Foundations and Trends® in Programming Languages
Vol. 3, No. 2-3 (2016) 95–230

DOI: 10.1561/2500000031

now
the essence of knowledge

Behavioral Types in Programming Languages

Davide Ancona, DIBRIS, Università di Genova, Italy
Viviana Bono, Dipartimento di Informatica, Università di Torino, Italy
Mario Bravetti, Università di Bologna, Italy / INRIA, France
Joana Campos, LaSIGE, Faculdade de Ciências, Univ de Lisboa, Portugal
Giuseppe Castagna, CNRS, IRIF, Univ Paris Diderot, Sorbonne Paris Cité, France
Pierre-Malo Deniélou, Royal Holloway, University of London, UK
Simon J. Gay, School of Computing Science, University of Glasgow, UK
Nils Gesbert, Université Grenoble Alpes, France
Elena Giachino, Università di Bologna, Italy / INRIA, France
Raymond Hu, Department of Computing, Imperial College London, UK
Einar Broch Johnsen, Institutt for informatikk, Universitetet i Oslo, Norway
Francisco Martins, LaSIGE, Faculdade de Ciências, Univ de Lisboa, Portugal
Viviana Mascardi, DIBRIS, Università di Genova, Italy
Fabrizio Montesi, University of Southern Denmark
Rumyana Neykova, Department of Computing, Imperial College London, UK
Nicholas Ng, Department of Computing, Imperial College London, UK
Luca Padovani, Dipartimento di Informatica, Università di Torino, Italy
Vasco T. Vasconcelos, LaSIGE, Faculdade de Ciências, Univ de Lisboa, Portugal
Nobuko Yoshida, Department of Computing, Imperial College London, UK

Contents

Abstract

A recent trend in programming language research is to use behavioral type theory to ensure various correctness properties of large-scale, communication-intensive systems. Behavioral types encompass concepts such as interfaces, communication protocols, contracts, and choreography. The successful application of behavioral types requires a solid understanding of several practical aspects, from their representation in a concrete programming language, to their integration with other programming constructs such as methods and functions, to design and monitoring methodologies that take behaviors into account. This survey provides an overview of the state of the art of these aspects, which we summarize as the *pragmatics* of behavioral types.

D. Ancona et al.. *Behavioral Types in Programming Languages*. Foundations and Trends® in Programming Languages, vol. 3, no. 2-3, pp. 95–230, 2016.

DOI: 10.1561/2500000031.

1

Introduction

Modern society is increasingly dependent on large-scale software systems that are distributed, collaborative and communication-centered. Correctness and reliability of such systems depend on compatibility between components and services that are newly developed or may already exist. The consequences of failure are severe, including security breaches and unavailability of essential services. Current software development technology is not well suited to producing these large-scale systems, because of the lack of high-level structuring abstractions for complex communication behavior.

A recent trend in current research is to use *behavioral type theory* as the basis for new foundations, programming languages, and software development methods for communication-intensive distributed systems. Behavioral type theory encompasses concepts such as interfaces, communication protocols, contracts, and choreography. Roughly speaking, a *behavioral type* describes a software entity, such as an object, a communication channel, or a Web Service, in terms of the sequences of *operations* that allow for a *correct interaction* among the involved entities. The precise notions of "operations" and of "correct interaction" are very much context-dependent. Typical examples of op-

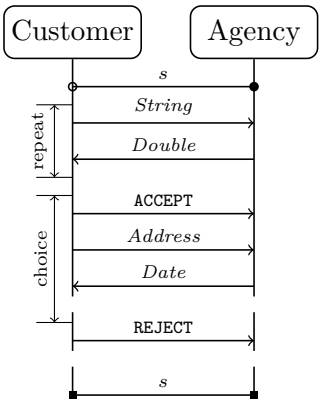


Figure 1.1: Graphical representation of the Customer-Agency protocol.

erations are invoking a method on an object, connecting a client with a Web Service in a distributed system, sending a message between cores in a parallel program. The notion of correct interaction may encompass both safety properties (such as the communication of valid method arguments, the absence of communication errors, the absence of deadlocks) as well as liveness properties (such as the eventual receipt of a message or the eventual termination of an interaction).

