1

Mungo and StMungo: Tools for Typechecking Protocols in Java

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

School of Computing Science, University of Glasgow, UK

1.1 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 [13, 14, 25]. 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

S. J. Gay and A. Ravara (eds), Behavioural Types: from Theory to Tools, 1–19.

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, 6, 7, 9, 16, 18, 20]. Gay *et al.* [10] proposed an integration of session types and object-oriented programming through the more general concept of typestate [23], in which methods are constrained to be called only in particular sequences. They defined a translation from the session types of communication channels into typestate specifications that constrained the use of send and receive methods on channel objects. Their notation for typestate specifications was itself inspired by the syntax of session types.

We (the first four authors of the present chapter) extended that work and implemented it as Mungo [17], a front-end typechecking tool for Java. We also generalised the translation from session types to typestate specifications, so that it handles multiparty [12] instead of binary session types, and made it concrete by implementing StMungo [17], a translator from the Scribble [21, 26] 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. Our previous paper [17] illustrated the use of Mungo and StMungo with a substantial case study based on the SMTP protocol [22], 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 1.2.1, illustrates Mungo by defining and checking a typestate specification for an iterator. The second, in Section 1.3, is a simple multiparty scenario based on a travel agency. Finally, in Section 1.4, we show how Mungo and StMungo can be used to typecheck a client for the POP3 protocol [19].

1.2 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 is modular enough to allow typechecking of 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. This allows programs that have been checked by Mungo to 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 1.5. If a class has a typestate specifiction, the Mungo typechecker analyses each object of that class in the program and extracts the method call behaviour (sequences of method calls) through the object'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 [11] framework, which is a Reference Attribute Grammar (RAG) meta-compiler suite compatible with the Java programming language. The JastAdd framework provides a Java parser which was used for the implementation of the Mungo typechecker. The JastAdd suite was also used to implement a parser for the Java-like typestate specification language.

Mungo supports typechecking for a subset of Java. The programmer can define both classes that follow a typestate specification and classes that do not. The typechecking procedure follows objects (instances of the former classes) 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 follow a typestate specification are only allowed to be defined in a class that also follows a typestate specification.

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 type-

checking the try body and if an exception is thrown a typestate violation may result.

1.2.1 Example: Iterator

We introduce some of the features of Mungo through an example that enforces correct usage of a Java Iterator. 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.

```
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 `void remove()` method. The implementation of method `Boolean 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 }
```

The enumeration is used to enable the external choice feature of the Mungo protocol (see below). Overall, the `StateIteratorProtocol` protocol will ensure that the Java Iterator will be accessed in an order that raises no exceptions (method `Object next()` raises the `NoSuchElementException` exception when there are no more elements on the underlying collection and method `void remove()` raises the `IllegalStateException` exception

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` where the only method available is `Boolean hasNext()`. If method `Object next()` were available then a possible `NoSuchElementException` might be thrown in the case where there are no (more) elements on the underlying collection. Similarly, the availability of method `void remove()` might result in the `IllegalStateException` exception. A call of method `Boolean hasNext()` enables an external choice on the continuation of the protocol, that depends on the return value of the method. In the case of `False` no further interaction with the iterator is possible, thus disallowing possible exceptions from a further interaction with the Iterator. If the value `True` is returned then the state changes to `Next`, which forces the programmer to call the `Object next()` method and proceed to state `HasNextOrRemove`. Should the method `void remove()` be available, then upon calling it an exception will be raised because there is no element to remove. Also, availability of the `Boolean hasNext()` method might result in a redundant call. In the `HasNextOrRemove` is an internal choice between methods `void remove` and `Boolean hasNext()`. In the former case the iterator can remove the current object via the `void remove` method and proceed to the `HasNext` state. Alternatively, it can enable an external choice by calling `Boolean hasNext()`, where upon having more elements the Iterator will proceed to state `NextOrRemove` or end 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 `Object next()` and proceeding to the `HasNextOrRemove` state.

To summarise, if we assume a correct implementation of the above protocol, then we can ensure exception-free access to the elements of the underlying collection. Specifically, it ensures: i) not calling the `Object next()` method on an empty collection; ii) not having redudant calls of the `Boolean hasNext()` method; and iii) not calling the `void remove()` when there is no element to remove from the underlying collection.

