University of Glasgow | School of Computing Science

A Portfolio of Case Studies using the Mungo and StMungo Protocol Typechecking Toolchain

Caitlin Norah MacFadyen

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

Date of submission: 05/09/2019

Abstract

Modern computing relies heavily on the use of communication to build large, complex distributed and concurrent systems. Developers creating these systems must ensure that their programs correctly follow the appropriate communication protocols in order to function properly. The incorporation of protocol typechecking tools into object-oriented programming languages has seen a wealth of research in recent years; one such example being the Scribble-StMungo-Mungo toolchain. It utilizes the theory of *session types* and *typestate* to provide static typechecking, helping programmers to build robust, communication-safe systems. Scribble is a language that can be used to represent the structure of multiparty communication protocols and project local protocols onto each participant. StMungo uses Scribble local protocols to generate typestate specifications and a corresponding Java API. Mungo statically typechecks any code that follows a typestate specification to ensure it is being followed correctly. This project serves as a simple, practical introduction to the Scribble-StMungo-Mungo toolchain and the theory of session types on which it is based. The project contributes to the Mungo website by providing a portfolio of three simple protocol examples–Travel Agent, Bookstore and Adder–which do not require an in-depth understanding of real internet protocols. Each example starts with a Scribble global protocol and describes the complete toolchain workflow, plus a video walkthrough of the step-by-step use of the tools. The portfolio can be used as a teaching tool that will benefit anyone wishing to learn how to use these typechecking tools and understand the theories they are based on.

Education Use Consent

Name: Caitlin Norah MacFadyen **Signature:** [illegible]

Acknowledgements

I would like to thank my supervisor Dr Ornela Dardha for her amazing support and guidance during the course of this project. I was very interested to learn more about session types and typechecking, and Ornela's help was really valuable when I had any concerns or confusion. I'm very glad I got the chance to work on this project–it was challenging but thoroughly enjoyable, inspiring and fascinating!

Contents

Chapter 1 Introduction 1

Chapter 2 Background 2

2.1 Session Types 2

2.2 Scribble 3

2.3 StMungo 4

2.4 Mungo 5

2.5 Analysis, Tools and Techniques 6

Chapter 3 Project Implementation 7

3.1 Using the Toolchain 7

3.2 Travel Agent Example 8

3.3 Bookstore Example 11

3.4 Adder Example 12

3.5 Challenges 15

Chapter 4 Related and Future Work 18

Chapter 5 Conclusions 19

Bibliography 20

Appendix A – Portfolio Setup Instructions 23

Chapter 1 Introduction

Communication is a key concept in modern software development. Systems are becoming increasingly more reliant on communication-based programming, from the increase in complexity and scale of concurrent and distributed systems to the use of multithreading and multi-core processing [1]. Protocols are fundamental to these communications–they describe the way in which participants must communicate with each other [2].

Traditionally, programming languages have been built upon the importance of correctly implementing data types. High-level programming languages have long provided support for typechecking, both static and dynamic, to enforce type-safety and hence minimise bugs produced by violation of data type definitions. However, this support does not currently extend to communication-based programming in mainstream languages, where correct following of protocols is essential for creating error-free systems. Thus, there is a pressing need for a standardised method of safely and correctly programming communications [1]. The idea of using typechecking techniques to improve communication-based programming has seen a growth of interest and led to significant research in recent years. The theory of *session types* is at the heart of this idea: they provide formal descriptions of protocols, describing the order of messages that must be sent and received by communicating participants, along with the data types associated with these messages [3]. Session types can hence be incorporated into a programming language's type system to offer programmers a method of communication protocol typechecking [1].

Two such tools that have been developed at the University of Glasgow based on the theory of session types are Mungo and StMungo. They form a toolchain that provides static protocol typechecking in Java, with the feature of incorporating *multiparty session types* [1, 4, 5]. Mungo implements the notion of *typestate* [6] to allow the association of classes with their corresponding typestate specifications using the `@Typestate` annotation. Typestate specifications describe the permitted sequence of send and receive methods on a given endpoint and can be simply viewed as state machines. Mungo can be used to separately typecheck any code which instantiates a typestate-following class, ensuring that its methods are called in a valid sequence. StMungo takes local protocols, produced by Scribble endpoint projections [2], and generates a suitable typestate specification for each local protocol. It also generates a skeletal Java API which provides an implementation of each endpoint in the communication [4].

The aim of this project is to present a portfolio of simple examples using the Scribble protocol language with the StMungo and Mungo typechecking toolchain. The portfolio shows how the toolchain ensures valid protocol adherence with implementations in Java. The examples used in this project have not previously been implemented using the current versions of the tools. They provide simple, easy-to-follow protocol implementations without the need for an in-depth knowledge of real-world internet protocols.

The concepts of session types and typestate checking can be daunting to those unfamiliar with them, so the portfolio aims to provide a simple introduction using three examples: Adder, Bookstore and Travel Agent. The concepts will first be explained in Chapter 2, where the Scribble, StMungo and Mungo tools will also be introduced. The production of the portfolio is discussed in Chapter 3, explaining in detail how each example is implemented. For each example there is an introduction to the protocol and an explanation of the toolchain usage, plus a full video walkthrough which can be found on the Mungo website [7] and YouTube [8, 9, 10]. Chapter 4 describes further work that could provide a more in-depth, practical example to include in the portfolio, and Chapter 5 contains concluding remarks.

Chapter 2 Background

2.1 Session Types

First introduced by Honda *et al* [11, 12], *session types* were developed for process calculi as a way of formalising the structure of communications. The collection of communications that form a program were introduced as *sessions*; the interactions in each session occur over a *channel* [11]. Session types define the sequences of send and receive actions that specify a communication over a channel, as well as the data types associated with each message and their direction. They also incorporate choices, both internal and external, that allow communications to proceed differently based on the selections made by particular participants [3]. These will be discussed further in Sections 2.2 and 3 in the context of the Scribble language and within the portfolio of examples.

There has been significant research in recent years on applying session types to mainstream programming languages in order to improve the quality, safety, efficiency and production cost of communication-based software. It is the focus of the ABCD project [13], running since 2013, that has given rise to many publications and resources including the Mungo and StMungo tools. Work by Gay *et al.* [14] in 2010 introduced the idea of using *typestate* to incorporate session types into object-oriented programming. Their work utilised the features of object-orientation to apply modular session types to both classes and individual communication channels, adding to the type system to allow protocol typechecking of these session-typed objects.

In order to understand both how typestate checking works, and the motivation for incorporating it into mainstream programming languages, the idea of *typestate* itself must be explored first. First introduced by Strom and Yemeni in 1986 [6], typestate is a concept built upon the notion of types in programming languages. It adds context to types, using pre- and post-conditions around methods to specify the current state of the object. It defines sequences of method calls that are allowed on objects when they are in a particular state, rather than the list of permitted methods being the same at any point during compile-time or runtime as with standard types. This has obvious benefits for communication-based programs, where the sequence, direction and type of messages are all key to their successful execution [1].

The Mungo and StMungo tools further build on previous research on session types and typestate in object-oriented languages, including Gay *et al*'s work [14]. The toolchain improves the previous work, most notably by incorporating the use of *multiparty session types* [5, 15], allowing more complex communications to be represented. Initially, research in the field focused on binary session types that relied on the concept of duality–where the send methods of the first participant must match the receive methods of the second, and vice versa [3]. Multiparty session types [15] extend the traditional notion of binary session types, which limited the modelling of communications to those that only involved two participants. In 2008, Honda, Yoshida and Carbone [15] applied the theory of multiparty session types to the pi-calculus. They introduced the idea of formalising *global types*: representations of multiparty communication protocols that take into account interleaving behaviours of each participant in the communication. Participants each own a *local endpoint* in the session channel; these endpoints can be projected from global types (see section 2.2). In essence, global types describe the communication between multiple endpoints as a whole, showing the order and type of interactions between all parties. This insight is incorporated into session type tools to aid programmers in several ways, for example to achieve communication safety or to eliminate deadlocks and other errors that arise in concurrent programming [15, 16].

2.2 Scribble

Multiparty session types form the basis of the Scribble protocol language [18]. Inspired by Kohei Honda's work at the (now closed) W3C Web Services Choreography Description Working Group [17], the development of Scribble sought to simplify the representation of application-level protocols [2]. Today, the Scribble tool provides several services to programmers to support the development of communicating systems. It serves as a human-readable language to describe *global* and *local protocols* that represent multiparty communications. In contrast to global protocols that describe communications from a global viewpoint, local protocols describe communications from the viewpoint of a single participant, showing how it interacts with the rest of the party [4]. Scribble's validation service checks the well-formedness and validity of global protocols. The tool also performs *endpoint projection*, as discussed in 2.1, that derives a local version of the global protocol for each role in the communication based on the theory of multiparty session types. In addition, Scribble provides endpoint API generation for both Java and Scala, plus endpoint monitoring to guarantee correct behaviour of endpoint implementations [2, 18]. This project will demonstrate the use of validation and endpoint projection with the Scribble-Java tool, which can be cloned or downloaded from the GitHub repository at [19]. To introduce the syntax of Scribble protocols and briefly introduce endpoint projection, a simple "Hello World" example [19] is provided below:

```
global protocol Hello(role C, role S) {
  Hello(String) from C to S;
}
```

Figure 1.1 – simple global protocol in Scribble

```
local protocol Hello_C(self C, role S) {
  Hello(String) to S;
}
```

Figure 1.2 – Endpoint projection of role C

```
local protocol Hello_S(role C, self S) {
  Hello(String) from C;
}
```

Figure 1.3 – Endpoint projection of role S

The global protocol `Hello` in *Figure 1.1* represents the global viewpoint of this simple two-party communication. The role declaration list `(role C, role S)` specifies each participant that can be projected to an endpoint, represented by a local protocol. After endpoint projection, the `Hello_C` local protocol in *Figure 1.2* represents a client which simply sends a hello message to the server S, represented by the `Hello_S` local protocol in *Figure 1.3*, whom receives this message. The `self C` and `self S` keywords denote that the local endpoints are `C` and `S` respectively. The body of the protocol contains the message signatures for messages that must be sent and received – in this case simply `Hello(String)`. This includes the message operator `Hello` and the payload type `(String)`. Payload type declarations must also be included to specify the host language [20], in this case Java; these can be seen in the project videos at [7, 8, 9, 10].

