Typechecking Java Protocols with [St]Mungo

A. Laura Voinea(✉), Ornela Dardha, and Simon J. Gay

School of Computing Science, University of Glasgow, Glasgow, UK
a.voinea.1@research.gla.ac.uk, {Ornela.Dardha,Simon.Gay}@glasgow.ac.uk

Abstract. This is a tutorial paper on [St]Mungo, a toolchain based on *multiparty session types* and their connection to *typestates* for safe distributed programming in Java language.

The StMungo ("Scribble-to-Mungo") tool is a bridge between multiparty session types and typestates. StMungo translates a *communication protocol*, namely a sequence of sends and receives of messages, given as a multiparty session type in the Scribble language, into a typestate specification and a Java API skeleton. The generated API skeleton is then further extended with the necessary logic, and finally typechecked by Mungo. The Mungo tool extends Java with (optional) typestate specifications. A typestate is a state machine specifying a *Java object protocol*, namely the permitted sequence of method calls of that object. Mungo statically typechecks that method calls follow the object's protocol, as defined by its typestate specification. Finally, if no errors are reported, the code is compiled with `javac` and run as standard Java code.

In this tutorial paper we give an overview of the stages of the [St]Mungo toolchain, starting from Scribble communication protocols, translating to Java classes with typestates, and finally to typechecking method calls with Mungo. We illustrate the [St]Mungo toolchain via a real-world case study, the HTTP client-server request-response protocol over TCP. During the tutorial session, we will apply [St]Mungo to a range of examples having increasing complexity, with HTTP being one of them.

Keywords: Multiparty session types · Typestate · Mungo · StMungo · HTTP protocol

1 Introduction

The concept of an *application programming interface* (API) is central to software architecture and implementation. An API is a specification of a collection

Supported by the UK EPSRC grant EP/K034413/1, "From Data Types to Session Types: A Basis for Concurrency and Distribution (ABCD)", by the EU HORIZON 2020 MSCA RISE project 778233 "BehAPI: Behavioural Application Program Interfaces", and by an EPSRC PhD studentship.

Published by Springer Nature Switzerland AG 2020
A. Gotsman and A. Sokolova (Eds.): FORTE 2020, LNCS 12136, pp. 208–224, 2020.
https://doi.org/10.1007/978-3-030-50086-3_12

of related programming language operations that enable the use of a particular kind of functionality. For example, in a typical programming language, the functionality for implementing graphical user interfaces is organised and described as an API. In an object-oriented language, an API is presented as a collection of classes, each with methods for a range of related operations. A specific example is the JavaFX API, which provides graphics and media functionality—in fact JavaFX is so large that it is better described as a collection of APIs for more specific purposes, such as media streaming and web rendering.

Nowadays, APIs are not only used to present the library functions of programming language implementations. They can also package up the functionality of distributed services, and be called remotely in networked applications. A significant trend is the development and publication of APIs to allow access to functions that were previously internal to a software application. For example, the developer of a student records database might publish an API to allow programmatic access to the data, and this could be used by third-party developers to produce applications that make use of student records to provide additional services. This evolution of the API concept has become a key aspect of open software development and service-oriented system architectures. In a commercial setting it has enabled the birth of an API economy in which the provision of APIs can be monetised. APIs have thus become a key focus of the software industry.

Typical methods in an API require parameters, and these can be specified using standard type-theoretic techniques. In a statically typed language, each method in an API has its type signature, specifying the types of its parameters and the type of any result that it returns. The standard techniques of typechecking, especially when implemented in an integrated development environment (IDE), are effective in supporting programmers to use APIs correctly, identifying errors during development rather than waiting until the testing phase when they are much more expensive to correct.

The description of an API as a collection of typed method signatures however, does not capture any constraints on the sequence in which methods can be called. For example, an API for working with files requires that a file must be successfully opened before it can be read or written. After the file has been closed, it cannot be read or written any more, and the only available method is open. Another standard example is the `Iterator` class in Java, in which the `hasNext` method must be called (and return `true`) before the `next` method can be called.

Another category of examples arises in APIs for communication, in concurrent or distributed systems. Typically the communication within a system is structured around various *communication protocols*, each of which specifies a permitted sequence of messages and the format (type) of each message. An API whose operations allow sending and receiving of messages in a given protocol has constraints so that the operation calls follow the protocol specification. These constraints cannot be expressed purely within the framework of typed method signatures, more expressive types and type systems are needed. In general, we

can speak of *behavioural APIs*, a term based on the term *behavioural types* for type systems that specify sequence-related properties involving multiple method calls.

