data classes, which had functions implemented with the purpose of convert the stored types in *Kotlin* to an identical *VDM-SL* representation of the values in that type.

3.3.5 Connecting to the Flight Simulator

- Implemented XPC into the flight simulator
- Allowed being able to
  - Read data from the simulator
  - Override dataref variables in the simulator
  - Execute other commands that can manipulate certain switches where otherwise unable to by changing the value of the dataref
- Made sure to check that the simulator is connected before running the test to avoid exceptions being thrown
- Logic behind doing an action is to fetch the action's initial state from the dataref variable name, run the action, then get the final state of the dataref
- There is an artificial delay added before running the action to try and simulate a delay of the crew's lag between reading the step of the checklist and doing the action
- Because of this, XPC had to be run asynchronously to prevent the GUI from hanging as a function is waiting to complete - prevents misleading user that the application has crashed, and it looks better

3.3.6 Testing

- Testing can be run with Gradle when it comes to running unit tests
- Decided to use JUnit 5 as it provides additional tools such as statistics, integration with IntelliJ to view code coverage, or being run in continuous integration tests
- The testable components in this project is mostly backend modules as the GUI made in Compose is not the focus of the project, and it would require a lot of extra time
- Unit tests have been made for the database and Koin
- Koin comes with tests that can be automatically be generated
- Ethos when testing was to try and find exploits, act as a user who may mishandle inputs, and stress testing functions by passing parameter with hundreds of objects

Testing for Resource Usage

- The application was tested using the *Profiler* tool on IntelliJ IDEA 2024 (Ultimate Edition) to find potential memory leaks
- One problem found was the initial VDMJ wrapper which would use the execute command instead of the interpreter, which would require reinitializing the entirety of VDMJ, which resulted in a slight memory leak and a massive write usage

3.4 Simulator Connector Plugin

3.4.1 Creating Maven Package

- XPC package is not published on a public Maven repository
- There has been a pull request that was merged to the *develop* branch that provides Maven POMs [37]. However, the maintainer for the project, at the time, did not have enough time to figure out the process of publishing the package to a Maven repository [38]
- Therefore, had to find an alternative way to implement
- Jitpack [39]
  - In theory, simple to publish a repository, all that is required is a GitHub repository and searching if one has already been created on JitPack or build and publish a specific version
  - However, due to the structure of the XPC repository, JitPack could not locate the build tools (Apache Maven in this case) as JitPack only searches on the root directory for the compatible build tools
- Gradle gitRepository [40]
  - There was not a lot of documentation
  - Ambiguous on how to define directory for where the Java library is located in the Git repository
  - However, as XPC was only built with Maven, Gradle was not able to add the dependency as `gitRepository()` only works with Gradle builds [41]
- Resorted to using a compiled Jar file and adding the dependency to Gradle
- Not happy about that because it means maintaining it will be more difficult as it is not as simple as just changing the version number
- Later, resorted to adding Gradle build files to XPC
- Used automatic conversion from Maven to Gradle using `gradle init` command [42]
- Had to add local dependencies due to how Gradle works differently

- Had to fix previous structure of Maven POM as the grouping as not good

Continuous Deployment of the Maven Package

- Used GitHub's template for Gradle package publishing
- Required some setup in Gradle build files

3.4.2 Submitting a Pull Request

- Adding the Gradle build tools can be seen as being helpful for others, as it would allow for the XPC library to be added as a dependency, especially if the NASA Ames Research Center Diagnostics and Prognostics Group were to add it to the GitHub repository, it would mean that it would be easier for people to access Maven Packages for XPC
- Therefore, to help improve the experience for other people who would want to develop with the XPC Java library, it would be logical to submit a pull request
- But it did mean making sure that the contribution would be perfect and not contain problems

Testing

- The XPC Java library includes a JUnit 4 test, however, implementing this with Gradle proved useless, as it was not able to get the results from the tests, which would be bad for not being able to catch problems with new builds
- Therefore, the tests were updated to JUnit 5, where most of the changes were adding asserts for throws [43] [6]

GitHub

- Made sure to add generated build files to .gitignore
- Changed the URL of the repository in Gradle to NASA's repository so that the Maven package can be published correctly on the GitHub repository
- From the beginning anyways, made sure to have insightful commit messages
- Submitted the pull request stating the changes made[7]

3.5 Scenarios

- Use a Quick Reference Handbook (QRH) to find potential list of checklists to test
- Look at previous accident reports that had an incident related to checklists and test it with my tool to see if it will pick it up
- These previous accident reports can be good metrics to know what statistics to look out for

3.6 Decisions

[6] The commit including the changes to the tests can be viewed here: https://github.com/smyalygames/XPlaneConnect/commit/e7b8d1e811999b4f8d7230f60ba94368e14f1148

[7] https://github.com/nasa/XPlaneConnect/pull/313

Chapter 4

Results

4.1 Final Prototype

4.1.1 Formal Model

- The model is mostly designed to imitate a Boeing 737-800, as the types modelled, have user inputs which are different from other aircraft types
  - For example, the Airbus A320 has push buttons whereas they are not there on the 737-800
  - However, further user input types could be added to the model and as a result, further aircraft types could have their procedures run through the formal model
- The `Procedure` type makes sure that the items on the procedure is completed in order, and if a step is missed, that would result in an invariant failure, resulting in the checklist test failing