As Scribble protocols become more complex, we see the introduction of ideas like *internal* and *external choice* and *recursive protocols*. The `choice` statement `choice at r { ... } or { ... }` is found at branch points at which the protocol can proceed in more than one direction. Here the *choice subject*, who makes the choice, is `r`. The choice in this example would be internal to `r` but external to the other roles in the communication. All three examples in the portfolio will utilise the choice feature. Recursive protocols, as represented by `rec X` in the Adder example in section 3.4, describe protocols that may loop back to a certain state based on the outcome of a choice [20]. Further Scribble features like *explicit* and *auxiliary global protocols* are available, but these are beyond the scope of the project. The Scribble language can be used in conjunction with Mungo and StMungo to take advantage of the benefits of protocol typechecking and build robust, communication-safe implementations of protocols.

2.3 StMungo

Once local protocols have been derived by Scribble endpoint projection, the StMungo tool [1, 4, 5] comes into play. StMungo, short for Scribble-to-Mungo, is so-called because it generates protocol implementation code that can be typechecked by Mungo. It takes Scribble local protocols and translates them into typestate specifications. It also provides a skeletal Java API for the implementation of each role in the protocol. StMungo works by abstracting each role in a communication into separate Java classes, each of which follows the associated typestate. These classes–together with the typestate specifications and main classes for instantiating role objects–can form the foundations of error-free communication systems. StMungo is a unique tool in that it offers the first integration of Scribble protocols and typestate specifications into an object-oriented language [4].

Typestate specifications, which define communication protocols as state machines, are given in a Java-like syntax. They are provided separately in .protocol files, like CProtocol.protocol and SProtocol.protocol in *Figures 2.1* and *2.2* below:

```
typestate CProtocol {
  State0 = {
    void send_HelloStringToS(String): end
  }
}
```

Figure 2.1 – CProtocol.protocol Typestate specification derived for role C

```
typestate SProtocol {
   State0 = {
      String receive_HelloStringFromC(): end
   }
}
```

Figure 2.2 – SProtocol.protocol Typestate specification derived for role S

The above typestate specifications are produced by running StMungo on the Scribble local protocols in *Figures 1.2* and *1.3*. Typestate specifications consist of one or more numbered states, which represent the state that the channel object is currently in. When methods are called on objects, it is likely that their typestate will change, proceeding to the next available state [5]. Both of these protocols contain only `State0` and `end`. This means that any object following the `CProtocol` typestate simply calls the `send_HelloStringToS(String)` method and the protocol proceeds to the `end` state, where it terminates. Any object following the `SProtocol` typestate first connects to the `CProtocol`-following object, then calls the `receive_HelloStringFromC()` method and terminates with the `end` state. More complex examples using multiparty communications will be discussed in Chapter 3.

StMungo produces a Role class for each local endpoint involved in the communication, which represents channel objects that follow the given typestate specification. In the above example, StMungo would produce `CRole` and `SRole` Java classes. Each Role class contains a basic implementation of each method specified by each state in the typestate, plus a constructor that contains Socket-based connection code. When the Role class is instantiated, the object connects to other Role objects participating in the session via Java Sockets [5].

StMungo also produces a main class to instantiate each role, for example `CMain` and `SMain` for the above example. These main classes create the Role object and use it to invoke each of the methods in the associated typestate in order, so they can then be separately typechecked by Mungo. The API produced by StMungo provides only a basic implementation of a communication, so it can be built upon based on the programmer's view for the final implementation. They can add extra logic to suit their requirements, as long as the typestate is not violated. The API itself does not need to be used, the typestate specifications alone can be utilised to build a separate implementation. The typestates themselves can also be altered, for example by naming states, to better suit the programmer's needs [4]. The StMungo tool is developed and maintained by Dr. Ornela Dardha. It can be downloaded from the Bitbucket repository [21].

2.4 Mungo

Mungo is a front-end tool for Java that performs static typestate checking, based on the idea of channel objects following typestate specifications [14]. It adds a *Typestate* definition to Java, which can be used alongside Java's standard type system to extend its coverage to include typestate checking. While typestate was traditionally introduced based on pre- and post-conditions on methods [6], Mungo's typestate checking technique diverges from this notion by defining each typestate in a separate file. This allows separate typechecking of each typestate-following object, which guarantees communication safety if every role adheres to its typestate [5]. A typestate specification is associated with a Java class–e.g. `CRole`– via the `@Typestate` annotation:

```
@Typestate("CProtocol")
public class CRole { … }
```

Here, the `CProtocol` typestate is adhered to by the class `CRole`, thus the class must implement the methods defined by each state [4]. The Mungo tool uses the JastAdd Reference Attribute Grammar meta-compiler suite to implement parsers for both the Mungo typechecker and the typestate specification language [1]. The Mungo typechecker can be run on any code which instantiates a channel object that follows a typestate specification. When it is run, an inferred typestate is produced, and the sequences of method calls on typestate-following objects are constructed and checked to ensure they follow the minimum inferred typestate. Aliasing can lead to inconsistencies in typestate, so Mungo controls this through linear use of objects [5]. The typestate inference system was formalised as a core object-oriented calculus in [5], however, this formalisation will not be covered in the project. If no errors are reported by Mungo, the code can be compiled and run as standard in Java 1.8 [4].

Mungo completes the session type toolchain by providing typestate checking for channel objects. It should be run once the final implementation of the communication has been completed, i.e., once Scribble and StMungo have been used and any appropriate alterations have been made to the API. Mungo typechecking currently covers a subset of Java. Coverage for generics, inheritance, exceptions and other features is not yet supported and require further work to be included in future releases [1]. Mungo was developed by Dr Dimitrios Kouzapas, and it is maintained by the ABCD team at University of Glasgow [13]. There is an older and a newer version of the Mungo tool, which run on different versions of Java: 1.4 and 1.8, respectively. Some work was required at the beginning of the project in order to cope with and deal with these differences, which will be discussed further in section 3.5. The new version of the tool (version 1.1) can be downloaded from the Mungo website [7].

2.5 Analysis, Tools and Techniques

As discussed in the previous chapters, the incorporation of typestate checking into mainstream programming languages could prove essential for the progression of modern computing [13]. This would mean that more programmers would need to gain a working knowledge of the theory of session types and the typechecking tools based on it. The aim of this project is to provide a portfolio of cases studies for the Mungo and StMungo tools that serves as a practical introduction to the subject, contributing to the Mungo website by providing step-by-step videos of the toolchain usage. The Background section in Chapter 2 has provided a comprehensive outline of the underlying concepts. Chapter 3 will now discuss the implementation of three examples using Scribble, StMungo and Mungo, namely Travel Agent, Bookstore and Adder. The first two provide examples of multiparty communications, the third demonstrates how a recursive protocol is dealt with. The general workflow of using the toolchain will be introduced, including the files that each tool runs on and the files produced. Each example will then be discussed in more depth and the chapter will conclude with the challenges that were faced during implementation. The following versions of each tool were used during the implementation of the project:

- Scribble-Java version 0.4, cloned from [19]
- StMungo version 1.1, cloned from [21]

- Mungo version 1.1, cloned from [22]
- Java version 1.8.0_211, downloaded from [23] – see section 3.5
- Cygwin64 Terminal version 3.0.7, downloaded from [24] – see section 3.5

In order to produce the video walkthroughs of the toolchain usage, the screen-capturing software Flashback Express Recorder was used. Version 5.36 was downloaded from [25]. To edit and add captions to the video, I used the free, open-source video editor OpenShot version 2.4.4, which was downloaded from [26].

Chapter 3 Project Implementation

3.1 Using the Toolchain

The general workflow of the Scribble-StMungo-Mungo toolchain is as follows:

1. Start by using the Scribble tool on Scribble global protocol (.scr) files for validation and endpoint projection. The Scribble-Java command line tool was used in the project.
   a. For each given example there is one global protocol, which projects to either two or three local protocols depending on how many participants there are.
   b. Endpoint projection derives local protocols that can be saved in separate Scribble files.
2. Once the local protocol files have been saved, run stmungo.jar on all local protocol (.scr) files.
   a. The local protocol module can be altered to specify where to place the files produced by StMungo, e.g. `module portfolio_Adder_C` will create a 'portfolio' package containing an 'Adder' folder in which the files will be saved.
3. Add logic to flesh out the skeleton socket-based Java API, or use typestate specification to create alternative API.
4. Run mungo.jar (using Java 1.8) on any Java code that instantiates a class that follows a typestate specification. Mungo provides useful error reporting that identifies the cause of any typestate violation.
5. If Mungo does not report any errors, compile and run the code as normal.