To illustrate some paradigmatic aspects of behavioral type theory more concretely, consider the diagram in Figure 1.1 depicting the interaction between two entities, named Customer and Agency. In this diagram, the horizontal lines represent interaction events between the two entities and the vertical lines represent their temporal ordering. The s-labeled line at the top of the diagram denotes the establishment of a connection between the two entities and the definition of an interaction scope that is often called *session*. The identifier s distinguishes this particular session from others (not depicted) in which Customer and Agency may be involved. We can think of s as the name of a communication channel that is known only to Customer and Agency. The proper interaction consists of two phases: the first one, marked as "repeat" in the figure, is made of an unbound number of queries issued by a Customer who is planning a trip through a travel Agency. Each query

includes the journey details, abstracted as a message of type *String*, to which the Agency answers with the price of the journey, represented as a message of type *Double*. In the second phase, marked as "choice", Customer decides whether to book one of the journeys, which it signals by sending either an `ACCEPT` message followed by the *Address* to which the physical documents related to the journey should be delivered at some *Date* estimated by Agency, or a `REJECT` message that simply terminates the interaction. Arrows in the diagram denote the direction of messages. The discontinuity in the vertical development of the protocol suggests that the sub-protocols beginning with the `ACCEPT` and `REJECT` messages are mutually exclusive, the decision being taken by Customer. Eventually, the interaction between Customer and Agency terminates and the session that connects them is closed. This is denoted by the s-labeled line at the bottom of the diagram. In summary, the diagram describes a communication protocol between Customer and Agency as a set of valid sequences of interactions. Making sure that some piece of code modeling either Customer or Agency adheres to this protocol is among the purposes of behavioral type systems, and the technical instrument through which this is accomplished is behavioral types.

In the setting of typed programming languages, the challenge posed by describing a channel like s with a type is that the *same entity* s is used for exchanging messages of *different types* (labels such as `ACCEPT` and `REJECT`, integers, strings, floating-point numbers, dates, etc.) at *different times* and traveling in *different directions* (both Customer and Agency send and receive messages on s). Therefore, it is not obvious which *unique* type should be given to a channel like s or, equivalently, to the functions/methods for using it. The solution adopted in conventional (*i.e.*, non-behavioral) type systems, and that is found in virtually all mainstream programming languages used today, is to declare that communication channels like s can be used for exchanging "raw" messages in the form of *byte arrays* or *strings*. It is up to the programmer to appropriately *marshal* data into such raw messages before transmission and correspondingly *unmarshal* raw messages into data when they reach their destination. In Java, for instance, the `InputStream` and `OutputStream` interfaces and related ones provide `read` and `write`

methods that respectively read data from a stream to a byte array and write data from a byte array to a stream. The main shortcoming of this approach is that it jeopardizes all the benefits and guarantees provided by the type system: such lax typing of channels and of the operations for using them provides no guarantee that the (un)marshalled data has the expected type, nor does it guarantee that messages flow along a channel in the direction intended by the protocol. Essentially, the approach corresponds to using *untyped* channels and establishes a border beyond which the type system of the programming language is no longer in effect. The resulting code is declared well typed by the compiler, but it may suffer from type-related errors (or other issues, such as deadlocks) at runtime.

The key idea of a behavioral type theory is to enrich the expressiveness of types so that it becomes possible to formally describe the sequences of messages (informally depicted in Figure 1.1) that are expected to be exchanged along the communication channel s that connects Customer and Agency. This type can then be used by a type checker to verify that the programs implementing Customer and Agency interact in accordance with the intended communication protocol. In fact, we can imagine *two* different types associated with channel s, depending on viewpoint we take, that of the Customer or that of the Agency. If we take the first viewpoint, we can describe s with a type T defined as

$$T = \bigoplus \left\{ \begin{array}{lcl} \texttt{QUERY} & : & !\mathit{String}.?\mathit{Double}.T \\ \texttt{ACCEPT} & : & !\mathit{String}.?\mathit{Date}.\texttt{end} \\ \texttt{REJECT} & : & \texttt{end} \end{array} \right\}$$

where:

- The symbol $\bigoplus$ denotes a choice of possible behaviors that Customer can attain to, each choice being represented by a symbolic label. In this case, the possible behaviors for Customer are querying the Agency (label `QUERY`), accepting an offer from the Agency (label `ACCEPT`), or quitting the interaction (label `REJECT`).
- The punctuation marks ! and ? respectively prefix the type of messages sent (*String*) and received (*Double* and *Date*) by Cus-

tomer. With these annotations, we can specify the intended direction of messages.

- The punctuation marks : and . represent the sequentiality of actions described by the type. In this case, a Customer that queries an Agency must first send a message of type *String* and *then* wait for a message of type *Double*. With these annotations, we can specify how the capabilities of the channel change as the channel is used for input/output operations.