Two established lines of research are relevant in this context. One is *typestates* [42], which is the idea of using static type systems to specify permitted sequences of method calls. The other is *session types* [26,28,44], which are type-theoretic descriptions of communication protocols. The StMungo and Mungo tools are the result of convergence between these two lines of research [2,17,34,35]. On the one hand, APIs for communication protocols are clearly a special case of behavioural APIs in general. On the other hand, transferring the concepts of session types from process calculi or functional languages to object-oriented languages requires embedding them in a more general setting that supports typestates. This is because it is natural to define methods that each perform several communication steps, and then the original communication protocol (session type) gives rise to different, although related, sequencing constraints on the methods.

StMungo is a bridge between session types and typestates, by translating multiparty session types (MPST) [29] written in the Scribble language [41] into typestate specifications for Java classes. The key steps are given in the following:

- Scribble is used as a specification language for *global protocols* (or global types) describing communication among all involved participants in a communication protocol in a distributed system.
- The Scribble tools are used to *validate* and *project* a global type into *local protocols* (or local types) for each participant involved.
- StMungo translates Scribble local types into typestate specifications for Java classes, describing the Java object protocols, namely the permitted sequences of method calls of an object.
- StMungo also generates an API implementation for each participant, which follows its typestate specification, described in the previous step.

At this stage we can run the Mungo tool. The key ideas and steps behind Mungo are given in the following:

- Typestate specifications are expressed as annotations of Java classes, so there is no change to the language itself.
- Linear typing is used to control aliasing, so that there is no possibility of inconsistent views of an object's state.
- The Mungo typechecker checks that method calls are performed following the object's protocol, as specified by its associated typestate.
- If Mungo typechecking is successful and no errors are reported, then the code is compiled with `javac` and run as standard Java code.
- The Mungo typechecker is formalised inspired by session types theory and the resulting type system is proved correct via the standard theorems of progress and subject reduction [34,35].

In the remainder of this tutorial paper we will describe the StMungo Sect. 2 and Mungo Sect. 3 tools via a real-world case study, the HTTP protocol.

In Sect. 4 we give step-by-step instructions on how to run the tools. In Sect. 5 we discuss related work and in Sect. 6 we conclude the paper and discuss future work.

2 StMungo

The StMungo tool is a Java-based transpiler implemented using the ANTLR v4.5 framework [6]. StMungo acts as a bridge between multiparty session types and typestate specifications. In particular it is the link between the Scribble specification language [27,41] and the Mungo tool Sect. 3. StMungo is the *first* tool to provide a practical embedding of Scribble multiparty session types into an object-oriented language with typestates.

In order to better understand the StMungo tool, we need to describe both the Scribble language and the typestate specifications. Let's start with Scribble.

The Scribble specification language is an implementation of multiparty session types (MPST) [29,41]. Participants in a distributed system communicate among each other by sending and receiving messages and following a predefined communication protocol. Such protocol is given as a *global protocol* (or global type) in Scribble. The Scribble tools can perform *validation* and *projection* of a global protocol. First, we must check if the specified global protocol is valid, meaning if it is correct with respect to transmitted data; there are no deadlocks within the global protocol; there are no un-notified participants for example, regarding session termination, and so on. These checks follow the MPST theory [29]. Once a global protocol is validated, with Scribble tools we can project it into *local protocols* (or local types) for each participant in the system.

The HTTP Protocol Case Study. Let us illustrate the notions of global and local protocols using our HTTP case study. HTTP (HyperText Transfer Protocol) [22] is the underlying data protocol used by the World Wide Web defining how messages are formatted and transmitted, and what actions servers and clients may take in response to various methods, such as `GET`, `PUT` or `POST`. An HTTP session is a sequence of network request-response transactions, initiated by the client sending a request over a TCP connection to a particular port of a server. Upon receiving the request, the server listening on that port sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own. The structure of the request and response messages exchanged is rich and complex, lending itself to be further specified through session types. Hence, we represent the HTTP global protocol in the style of Hu [31] where an HTTP request and response are broken down respectively into sending and receiving a request line – `request`, followed by zero or more header-fields – `host` or `usera` terminated by a new-line. This fine grained representation of the protocol is made possible by the message being broken down via TCP bit streams, in a manner that is transparent to the parties involved.