The following alterations were made in each example. The local protocol modules were altered in order to save files into a portfolio package. After running StMungo, the Sockets and ServerSockets in each produced Java file had to be properly configured to allow communication between each party, i.e., by changing port numbers appropriately. To achieve a complete run-through of the communications, some of the methods had to be altered, Typestate had to be imported from Mungo.lib in Role classes, and extra dialog was added to make the communications easier to follow. The programs in each communication were run on separate terminals to represent each of the parties involved in the communication. The following sections will discuss the individual implementations and workflow of each example in the portfolio. Appendix A provides instructions on how to set up and run the portfolio examples using the tools.

3.2 Travel Agent Example

The first example in the portfolio is that of the travel agent. This example models a three-party communication using a scenario where a university researcher wishes to organise travel for research purposes. The three participants involved in the example are the Researcher (R) who wishes to travel, the Agent (A) who makes the bookings, and the Finance department (F) who controls the university's budget. The communication begins when the Researcher requests a quote from the Agent, who sends it back. The Researcher then sends the quote to Finance who must approve or refuse travel. If Finance approves, they must send an approval code to both the Agent and the Researcher; the Agent then sends a ticket String to the Researcher and an invoice code back to Finance. When the Researcher's ticket has been received, their role in the communication is complete. Finally, when Finance receives the invoice code, they can send back the payment to the Agent and the communication is complete. Conversely, if Finance refuses the travel request, they simply send a String explaining the refusal to the Researcher and the Agent.

In order to implement this example, I started by reviewing the following BuyTicket Scribble global protocol, found at [1]:

```
global protocol BuyTicket(role R, role A, role F) {
   request(String) from R to A;
   quote(int) from A to R;
   check(int) from R to F;
   choice at F {
      approve(int) from F to R;
      approve(int) from F to A;
      ticket(String) from A to R;
      invoice(int) from A to F;
      payment(int) from F to A;
   } or {
      refuse(String) from F to R;
      refuse(String) from F to A;
   }
}
```

Figure 3.1 – BuyTicket (Travel Agent) global protocol

The local protocol files had already been defined at the StMungo Bitbucket repository [21], which were used initially, but the final example begins with the Scribble global protocol as seen in *Figure 3.1* above, and in the video [7, 10].
I began this example by running StMungo on the local protocols. I did run into some difficulties during this process which became apparent when trying to implement the logic in the resulting Java API; this will be discussed in detail in Section 3.5. Once the appropriate changes to the global protocol were made, I was able to successfully run StMungo on the three Scribble local protocol files – travel-agent_Agent.scr, travel-agent_Research.scr and travel-agent_Finance.scr. The local protocol for the Finance role, as produced by StMungo, is given below in *Figure 3.2*.

```
local protocol BuyTicket_F(role R, role A, self F) {
  check(int) from R;
  choice at F {
    approve(int) to R;
    approve(int) to A;
    invoice(int) from A;
    payment(int) to A;
  } or {
    refuse(String) to R;
    refuse(String) to A;
  }
}
```

Figure 3.2 – travel-agent_Finance.scr local protocol

The following skeleton files were produced as a result of running StMungo on the above file:

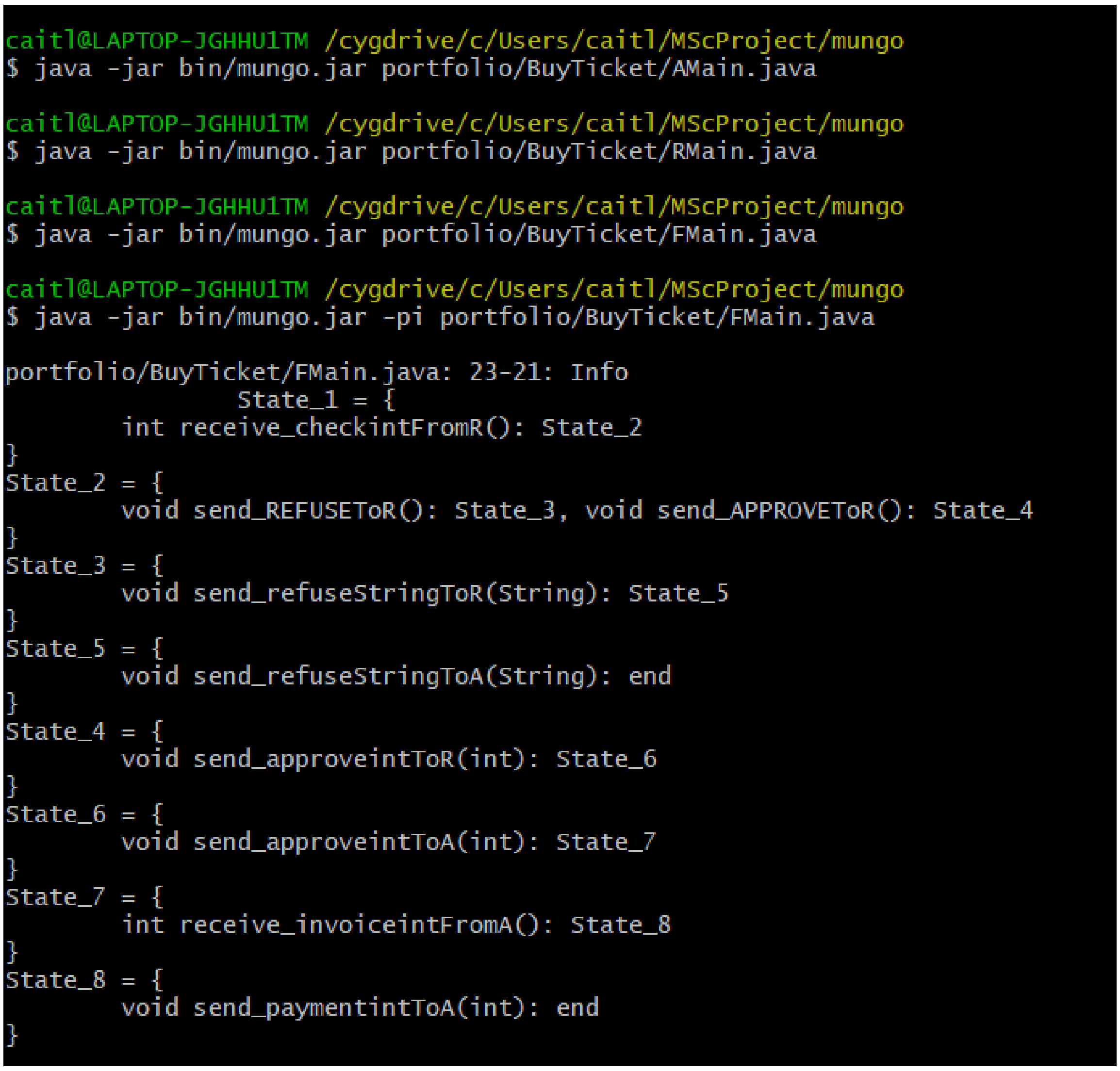
- FProtocol.protocol
- FRole.java
- FMain.java
- Choice1.java

The typestate specification in FProtocol.protocol was defined as follows:

```
typestate FProtocol {
  State0 = {
    int receive_checkintFromR(): State1
  }
  State1 = {
    void send_APPROVEToR(): State2,
    void send_REFUSEToR(): State6
  }
  State2 = {
    void send_approveintToR(int): State3
  }
  State3 = {
    void send_approveintToA(int): State4
  }
  State4 = {
    int receive_invoiceintFromA(): State5
  }
  State5 = {
    void send_paymentintToA(int): end
  }
  State6 = {
    void send_refuseStringToR(String): State7
  }
  State7 = {
    void send_refuseStringToA(String): end
  }
}
```

Figure 3.3 – Fprotocol.protocol typestate specification

As seen in *Figures 3.2* and *3.3* above, StMungo converts each *message* in the local protocol to a *method* in the typestate specification, incorporating the return type and payload type. Each line also defines which state the associated object proceeds to after it returns from a method [5]. The subsequent state can differ depending on the result of a choice, as seen at `State1`, which represents an internal choice at `F`. Here, the object will either proceed to `State2` if `APPROVE` is chosen, or `State6` if `REFUSE` is chosen. These choices are represented as enumerated types in Choice1.java, produced by StMungo. For each local protocol, StMungo produces a Role class, a Main class and a Protocol typestate specification. The Role classes are those which follow the typestate specification. These classes make use of the `@Typestate` annotation to denote the protocol that should be followed by any class that instantiates a Role object. For example, any class that instantiates an `FRole` object must make sure it follows the `FProtocol` typestate specification, calling the methods in the correct sequence in `FMain`. This file can then be typechecked by Mungo:



```
caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/mungo
$ java -jar bin/mungo.jar portfolio/BuyTicket/AMain.java

caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/mungo
$ java -jar bin/mungo.jar portfolio/BuyTicket/RMain.java

caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/mungo
$ java -jar bin/mungo.jar portfolio/BuyTicket/FMain.java

caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/mungo
$ java -jar bin/mungo.jar -pi portfolio/BuyTicket/FMain.java

portfolio/BuyTicket/FMain.java: 23-21: Info
                State_1 = {
        int receive_checkintFromR(): State_2
}
State_2 = {
        void send_REFUSEToR(): State_3, void send_APPROVEToR(): State_4
}
State_3 = {
        void send_refuseStringToR(String): State_5
}
State_5 = {
        void send_refuseStringToA(String): end
}
State_4 = {
        void send_approveintToR(int): State_6
}
State_6 = {
        void send_approveintToA(int): State_7
}
State_7 = {
        int receive_invoiceintFromA(): State_8
}
State_8 = {
        void send_paymentintToA(int): end
}
```

Figure 3.4 – Running Mungo on Travel agent main classes

Figure 3.4 shows how Mungo is run in the Cygwin64 terminal: since it displays no error messages, it means there are no typestate violations. The use of the `-pi` flag is also shown, which displays the inferred types that the Mungo typechecker uses to check against the method calls in the `FRole` object's lifetime [1]. As seen in the video walkthrough [7, 10], when a method call is removed from `FMain`, Mungo reports a semantic error that describes a typestate mismatch, because the expected order of method calls is incorrect. When the method call is replaced, the error is resolved.