4.1.2 Checklist Tester

- The main features of GUI have been completed, it has all the sections desired
  - Projects can be created to split up different aircraft or revisions of checklists
  - Procedures can be created and tested
  - These procedures get tested in the flight simulator automatically and gives the results of how the procedure has been doing in real time

4.1.3 Setting up Tests

- Each test is set up by defining each action in the procedure, on the Procedure screen
- To be able to define each action is supposed to do, it uses the Dataref variables in X-Plane, which is what stores the state of the aircraft. Each switch has their own unique Dataref
- In the checklist tester then, each action asks for a Dataref and a desired goal value
- Some Datarefs are read only, but there are other Datarefs for the item desired, but are only 'command's, which can only be called and not have its value changed; this can be run by setting the desired goal value to be -988 (because XPC uses that value)

Running Tests

- Tests are run by connecting to the flight simulator, X-Plane
- The tester goes through each action in the procedure one by one and waits for the current action to complete before proceeding on to the next one

- The checklist tester is not advanced enough to be able to control fly the aircraft; hence the tester would be able to engage autopilot first, or control the aircraft themselves, where the checklist tester would be acting like a first officer

Storing Test Results

- There is a database storing the results of each of the tests
- Each tests store

  Time taken for each of the actions in the procedure to complete

  Start state for the state that the action in the procedure was at

  End state for the state that the action in the procedure finished the item at

  Overall test time Stores the time taken from when the test started to when the test ended

- This gives feedback/statistics for the checklist designers to find areas of improvement on the procedure, such as one action in the procedure taking too long, may point out a potential flaw to the designer and as a result aid finding potential alternative options for that step in the procedure

4.2 Problems Found

4.3 LOC?

4.4 Reflection

4.4.1 Planning

Gantt Chart

Used Gantt chart to create a plan for what would be needed from this project

Pros:

- Was useful for the first part because it set expectations of what was needed and how much time there was to complete them
- Helped visualize the different components of the project
- Helped in the beginning being accompanied by a Kanban in Leantime[1]

Cons:

- Was not detailed enough, and a design document would have been useful to accompany the Gantt chart for each section
- The lack of detail was not helpful when falling behind as having attention deficit hyperactivity disorder (ADHD) added to the burden of feeling like each section was a massive project
- Leantime's claim for being 'built with ADHD [...] in mind' felt misleading as navigating through it felt worse than using the front page of Stack Overflow[2]
- Todoist[3] was a good alternative though

GUI Design

Figma was very useful in implementations as

Pros:

- It helped with timing and knowing what to do

[1] https://leantime.io/
[2] https://stackoverflow.com/
[3] https://todoist.com/

- Made things feel manageable as it was split up to different sections
- Meant features will not be forgotten

Cons:

- Certain features being too simple and annoying to use
- A bit of a learning curve for using other components, compared to using plugins

4.4.2 Implementation

Checklist Tester

- Implementing the GUI was useful to split up the sections required for the project and having a goal for what to be done
- However, a bit too much time was spent on creating a GUI when it could have been used for development
- It was useful for motivational reasons to feel like something materialistic has been produced rather than something theoretical
- Was originally intended to be used to interact with custom plugin for X-Plane as it would have been difficult otherwise

Connecting to the Flight Simulator

- Would have been more useful to search a bit further if there was another plugin available, as found Dataref Editor on the X-Plane docs, so could have looked for a similar plugin for connecting to X-Plane
- At first spent about a week developing a C++ X-Plane plugin from scratch, requiring to figure out sockets
- At the same time finding out XPC exists and having wasted that time
- However, it did teach me more about understanding how sockets work and more about C++ and setting up a project with CMake and adding packages with vcpkg

4.5 Time Spent

- Time spent was recorded using Wakatime, other than time spent researching, which had to be recorded manually, using Leantime
- The time spent on GUI is also time spent on connecting other tools such as the VDMJ wrapper, XPC, and the database

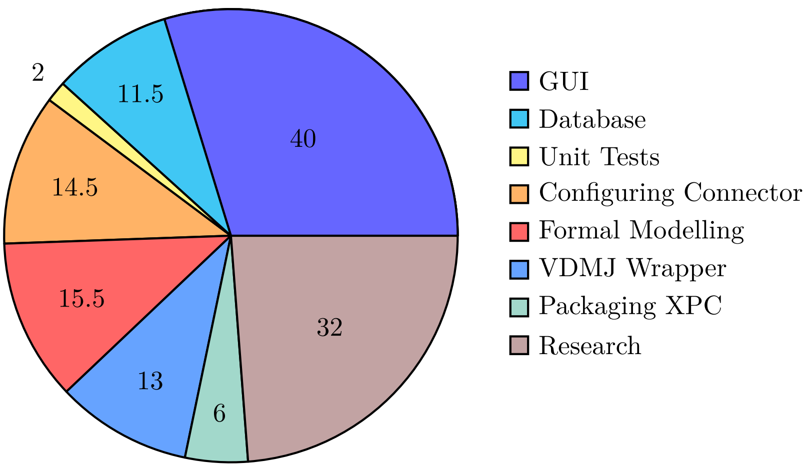

Figure 4.1: Time spent on sections of project (in hours)