The code below shows the creation and usage of a `StateIterator` object:

```
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(); continue iterate;
10     case False: break iterate;
11   }
12 } while(true);
```

A new collection is created and filled with elements. The collection's iterator is then wrapped in a `StateIterator` object, that is subsequently used according to its protocol. The switch statement is used to implement an external choice. The pattern `label: do { ... } while(true);` together with the `continue label;` and `break label;` statements is used when an external choice controls a loop structure. Also, notice the internal choice (`if(i%2 == 0)` in line 9) between calling the `void remove()` method and continue with the loop that removes all even numbers from the collection and corresponds to the implementation of the protocol state `HasNextOrRemove`.

1.3 StMungo: Typestate from Communication Protocols

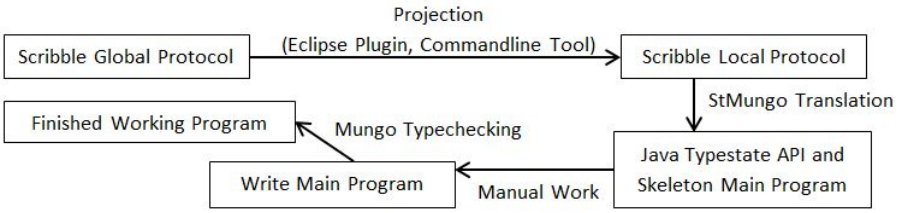
StMungo (Scribble to Mungo) is a transpiler from Scribble to Java. It is based on the integration of session types and typestate [10] which

consists of a formal translation of session types for communication channels into typestate specifications for channel objects, the latter defines 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 and extend this formal translation from binary to multiparty session types [12] and implement it as StMungo, which translates Scribble local protocols into typestate specifications and skeleton socket-based implementation code. The resulting code is then 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 from StMungo together with the Mungo typechecker can guide the user in the design and implementation of distributed multiparty communication-based programs with guarantees on communication safety and soundness. StMungo is the first tool to provide a practical embedding of Scribble multiparty protocols into object-oriented languages with typestate.

We are now ready to describe the toolchain composed by Scribble, StMungo and Mungo. To do so, first we give a diagram showing how these tools are used step by step, and afterwards we show an example.

The diagram below depicts the process of generating a Java program starting from Scribble protocols.



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 skeletal main program. After

completing the main program, we are ready to run Mungo on it, which produces standard Java code.

1.3.1 Example: Travel Agency

At this point we want to illustrate the toolchain of Scribble, StMungo and Mungo by the 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 [26] 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 following global protocol, after the `quote` and the `check` message requesting authorisation for a trip, Finance 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   } }
```

Scribble 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 [12]. This is known as *endpoint projection*. Here we show the projection for Researcher, which describes only the messages involving that role. The `self` keyword indicates that `R` is the local endpoint.

```
local protocol BuyTicket_R(self R, role A, role F) {
  request(Travel) to A;
  quote(Price) from A;
  check(Price) to F;
  choice at F {
    approve(Code) from F;
    ticket(String) from R;
  } or {
    refuse(String) from F;
  } }
```

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

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

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 complete. However, in the POP3 example (Section 1.4), an extra layer is necessary 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 definition 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 Researcher endpoint.
3. `RMain`, a skeletal Java implementation of the Researcher endpoint. This runs as a Java process, and provides a `main()` method which uses `RRole` to communicate with the other parties in the session.

To complete the ticket buying example, we now describe the result of translating the local protocol for Researcher. For each choice there is an enumerated type.

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

The typestate specification defines the allowed sequences of method calls. It plays a similar role to an interface, as method headers are included. The initial state is the first one that is defined. A constructor is also generated, but it does not appear in the typestate specification. The generated `RProtocol` definition is as follows:

```
typestate RProtocol {
  State0 = { void send_requestTravelToA(Travel): State1 }
  State1 = { Price receive_quotePriceFromA(): State2 }
  State2 = { void send_checkPriceToF(Price): State3 }
  State3 = { Choice1 receive_Choice1LabelFromF():
                  <APPROVE: State4, REFUSE: State6>  }
  State4 = { Code receive_approveCodeFromF(): State5 }
  State5 = { String receive_ticketStringFromA(): end }
  State6 = { String receive_refuseTravelFromF(): end } }
```

The API is defined by the class `RRole`, which is also generated.