The global protocol for HTTP specified in Scribble is given in Listing 1.1. Line 1 contains the `module` declaration, made up of an optional package prefix

i.e., `http`, and the name of the file containing the module, `Http`. Line 2 contains a payload type declaration `type <java>...`, which gives an alias (`str`) to a data type (`String`) from an external language `java` which can be used in the payload of a message signature. A module can contain zero or more *global protocol declarations*, consisting of a protocol signature (line 4), choices (lines 5 and 27), message passing (line 6), and recursion (line 7). Lines 11–46 model a correctly formatted client request and lines 49–91 a server response.

```
1  module http.Http;
2  type <java> "java.lang.String" from "rt.jar" as str;
3
4  global protocol Http(role C, role S){
5    choice at C{ // Request
6      request(str) from C to S;  //GET / HTTP/1.1
7      rec X{ choice at C{
8                host(str) from C to S;//Host: www.google.co.uk
9                continue X;
10             }or{
11               userA(str) from C to S;//User-Agent:...
12               continue X;
13             }or{
14               acceptT(str) from C to S;//Accept: text/html...
15               continue X;
16             }or{
17               ... //other header fields
18               body(str) from C to S;
19             }}}
20   //Response
21   httpv(str) from S to C;//HTTP/1.1
22   choice at S{
23     200(str) from S to C;//200 OK
24   }or{
25     404(str) from S to C;//404 Bad Request
26   }
27   rec Y{
28     choice at S{
29       date(str) from S to C;//Date: ...
30       continue Y;
31     }or{
32       server(str) from S to C;//Server:...
33       continue Y;
34     }or{
35       strictTS(str) from S to C;//Strict-Transport-Security
36       continue Y;
37     }or{
38       ...//other header fields
39       body(str) from S to C;
40     }}}
```

Listing 1.1. HTTP Global Protocol

Using the Scribble tools, we can project the HTTP global protocol onto local protocols for the server `S` and the client `C`. In this tutorial we will focus only on the client side as we will interact with real-world HTTP servers. The local protocol for the HTTP client `C`, given in Listing 1.2, describes the behaviour of this role. The `_C` in the protocol name indicates that `C` is the local endpoint. For simplicity, we limit this protocol to the `GET` command only, with the rest being represented in a similar manner.

```
1  ...
2  local protocol Http_C(role C, role S) {
3    choice at C {
4      request(str) to S;
5      rec X { choice at C {
6                  host(str) to S;
7                  continue X;
8              } or {
9                  userA(str) to S;
10                 continue X;
11             } or {
12                 acceptT(str) to S;
13                 continue X;
14             } or {
15                 ...//other header fields
16                 body(str) to S;
17             }}}
18   httpv(str) from S;
19   choice at S {
20      200(str) from S;
21    } or {
22     404(str) from S;
23    }
24   rec Y { choice at S {
25             date(str) from S;
26             continue Y;
27           } or {
28             server(str) from S;
29             continue Y;
30           } or {
31             strictTS(str) from S;
32             continue Y;
33           } or {
34             ...//other header fields
35             body(str) from S;
36           }}}
```

Listing 1.2. HTTP Client Protocol

The client can send a request line `request` (line 4), followed by zero or more header-fields—`host`, or `userA` and so on. The server responds with a line containing the HTTP version—`httpv` (line 18) followed by the status of the request, either—`200` for a found resource, or—`404` for a bad request. The server can choose zero or more header-fields to follow this message with. The StMungo tool takes in input a Scribble local protocol for a `role` and translates it into a typestate specification for a Java API skeleton. This translation is based on the principle that each `role` in the multiparty session communication following its local protocol, can be abstracted as a Java class following its typestate specifica-

tion. A typestate is a state machine defining the permitted sequence of method calls of a Java object, thus defining the object's protocol.

The HTTP Protocol Case Study (Continued). Running StMungo on the HTTP client protocol Listing 1.2 produces the following files, where `C` at the beginning of each file name stands for client.

1. `CProtocol.protocol`: the typestate specification representing the HTTP client's local protocol. The send and receive operations are translated as Java methods (Listing 1.3 below in this section).
2. `CRole.java`: the Java API implementing the HTTP client. This class implements the typestate `CProtocol` over Java sockets (Listing 1.4, Sect. 3).
3. `CMain.java`: this can be an optional file. It gives a minimum logic of the client `CRole` and provides a `main()` method (Listing 1.5, Sect. 3).

