14

Mungo and StMungo: Tools for Typechecking Protocols in Java

Ornela Dardha[1], Simon J. Gay[1], Dimitrios Kouzapas[1], Roly Perera[1,2], A. Laura Voinea[1] and Florian Weber[1]

[1]School of Computing Science, University of Glasgow, UK
[2]School of Informatics, University of Edinburgh, UK

Abstract

We present two tools that support static typechecking of communication protocols in Java. Mungo associates Java classes with typestate specifications, which are state machines defining permitted sequences of method calls. StMungo translates a communication protocol specified in the Scribble protocol description language into a typestate specification for each role in the protocol by following the message sequence. Role implementations can be typechecked by Mungo to ensure that they satisfy their protocols, and then compiled as usual with `javac`. We demonstrate the Scribble, StMungo and Mungo toolchain via a typechecked POP3 client that can communicate with a real-world POP3 server.

14.1 Introduction

Modern computing is dominated by communication, at every level from manycore architectures through multithreaded programs to large-scale distributed systems; this contrasts with the original emphasis on data processing. Early recognition of the importance of structured data meant that high-level programming languages have always incorporated data types and supported programmers through the techniques of static and dynamic typechecking. The foundational status of structured data was explicitly recognised in the title of Wirth's classic 1976 text *Algorithms + Data Structures = Programs*, but a more appropriate modern slogan would be *Programs + Communication*

Structures = Systems. The new reality of communication-based software development needs to be supported by programming tools based on structuring principles and high-level abstractions. Given the success of data types, it is natural to apply type-theoretic techniques to the specification and verification of communication-based code. During the last twenty years, this goal has been pursued by the expanding and increasingly active research community on session types [12, 13, 24]. A session type is a formal structured description of a communication protocol, specifying the type, sequence and direction of messages. By embedding this description in the type system of a programming language, adherence to the protocol can be verified by static typechecking; if desired, dynamic monitoring can be introduced into the runtime system.

Several researchers have worked towards making typechecked communication structures available for mainstream software development, by transferring session types from their original setting of pi-calculus to functional and object-oriented languages [3, 5–8, 15, 17, 19]. Gay *et al.* [9] proposed an integration of session types and object-oriented programming through the concept of typestates [22], in which methods are constrained to be called only in particular sequences. They defined a translation from the session type of a communication channel endpoint into a typestate specification that constrains the use of send and receive methods on an object representing the channel endpoint. Their notation for typestate specifications was inspired by the syntax of session types.

Dardha, Gay, Kouzapas and Perera extended that work and implemented it as Mungo [16], a front-end typechecking tool for Java. They also generalised the translation from session types to typestate specifications, so that it handles multiparty [11] instead of binary session types, and made it concrete by implementing StMungo [16], a translator from the Scribble [20, 25] protocol description language into Mungo specifications. The Scribble description of a protocol is translated into an API with which to program implementations of protocol roles; the typestate specification associated with the API permits static checking of the correctness of the implementation of a role. Typestate specifications do not represent the notion of duality of session types; compatibility between roles depends on the assumption that their typestate specifications are derived from a single global session type. The paper by Kouzapas *et al.* [16] illustrated the use of Mungo and StMungo with a substantial case study of an SMTP client [21], including the low-level implementation details necessary to enable communication with standard SMTP servers. This achieved the long-standing goal of using session types to specify and verify implementations of real internet protocols.

The present chapter describes Mungo and StMungo in relation to three examples. The first, in Section 14.2.1, illustrates Mungo by defining and checking a typestate specification for an iterator. The second, in Section 14.3, is a simple multiparty scenario based on a travel agency. Finally, in Section 14.4, we show how Mungo and StMungo can be used to typecheck a client for the POP3 protocol [18].