```
@Typestate("RProtocol") class RRole {
  // Constructor and method definitions.
}
```

The `RRole` class provides an implementation of `RProtocol` based on Java sockets. When instantiated, it connects to the other role objects in the session (`ARole` and `FRole`); we omit the details here. The `RMain` class

provides a skeletal implementation of the Researcher endpoint, using the `RRole` class to communicate with the other roles in the system.

Mungo is able to statically check the correctness of a Researcher implementation, by checking that methods are called in allowed sequences and that all possible responses are handled. For example, the following method is correct.

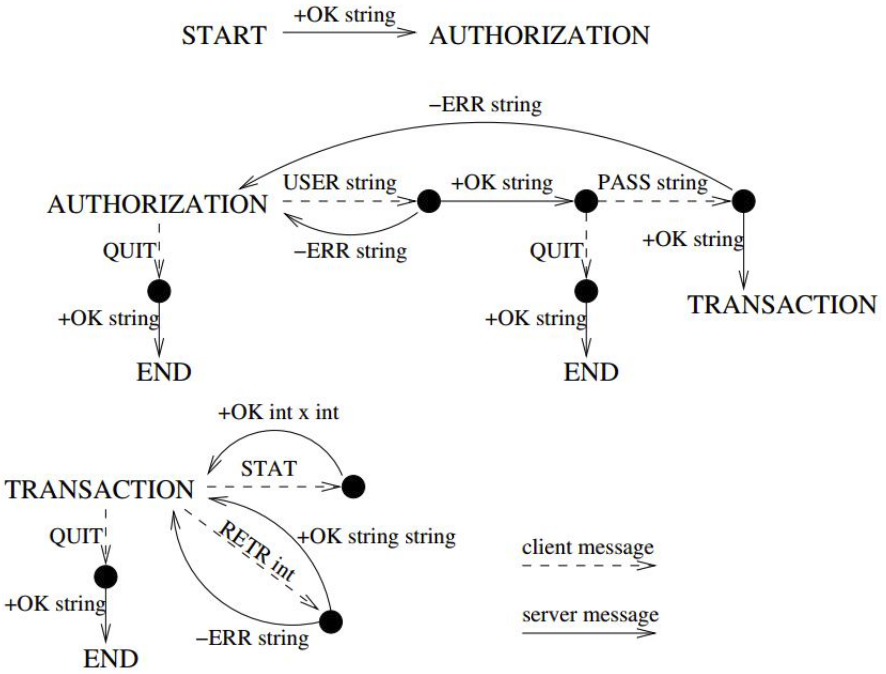
```
1  public static void main(String[] args) {
2    RRole r = new RRole();
3    Travel t = // input travel;
4    r.send_requestTravelToA(t);
5    Price p = r.receive_quotePriceFromA();
6    r.send_checkPriceToF(p);
7    switch(r.receive_Choice1LabelFromF().getEnum()) {
8      case APPROVE:
9        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 flesh out the skeletal implementation with extra business logic. Mungo is able to statically check `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.

1.4 POP3: Typechecking an Internet Protocol Client

As a more substantial example, we use a standard internet protocol, POP3 [19] (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 [8] is based on RFC 1939

[19], the official specification of the protocol. 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 `STAT` (mailbox statistics) and `RETR` (retrieve a message). 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 also choose to end the transaction. 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   OK(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                  RETR_n(int) 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(String) 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              OK(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            OK(String) from S to C; } }
41      } or {
```

```
        ERR(String) from S to C;
        continue authentication_username; }
    } or {
      QUIT() from C to S;
      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, and to reduce the size of the listings we omit the authentication phase.

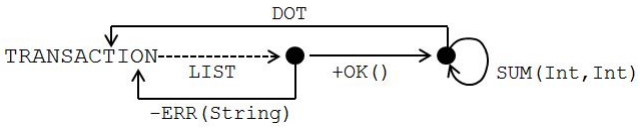
```
local protocol POP3 (role S,self C) {
  OK(String) from S;
  ...
              rec transaction {
                choice at C {
                  STAT() to S;
                  OKN(int,int) from S;
                  continue transaction;
                } or {
                  RETR_n(int) to S;
                  choice at S {
                    OK(String) from S;
                    rec summary_choice_retrieve {
                      choice at S {
                        DOT() from S;
                        continue transaction;
                      } or {
                        SUM(String) from S;
                        continue summary_choice_retrieve; } }
                  } or {
                    ERR(String) from S;
                    continue transaction; }
                } or {
                  QUIT() to S;
                  OK(String) from S; } }
          ...
      QUIT() to S;
      OKN(String) from S; } } }
```