3.3 Bookstore Example

The Bookstore example is similar to the Travel Agent in that it involves a multiparty communication between three participants. The Bookstore global protocol in *Figure 4.1* was adapted from the TwoBuyer example in the Scribble-Java GitHub repository [19].

```
global protocol Bookstore(role Seller, role Buyer2, role Buyer1) {
    book(String) from Buyer1 to Seller;
    book(int) from Seller to Buyer1;
    quote(int) from Buyer1 to Buyer2;
    choice at Buyer2 {
          agree(String) from Buyer2 to Buyer1;
          agree(String) from Buyer2 to Seller;
          transfer(int) from Buyer1 to Seller;
          transfer(int) from Buyer2 to Seller;
    } or {
          quit(String) from Buyer2 to Buyer1;
          quit(String) from Buyer2 to Seller;
    }
}
```

Figure 4.1 – Bookstore.scr global protocol

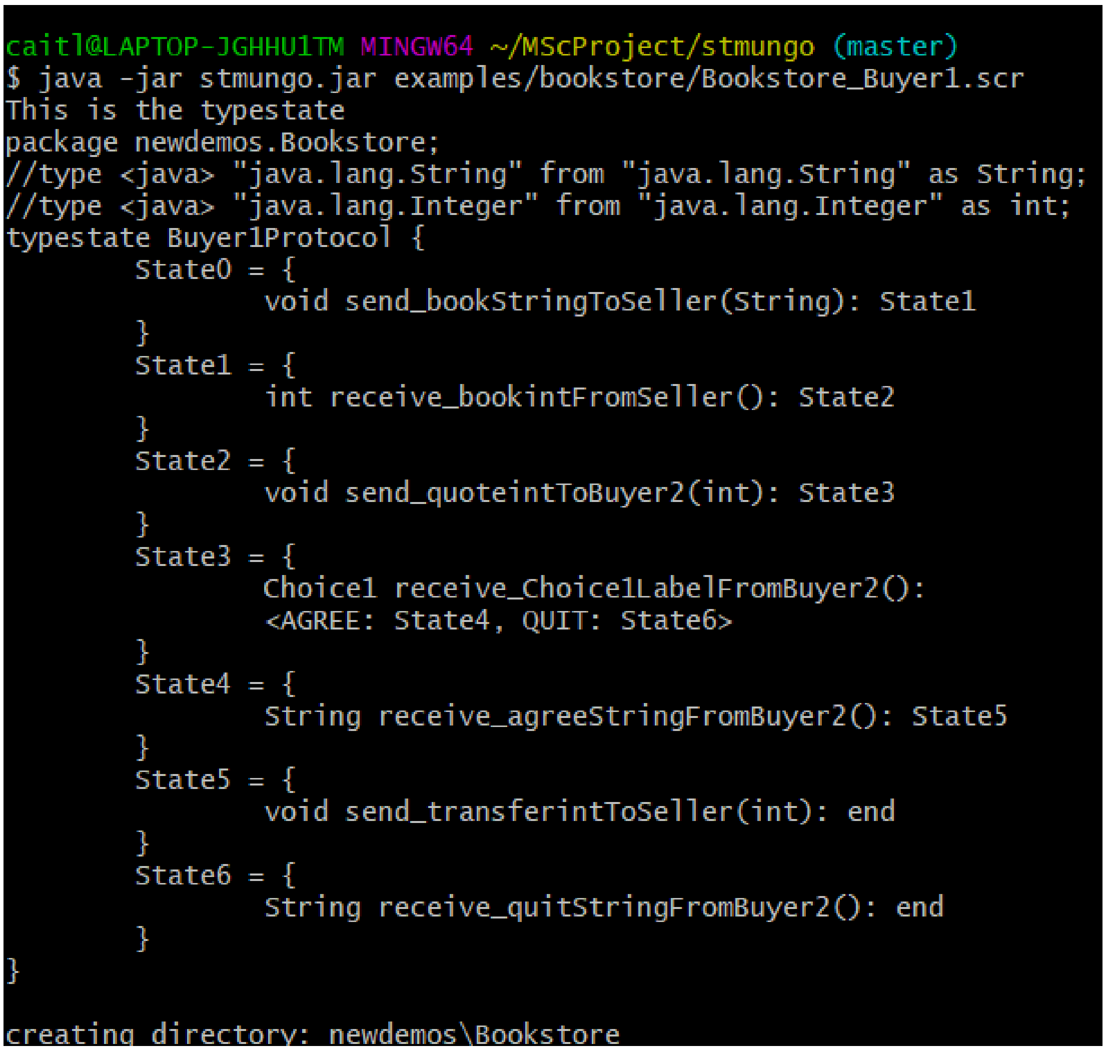
As with the Travel Agent, the Scribble local protocol files for Buyer1, Buyer2 and Seller can be found at the StMungo Bitbucket repository [21]. In this example, there is the Seller, the first buyer (Buyer1) and the second buyer (Buyer2). The scenario is that Buyer1 sends the title of the book they wish to purchase to the Seller, who returns its price. Buyer1 cannot afford the book alone, so sends a quote to Buyer2 who will potentially pay a portion of the cost. There is a choice in Buyer2's protocol specification as they decide whether to agree to send the quoted payment to the Seller. If Buyer2 agrees, they send a String confirming their agreement to both other participants. Then Buyer1 sends their part-payment to the Seller, and Buyer2 follows with the rest of the payment. If they quit, a quit message is sent to Buyer1 and Seller respectively.

```
local protocol Bookstore_Buyer1(role Seller, role Buyer2, self
Buyer1) {
    book(String) to Seller;
    book(int) from Seller;
    quote(int) to Buyer2;
    choice at Buyer2 {
        agree(String) from Buyer2;
        transfer(int) to Seller;
    } or {
        quit(String) from Buyer2;
    }
}
```

Figure 4.2 – Bookstore_Buyer1.scr local protocol

Figure 4.2 above demonstrates how the external choice at `Buyer2`'s local protocol is seen in `Buyer1`'s local protocol. While the choice must be made by `Buyer2`, it is also present as an external choice within the local protocols of `Seller` and `Buyer1`. This is because the progression of all three parties depends on the result of `Buyer2`'s choice, namely `Choice1.AGREE` or `Choice1.QUIT`.

In *Figure 4.3* below, we see the command used to run StMungo on `Buyer1`'s local protocol. The derived typestate specification is seen along with the directory in which all the produced files are saved. Note that here the files are saved in the 'newdemos' directory; the final versions were moved to the 'portfolio' directory by changing the module specification in the local protocol.



```
caitl@LAPTOP-JGHHU1TM MINGW64 ~/MScProject/stmungo (master)
$ java -jar stmungo.jar examples/bookstore/Bookstore_Buyer1.scr
This is the typestate
package newdemos.Bookstore;
//type <java> "java.lang.String" from "java.lang.String" as String;
//type <java> "java.lang.Integer" from "java.lang.Integer" as int;
typestate Buyer1Protocol {
        State0 = {
                void send_bookStringToSeller(String): State1
        }
        State1 = {
                int receive_bookintFromSeller(): State2
        }
        State2 = {
                void send_quoteintToBuyer2(int): State3
        }
        State3 = {
                Choice1 receive_Choice1LabelFromBuyer2():
                <AGREE: State4, QUIT: State6>
        }
        State4 = {
                String receive_agreeStringFromBuyer2(): State5
        }
        State5 = {
                void send_transferintToSeller(int): end
        }
        State6 = {
                String receive_quitStringFromBuyer2(): end
        }
}

creating directory: newdemos\Bookstore
```

Figure 4.3 – StMungo running on Bookstore_Buyer1.scr

The complete code produced by StMungo is seen in the video, along with a full demonstration of the typechecking and running of the final programs [7, 9].