Chapter 5

Conclusion

5.1 Changes

- Added the checklist manager which was not a part of the original objectives
  - Helped more to visualize the project
  - Aided in gathering statistics for how well the checklist performed
  - Using Kotlin helped speed up development, it simplifies parts of Java and omitted a lot of boilerplate code that is required in Java, such as setters and getters
- How the Formal Model would interact was modified
  - Initially was designed so that the formal model would complete the entirety of the checklist, however, it was not useful for interacting with the flight simulator
  - Modified the model to provide it would be similar to actions pilots can do in the cockpit
  - Therefore acts like Read Checklist → Pilot Logic (VDM) → Do Action (XPC)
- Originally was supposed to write an original plugin to connect to the flight simulator
  - Whilst creating the plugin, sockets were confusing and accidentally stumbled on the X-Plane Connect GitHub repository
  - This could have been prevented if a design document was created and time was spent researching for tools in obscure places

5.2 Objectives

- Most of the objectives were met
- One of the original objectives was to research pilot reaction times and how long it takes pilots to complete an action
  - However, not able to do that as there are too many factors that can affect a pilot's reaction time, such as age, experience on an aircraft, total experience, how far a button is from the pilot, etc.
- Objective 2.a. was met to an extent
  - Currently, the states of the aircraft monitored are only the actions specified in the test, in the checklist tester
  - There could be more variables that could be monitored. Such as engine fire, could monitor the engine temperature or thrust produced by engine
  - This would have required a substantial amount of planning as checklists do have conditional statements, for example 'If APU is available, then do Step 3 else do Step 4'

- Objective 2.b. was also met to an extent
  - Currently, this can be met by re-running the test multiple times manually
  - However, it is manual at this stage due to limitations of XPC and setting up the aircraft
  - The test data is stored on the database, hence test results can be analysed to see the consistency between each test
- The Checklist Tester does not currently run actions from the Formal Model due to implementing the functions from VDMJ being laborious
- Hence focus was put on XPC first, as it would produce direct results

5.3 What Next

The most important next steps to implement would be linking the formal mode, adding options of what parts of the aircraft to monitor

- Formal Model
  - Implemented either by creating an automatic wrapper. Done by either potentially linking the VDMJ LSP, or creating a plugin for VDMJ
  - Or doing string manipulation on the VDM results for each of the functions as a lot of it is copy and paste - can be bad practice as it requires a lot of hard-coded code
- Monitoring more of the aircraft
  - Done by adding options in the Checklist Tester for extra Datarefs to monitor
  - Modifying the `Aircraft` record type to include a states type that checks multiple times throughout the procedure if this state has violated a constraint or if the goal of the state has been achieved (e.g. Engine is no longer on fire)
- Expanding out of the scope of the objectives, conditional logic, such as if statements, to the checklist would be the next logical step
  - VDM-SL would be really helpful for this, as can be used to design logic to be used outside of Kotlin
  - This would allow for further automation of checklists, rather than only testing linearly, which at this current state would require writing the test multiple times
- Adding more detailed test results
  - Use analysis of previous test results to gain an understanding of the reproducibility of the procedure
  - Keep track of aircraft state, such as speed or altitude aiding in understanding if the procedure may impose a safety risk

Appendix A

Formal Model

```
module Checklist
exports all
definitions

values
    -- Before Start Checklist
    -- Items in Aircraft
    -- Flight Deck... (can't check)
    fuel: ItemObject = mk_ItemObject(<SWITCH>, mk_Switch(<OFF>, false));
    pax_sign: ItemObject = mk_ItemObject(<SWITCH>, mk_Switch(<OFF>, true));
    windows: ItemObject = mk_ItemObject(<SWITCH>, mk_Switch(<ON>, false));
    -- Preflight steps
    acol: ItemObject = mk_ItemObject(<SWITCH>, mk_Switch(<OFF>, false));

    aircraft_panels: Items = {"Fuel Pump" |-> fuel, "Passenger Signs" |-> pax_sign, "Windows" |-> windows, "Anti Collision Lights" |-> acol};

    -- Checklist
    -- Flight Deck... (can't check)
    fuel_chkl: ChecklistItem = mk_ChecklistItem("Fuel Pump", <SWITCH>, <ON>, false);
    pax_sign_chkl: ChecklistItem = mk_ChecklistItem("Passenger Signs", <SWITCH>, <ON>, false);
    windows_chkl: ChecklistItem = mk_ChecklistItem("Windows", <SWITCH>, <ON>, false);
    -- Preflight steps
    acol_chkl: ChecklistItem = mk_ChecklistItem("Anti Collision Lights", <SWITCH>, <ON>, false);

    before_start_procedure: Procedure = [fuel_chkl, pax_sign_chkl, windows_chkl, acol_chkl];

    aircraft = mk_Aircraft(aircraft_panels, before_start_procedure);
types
    --@doc The dataref name in X-Plane
    Dataref = seq1 of char;