14.2 Mungo: Typestate Checking for Java

Mungo is a static analysis tool that checks typestate properties in Java programs. Mungo implements two main components. The first is a Java-like syntax to define typestate specifications for classes, and the second is a typechecker that checks whether objects that have typestate specifications are used correctly. Mungo typechecks standard Java code without syntactic extensions; typestate specifications are defined in separate files and are associated with Java classes by means of the Java annotation mechanism. After typechecking with Mungo, programs can be compiled and run using standard Java tools. The declaration of a typestate specification in a single file contrasts with other approaches that take the viewpoint of typestate as pre- and post-conditions on methods; we discuss this point in Section 14.5. If a class has a typestate specification, the Mungo typechecker analyses each variable of that class in the program and extracts the method call behaviour (sequences of method calls) through the variable's life. Finally, it checks the extracted information against the sequences of method calls allowed by the typestate specification.

Mungo is implemented in the JastAdd [10] framework, which is a Reference Attribute Grammar (RAG) meta-compiler suite compatible with Java. JastAdd provides a Java parser and typechecker, and was also used to implement a parser for the typestate specification language.

Mungo supports typechecking for a subset of Java. The programmer can define classes with typestate specifications and classes without them. The typechecking procedure tracks variables storing instances of classes with typestate specifications, through argument passing and return values. Moreover, the typechecking procedure for the fields of a class follows the typestate specification of the class to infer a typestate usage for the fields. For this reason fields that have typestate specifications must be defined in a class that also has a typestate specification.

Mungo first runs the Java typechecker provided by the JastAdd framework. If there are no errors then Mungo performs additional well-formedness checks before it runs the typestate checking procedure. First, the tool checks for well-formed typestate specifications: they must be deterministic and all states must be reachable from the initial state. Second, it checks that a class with a typestate specification implements all the methods required in the typestate. Third, arrays cannot store objects that have typestates, because array access, and thus inference for objects that are stored in an array, cannot be determined at statically. Finally, fields with typestate specifications must be private and non-static, to disallow external interference with their state.

Completing the coverage of Java will require further work. Some features we anticipate to be relatively straightforward extensions, such as synchronised statements, the conditional operator ?:, inner and anonymous classes, and static initialisers. Generics, inheritance and exceptions are non-trivial. Currently, generics are not supported, while inheritance is supported for classes without associated typestate behaviour. Exceptions are supported syntactically but are type-checked under the (unsound) assumption that no exceptions are thrown; a `try{...} catch(Exception e) {...}` statement is typechecked by typechecking the `try` block but not the `catch` block. If the program does not throw exceptions then there will be no violations of typestate specifications, but exception handlers may violate typestates.

14.2.1 Example: Iterator

We introduce some of the features of Mungo through an example that enforces correct usage of a Java Iterator. The example shows how a programmer can define an API and associate it with a typestate specification in order to constrain the order in which methods can be called. In the code below we define class `StateIterator` to wrap a Java Iterator. We use the Java annotation syntax `@Typestate("StateIteratorProtocol")` to associate the class `StateIterator` with the typestate specification `StateIteratorProtocol`. We often refer to a typestate specification as a protocol, following the established terminology of "object protocol" in the typestate literature.

```
1  package iterator;
2  import java.util.Iterator;
3
4  @Typestate("StateIteratorProtocol")
5  class StateIterator {
```

```
6     private Iterator iter;
7
8     public StateIterator(Iterator i) { iter = i; }
9     public Object next()             { return iter.next(); }
10    public void remove()             { iter.remove(); }
11    public Boolean hasNext() {
12      if(iter.hasNext() == true)
13        return Boolean.True;
14      return Boolean.False;
15    } }
```

We assume that the underlying implementation of the Java Iterator includes the `remove()` method. The implementation of method `hasNext()` uses the Iterator to discover whether the underlying collection has more elements. It assumes the definition of the enumeration

```
1    enum Boolean { True, False }
```

which is provided as part of the Mungo framework. This enumeration is used to specify dependency of the protocol on the result of a method.