The typestate specification `CProtocol.protocol` for the HTTP client is given in Listing 1.3.

```
typestate CProtocol {
  State0 = {void send_REQUESTToS(): State1}
  State1 = {void send_requestStrToS(String): State2}
  State2 = {void send_HOSTToS(): State3,
            void send_USERAToS(): State4,
            void send_ACCEPTTToS(): State5,
            ... //send other labels
            void send_BODYToS(): State12}
  State3 = {void send_hostStrToS(String): State2}
  State4 = {void send_userAStrToS(String): State2}
  ... //send other main messages
  State12 = {void send_bodyStrToS(String): State13}
  State13 = {String receive_httpvStrFromS(): State14}
  State14 = {Choice1 receive_Choice1LabelFromS():
            <_200: State15, _404: State16>}
  State15 = {String receive_200StrFromS(): State17}
  State16 = {String receive_404StrFromS(): State17}
  State17 = {Choice2 receive_Choice2LabelFromS():
            <DATE: State18, SERVER: State19,
            STRICTTS: State20, ..., BODY: State28>}
  State18 = {String receive_dateStrFromS(): State17}
  State19 = {String receive_serverStrFromS(): State17}
  State20 = {String receive_strictTSStrFromS(): State17}
  ...
  State28 = {String receive_BODYStrFromS(): end}}
```

Listing 1.3. Typestate Specification

A typestate is a state machine (Fig. 1) with states labelled `State0` (initial state), `State1`, `State2` ... Each state offers a set of methods that must be a subset of the methods defined by the class; each method specifies a transition to a successor state, such that when called at runtime allows the object to change state as specified by its typestate.