```

```
-- Aircraft

-- Switches
--@doc The state a switch can be in
SwitchState = <OFF> | <MIDDLE> | <ON>;

    --@LF why have a type kist as a rename?
ItemState = SwitchState; --@TODO | Button | ...

--@doc A switch, with the possible states it can be in, and the state that it is in
Switch ::
    position : SwitchState
    middlePosition : bool
    inv s ==
        (s.position = <MIDDLE> => s.middlePosition);

-- Knob
Knob ::
    position : nat
    --@LF how can a state be an int? perhaps a proper type (i..e. subset of int range or a union?)
    states : set1 of nat
    inv k ==
        k.position in set k.states;

Lever = nat
    inv t == t <= 100;

Throttle ::
    thrust: Lever
    reverser: Lever
    inv t ==
        (t.reverser > 0 <=> t.thrust = 0);

--@doc The type that the action of the button is
ItemType = <SWITCH> | <KNOB> | <BUTTON> | <THROTTLE>;

--@doc The unique switch/knob/etc of that aircraft
ObjectType = Switch | Knob | Throttle;
ItemObject ::
    type : ItemType
    object : ObjectType
    inv mk_ItemObject(type, object) ==
            cases type:
                    <SWITCH> -> is_Switch(object),
                    <KNOB>   -> is_Knob(object),
                    <THROTTLE>-> is_Throttle(object),
                    --<BUTTON> -> true
                    others -> true
            end;

--@doc Contains each ItemObject in the Aircraft, e.g. Fuel Pump switch
Items = map Dataref to ItemObject;

--@doc Contains the panels (all the items in the aircraft) and the procedure
```

```
    Aircraft ::
        items : Items
        procedure : Procedure
        inv mk_Aircraft(i, p) ==
        ({ x.procedure | x in seq p } subset dom i);

    -- Checklist

    --@doc Item of a checklist, e.g. Landing gear down
    ChecklistItem ::
        --@LF again, empty string here doesn't make sense.
        procedure : Dataref
        type : ItemType
        --TODO Check is not only SwitchState
        check : SwitchState
        checked : bool;

    --@doc This is an item in the aircraft that complements the item in the procedure
    ItemAndChecklistItem ::
        item : ItemObject
        checklistItem: ChecklistItem
        inv i == i.item.type = i.checklistItem.type;

    --@doc A section of a checklist, e.g. Landing Checklist
    --@LF shouldn't this be non-empty? What's the point to map a checklist name to an empty procedure? Yes.
    Procedure = seq1 of ChecklistItem
        inv p ==
            --@LF the "trick" for "false not in set S" is neat. It forces a full evaluation, rather than short circuited (i.e. stops at first false).
            --    I presume this was intended.
            false not in set {
                let first = p(x-1).checked, second = p(x).checked in
                        --@LF boolean values don't need equality check
                    second => first--((first = true) and (second = false))
                | x in set {2,...,len p}};

functions
    -- PROCEDURES
    --@doc Finds the index of the next item in the procedure that needs to be completed
    procedure_next_item_index: Procedure -> nat1
    procedure_next_item_index(p) ==
        hd [ x | x in set {1,...,len p} & not p(x).checked ]--p(x).checked = false]
    pre
        -- Checks procedure has not already been completed
        not procedure_completed(p)--procedure_completed(p) = false
    post
        -- Checks that the index of the item is the next one to be completed
        --@LF your def is quite confusing (to me)
        --@LF how do you know that RESULT in inds p? Well, the definition above okay.
```

```
        --    but you can't know whether p(RESULT-1) will! What if RESULT=1? p(RESULT-1)=p(0) which is invalid!
        (not p(RESULT).checked)
        and
        (RESULT > 1 => p(RESULT-1).checked)
        --p(RESULT).checked = false
        --and if RESULT > 1 then
        --    p(RESULT-1).checked = true
        --else
        --    true
        ;

    -- --@doc Checks if all the procedures have been completed
    -- check_all_proc_completed: Checklist -> bool
    -- check_all_proc_completed(c) ==
    --     false not in set { procedure_completed(c(x)) | x in set {1,...,len c} };

    -- --@doc Gives the index for the next procedure to complete
    -- next_procedure: Checklist -> nat1
    -- next_procedure(c) ==
    --     hd [ x | x in set {1,...,len c} & not procedure_completed(c(x))]
    -- post
    --     RESULT <= len c;

    --@doc Checks if the procedure has been completed
    procedure_completed: Procedure -> bool
    procedure_completed(p) ==
        false not in set { p(x).checked | x in set {1,...,len p} };

    --@doc Checks if the next item in the procedure has been completed
    check_proc_item_complete: Procedure * Aircraft -> bool
    check_proc_item_complete(p, a) ==
        --@LF here you have a nice lemma to prove: procedure_next_item_index(p) in set inds p!
        --       I think that's always true
        let procItem = p(procedure_next_item_index(p)),
                --@LF here you can't tell whether this will be true? i.e. procItem.procedure in set dom a.items?
            item = a.items(procItem.procedure) in

            --TODO need to be able to check for different types of Items
            procItem.check = item.object.position
    pre
        procedure_completed(p) = false
        --@LF perhaps add
        --and
        --p(procedure_next_item_index(p)).procedure in set dom a.items?
        ;

    --@doc Marks next item in procedure as complete
    mark_proc_item_complete: Procedure -> Procedure
    mark_proc_item_complete(p) ==
        let i = procedure_next_item_index(p), item = p(i) in
            p ++ {i |-> complete_item(item)}
```

```
        pre
            procedure_completed(p) = false;

    --@doc Completes an item in the procedure
    do_proc_item: ItemObject * ChecklistItem -> ItemAndChecklistItem
    do_proc_item(i, p) ==
        let objective = p.check,
            checkckItem = complete_item(p) in
            -- Checks if the item is in the objective desired by the checklist
            if check_item_in_position(i, objective) then
                mk_ItemAndChecklistItem(i, checkckItem)
            else
                mk_ItemAndChecklistItem(move_item(i, p.check), checkckItem)
    pre
        p.checked = false
    post
        -- Checks the item has been moved correctly
        check_item_in_position(RESULT.item, p.check);

    --@doc Completes a procedure step by step
    -- a = Aircraft
    complete_procedure: Aircraft -> Aircraft