Overall, the `StateIteratorProtocol` protocol ensures that the Java Iterator will be used in a way that throws no exceptions (method `next()` throws `NoSuchElementException` when there are no more elements in the underlying collection, and method `remove()` throws `IllegalStateException` when there is no element to removed). The code below defines the typestate specification `StateIteratorProtocol`.

```
1   package iterator;
2
3   typestate StateIteratorProtocol {
4     HasNext = { Boolean hasNext(): <True: Next, False: end> }
5     Next =    { Object next(): HasNextOrRemove }
6     HasNextOrRemove = {
7       void remove(): HasNext,
8       Boolean hasNext(): <True: NextOrRemove, False: end>
9     }
10    NextOrRemove = {
11      void remove(): Next,
12      Object next(): HasNextOrRemove
13    } }
```

A new iterator object is in state HasNext, because that is the first state in the definition. The only method available is hasNext(). If method next() were available then NoSuchElementException might be thrown in the case where there are no (more) elements in the underlying collection. Similarly, the availability of method remove() might result in IllegalStateException. A call of method hasNext() means that the continuation of the protocol depends on the return value of the method. In the case of False no further interaction with the iterator is possible, thus preventing possible exceptions. If the value True is returned then the state changes to Next, which forces the programmer to call the next() method and proceed to state HasNextOrRemove. Method remove() is not available because it should only be called after next() in order to remove the element returned by next(). Method hasNext() is not available because calling it would be redundant.

The state HasNextOrRemove offers a choice between methods remove and hasNext(). In the former case the iterator removes the current object and proceeds to the HasNext state. Alternatively, calling hasNext() either proceeds to state NextOrRemove or ends the protocol otherwise. In state NextOrRemove there is still the possibility of removing the last returned object and proceeding to the Next state (this is because a poll has already been done), or getting the next element of the collection using method next() and proceeding to the HasNextOrRemove state.

To summarise, if we assume semantic correctness of the methods of iter (for example, that iter.hasNext() correctly reports the state of iter), then by using Mungo to typecheck code that uses a StateIterator, we can ensure that NoSuchElementException and IllegalStateException will not occur. Specifically, we guarantee: i) not calling the next() method on an empty collection; ii) not calling the remove() when there is no element to remove from the underlying collection; iii) additionally, not having redundant calls of the hasNext() method.

To avoid conflicting state changes, objects with typestates must not be aliased. Mungo uses linear typing to prevent aliasing.

The code below, which is well-typed according to Mungo, creates and uses a StateIterator object. It creates a HashSet containing the positive integers smaller than 32, and then removes the even numbers.

```
1 Collection c = new HashSet();
2 Integer i = 0; while(i < 32) c.add(i++);
3 StateIterator iter = new StateIterator(c.iterator());
4 iterate:
```

```
5  do {
6    switch(iter.hasNext()) {
7      case True:
8        System.out.println(i = (Integer) iter.next());
9        if(i%2 == 0) iter.remove();
10       continue iterate;
11     case False:
12       break iterate;
13   }
14 } while(true);
```

The `HashSet`'s iterator is wrapped in a `StateIterator` object, which is subsequently used according to its protocol. The loop structure in the protocol is matched by the pattern `label: do { ... } while(true);` together with the `continue label;` and `break label;` statements. The `switch` statement handles the possible results of `hasNext()`, controlling the continuation or termination of the loop. The code on line 9 chooses whether or not to call `remove()`; the state here is `HasNextOrRemove`.

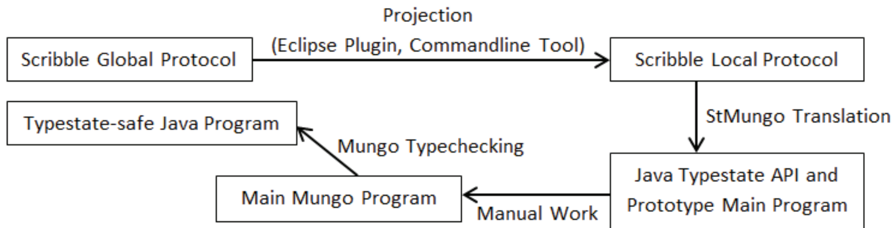
14.3 StMungo: Typestates from Communication Protocols

StMungo (Scribble to Mungo) is a transpiler from Scribble to Java, which also generates Mungo typestate specifications. It is based on the integration of session types and typestates [9] which consists of a formal translation of session types for communication channels into typestate specifications for channel objects. The latter define the order in which the methods of the channel objects can be called. This specification of the permitted sequences of method calls is naturally viewed as a channel protocol. We take a step further: we extend this formal translation from binary to multiparty session types [11] and implement it as StMungo, which translates Scribble local protocols into typestate specifications and prototype implementation code based on TCP/IP sockets. After refinement, the implementation is typechecked using Mungo.