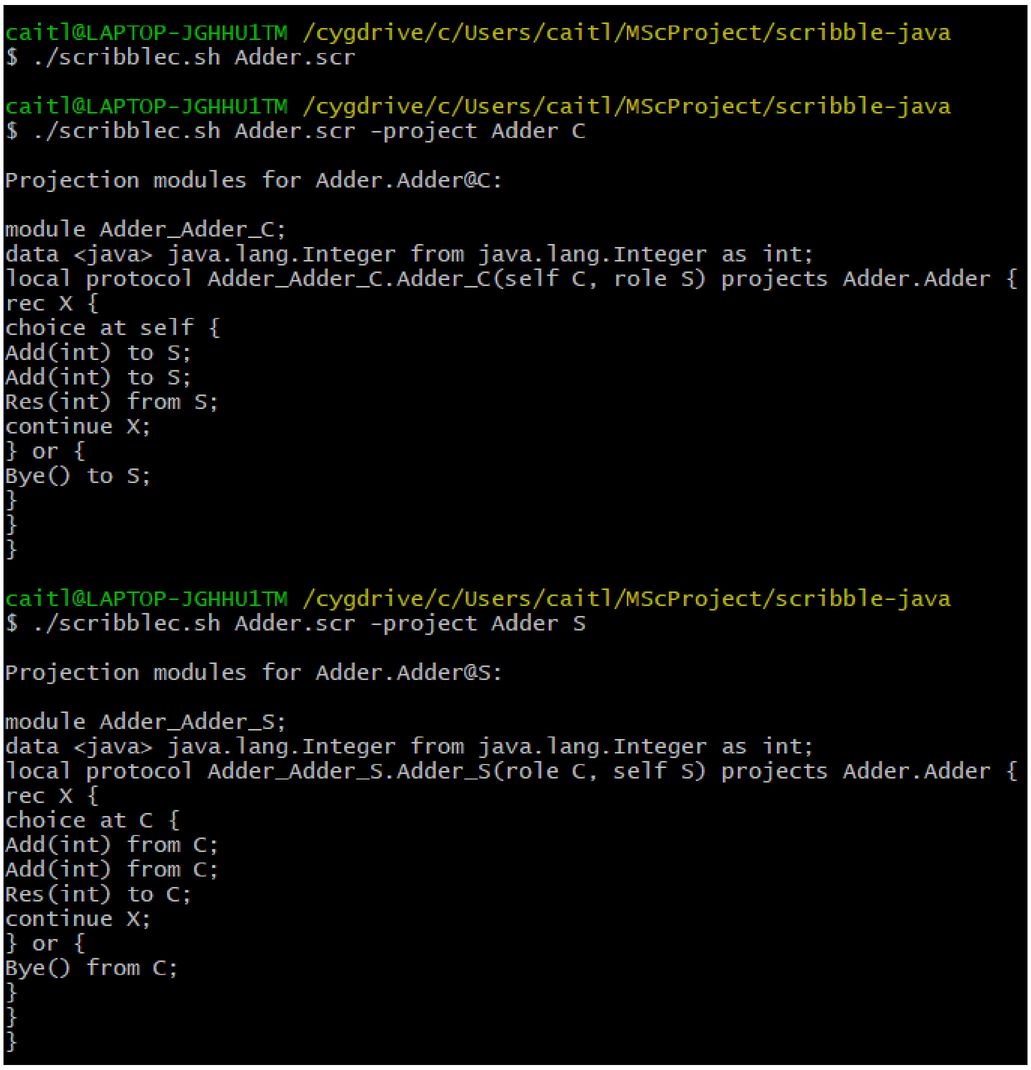
3.4 Adder Example

The third and final example in the portfolio is the recursive Adder example, which involves two participants: C and S. The body of the protocol essentially involves the client, C, sending two integers to the server, S, which adds the two numbers and returns the result to C. This example, however, also shows the use of recursive protocols and how they are handled by Scribble and StMungo. The Adder global protocol is defined below in *Figure 5.1*.

```
global protocol Adder(role C, role S) {
   rec X {
      choice at C {
         Add(int) from C to S;
         Add(int) from C to S;
         Res(int) from S to C;
         continue X;
      } or {
         Bye() from C to S;
      }
   }
}
```

Figure 5.1 – Adder.scr global protocol

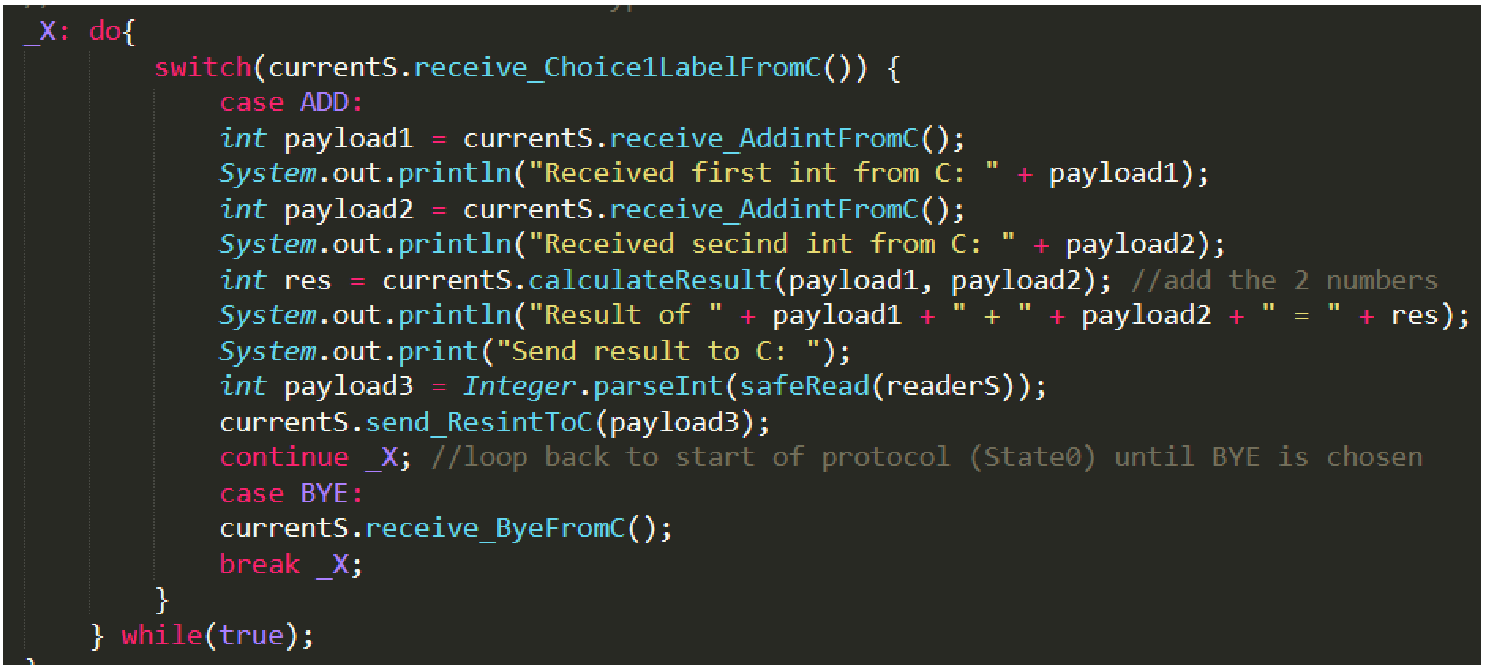
Recursion is defined by the `rec X {… continue X}` statement around the body of the protocol. The choice at `role C` requires the user to choose between the enumerations `ADD` and `BYE`. If `ADD` is chosen, the two integers are sent to `S` and the result of the addition is sent back. When `continue X` is reached, the protocol loops back to `rec X` and asks the user to choose again. This continues indefinitely until `BYE` is chosen, whereupon the `Bye()` method is called and the protocol terminates. The Adder global protocol can be used by Scribble as follows:



```
caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/scribble-java
$ ./scribblec.sh Adder.scr

caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/scribble-java
$ ./scribblec.sh Adder.scr -project Adder C

Projection modules for Adder.Adder@C:

module Adder_Adder_C;
data <java> java.lang.Integer from java.lang.Integer as int;
local protocol Adder_Adder_C.Adder_C(self C, role S) projects Adder.Adder {
rec X {
choice at self {
Add(int) to S;
Add(int) to S;
Res(int) from S;
continue X;
} or {
Bye() to S;
}
}
}

caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/scribble-java
$ ./scribblec.sh Adder.scr -project Adder S

Projection modules for Adder.Adder@S:

module Adder_Adder_S;
data <java> java.lang.Integer from java.lang.Integer as int;
local protocol Adder_Adder_S.Adder_S(role C, self S) projects Adder.Adder {
rec X {
choice at C {
Add(int) from C;
Add(int) from C;
Res(int) to C;
continue X;
} or {
Bye() from C;
}
}
}
```

Figure 5.2 – Scribble validation and endpoint projection on Adder.scr

The global protocol is validated by Scribble as given in the first command in *Figure 5.2*. The endpoint projection is then carried out using the `-project` command with module (`Adder`) and role (`C`) arguments. Local protocols can simply be copied and pasted into appropriate files to be used by StMungo.
StMungo deals with this recursive example by producing a labelled `do-while(true)` loop in each main class:



```
_X: do{
        switch(currentS.receive_Choice1LabelFromC()) {
            case ADD:
            int payload1 = currentS.receive_AddintFromC();
            System.out.println("Received first int from C: " + payload1);
            int payload2 = currentS.receive_AddintFromC();
            System.out.println("Received secind int from C: " + payload2);
            int res = currentS.calculateResult(payload1, payload2); //add the 2 numbers
            System.out.println("Result of " + payload1 + " + " + payload2 + " = " + res);
            System.out.print("Send result to C: ");
            int payload3 = Integer.parseInt(safeRead(readerS));
            currentS.send_ResintToC(payload3);
            continue _X; //loop back to start of protocol (State0) until BYE is chosen
            case BYE:
            currentS.receive_ByeFromC();
            break _X;
        }
    } while(true);
```