    complete_procedure(a) ==
        let procedure = a.procedure in
            mk_Aircraft(
                a.items ++ { x.procedure |-> do_proc_item(a.items(x.procedure), x).item | x in seq procedure },
                [ complete_item(x) | x in seq procedure ]
            )
    pre
        not procedure_completed(a.procedure)
    post
        procedure_completed(RESULT.procedure);

    -- AIRCRAFT ITEMS
    --@doc Marks ChecklistItem as complete
    complete_item: ChecklistItem -> ChecklistItem
    complete_item(i) ==
        mk_ChecklistItem(i.procedure, i.type, i.check, true)
    pre
        i.checked = false;

    --@doc Moves any type of Item
    move_item: ItemObject * ItemState -> ItemObject
    move_item(i, s) ==
        -- if is_Switch(i) then (implement later)
            let switch: Switch = i.object in
                if check_switch_onoff(switch) and (s <> <MIDDLE>) and switch.middlePosition then
                    mk_ItemObject(i.type, move_switch(move_switch(switch, <MIDDLE>), s))
                else
                    mk_ItemObject(i.type, move_switch(switch, s))
    pre
        wf_item_itemstate(i, s)
        and not check_item_in_position(i, s);
```

```
        -- and wf_switch_move(i.object, s);

    --@doc Moves a specific switch in the aircraft
    move_switch: Switch * SwitchState -> Switch
    move_switch(i, s) ==
        mk_Switch(s, i.middlePosition)
    pre
        wf_switch_move(i, s)
    post
        RESULT.position = s;

    --@doc Checks if the switch is in the on or off position
    check_switch_onoff: Switch -> bool
    check_switch_onoff(s) ==
        let position = s.position in
            position = <OFF> or position = <ON>
    post
        -- Only one can be true at a time
        -- If the switch is in the middle position, then RESULT cannot be true
        -- If the switch is in the on/off position, then the RESULT will be true
        (s.position = <MIDDLE>) <> RESULT;

    --@doc Checks if the item is already in position for the desired state for that item
    check_item_in_position: ItemObject * ItemState -> bool
    check_item_in_position(i, s) ==
        -- if is_Switch(i) then (implement later)
            i.object.position = s
    pre
        wf_item_itemstate(i,s);

    --@doc Checks if the Item.object is the same type for the ItemState
    wf_item_itemstate: ItemObject * ItemState -> bool
    wf_item_itemstate(i, s) ==
        (is_Switch(i.object) and is_SwitchState(s) and i.type = <SWITCH>)
        --TODO check that the item has not already been completed before moving item
        --TODO add other types of Items
        ;

    --@doc Checks if the move of the Switch is a valid
    wf_switch_move: Switch * SwitchState -> bool
    wf_switch_move(i, s) ==
        -- Checks that the switch not already in the desired state
        i.position <> s and
        -- The switch has to move one at a time
        -- Reasoning for this is that some switches cannot be moved in one quick move
        if i.middlePosition = true then
            -- Checks moving the switch away from the middle position
            (i.position = <MIDDLE> and s <> <MIDDLE>)
            -- Checks moving the siwtch to the middle position
            <> (check_switch_onoff(i) = true and s = <MIDDLE>)
        else
```

```
            check_switch_onoff(i) and s <> <MIDDLE>;


end Checklist

/*
//@LF always a good idea to run "qc" on your model. Here is its output. PO 21 and 22 show a problem.
//@LF silly me, this was my encoding with the cases missing one pattern :-). I can see yours has no issues. Good.

> qc
PO #1, PROVABLE by finite types in 0.002s
PO #2, PROVABLE by finite types in 0.0s
PO #3, PROVABLE by finite types in 0.0s
PO #4, PROVABLE by finite types in 0.0s
PO #5, PROVABLE by finite types in 0.0s
PO #6, PROVABLE by finite types in 0.0s
PO #7, PROVABLE by finite types in 0.0s
PO #8, PROVABLE by finite types in 0.0s
PO #9, PROVABLE by finite types in 0.001s
PO #10, PROVABLE by finite types in 0.001s
PO #11, PROVABLE by direct (body is total) in 0.003s
PO #12, PROVABLE by witness s = mk_Switch(<MIDDLE>, true) in 0.001s
PO #13, PROVABLE by direct (body is total) in 0.001s
PO #14, PROVABLE by witness k = mk_Knob(1, [-2]) in 0.0s
PO #15, PROVABLE by direct (body is total) in 0.0s
PO #16, PROVABLE by witness t = 0 in 0.0s
PO #17, PROVABLE by direct (body is total) in 0.001s
PO #18, PROVABLE by witness t = mk_Throttle(0, 0) in 0.001s
PO #19, PROVABLE by direct (body is total) in 0.002s
PO #20, PROVABLE by witness i = mk_ItemObject(<KNOB>, mk_Knob(1, [-1])) in 0.002s
PO #21, FAILED in 0.002s: Counterexample: type = <BUTTON>, object = mk_Knob(1, [-1])
Causes Error 4004: No cases apply for <BUTTON> in 'Checklist' (formal/checklist.vdmsl) at line 119:13
----
ItemObject':␣total␣function␣obligation␣in␣'Checklist'␣(formal/checklist.vdmsl)␣at␣line␣118:13
(forall␣mk_ItemObject'(type, object):ItemObject'!␣&
␣␣is_(inv_ItemObject'(mk_ItemObject'!(type,␣object)),␣bool))

PO␣#22,␣FAILED␣by␣direct␣in␣0.005s:␣Counterexample:␣type␣=␣<BUTTON>
PO␣#23,␣PROVABLE␣by␣witness␣type␣=␣<KNOB>,␣object␣=␣mk_Knob(1,␣[-1])␣in␣0.002s
PO␣#24,␣PROVABLE␣by␣direct␣(body␣is␣total)␣in␣0.001s
PO␣#25,␣PROVABLE␣by␣witness␣i␣=␣mk_ItemAndChecklistItem(mk_ItemObject(<KNOB>,␣mk_Knob(1,␣[-1])),␣mk_ChecklistItem([],␣<KNOB>,␣<MIDDLE>,␣true))␣in␣0.001s
PO␣#26,␣MAYBE␣in␣0.003s
PO␣#27,␣MAYBE␣in␣0.003s
PO␣#28,␣MAYBE␣in␣0.002s
PO␣#29,␣PROVABLE␣by␣witness␣p␣=␣[mk_ChecklistItem([],␣<BUTTON>,␣<MIDDLE>,␣true)]␣in␣0.001s
PO␣#30,␣MAYBE␣in␣0.002s
PO␣#31,␣MAYBE␣in␣0.001s
PO␣#32,␣MAYBE␣in␣0.003s
```

```
PO␣#33,␣MAYBE␣in␣0.002s
PO␣#34,␣MAYBE␣in␣0.001s
PO␣#35,␣MAYBE␣in␣0.002s
PO␣#36,␣MAYBE␣in␣0.009s
PO␣#37,␣MAYBE␣in␣0.008s
PO␣#38,␣MAYBE␣in␣0.007s
PO␣#39,␣MAYBE␣in␣0.009s
PO␣#40,␣MAYBE␣in␣0.002s
PO␣#41,␣MAYBE␣in␣0.001s
PO␣#42,␣MAYBE␣in␣0.001s
PO␣#43,␣MAYBE␣in␣0.002s
PO␣#44,␣MAYBE␣in␣0.002s
PO␣#45,␣MAYBE␣in␣0.003s
PO␣#46,␣MAYBE␣in␣0.002s
PO␣#47,␣MAYBE␣in␣0.002s
PO␣#48,␣MAYBE␣in␣0.001s
PO␣#49,␣MAYBE␣in␣0.001s
PO␣#50,␣MAYBE␣in␣0.0s
PO␣#51,␣MAYBE␣in␣0.0s
PO␣#52,␣MAYBE␣in␣0.005s
PO␣#53,␣PROVABLE␣by␣trivial␣p␣in␣set␣(dom␣checklist)␣in␣0.001s
PO␣#54,␣MAYBE␣in␣0.006s
PO␣#55,␣MAYBE␣in␣0.0s
PO␣#56,␣MAYBE␣in␣0.001s
PO␣#57,␣MAYBE␣in␣0.001s
PO␣#58,␣MAYBE␣in␣0.001s
PO␣#59,␣MAYBE␣in␣0.001s
PO␣#60,␣MAYBE␣in␣0.001s
PO␣#61,␣MAYBE␣in␣0.001s
PO␣#62,␣MAYBE␣in␣0.0s
PO␣#63,␣PROVABLE␣by␣finite␣types␣in␣0.001s
PO␣#64,␣PROVABLE␣by␣finite␣types␣in␣0.001s
PO␣#65,␣PROVABLE␣by␣finite␣types␣in␣0.001s
PO␣#66,␣MAYBE␣in␣0.001s
>
*/
```