A Scribble local protocol describes the communication between one role and all the other participants in a multiparty scenario, including the way in which messages sent to different participants are interleaved. StMungo is based on the principle that each role in the multiparty communication can be abstracted as a Java class following the typestate corresponding to the role's local protocol. The typestate specification generated by StMungo, together with the Mungo typechecker, guide the programmer in the design and implementation of distributed multiparty communication-based programs with

guarantees of communication safety and soundness. StMungo is the first tool to provide a practical embedding of multiparty session type protocols into object-oriented languages with typestate specifications.

The diagram shows how the toolchain consisting of Scribble, StMungo and Mungo is used to generate a Java program from a Scribble protocol.



We start with a global protocol written in Scribble, which is then validated and projected into local protocols, one for each role specified in the global protocol. At this point we run StMungo on the local projections for which we want to generate a typestate. The tool generates a typestate specification, a Java API and a prototype main program. After completing the main program, typechecking with Mungo verifies that it correctly implements the protocol.

14.3.1 Example: Travel Agency

We now illustrate the toolchain of Scribble, StMungo and Mungo by means of a travel agency example, which models the process of booking a flight through a university travel agent.

Three participants are involved: Researcher (abbreviated `R`), who intends to travel; Agent (`A`), who is able to make travel reservations; and Finance (`F`), who approves expenditure from the budget. In the Scribble [25] language, we first define the global protocol among three *roles*, which are abstract representations of the participants. The protocol consists of sequences of interactions. Every message (e.g. `request`) can be associated with a payload type (e.g. `Travel`), a sender, and one or more receivers. Typically payload types are structured data types defined separately from the protocol specification.

In the global protocol, after the `check` message requesting authorisation for a trip, `F` can choose to `approve` or `refuse` the request.

```
1  global protocol BuyTicket(role R, role A, role F) {
2    request(Travel) from R to A;
3    quote(Price) from A to R;
```

```
4    check(Price) from R to F;
5    choice at F {
6      approve(Code) from F to R,A;
7      ticket(String) from A to R;
8      invoice(Code) from A to F;
9      payment(Price) from F to A;
10   } or {
11     refuse(String) from F to R,A;
12   } }
```

The Scribble tools can be used to validate the protocol definition and to derive a *local* version of the protocol for each role, according to the theory of multiparty session types [11]. This is known as *endpoint projection*. Here is the projection for `R`, which describes only the messages involving that role. The `self` keyword indicates that `R` is the local endpoint.

```
1  local protocol BuyTicket_R(self R, role A, role F) {
2    request(Travel) to A;
3    quote(Price) from A;
4    check(Price) to F;
5    choice at F {
6      approve(Code) from F;
7      ticket(String) from A;
8    } or {
9      refuse(String) from F;
10   } }
```

Notice that the exchange of `invoice` and `payment` between `A` and `F` is not included. Similarly, the local projection for `A` omits the `check` message; we omit its local projection. Finally, the local projection for `F` omits the `request`, `quote` and `ticket` messages.

```
1 local protocol BuyTicket_F(role R, role A, self F) {
2   check(Price) from R;
3   choice at F {
4     approve(Code) to R,A;
5     invoice(Code) from A;
6     payment(Price) to A;
7   } or {
8     refuse(String) from F to R,A;
9   } }
```

The common theme between protocols and typestate specifications is the requirement to do operations in particular orders. Our methodology for implementing the roles in a Scribble protocol is to define a Java class that encapsulates socket connections to provide the necessary communication, and provides methods that send and receive the messages in the protocol. This class constitutes an API for role programming. To ensure that communication methods are called in the order required by the protocol, we associate a typestate specification with the API, so that Mungo can check the correctness of code that uses the API. StMungo generates a Java API and a Mungo specification. If we are implementing all of the endpoints in a system, then the generated APIs are immediately interoperable with each other. However, interoperability with pre-existing endpoints such as a POP3 server (Section 14.4) typically requires an extra layer in order to translate between the abstract message labels defined in Scribble and the detailed textual message formats required by the protocol.