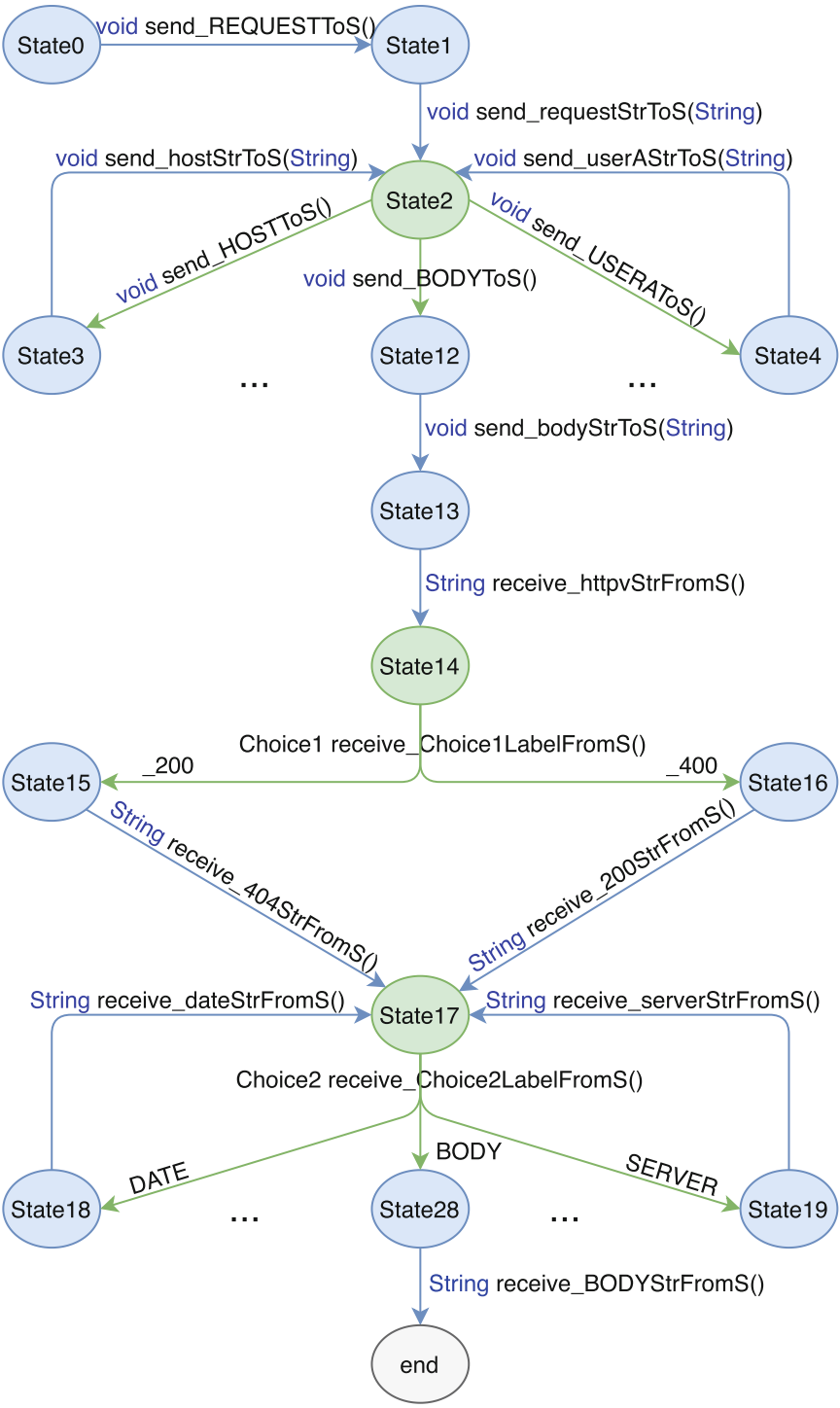


Fig. 1. State machine for CProtocol

The send and receive operations given in the client's local protocol are translated as typestate methods in `CProtocol.protocol`. For example, the message `request(str) to S` (line 4, Listing 1.2) where the client sends a `request` message of type `str` to the server, is translated as two method calls due to formatting and parsing (lines 2–3 in Listing 1.3). Calling the first method `void send_REQUESTToS()` specifying the method and calling the second method `void send_requestStrToS(String)` requests the rest of the message of type `String` (further details in Sect. 3).

We will comment on the other two files `CRole.java` and `CMain.java` in Sect. 3.

3 Mungo

The Mungo tool is a Java front-end tool used to statically typecheck typestate specifications for Java classes. The tool is implemented in Java using the ExtendJ framework [25,38], a meta-compiler based on reference attribute grammars.

Mungo extends a Java class with a typestate specification, which is saved in a separate file (such as `CProtocol.protocol` in Sect. 2) and is attached to a Java class using the annotation `@Typestate("ProtocolName")`, where `"ProtocolName"` names the file where the typestate is defined. The typestate inference algorithm given by the formalisation of the Mungo tool in [34,35] constructs the sequences of methods called on all objects associated with a typestate, and then checks if the inferred typestate is a subtype of the object's declared typestate. The formalisation of the typestate inference system and its soundness properties are beyond the scope of this paper and the reader is referred to [34,35].

Source files are typechecked in two phases: first, according to the standard Java type system, and then to the typestate type system via Mungo. The source files can then be compiled using standard `javac` and executed in the standard Java runtime environment.

The typestate specification generated from StMungo together with the Mungo typechecker can guide the user in the design and development of distributed multiparty communication-based systems with guarantees of communication safety and soundness.

We will now describe the use of Mungo via our running example, the HTTP protocol, and in particular we will do so by commenting on the last two files `CRole.java` and `CMain.java` generated by StMungo for the HTTP client `C`.

The HTTP Protocol Case Study (Continued). The HTTP client API is given by Listing 1.4 annotated by the typestate `CProtocol`, defined in Listing 1.3.

Lines 3–9 define the client's constructor where the connection phase over Java sockets takes place. The rest of `CRole` contains a minimal implementation of the methods specified in the typestate `CProtocol`. The methods for sending and receiving messages contain basic formatting and parsing, which can be further improved by the programmer.

```
1  @Typestate("CProtocol")
2  public class CRole {
3    public CRole() { ...//Bind the sockets and accept a client
4      connection
5      try { // Create the read and write streams
6          socketSIn = new BufferedReader(...);
7          socketSOut = new PrintWriter(...);}
8      catch (IOException e) {...}}
9    public void send_REQUESTToS(){this.socketSOut.print("GET")
       ;}
10   public void send_requestStrToS(String payload){this.
       socketSOut.println(payload);}
11   ... // Define all other send methods in CProtocol
12   public String receive_httpvStrFromS()() {
13     String line = "";
14     try {line = this.socketSIn.readLine();}
15     catch (IOException e) {...}
16     return line;}
17   public Choice1 receive_Choice1LabelFromS() {
18     try {stringLabelChoice1 = this.socketSIn.readLine();}
19     catch (IOException e) {...}
20     switch (stringLabelChoice1) {
21       case "200":
22         return Choice1._200;
23       case "404":
24       default:
25         return Choice1._404;}}
26   public String receive_200StrFromS() {
27     String line = "";
28     try {line = this.socketSIn.readLine();}
29     catch (IOException e) {...}
30     return line;}
31   public String receive_404StrFromS() {
32     String line = "";
33     try {line = this.socketSIn.readLine();}
34     catch (IOException e) {...}
35     return line;}
36      .../*Define all other receive methods in CProtocol*/}}}
```

Listing 1.4. Client API

Lines 8–9 define the two methods for sending the initial, mandatory, request line—`send_REQUESTToS` (for the method, i.e."`GET`") and `send_requestStrToS` (for the rest of the message). Lines 11–34 define methods for receiving the first line in a response, composed of the HTTP version—`receive_httpvStrFrom` and the status. The method in line 16 `Choice1 receive_Choice1LabelFromS` captures the status. This method returns a `Choice1` type, which is an enumerated type defined as:

```
1  enum Choice1 {_200, _404;}
```

For each choice there is an enumerated type, named by StMungo according to the position of the choice in the sequence of choices within the local protocol.