Appendix B

Database

B.1 SQL Schemas

```sql
CREATE TABLE IF NOT EXISTS Project (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    name TEXT NOT NULL,
    aircraftType TEXT NOT NULL,
    createdUTC TEXT NOT NULL,
    modifiedUTC TEXT
);

createProject:
INSERT INTO Project(name, aircraftType, createdUTC)
VALUES (?, ?, ?);

selectAllProjects:
SELECT * FROM Project;

selectProjectById:
SELECT * FROM Project
WHERE id = ?;

countProjects:
SELECT COUNT(*) FROM Project;
```

Listing B.2: SQL Schema for Project

```sql
CREATE TABLE IF NOT EXISTS Procedure (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    projectId INTEGER NOT NULL,
    name TEXT NOT NULL,
    type TEXT NOT NULL,
    description TEXT NOT NULL,
    createdUTC TEXT NOT NULL,
    modifiedUTC TEXT,
    FOREIGN KEY (projectId) REFERENCES Project(id)
);

createProcedure:
INSERT INTO Procedure(projectId, name, type, description, createdUTC)
VALUES (?, ?, ?, ?, ?);

selectProcedures:
SELECT * FROM Procedure
WHERE projectId = ?;

selectProcedureById:
SELECT * FROM Procedure
WHERE id = ?;

countProcedures:
SELECT COUNT(*) FROM Procedure
WHERE projectId = ?;
```