For the `R` role, StMungo converts the `BuyTicket_R` local projection into the following Mungo definitions:

1. `RProtocol`, a typestate specification capturing the interactions local to the `R` role.
2. `RRole`, a Java class that implements `RProtocol` by communication over Java sockets. This is an API that can be used to implement the `R` endpoint.
3. `RMain`, a prototype Java implementation of the `R` endpoint. This runs as a Java process, and provides a `main()` method which uses `RRole` to communicate with the other parties in the session. For testing purposes it provides a command-line interface to choose and display message parameters.

To complete the ticket buying example, we now describe the result of translating the local protocol for `R`. For each choice there is an enumerated type, named according to the numerical 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 the choice in `BuyTicket_R` we have the following definition.

```
1  enum Choice1 { APPROVE, REFUSE; }
```

Every role involved in the choice will have an enumerated type with the same set of values, but the names of the types are not necessarily the same for every role.

The typestate specification `RProtocol` defines the allowed sequences of method calls. As it includes method headers, it also provides similar documentation to an interface. The initial state is the first one defined.

```
1  typestate RProtocol {
2    State0 = { void send_requestTravelToA(Travel): State1 }
3    State1 = { Price receive_quotePriceFromA(): State2 }
4    State2 = { void send_checkPriceToF(Price): State3 }
5    State3 = { Choice1 receive_Choice1LabelFromF():
6                        <APPROVE: State4, REFUSE: State6>  }
7    State4 = { Code receive_approveCodeFromF(): State5 }
8    State5 = { String receive_ticketStringFromA(): end }
9    State6 = { String receive_refuseTravelFromF(): end } }
```

The API is defined by the class `RRole`, which is also generated. When instantiated, it establishes socket connections to the other role objects in the session (`ARole` and `FRole`); we omit the details here.

```
1  @Typestate("RProtocol") public class RRole {
2    public RRole(){
3      ... // Bind the sockets and accept a client connection
4      try { // Create the read and write streams
5        socketAIn = new BufferedReader(..);
6        socketAOut = new PrintWriter(..);
7       } catch (IOException e) {
8         System.out.println("Read failed");  System.exit(-1);
9       } }
10   public void send_requestTravelToA(Travel payload) {
11     this.socketAOut.println(payload);  }
12   public Price receive_quotePriceFromA() {
13     String line = "";
14     try {  line = this.socketAIn.readLine();
15     } catch (IOException e) {
16       System.out.println("Input/Output error."); System.exit(-1);
17     }
18     // Parse line to the appropriate type and then return it
19     return Price.parsePrice(line);  }
20     ... // Define all other methods in RProtocol }
```

The `RMain` class provides a prototype implementation of the `R` endpoint, using the `RRole` class to communicate with the other roles in the system.

Mungo statically checks the correctness of an `R` implementation (either based on the prototype or written separately), by checking that methods are called in allowed sequences and that all possible responses are handled. For example, `main` below is correct.

```
1  public static void main(String[] args) {
2    RRole r = new RRole();
3    Travel t = // input travel;
4    r.send_requestTravelToA(t);
5    Price p = r.receive_quotePriceFromA();
6    r.send_checkPriceToF(p);
7    switch(r.receive_Choice1LabelFromF().getEnum()) {
8      case APPROVE:
9        Code c = r.receive_approveCodeFromF();
10       println(r.receive_ticketStringFromA());
11       break;
12     case REFUSE:
13       println(r.receive_refuseStringFromF());
14       break;
15   } }
```

This code is checked by computing the sequences of method calls that are made on an `RRole` object, inferring the minimal typestate specification that allows those sequences, and then comparing this specification with the declared specification `RProtocol`. The comparison is based on a simulation relation. Typically the programmer would modify the prototype implementation by defining extra business logic, but she is also free to rewrite it completely. Mungo statically checks `RMain`, or any client of the `RRole` class, to ensure that methods of the protocol are called in a valid sequence and that all possible responses are handled.