Figure 5.3 – `do-while(true)` loop in SMain.java

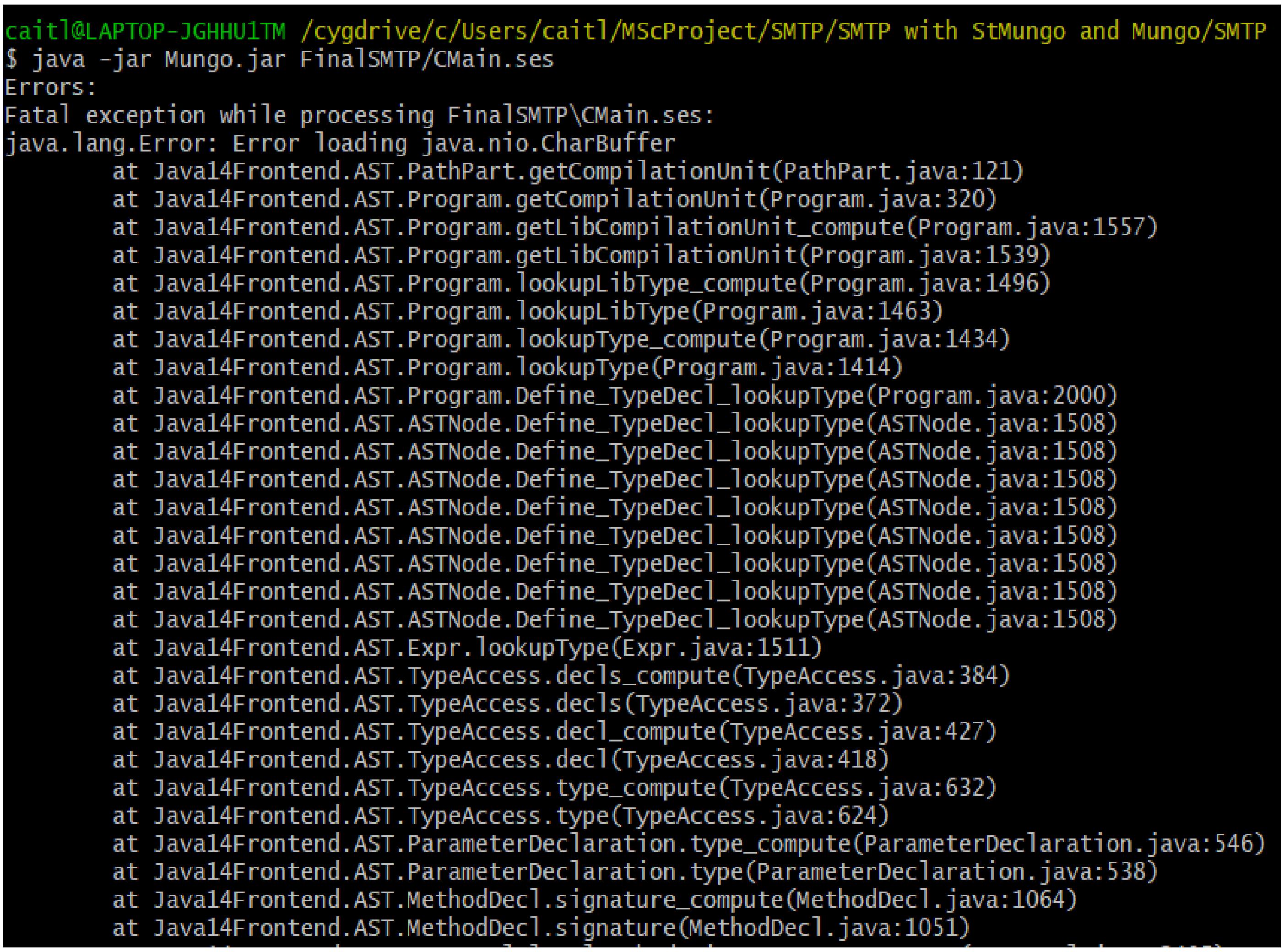
The code combines the loop, labelled `_X`, with a `switch` that deals with the choice itself by inspecting the `Choice1` enumeration returned by the method `currentS.receive_Choice1LabelFromC()`. Each `case` represents one of the enumerated choices, which are used to determine whether to return to the start of the loop using `continue _X` or terminate by `break _X` [5].

As with the rest of the examples, video recordings have been made demonstrating a full walkthrough of the toolchain usage; starting from validating and projecting Scribble global protocols to running StMungo on each local protocol and finally typechecking with Mungo and running the final programs. These videos can be found at the Mungo website [7] and on YouTube [8].

3.5 Challenges

A number of challenges arose during this project that had to be overcome in order to successfully complete the portfolio of work. At the beginning of the project, I was completely unfamiliar with the concept of session types and typechecking, and was relatively new to socket programming and command line usage. I started by reviewing the existing literature on session types and the uses of typechecking in modern computing; I was glad to have the opportunity to work on such an exciting, thought-provoking field. Achieving an inclusive theoretical knowledge of the subject proved to be a steep learning curve which required significant time and effort.

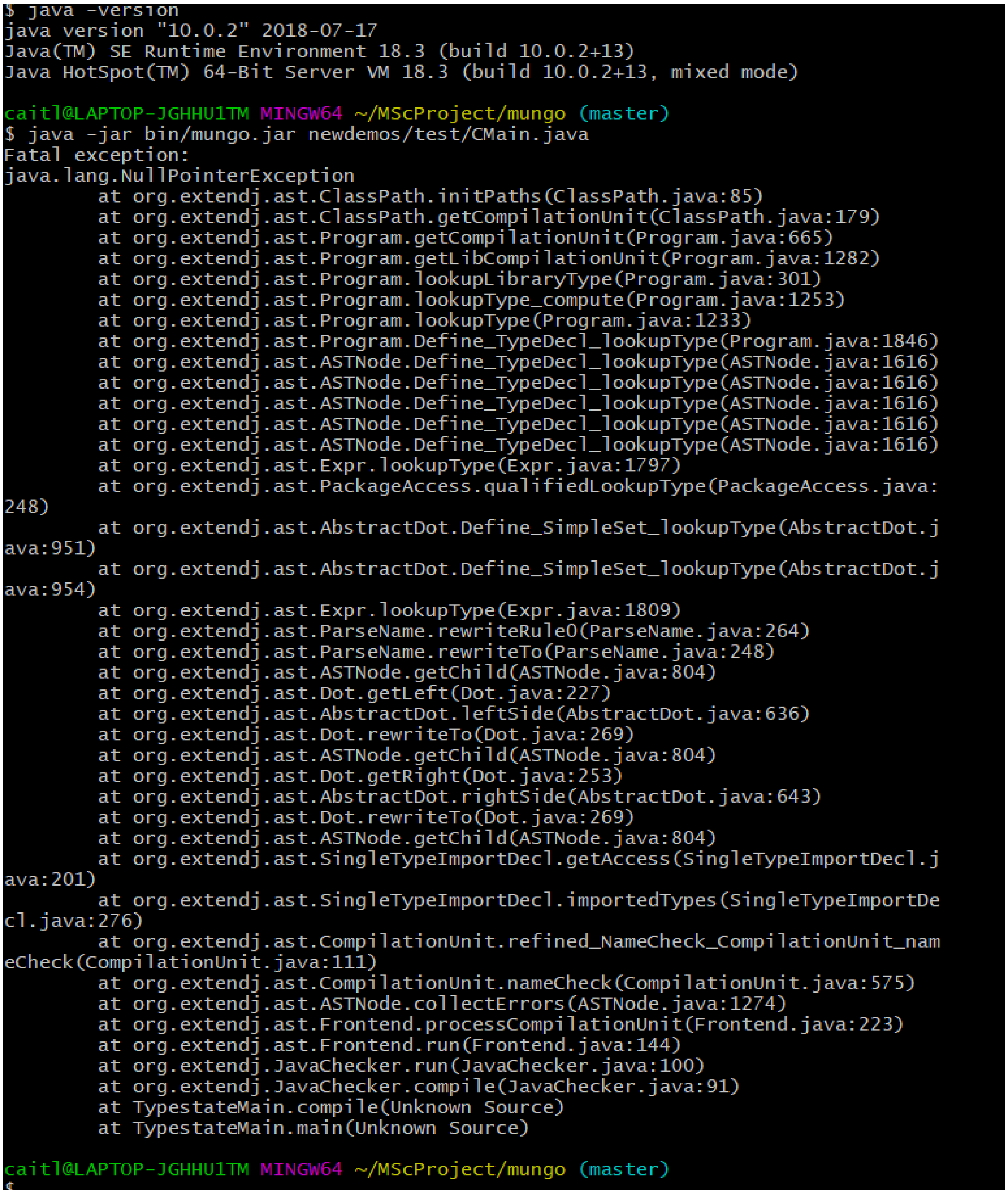
The original goal of the project was to use the Scribble-StMungo-Mungo toolchain to demonstrate how it can be used to typecheck a Domain Name System (DNS) client [27, 28, 29]. This was inspired by the Simple Mail Transfer Protocol (SMTP) example developed by Dr Ornela Dardha using the older version of the Mungo tool [4, 5] and demonstrated in a video on the Mungo website [7]. When I began work on the project, I was running Java version 10.0.2 and I had access to the older version of Mungo. I started out by trying to test-run it on the completed SMTP code, to gain a better understanding of the example in order to implement the DNS in a similar way. However, when I ran the old version of Mungo (which ran on .ses files) on the SMTP example, a number of errors were raised:



```
caitl@LAPTOP-JGHHU1TM /cygdrive/c/Users/caitl/MScProject/SMTP/SMTP with StMungo and Mungo/SMTP
$ java -jar Mungo.jar FinalSMTP/CMain.ses
Errors:
Fatal exception while processing FinalSMTP\CMain.ses:
java.lang.Error: Error loading java.nio.CharBuffer
        at Java14Frontend.AST.PathPart.getCompilationUnit(PathPart.java:121)
        at Java14Frontend.AST.Program.getCompilationUnit(Program.java:320)
        at Java14Frontend.AST.Program.getLibCompilationUnit_compute(Program.java:1557)
        at Java14Frontend.AST.Program.getLibCompilationUnit(Program.java:1539)
        at Java14Frontend.AST.Program.lookupLibType_compute(Program.java:1496)
        at Java14Frontend.AST.Program.lookupLibType(Program.java:1463)
        at Java14Frontend.AST.Program.lookupType_compute(Program.java:1434)
        at Java14Frontend.AST.Program.lookupType(Program.java:1414)
        at Java14Frontend.AST.Program.Define_TypeDecl_lookupType(Program.java:2000)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1508)
        at Java14Frontend.AST.Expr.lookupType(Expr.java:1511)
        at Java14Frontend.AST.TypeAccess.decls_compute(TypeAccess.java:384)
        at Java14Frontend.AST.TypeAccess.decls(TypeAccess.java:372)
        at Java14Frontend.AST.TypeAccess.decl_compute(TypeAccess.java:427)
        at Java14Frontend.AST.TypeAccess.decl(TypeAccess.java:418)
        at Java14Frontend.AST.TypeAccess.type_compute(TypeAccess.java:632)
        at Java14Frontend.AST.TypeAccess.type(TypeAccess.java:624)
        at Java14Frontend.AST.ParameterDeclaration.type_compute(ParameterDeclaration.java:546)
        at Java14Frontend.AST.ParameterDeclaration.type(ParameterDeclaration.java:538)
        at Java14Frontend.AST.MethodDecl.signature_compute(MethodDecl.java:1064)
        at Java14Frontend.AST.MethodDecl.signature(MethodDecl.java:1051)
```