Listing B.3: SQL Schema for Procedure

```sql
CREATE TABLE IF NOT EXISTS Action (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    procedureId INTEGER NOT NULL,
    step INTEGER NOT NULL,
    type TEXT NOT NULL,
    goal TEXT NOT NULL,
    FOREIGN KEY (procedureId) REFERENCES Procedure(id)
);

createAction:
INSERT INTO Action(procedureId, step, type, goal)
VALUES (?, ?, ?, ?);

selectActions:
SELECT * FROM Action
WHERE procedureId = ?;

countActions:
SELECT COUNT(*) FROM Action
WHERE procedureId = ?;

deleteByProcedure:
DELETE FROM Action
WHERE procedureId = ?;
```

Listing B.4: SQL Schema for Action

```sql
CREATE TABLE IF NOT EXISTS Test (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    procedureId INTEGER NOT NULL,
    startUTC TEXT NOT NULL,
    endUTC TEXT,
    FOREIGN KEY (procedureId) REFERENCES Procedure(id)
);

startTest:
INSERT INTO Test(procedureId, startUTC)
VALUES (?, ?);

endTest:
UPDATE Test
SET endUTC = ?
WHERE id = ?;

lastInsertedRowId:
SELECT last_insert_rowid();
```

Listing B.5: SQL Schema for Test

```sql
CREATE TABLE IF NOT EXISTS ActionResult (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    testId INTEGER NOT NULL,
    actionId INTEGER NOT NULL,
    initState TEXT NOT NULL,
    endState TEXT,
    startUTC TEXT NOT NULL,
    endUTC TEXT,
    FOREIGN KEY (testId) REFERENCES Test(id),
    FOREIGN KEY (actionId) REFERENCES Action(id)
);

startResult:
INSERT INTO ActionResult(testId, actionId, initState, startUTC)
VALUES (?, ?, ?, ?);

finishResult:
UPDATE ActionResult
SET endState = ?, endUTC = ?
WHERE id = ?;

lastInsertedRowId:
SELECT last_insert_rowid();
```

Listing B.6: SQL Schema for ActionResult

References

[1] Immanuel Barshi, Robert Mauro, Asaf Degani et al. *Designing Flightdeck Procedures*. eng. Ames Research Center, Nov. 2016. URL: https://ntrs.nasa.gov/citations/20160013263.

[2] Atul Gawande. *The Checklist Manifesto: How To Get Things Right*. Main Edition. Profile Books, July 2010. ISBN: 9781846683145.

[3] Civil Aviation Authority. *Aircraft Emergencies: Considerations for air traffic controllers*. CAP 745. Mar. 2005. URL: https://www.caa.co.uk/cap745.

[4] National Tranportation Safety Board. *Loss of Thrust in Both Engines After Encountering a Flock of Birds and Subsequent Ditching on the Hudson River*. Technical Report PB2010-910403. May 2010. URL: https://www.ntsb.gov/investigations/Pages/DCA09MA026.aspx.

[5] William R. Knecht and Michael Lenz. *Causes of General Aviation Weather-Related, Non-Fatal Incidents: Analysis Using NASA Aviation Safety Reporting System Data*. Tech. rep. DOT/FAA/AM-10/13. FAA Office of Aerospace Medicine Civil Aerospace Medical Institute, Sept. 2010.

[6] Civil Aviation Authority. *Guidance on the Design, Presentation and Use of Emergency and Abnormal Checklists*. CAP 676. Aug. 2006. URL: https://www.caa.co.uk/cap745.

[7] Transport Safety Board of Canada. *Aviation Investigation Report In-Flight Fire Leading to Collision with Water Swissair Transport Limited McDonnell Douglas MD-11 HB-IWF Peggy's Cove, Nova Scotia 5 nm SW 2 September 1998*. A98H0003. Feb. 2003. URL: https://www.tsb.gc.ca/eng/rapports-reports/aviation/1998/a98h0003/a98h0003.pdf.

[8] National Tranportation Safety Board. *Aircraft Accident Report Northwest Airlines, Inc c-Donnell Douglas DC-9-82, N312RC, Detroit Metropolitan Wayne County Airport Romulus, Michigan*. PB88-910406. Aug. 1987. URL: https://www.ntsb.gov/investigations/AccidentReports/Reports/AAR8805.pdf.

[9] NASA Langley Formal Methods Research Program. *Langley Formal Methods Program • What is Formal Methods*. URL: https://shemesh.larc.nasa.gov/fm/fm-what.html (visited on 20/05/2024).

[10] Odile Laurent. 'Using Formal Methods and Testability Concepts in the Avionics Systems Validation and Verification (V&V) Process'. In: *2010 Third International Conference on Software Testing, Verification and Validation*. 2010, pp. 1–10. DOI: 10.1109/ICST.2010.38.

[11] Nick Battle. *VDMJ*. URL: https://github.com/nickbattle/vdmj (visited on 21/04/2024).

[12] Peter Gorm Larsen, Kenneth Lausdahl, Peter Jørgensen et al. *Overture VDM-10 Tool Support: User Guide*. TR-2010-02. Apr. 2013. Chap. 16, pp. 81–98. URL: https://raw.github.com/overturetool/documentation/editing/documentation/UserGuideOvertureIDE/OvertureIDEUserGuide.pdf.

[13] Kyushu University. *The VDM Toolbox API*. Version 1.0. 2016. URL: https://github.com/vdmtools/vdmtools/raw/stable/doc/api-man/ApiMan_a4E.pdf.

[14] Raoul-Gabriel Urma. 'Alternative Languages for the JVM'. In: *Java Magazine* (July 2014). URL: https://www.oracle.com/technical-resources/articles/java/architect-languages.html (visited on 05/05/2024).