14.4 POP3: Typechecking an Internet Protocol Client

As a more substantial example, we use a standard internet protocol, POP3 [18] (Post Office Protocol Version 3), to show the applicability of session types in the real world and the use of session type tools to typecheck protocols. The protocol allows an email client to retrieve messages from a server. The diagram below is based on RFC 1939 [18], the official specification of the protocol. The labels "+OK" and "-ERR" are part of the textual message format. For simplicity, several transitions from state `TRANSACTION` have been omitted.

The protocol starts with the client connecting to the server and the server authenticating the connection. The client then has the choice to either submit a username to log into a mailbox, or to end the authorization. Upon receiving the username, the server has the choice to accept the username or to send an error message to the client, for example if the username does not exist. After the username has been accepted, the client is then required to send a password or to end the authorization. If the password is accepted, the transaction stage begins. In the transaction stage, the client has a choice of various commands: the diagram shows just `STAT` (status) and `LIST` (summary list). Some of these requests involve a choice at the server side to either fulfil the request or to send an error message.

Alternatively the client can choose to `QUIT`. The specification of the messages and state transitions of POP3 can be converted into a Scribble global protocol, as shown below.

```
1  global protocol POP3(role S, role C) {
2    OKN(String) from S to C;
3    rec authentication_username {
4      choice at C {
5        USER(String) from C to S;
6        choice at S {
7          OK(String) from S to C;
8          rec authentication_password {
9            choice at C   {
```

```
10              PASS(String) from C to S;
11              choice at S {
12                OK(String) from S to C;
13                rec transaction {
14                  choice at C {
15                    STAT() from C to S;
16                    OKN(int, int) from S to C;
17                    continue transaction;
18                  } or {
19                    LIST() from C to S;
20                choice at S {
21                  OK(String) from S to C;
22                  rec summary_choice_retrieve {
23                    choice at S {
24                      DOT() from S to C;
25                      continue transaction;
26                    } or {
27                      SUM(int, int) from S to C;
28                      continue summary_choice_retrieve; } }
29                } or {
30                  ERR(String) from S to C;
31                  continue transaction; }
32              } or {
33                QUIT() from C to S;
34                OKN(String)from S to C; } }
35              } or {
36                ERR(String) from S to C;
37                continue authentication_password; }
38            } or {
39              QUIT() from C to S;
40              OKN(String) from S to C; } }
41        } or {
42          ERR(String) from S to C;
43          continue authentication_username; }
44      } or {
45        QUIT() from C to S;
46        OKN(String) from S to C; } } }
```

Projection using the Scribble tools produces local protocols for the client and the server. For the rest of this section we focus on the client protocol. For brevity we omit the authentication phase.

```
1  local protocol POP3 (role S,self C) {
2    OKN(String) from S;
3    ...
4                  rec transaction {
5                    choice at C {
6                      STAT() to S;
7                      OKN(int,int) from S;
8                      continue transaction;
9                    } or {
10                     LIST() to S;
11                     choice at S {
12                       OK(String) from S;
13                       rec summary_choice_retrieve {
14                         choice at S {
15                           DOT() from S;
16                           continue transaction;
17                         } or {
18                           SUM(int,int) from S;
19                           continue summary_choice_retrieve; } }
20                     } or {
21                       ERR(String) from S;
22                       continue transaction; }
23                   } or {
24                     QUIT() to S;
25                     OKN(String) from S; } }
26           ...
27       QUIT() to S;
28       OKN(String) from S; } } }
```