Figure 6.1 – Errors produced by running old version of Mungo on SMTP with Java 10

I had already cloned the new Mungo (version 1.1) so I was advised to try out this version. I started out with a simple test example similar to the one described in Chapter 2. I was able to successfully run StMungo 1.1 on the local protocols using Java 10, but when I proceeded to typecheck the resulting Java files using the new Mungo, several different errors were raised:



```
$ java -version
java version "10.0.2" 2018-07-17
Java(TM) SE Runtime Environment 18.3 (build 10.0.2+13)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.2+13, mixed mode)

caitl@LAPTOP-JGHHU1TM MINGW64 ~/MScProject/mungo (master)
$ java -jar bin/mungo.jar newdemos/test/CMain.java
Fatal exception:
java.lang.NullPointerException
        at org.extendj.ast.ClassPath.initPaths(ClassPath.java:85)
        at org.extendj.ast.ClassPath.getCompilationUnit(ClassPath.java:179)
        at org.extendj.ast.Program.getCompilationUnit(Program.java:665)
        at org.extendj.ast.Program.getLibCompilationUnit(Program.java:1282)
        at org.extendj.ast.Program.lookupLibraryType(Program.java:301)
        at org.extendj.ast.Program.lookupType_compute(Program.java:1253)
        at org.extendj.ast.Program.lookupType(Program.java:1233)
        at org.extendj.ast.Program.Define_TypeDecl_lookupType(Program.java:1846)
        at org.extendj.ast.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1616)
        at org.extendj.ast.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1616)
        at org.extendj.ast.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1616)
        at org.extendj.ast.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1616)
        at org.extendj.ast.ASTNode.Define_TypeDecl_lookupType(ASTNode.java:1616)
        at org.extendj.ast.Expr.lookupType(Expr.java:1797)
        at org.extendj.ast.PackageAccess.qualifiedLookupType(PackageAccess.java:
248)
        at org.extendj.ast.AbstractDot.Define_SimpleSet_lookupType(AbstractDot.j
ava:951)
        at org.extendj.ast.AbstractDot.Define_SimpleSet_lookupType(AbstractDot.j
ava:954)
        at org.extendj.ast.Expr.lookupType(Expr.java:1809)
        at org.extendj.ast.ParseName.rewriteRule0(ParseName.java:264)
        at org.extendj.ast.ParseName.rewriteTo(ParseName.java:248)
        at org.extendj.ast.ASTNode.getChild(ASTNode.java:804)
        at org.extendj.ast.Dot.getLeft(Dot.java:227)
        at org.extendj.ast.AbstractDot.leftSide(AbstractDot.java:636)
        at org.extendj.ast.Dot.rewriteTo(Dot.java:269)
        at org.extendj.ast.ASTNode.getChild(ASTNode.java:804)
        at org.extendj.ast.Dot.getRight(Dot.java:253)
        at org.extendj.ast.AbstractDot.rightSide(AbstractDot.java:643)
        at org.extendj.ast.Dot.rewriteTo(Dot.java:269)
        at org.extendj.ast.ASTNode.getChild(ASTNode.java:804)
        at org.extendj.ast.SingleTypeImportDecl.getAccess(SingleTypeImportDecl.j
ava:201)
        at org.extendj.ast.SingleTypeImportDecl.importedTypes(SingleTypeImportDe
cl.java:276)
        at org.extendj.ast.CompilationUnit.refined_NameCheck_CompilationUnit_nam
eCheck(CompilationUnit.java:111)
        at org.extendj.ast.CompilationUnit.nameCheck(CompilationUnit.java:575)
        at org.extendj.ast.ASTNode.collectErrors(ASTNode.java:1274)
        at org.extendj.ast.Frontend.processCompilationUnit(Frontend.java:223)
        at org.extendj.ast.Frontend.run(Frontend.java:144)
        at org.extendj.JavaChecker.run(JavaChecker.java:100)
        at org.extendj.JavaChecker.compile(JavaChecker.java:91)
        at TypestateMain.compile(Unknown Source)
        at TypestateMain.main(Unknown Source)

caitl@LAPTOP-JGHHU1TM MINGW64 ~/MScProject/mungo (master)
$
```

Figure 6.2 – Errors produced by running new version of Mungo with Java 10

The errors shown in *Figures 6.1* and *6.2* were caused by the two different versions of the tools–touched on in Chapter 2–and their compatibility with Java versions. This caused confusion as I did not initially know the two versions ran on different versions of Java. I was advised to try out the old version using Java 1.4 and the new version using Java 1.8. I had to install both of these Java versions and research on how to change between them. While I was not able to run the old Mungo, I was eventually able to run the new Mungo (version 1.1) when I had installed and switched to Java version 1.8.0_211 by changing the Path environment variable to point to the JDK for Java 1.8. This proved difficult in itself as I did not have previous experience in doing this.

The DNS example originally intended for the project proved to be too complicated due to the errors produced and time taken trying to run the old Mungo version. In order to complete a useful, practical example within the given timeframe, the focus of the project shifted to using the new version of Mungo to implement an introduction to the toolchain with video demonstrations similar to the SMTP example at [7]. Once the new goal of the project was established, I began work on the portfolio examples. I encountered a few issues with the code produced by StMungo, based on the global and local protocols. For example, the travel-agent_Finance.scr file had the following lines:

```
approve(Code) to Researcher, Agent
invoice(Price, Code) from Agent
```

The `Code` and `Price` payload types represented `ints` in Java. Initially I had created separate classes for these, but for simplicity I changed the types to `int` in the global protocol and re-ran Scribble endpoint projection. The commas in each line caused StMungo to produce method names that included commas in the Java classes and typestate specifications. These caused syntax errors when compiling, and caused typestate violations if the commas were removed from the method names. I simply removed the commas and put each statement on a separate line in the global protocol, which resolved the issue.

There were several more challenges, both related to the tools and related to my relative inexperience with programming in general. I found that the Scribble shell script would not run on the Git Bash terminal that I originally used. I downloaded and switched to the Cygwin64 Terminal which allowed me to run the script. I was new to socket programming in Java when the project began, so it took effort to figure out how to configure the port numbers to achieve multiparty communication. I am also fairly new to command line use, especially adding classpaths when compiling and running scripts. Finally, video recording, editing and uploading to YouTube is a new – and enjoyable – challenge for me. This project presented a variety of steep learning curves, giving me the opportunity to study an exciting new concept in-depth and learn useful skills, related both to the field of typestate checking and to more general programming knowledge. I was very glad to work on these skills and gain some experience which I believe will be extremely valuable in my future career.