[15] JetBrains s.r.o. *Kotlin Programming Language*. URL: https://kotlinlang.org/ (visited on 21/04/2024).

[16] Google LLC. *Kotlin and Android | Android Developers*. URL: https://developer.android.com/kotlin (visited on 21/04/2024).

[17] OpenJFX. *JavaFX*. URL: https://openjfx.io/ (visited on 21/04/2024).

[18] FormDev Software GmbH. *FlatLaf - Flat Look and Feel | FormDev.* URL: https://www.formdev.com/flatlaf/ (visited on 21/04/2024).

[19] JetBrains s.r.o. *Compose Multiplatform UI Framework | JetBrains | JetBrains: Developer Tools for Professionals and Teams.* URL: https://www.jetbrains.com/lp/compose-multiplatform/ (visited on 21/04/2024).

[20] Google LLC. *Flutter - Build apps for any screen.* URL: https://flutter.dev/ (visited on 21/04/2024).

[21] Laminar Research. *X-Plane | The world's most advanced flight simulator.* URL: https://www.x-plane.com/ (visited on 21/04/2024).

[22] Lockheed Martin Corporation. *Prepar3D – Next Level Training. World class simulation. Be ahead of ready with Prepar3D.* URL: https://www.prepar3d.com/ (visited on 21/04/2024).

[23] NASA Ames Research Center Diagnostics and Prognostics Group. *X-Plane Connect.* URL: https://github.com/nasa/XPlaneConnect (visited on 21/04/2024).

[24] Nick Battle. *Release 4.5.0 Release · nickbattle/vdmj.* URL: https://github.com/nickbattle/vdmj/releases/tag/4.5.0-release (visited on 22/05/2024).

[25] Koen Claessen and John Hughes. 'Testing Monadic Code with QuickCheck'. In: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* 37 (June 2002). DOI: 10.1145/636517.636527.

[26] Google LLC. *Lists – Material Design 3.* URL: https://m3.material.io/components/lists/guidelines (visited on 13/05/2024).

[27] Google LLC. *Top app bar – Material Design 3.* URL: https://m3.material.io/components/top-app-bar/guidelines (visited on 13/05/2024).

[28] Adriel Café. *Overview | Voyager.* URL: https://voyager.adriel.cafe/ (visited on 13/05/2024).

[29] Koin and Kotzilla. *Koin - The pragmatic Kotlin Injection Framework - developed by Kotzilla and its open-source contributors.* URL: https://insert-koin.io/ (visited on 13/05/2024).

[30] Adriel Café. *Koin integration | Voyager.* URL: https://voyager.adriel.cafe/screenmodel/koin-integration (visited on 13/05/2024).

[31] Square, Inc. *Overview - SQLDelight.* Version 2.0.2. URL: https://cashapp.github.io/sqldelight/2.0.2/ (visited on 14/05/2024).

[32] Hipp, Wyrick & Company, Inc. *How SQLite Is Tested.* URL: https://www.sqlite.org/testing.html (visited on 14/05/2024).

[33] Docker Inc. *What is a Container? | Docker.* URL: https://www.docker.com/resources/what-container/ (visited on 14/05/2024).

[34] Nick Battle. *vdmj/LICENCE at master · nickbattle/vdmj.* URL: https://github.com/nickbattle/vdmj/blob/master/LICENCE (visited on 14/05/2024).

[35] Free Software Foundation, Inc. *The GNU General Public License v3.0 - GNU Project - Free Software Foundation.* URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 14/05/2024).

[36] Free Software Foundation, Inc. *Frequently Asked Questions about the GNU Licenses - GNU Project - Free Software Foundation.* URL: https://www.gnu.org/licenses/gpl-faq.html#IfLibraryIsGPL (visited on 14/05/2024).

[37] Mike Frizzell. *Maven Folder Structure Re-org by frizman21 · Pull Request #227 · nasa/X-PlaneConnect.* URL: https://github.com/nasa/XPlaneConnect/pull/227 (visited on 13/05/2024).

[38] Jason Watkins. *Publish Java library to maven repo · Issue #223 · nasa/XPlaneConnect - Comment.* URL: https://github.com/nasa/XPlaneConnect/issues/223#issuecomment-870819396 (visited on 13/05/2024).

[39] JitPack. *JitPack | Publish JVM and Android libraries.* URL: https://jitpack.io/ (visited on 13/05/2024).

[40] Gradle Inc. *gitRepository.* URL: https://docs.gradle.org/current/kotlin-dsl/gradle/org.gradle.vcs/-source-control/git-repository.html (visited on 13/05/2024).

[41] Jendrik Johannes. *Git repository at <url> did not contain a project publishing the specified dependency.* URL: https://discuss.gradle.org/t/git-repository-at-url-did-not-contain-a-project-publishing-the-specified-dependency/34019/2 (visited on 13/05/2024).

[42] Gradle Inc. *Migrating Builds From Apache Maven.* Version 8.7. 2023. URL: https://docs.gradle.org/current/userguide/migrating_from_maven.html#migmvn:automatic_conversion.

[43] The JUnit Team. *JUnit 5 User Guide - Migrating from JUnit 4*. URL: https://github.com/smyalygames/XPlaneConnect/commit/e7b8d1e811999b4f8d7230f60ba94368e14f1148 (visited on 15/05/2024).