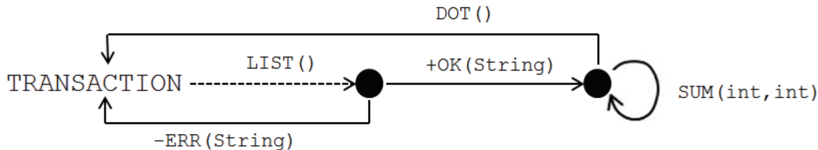
We use StMungo to translate the Scribble local protocol into a typestate specification `CProtocol`, which defines the order in which the communication methods are called.

```
1  typestate CProtocol {
2  State0 = {String receive_OKStringFromS(): State1}
3  ...
4  State9 = {void send_STATToS(): State10,
```

```
5             void send_RETR_NToS(): State12,
6             void send_QUITToS(): State19}
7   State10 = {void send_STATToS(): State11}
8   State11 = {IntInt receive_OKNIntIntFromS(): State9}
9   State12 = {void send_LISTToS(): State13}
10  State13 = {Choice1 receive_Choice1LabelFromS():
11      <OK: State14, ERR: State18>}
12  State14 = {String receive_OKStringFromS(): State15}
13  State15 = {Choice2 receive_Choice2LabelFromS():
14      <DOT: State16, SUM: State17>}
15  State16 = {void receive_DOTFromS(): State9}
16  State17 = {String receive_SUMIntIntFromS(): State15}...}
```

14.4.1 Challenges of Using Mungo and StMungo in the Real World

Programming with loops A POP3 server responds to the `LIST` command by sending any number of lines, terminated by the `DOT` message.



The Scribble description of the state reached by `+OK()` uses explicit recursion in which `continue` jumps to a named state.

```
1  rec summary_choice_list {
2    choice at S    {
3      DOT() from S to C
4      continue transaction;
5    } or {
6      SUM(int, int) from S to C;
7      continue summary_choice_list; } }
```

The Java code generated by StMungo is a direct translation, using labelled statements and `continue`. Given that we are generating imperative code rather than recursive functions, this seems to be the only systematic way to handle the arbitrary structure of Scribble's `rec`. Although `continue` is a `goto`, its use is

controlled and checked by Mungo: it is only allowed when the recursion point in the protocol has been reached. The SJ language [15] introduces `sendWhile` and `receiveWhile` loops to match particular protocol patterns, but we have chosen not to extend Java with new loop constructs.

```
1  _summary_choice_list: do {
2    switch(currentC.receive_Choice2LabelFromS().getEnum()){
3      case Choice2.DOT:
4        Void payload10 = currentC.receive_DOTVoidFromS();
5        System.out.println("Received from S: ." + payload10);
6        continue _transaction;
7      case Choice2.SUM:
8        SUMIntInt payload11 = currentC.receive_SUMIntIntFromS();
9        System.out.println("Received from S: " + payload11);
10       continue _summary_choice_list; } }
11 while(true);
```

Abstract vs. concrete messages When designing a complete system and implementing all the roles, StMungo can generate concrete textual messages in a uniform way; alternatively, we could use a structured message format such as JSON. However, in POP3 and other standard protocols, the client has to work with the textual message formats defined by the protocol. For example, the Scribble message `OK(int, int) from S to C;` corresponds to a line of text such as `+OK 2 200`. In the current implementation of the POP3 example, conversion between abstract and concrete messages is done by handwritten code, but we are working on a tool to generate message converters from a specification.

Naming StMungo converts Scribble message names into Java method names. The method definition depends on whether or not the message appears at the beginning of a Scribble `choice`, and this cause naming conflicts if the same name is used for messages in both kinds of position. For example, `OK` and `OKN` in POP3 would more naturally both be `OK`.

Non-standard implementations Real-world servers do not always follow the RFC exactly. The specification of POP3 states that if the client sends an unknown username, it is rejected and the username must be sent again. However, the server used for this case study, namely `GMX.co.uk`, accepts an unknown username and expects the client to send the password again. Consequently, even after completing the prototype client generated by StMungo and checking it with Mungo, it is necessary to test the client thoroughly

with existing servers if we want to ensure correct operation in all cases. When deviations from the RFC are discovered, the Scribble definition of the protocol can be generalised accordingly. This problem could be reduced by promoting the use of formal protocol descriptions within RFCs.

14.5 Related Work

Session types. The main pieces of related work on session types and Java are the Session Java (SJ) language [15] and the API generation approach [14], both by Hu *et al.* The API generation approach has been used to to analyse an SMTP client in Java. The API for SMTP implements multiparty session types using a pattern in which each communication method returns the receiver object with a new type that determines which communication methods are available at the next step. Standard Java typechecking can verify the correctness of communication when the pattern is used properly, with runtime monitoring being used to ensure linearity constraints are fulfilled. In contrast with this approach, Mungo's approach is completely static.