Chapter 4 Related and Future Work

The portfolio completed in this project provides simple introductory examples to the Scribble-StMungo-Mungo toolchain. In-depth examples modelling substantial real-world protocols like the Simple Mail Transfer Protocol (SMTP) by Kouzapas *et al* [4, 5] and Post Office Protocol v.3 (POP3) by Dardha *et al* [1] have already been implemented. The Mungo website [7] contains a video walkthrough of the SMTP example—recorded by Dr Ornela Dardha—which was the inspiration for the videos created for this project. The SMTP and POP3 examples both involve clients sending and receiving messages from email servers. They follow the general workflow as described here in Chapter 3, but they generally require extra layers of implementation to work with a real server. The POP3 example in [1] states that some servers do not precisely adhere to the RFC specification, which can require workarounds during implementation. There is scope here for incorporating the Scribble language as standard into RFCs, thus encouraging the use of the full typechecking toolchain when implementing protocols. The authors concur, however, that this is currently not likely [1].

Future work could add to this project by including existing internet protocols that would show how the toolchain can be used to help develop a robust communication system. The Domain Name System would be a useful protocol to implement using the toolchain; this project was originally planned around it. However, as discussed above, the decision was made early on to focus on smaller examples, based on the challenges faced and the complexity and scale of the DNS protocol. The DNS, first described in RFC 882 [27] and extended in RFC 1034 [28] and RFC 1035 [29], is the system used to convert human-readable web addresses into IP addresses; it is a vital component of the functioning of the Internet. In essence, it defines the hierarchical structuring of the domain name space, describing the structured labels that compose web addresses. The top of this hierarchical tree starts at the least-specific root name (e.g. the rightmost dot in `www.example.com.`) and works down to the most-specific (e.g. the host name `www`). This information may be stored across several name servers, each of which holds information about certain subsets of the domain name space [28]. When a webpage is accessed, a DNS query is sent by the client to resolve the domain's IP address from the appropriate name servers. Like the SMTP and POP3 examples described above, DNS queries therefore involve sequences of messages being sent between clients and servers [28], and so can be represented in the Scribble protocol language. Fowler [16] introduces a framework for monitoring multiparty session types in the actor-based language Erlang. The framework includes a Scribble global protocol specification of a DNS server, found at [30]. This could be used by Scribble to project to local endpoints, and kick-start the process described in section 3.1 to build a typechecked Java implementation of the DNS using StMungo and Mungo.

Chapter 5 Conclusions

This project presents a portfolio that adds to the Mungo website and the existing literature with an introductory teaching tool for the Scribble-StMungo-Mungo toolchain. The theory of session types is introduced in Chapter 2, describing its relevance as the formal foundation of the toolchain. The functioning of the tools is then described in detail: First, the Scribble language is used to represent structured communication protocols and project them to session channel endpoints. Then, StMungo takes these local endpoints and produces a Java-like typestate specification and a Java API for each one. Finally, after the code has been suitably altered to add business logic, the Mungo typechecking tool is run. Each example in the portfolio goes through this process, which has been fully documented in videos. The Travel Agent and Bookstore examples demonstrate multiparty communications, while the Adder example contributes by showing the use of recursion. As stated in the ABCD project's introduction [13]:

"*Without a way to routinely and reliably build concurrent and distributed systems, a half century of unprecedented technical progress will draw to a close*."

In the near future, this toolchain and similar session type tools will therefore almost certainly have to be incorporated as standard into a programmer's skillset. This portfolio, along with the videos and information provided in this dissertation, can serve as a useful tool for anyone wishing to get a head-start in the field.

Bibliography

[1] Ornela Dardha, Simon J. Gay, Dimitrios Kouzapas, Roly Perera, A. Laura Voinea and Florian Weber. Mungo and StMungo: Tools for Typechecking Protocols in Java. In *Behavioural Types: from Theory to Tools*, pages 309-328. River Publishers, 2017.

[2] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In *Trustworthy Global Computing '13*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013.

[3] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Information and Computation*, vol. 256, pages 253-286, 2017.

[4] Dimitrios Kouzapas, Ornela Dardha, Roly Perera and Simon J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming*, vol. 155, pages 52-75, 2018.

[5] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP),* pages 146–159. ACM, 2016.

[6] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, vol. 12, no.1, pages 157–171, 1986.

[7] Mungo homepage. University of Glasgow, 2019. `http://www.dcs.gla.ac.uk/research/mungo/index.html.` [Accessed 01/09/2019].

[8] Caitlin MacFadyen. Mungo Typechecking – Adder Example. YouTube, 2019. `https://www.youtube.com/watch?v=Vdt1lgMmmIY.` [Accessed 02/09/2019].

[9] Caitlin MacFadyen. Mungo Typechecking – Bookstore Example. YouTube, 2019. `https://www.youtube.com/watch?v=UvnsqT3w4Ck.` [Accessed 02/09/2019].

[10] Caitlin MacFadyen. Mungo Typechecking – Travel Agent Example. YouTube, 2019. `https://www.youtube.com/watch?v=4N0s2dVIDMk.` [Accessed 02/09/2019].

[11] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming (ESOP),* volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[12] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *Lecture Notes in Computer Science,* pages 398–413. Springer, 1994.

[13] ABCD Project homepage. 2019. `https://groups.inf.ed.ac.uk/abcd/` [Accessed 29/08/2019].

[14] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL),* pages 299–312. ACM, 2010.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL*), pages 273–284. ACM, 2008.

[16] Simon Fowler. An Erlang Implementation of Multiparty Session Actors. *Electronic Proceedings in Theoretical Computer Science*, no. 223, pages 36-50, 2016.

[17] Web Services Choreography Working Group homepage. W3C, 2003. `https://www.w3.org/2002/ws/chor/`. [Accessed 23/08/2019].

[18] Scribble Language homepage. 2019. `http://www.scribble.org/`. [Accessed 29/08/2019].

[19] Scribble-Java Repository. GitHub, 2019. `https://github.com/scribble/scribble-java`. [Accessed 15/08/2019].

[20] Scribble-Java Tutorial webpage. 2019. `http://www.scribble.org/docs/scribble-java.html#SCRIBSIG`. [Accessed 01/09/2019].

[21] StMungo Repository. Bitbucket, 2017. `https://bitbucket.org/abcd-glasgow/stmungo/src/master/examples/travel-agent/`. [Accessed 30/08/2019].

[22] Mungo Repository. Bitbucket, 2017. `https://bitbucket.org/abcd-glasgow/mungo/src/master/`. [Accessed 30/08/2019].

[23] Java SE Development Kit 8u221 Download webpage. 2019. `https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html?printOnly=1`. [Accessed 04/07/2019].

[24] Cygwin64 Terminal (for Windows 64) Installation webpage. 2019. `https://cygwin.com/install.html`. [Accessed 30/07/2019].

[25] Flashback Express Recorder homepage. 2019. `https://www.flashbackrecorder.com/express/`. [Accessed 06/08/2019].

[26] OpenShot Video Editor homepage. 2019. `https://www.openshot.org/`. [Accessed 06/08/2019].

[27] Paul Mockapetris. Domain Names – Concepts and Facilities, RFC 882. Internet Engineering Task Force, 1983.
`https://tools.ietf.org/html/rfc882`. [Accessed 28/08/2019].

[28] Paul Mockapetris. Domain Names – Concepts and Facilities, RFC 1034. Internet Engineering Task Force, 1987.
`https://www.ietf.org/rfc/rfc1034.txt` [Accessed 28/08/2019].

[29] Paul Mockapetris. Domain Names – Implementation and Specification, RFC 1035. Internet Engineering Task Force, 1987.
`https://www.ietf.org/rfc/rfc1035.txt` [Accessed 28/08/2019].

[30] Simon Fowler. Monitored Session Erlang. GitHub, 2015.
`https://github.com/SimonJF/monitored-session-erlang/blob/master/mockups/DNSServer.scr`. [Accessed 29/08/2019].

Appendix A – Portfolio Setup Instructions

The code for each of the portfolio examples has been provided in the submission. The examples can be run as follows:

1. First, download and install Scribble-Java from [19], StMungo from [21] and Mungo from [22], following their setup instructions.
2. Once the Scribble-Java project is set up, add the folder **scribble-portfolio** (containing Adder.scr, Bookstore.scr and BuyTicket.scr) into the same directory as the **scribblec.sh** file. A Cygwin terminal may need to be used on Windows to run the script.
3. Validation (a) and endpoint projection (b) can be carried out with the following commands, in this case on the Adder example:

    a. `$ ./scribblec.sh scribble-portfolio/Adder.scr`
    b. `$ ./scribblec.sh scribble-portfolio/Adder.scr -project Adder C`

4. Place the **portfolio-examples** folder in the StMungo main directory (where stmungo.jar should also be found after building the project) and run the following command, replacing `Adder_C.scr` with the appropriate local protocol:

    a. `$ java -jar stmungo.jar portfolio-examples/adder/Adder_C.scr`

5. The **portfolio** folder contains fully-implemented code that can be typechecked by Mungo. The code can be compiled as follows:

    a. `$ javac -cp bin/mungo.jar portfolio/Adder/*.java`

6. Place this folder in the main Mungo directory and run the following command for typechecking, replacing `portfolio/adder/CMain.java` with the filepath to any class that instantiates a role object:

    a. `$ java -jar bin/mungo.jar portfolio/Adder/CMain.java`

7. Finally, the programs can be compiled and run on separate terminals if the Mungo typechecking is error-free. In order to properly establish the connections, the programs must be run in the following order:
    a. **Travel Agent:** `RMain.java -> AMain.java -> FMain.java`
    b. **Bookstore:** `SellerMain.java -> Buyer2Main.java -> Buyer1Main.java`
    c. **Adder:** `CMain.java -> SMain.java`