The values of the enumerated type are the names of the first message in each branch of the choice, for example for `Choice1` they are `_200` or `_400`. Thus, the method `receive_Choice1LabelFromS` receives a message which represents one of the two status codes, and it returns the corresponding `enum` value.

Let's move now onto the `CMain.java` given in Listing 1.5. `CMain.java` contains a minimal implementation of the client endpoint using the CRole class to communicate with the server endpoint. Below we give the main method, omitting any auxiliary methods generated by StMungo. The code is modified from the generated version by adding the request and host messages needed to request the home page from `www.google.co.uk`.

```
1   public static void main(String[] args) {
2     CRole currentC = new CRole();
3     String sread = //input REQUEST
4     if ("REQUEST".equals(sread)) {
5       currentC.send_REQUESTToS();
6       currentC.send_requestStrToS("/ HTTP/1.1");
7       _X: do { sread = //input header choice
8         switch (sread) {
9           case ("HOST"):
10            currentC.send_HOSTToS();
11            currentC.send_hostStrToS("www.google.co.uk");
12            continue _X;
13           ... //other cases corresponding to header fields
14          case ("BODY"):
15            currentC.send_BODYToS();
16            currentC.send_bodyStrToS("/r/n");
17            break _X;
18        }} while (true);}
19    currentC.receive_httpvStrFromS();
20    switch (currentC.receive_Choice1LabelFromS()) {
21      case _200:
22        currentC.receive_200StrFromS();
23        break;
24      case _404:
25        currentC.receive_404StrFromS();
26        break;}
27  _Y:do {
28    switch (currentC.receive_Choice2LabelFromS()) {
29      case DATE:
30        currentC.receive_dateStrFromS();
31        continue _Y;
32      ... //other cases corresponding to the header fields
33      case BODY:
34        currentC.receive_bodyStrFromS();
35        break _Y;}
36    } while (true);}
```

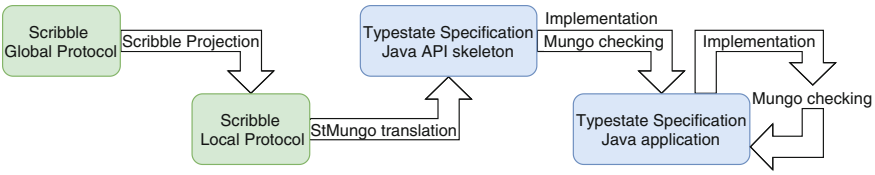
Listing 1.5. Client Implementation

In line 2 we create a new HTTP client, `currentC`, and proceed by showing the code for a small correctly formatted request, with the initial, mandatory request line messages being sent first (lines 5–6); then among the recursive choice cases

we show the code for sending the the host field (lines 10–11), before concluding the request by an empty body (lines 15–16). Then `currentC` will receive the response status line (lines 19–26) followed by recursive choice cases for the fields to be received from the server (lines 27–36).

To ensure that methods of the protocol are called in a valid sequence and that all possible responses are handled, the `CMain` implementation is checked by computing the sequences of method calls that are made on the `currentC` object, inferring the minimal typestate specification that allows them, and then comparing it with the specification declared in `CProtocol`.

4 How to Run [St]Mungo: A Step-by-Step Tutorial



The tools together with the HTTP example and further examples can be obtained from the [St]Mungo repository [1].

The tools come prebuilt and ready to use as runnable jar files: `stmungo.jar` and `mungo.jar`. In the same repository we also provide the latest release—0.4.3, of the command line tool for Scribble.

We show how to use these tools via the HTTP example, assuming the root folder of the repository linked above.

To run the Scribble tool on the global protocol for validation only: `./scribble-0.4.3/scribblec.sh demos/http/Http.scr`

To run the Scribble tool on the global protocol and project the client role: `./scribble-0.4.3/scribblec.sh demos/http/Http.scr -project http C`

To run StMungo and obtain the Java prototype implementation: `java -jar stmungo.jar demos/http/Http_C.scr`

To run Mungo: `java -jar mungo.jar demos/http/CMain.java`

Finally, if no errors are reported, the code can be compiled with `javac` and run as standard Java code.

5 Related Work

There is a huge and growing literature on session types and other forms of behavioural types, going back to the original papers on binary session types [26,28,44] and multiparty session types [29,30]. The BETTY project[1] produced three survey articles: one on foundations of behavioural types [33], one on behavioural types and security [13] and one on behavioural types in programming

[1] COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY), `www.behavioural-types.eu`.

languages [5]. The project also produced a book [24] describing implementations of programming languages and tools based on behavioural types. The ABCD project[2] has produced a list of implementations of session types in programming languages.