SJ [15] builds on earlier work [4, 5, 7] to add binary session type channels to Java. SJ implements a library for binary sessions that have a pre-defined interface. The syntax of Java is extended with communication statements to allow typechecking. The scope of a session is restricted to the body of a single method. Mungo removes this restriction by allowing the abstraction of multiparty session types as user-defined objects that can be passed and used throughout different program scopes.

Typestates. There have been many projects that add typestates to practical languages, since the introduction of the concept by Strom and Yemini [22]. Plural [2] is a noteworthy example. It is based on Java and has been used to study access control systems. Plural implements typestates by using annotations to define pre- and post-conditions on methods, referring to abstract states and predicates on instance variables. By contrast, Mungo explicitly defines the possible sequences of method calls. Plural and Mungo both allow the typestate after a method call to depend on the return value.

Plaid [1, 23] introduces typestate-oriented programming as a paradigm. 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. Similarly to classes, states can be structured into an inheritance hierarchy. As opposed to Plaid, Mungo focuses on the object-oriented paradigm in order to be applicable to Java.

Our previous paper [16] discusses related work in more detail.

Acknowledgements This research was supported by the UK EPSRC project "From Data Types to Session Types: A Basis for Concurrency and Distribution" (EP/K034413/1) and by COST Action IC1201 "Behavioural Types for Reliable Large-Scale Software Systems". We thank the reviewers for their detailed comments.

References

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA '09*, pages 1015–1022. ACM Press, 2009.
[2] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09*, volume 5653 of *Springer LNCS*, pages 195–219, 2009.
[3] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, and E. Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theoret. Comp. Sci.*, 410:142–167, 2009.
[4] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and session types. *Information and Computation*, 207(5): 595–641, 2009.
[5] M. Dezani-Ciancaglini, E. Giachino, S. Drossopoulou, and N. Yoshida. Bounded session types for object oriented languages. In *FMCO '06*, volume 4709 of *Springer LNCS*, pages 207–245, 2006.
[6] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopolou. Session types for object-oriented languages. In *ECOOP '06*, volume 4067 of *Springer LNCS*, pages 328–352, 2006.
[7] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopolou. A distributed object-oriented language with session types. In *TGC '05*, volume 3705 of *Springer LNCS*, pages 299–318, 2005.
[8] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.
[9] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL '10*, pages 299–312. ACM Press, 2010.
[10] G. Hedin. An introductory tutorial on JastAdd attribute grammars. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Springer LNCS*, pages 166–200, 2011.
[11] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM Press, 2008.

[12] K. Honda. Types for dyadic interaction. In *CONCUR '93*, volume 715 of *Springer LNCS*, pages 509–523, 1993.
[13] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98*, volume 1381 of *Springer LNCS*, pages 122–138, 1998.
[14] R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE 16*, volume 9633 of *Springer LNCS*, pages 401–418, 2016.
[15] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in Java. In *ECOOP '08*, volume 5142 of *Springer LNCS*, pages 516–541, 2008.
[16] D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP '16*, pages 146–159. ACM Press, 2016.
[17] M. Neubauer and P. Thiemann. An implementation of session types. In *PADL '04*, volume 3057 of *Springer LNCS*, pages 56–70, 2004.
[18] Post office protocol version 3, RFC 1939. `https://www.ietf.org/rfc/rfc1939`.
[19] R. Pucella and J. A. Tov. Haskell session types with (almost) no class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 25–36. ACM Press, 2008.
[20] Scribble project homepage. `www.scribble.org`.
[21] Simple mail transfer protocol, RFC 821. `https://tools.ietf.org/html/rfc821`.
[22] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1): 157–171, 1986.
[23] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter. First-class state change in Plaid. In *OOPSLA '11*, pages 713–732. ACM Press, 2011.
[24] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE '94*, volume 817 of *Springer LNCS*, pages 398–413, 1994.
[25] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC '13*, volume 8358 of *Springer LNCS*, pages 22–41, 2013.