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

```
typestate CProtocol {
State0 = {String receive_OKStringFromS(): State1}
...
State9 = {void send_STATToS(): State10,
    void send_RETR_NToS(): State12,
    void send_QUITToS(): State19}
State10 = {void send_STATToS(): State11}
State11 = {intInt receive_OKNIntIntFromS(): State9}
State12 = {void send_RETR_nintToS(int): State13}
State13 = {Choice1 receive_Choice1LabelFromS():
    <OK: State14, ERR: State18>}
State14 = {String receive_OKStringFromS(): State15}
State15 = {Choice2 receive_Choice2LabelFromS():
    <DOT: State16, SUM: State17>}
State16 = {void receive_DOTFromS(): State9}
State17 = {String receive_SUMStringFromS(): State15}...}
```

1.4.1 Challenges of using Scribble in the real world

Programming with loops In response to the `LIST` command in POP3, the server can send any number of lines, terminated by the `DOT` message.



The corresponding Scribble description uses explicit recursion in which `continue` jumps to a named state.

```
rec summary_choice_list {
  choice at S   {
    DOT() from S to C
    continue transaction;
  } or {
```

```
    SUM(int, int) from S to C;
    continue summary_choice_list; } }
```

The skeleton Java code generated by StMungo is a direct translation, using labelled statements and `continue`. This might, however, be considered an unnatural programming style in Java.

```
summary_choice_list: do {
  switch(currentC.receive_Choice2LabelFromS().getEnum()){
    case Choice2.DOT:
      Void payload10 = currentC.receive_DOTVoidFromS();
      System.out.println("Received from S: ." + payload10);
      continue _transaction;
    case Choice2.SUM:
      SUMIntInt payload11 = currentC.receive_SUMIntIntFromS();
      System.out.println("Received from S: " + payload11);
      continue _summary_choice_list; } }
while(true);
```

Abstract vs. concrete messages When designing a complete system and implementing all of the roles, it is possible for StMungo to generate concrete textual messages in a uniform way; alternatively, it would be possible to use a structured message format such as JSON. However, in POP3 and other standard protocols, the client has to work with the specific 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 send as `+OK 2 200`. In the current implementation of the POP3 example, conversion between abstract and concrete messages is done by hand-written 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, which can cause naming conflicts if the same name is used for messages with different formats. For example, the messages `OK` and `OKN` would more naturally both be called `OK`, which is allowed in Scribble despite their different payload types but would result in overloaded Java methods without disambiguating parameter types. The general point is that it is difficult to write Scribble in a truly language-independent style.

Non-standard implementations Real-world servers do not always follow the RFC exactly. The specification of POP3 says 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 skeleton client generated by StMungo and checking it with Mungo, it is necessary to test the client thoroughly with existing servers. This problem could be avoided by mandating the use of Scribble within RFCs and requiring implementations to demonstrate compliance with the Scribble specification. We recognise that such a situation is unlikely to be achieved.

1.5 Related Work

Session types. The main instances of related work on session types and Java are the Session Java (SJ) language [16] and the API generation approach [15], 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, Mungo's analysis of SMTP and POP3 statically checks all aspects of the protocol implementation.

SJ [16] builds on earlier work [5, 4, 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.

Typestate. There have been many projects that add typestate to practical languages, since the introduction of the concept by Strom and Yemini [23]. Plural [2] is a noteworthy example. It is based on Java and has been used to study access control systems. Plural implements typestate by using annotations to define pre- and post-conditions on

methods, referring to abstract states and predicates on instance variables. In 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, 24] 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.

A more detailed discussion of related work can be found in our previous paper [17].

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".

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, A. Ravara, and V. T. Vasconcelos. Session types for inter-process communication. Technical Report TR-2003-133, Comp. Sci., Univ. Glasgow, 2003.

[9] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[10] 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.

[11] 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.

[12] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM Press, 2008.

[13] K. Honda. Types for dyadic interaction. In *CONCUR '93*, volume 715 of *Springer LNCS*, pages 509–523, 1993.

[14] 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.

[15] R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE 16*, volume 9633 of *Springer LNCS*, pages 401–418, 2016.

[16] 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.

[17] 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.

[18] M. Neubauer and P. Thiemann. An implementation of session types. In *PADL '04*, volume 3057 of *Springer LNCS*, pages 56–70, 2004.

[19] Post office protocol version 3, RFC 1939. `https://www.ietf.org/rfc/rfc1939`.

[20] 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.

[21] Scribble project homepage. `www.scribble.org`.

[22] Simple mail transfer protocol, RFC 821. `https://tools.ietf.org/html/rfc821`.

[23] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[24] 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.

[25] 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.

[26] 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.