Since the introduction of typestate [42], there have been several projects to add the concept to practical programming languages. Vault [18,21] is an extension of C, and Fugue [19] applies similar ideas to C#. Plural [10] is based on Java and has been used to study access control systems [9] and transactional memory [8], and to evaluate the effectiveness of typestate in Java APIs [10]. Sing# [20] is an extension of C# which was used to implement Singularity, an operating system based on message-passing. It incorporates typestate-like contracts, which are a form of session type, to specify protocols. Bono et al. [12] have formalised a core calculus based on Sing# and proved type safety.

The Plaid programming language [3,43] proposes a new paradigm of typestate-oriented programming. Instead of class definitions, a program consists of state definitions containing methods that cause transitions to other states. Transitions are specified in a similar way to Plural's pre- and post-conditions. Like classes, states are organised into an inheritance hierarchy. Recent work [23,45] uses gradual typing to integrate static and dynamic typestate checking.

Bodden and Hendren [11] developed the Clara framework, which combines static typestate analysis with runtime monitoring. The monitoring is based on the *trace matches* approach [4], using regular expressions to define allowed sequences of method calls. The static analysis attempts to remove the need for runtime monitoring, but if this is not possible, the runtime monitor is optimised.

A challenge in typestate systems is aliasing. State changes to a given object must be reflected in all references that point to that object, otherwise inconsistency can result in violations of type safety. The literature includes several approaches to alias control. Some work, including ours, uses linear typing to forbid aliasing completely. The *adoption and focus* approach of Vault and Fugue, and the permission-based approaches of Plural and Plaid, are more flexible. Militão et al. [36] present an expressive fine-grained system. Crafa and Padovani [16,40] present an approach to concurrent typestate-oriented programming, allowing objects to be accessed and modified concurrently by several processes, each potentially changing only part of their state. Some work [32,39] combines static checking of typestate (or session type) properties with dynamic monitoring of (non-)aliasing properties. Balzer et al. [7] augment session types with points at which locks need to be acquired in order to perform state-changing operations; this approach has not yet been applied to a typestate system.

There is relatively little work combining behavioural types and typestate in the way that Mungo and StMungo do. The only other research we are aware of is the *API generation* approach of Hu [31]. The idea is to translate a Scribble protocol into a collection of classes for a standard language such as Java [32],

[2] EPSRC EP/K034413/1 From Data Types to Session Types: A Basis for Concurrency and Distribution (ABCD), `groups.inf.ed.ac.uk/abcd/`.

F# [37] or Go [15]. Each class represents a particular state in a protocol, with the methods available in that state. Each method returns the object on which it was called, but with a different class corresponding to the new state of the object. Because each state has its own class, standard IDEs can show the programmer which methods are available; however, for a complex protocol there can be a large number of classes. Runtime monitoring is used to check absence of aliasing.

For this tutorial we have used an example based on a standard internet protocol, HTTP. In previous work with Mungo and StMungo we have analysed SMTP [34,35] and POP3 [17]. Hu et al. also use SMTP [32,37] and HTTP [31] as case studies.

6 Conclusion and Future Work

We have presented a tutorial on using the [St]Mungo toolchain for static typechecking of a communication protocols. StMungo connects the Scribble specification language, used to define communication protocols, to Mungo by translating multiparty session types into typestate specifications. Mungo extends Java with typestate specifications, which annotate classes and define the permitted sequence of method calls of Java objects. We illustrate the workflow of both tools through implementing a substantial case study, an HTTP client. We use this client to communicate with a real-world server, the `www.google.co.uk` server.

While the toolchain is effective for statically typechecking the correct implementation of communication protocols, we intend to further improve its features for distributed programming in Java. On the StMungo side, we will keep it up to date with any changes in the Scribble specification language. On the Mungo side, we aim to offer static typechecking of generics and exceptions. To support generics, method calls on an object whose type is a generic parameter must be typechecked against the typestate specification of the parameter's upper bound. To support typechecking of exception handlers, typestate specifications must define the state transitions corresponding to exceptions, and check the transitions are consistent with the states of fields at the point where an exception is thrown. While existing work on exceptions in session types [14] provides inspiration, the complexities of Java's exception mechanism need to be accounted for as well. Another aim is to improve Mungo's error messages to better allow debugging.

References

1. Mungo Repository. https://bitbucket.org/abcd-glasgow/mungo-tools/src/master/
2. Mungo Webpage. http://www.dcs.gla.ac.uk/research/mungo/
3. Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: OOPSLA Companion, pp. 1015–1022. ACM (2009). https://doi.org/10.1145/1639950.1640073
4. Allan, C., et al.: Adding trace matching with free variables to AspectJ. In: OOPSLA, pp. 345–364. ACM (2005). https://doi.org/10.1145/1094811.1094839
5. Ancona, D., et al.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016). https://doi.org/10.1561/2500000031

6. ANTLR Project Homepage. www.antlr.org
7. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Caires, L. (ed.) ESOP 2019. LNCS, vol. 11423, pp. 611–639. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17184-1_22
8. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. In: OOPSLA, pp. 227–244. ACM (2008). https://doi.org/10.1145/1449764.1449783
9. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA, pp. 301–320. ACM (2007). https://doi.org/10.1145/1297027.1297050
10. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API protocol checking with access permissions. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 195–219. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03013-0_10
11. Bodden, E., Hendren, L.J.: The clara framework for hybrid typestate analysis. STTT **14**(3), 307–326 (2012). https://doi.org/10.1007/s10009-010-0183-5
12. Bono, V., Messa, C., Padovani, L.: Typing copyless message passing. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 57–76. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_4
13. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Information flow safety in multiparty sessions. Math. Struct. Comput. Sci. **26**(8), 1352–1394 (2016). https://doi.org/10.1017/S0960129514000619
14. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions in session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_32
15. Castro-Perez, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. PACMPL **3**(POPL), 29:1–29:30 (2019). https://doi.org/10.1145/3290342
16. Crafa, S., Padovani, L.: The chemical approach to typestate-oriented programming. In: OOPSLA, pp. 917–934. ACM (2015). https://doi.org/10.1145/2814270.2814287
17. Dardha, O., Gay, S.J., Kouzapas, D., Perera, R., Voinea, A.L., Weber, F.: Mungo and StMungo: tools for typechecking protocols in Java. In: Behavioural Types: From Theory to Tools. River Publishers (2017)
18. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 59–69. ACM (2001). https://doi.org/10.1145/378795.378811
19. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24851-4_21
20. Fähndrich, M., et al.: Language support for fast and reliable message-based communication in singularity OS. In: EuroSys, pp. 177–190. ACM (2006). https://doi.org/10.1145/1217935.1217953
21. Fähndrich, M., DeLine, R.: Adoption and focus: practical linear types for imperative programming. In: PLDI, pp. 13–24. ACM (2002). https://doi.org/10.1145/512529.512532
22. Fielding, R.T., Reschke, J.F.: Hypertext transfer protocol (HTTP/1.1): message syntax and routing. RFC 7230, pp. 1–89 (2014)

23. Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. ACM Trans. Program. Lang. Syst. **36**(4), 12:1–12:44 (2014). https://doi.org/10.1145/2629609
24. Gay, S.J., Ravara, A. (eds.): Behavioural Types: From Theory to Tools. River Publishers, Denmark (2017)
25. Hedin, G.: An introductory tutorial on JastAdd attribute grammars. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 166–200. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18023-1_4
26. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
27. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19056-8_4
28. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
29. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
30. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
31. Hu, R.: Distributed programming using Java APIs generated from session types. Behavioural Types: from Theory to Tools, pp. 287–308 (2017)
32. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
33. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016). https://doi.org/10.1145/2873052
34. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with mungo and StMungo. In: PPDP, pp. 146–159. ACM (2016). https://doi.org/10.1145/2967973.2968595
35. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo: a session type toolchain for java. Sci. Comput. Program. **155**, 52–75 (2018). https://doi.org/10.1016/j.scico.2017.10.006
36. Militão, F., Aldrich, J., Caires, L.: Aliasing control with view-based typestate. In: FTfJP@ECOOP, pp. 7:1–7:7. ACM (2010). https://doi.org/10.1145/1924520.1924527
37. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in F#. In: CC, pp. 128–138. ACM (2018). https://doi.org/10.1145/3178372.3179495
38. Öqvist, J.: ExtendJ: extensible java compiler. In: Programming, pp. 234–235. ACM (2018). https://doi.org/10.1145/3191697.3213798
39. Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, e4 (2017). https://doi.org/10.1017/S0956796816000289
40. Padovani, L.: Deadlock-free typestate-oriented programming. Program. J. **2**(3), 15 (2018). https://doi.org/10.22152/programming-journal.org/2018/2/15
41. Scribble Project Homepage. www.scribble.org

42. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
43. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, É.: First-class state change in Plaid. In: OOPSLA, pp. 713–732 (2011). https://doi.org/10.1145/2048066.2048122
44. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D., Philokyprou, G., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58184-7_118
45. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual typestate. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 459–483. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22655-7_22