- The occurrence of T on the right hand side of the equation indicates that T is a *recursive* type, therefore allowing for an unbounded number of queries from Customer to Agency. This makes it possible to specify recursive protocols.

- `end` marks the points in which the interaction between Customer and Agency terminates and no more messages are supposed to be exchanged.

If we take the Agency viewpoint, it is reasonable to expect that the type of s should express complementary behaviors: the Agency offers choices when Customer selects one, the Agency receives a message when the Customer sends one, and vice versa. Customer and Agency should also agree on the moments in which the interaction terminates. This relation between the behaviors of Customer and Agency can be formalized as a notion of *duality* between the two types of s. In particular, the *dual* of T is the type S defined as

$$S = \sum \left\{ \begin{array}{lcl} \texttt{QUERY} & : & ?\mathit{String}.!\mathit{Double}.S \\ \texttt{ACCEPT} & : & ?\mathit{String}.!\mathit{Date}.\texttt{end} \\ \texttt{REJECT} & : & \texttt{end} \end{array} \right\}$$

obtained from T by swapping choices $\bigoplus$ with offers $\sum$, inputs ? with outputs !, and leaving `end` unchanged. Now, checking that Customer and Agency use the respective ends of s according to T and S makes sure that choices and offers match and messages of the right type are exchanged at the right time. In summary, that Customer and Agency interact correctly.

The successful application of behavioral types to the development of reliable, large-scale software requires both the study of formal type theories but also understanding and addressing more practical aspects, including the representation of behavioral types such as T and S in a concrete programming language, the integration of behavioral type checking with other programming constructs like methods and functions, and also design methodologies that take behaviors into account. The aim of this survey is to provide a first comprehensive overview of the state of the art of these aspects, which we may summarize as the *pragmatics* of behavioral types. The survey is structured as a series of chapters, each covering a particular programming paradigm or methodology. Below is an account of the content of each chapter:

- Chapter 2 is devoted to the integration of behavioral types into Object-Oriented languages. Object-oriented languages are relevant for their widespread adoption in the current development of software, for the wealth and popularity of tools that are available, and because objects nicely fit a distribution model to which behavioral types can be applied naturally. The integration can be achieved in different ways: either by *enriching* the languages with constructs (in particular, sessions) that call for a corresponding extension at the type level, or by *amalgamating* sessions and objects to the point that the objects themselves become the entities for which a behavioral description is required, for example to specify the order in which methods must/can be invoked. We also survey a parallel, but related line of research concerning *typestate*. This concept, originally introduced for discriminating the state of imperative variables (uninitialized, initialized, finalized), finds a natural application to describing object protocols and has been recently converging to behavioral typing.

- Chapter 3 explores the integration of behavioral types within functional languages. Functional languages are relevant for their qualities of being easily endowed with high-level type-theoretic and concurrent extensions, for their natural support to parallelism, and since they permit rapid prototyping. We survey three different approaches, one akin to an effect system, one based on

explicit continuation passing, and one based on monads. Besides providing an out-of-the-box application of behavioral types to a concrete programming language, the continuation-based and monadic approaches can take advantage of the *type inference* engine of the language so that the programmer is not required to explicitly write (or annotate programs with) behavioral types, which can be automatically reconstructed from the source code of the program.

- High-performance computing often relies on parallel processes that synchronize by means of message passing. Chapter 4 describes the use of behavioral types in conjunction with Message Passing Interface (MPI) which is the *de facto* standard API for high-performance computing. Also in this case, behavioral types provide an effective means for making sure that communications occur without errors. We survey three alternative approaches making use of behavioral types in this context: one based on higher-level structuring abstractions, one based on source code verification, and one based on source code generation.

- Chapter 5 describes an application of behavioral types to multi-agent systems. The latter have been proved to be an industrial-strength technology for integrating and coordinating autonomous and heterogeneous entities in *open* systems. In this setting, the possibility of formally describing interaction protocols in the form of behavioral types enables forms of runtime monitoring for multi-agent systems.

- Chapter 6 provides an overview of the use of behavioral types in Singularity OS, a prototype Operating System developed by Microsoft that adopts communication as the fundamental and only synchronization mechanism between processes. Sing$^{\#}$, the programming language used for the implementation of Singularity OS, is an extension of C$^{\#}$ that comprises both object-oriented and functional constructs and provides a native notion of *channel contract*, a form of behavioral type. The formal investigation of behavioral types in this setting has led to the discovery of un-

forseen system configurations that yield memory leaks and to the development of refined behavioral type theories preventing them.

- The WSDL and UDDI standards are technologies currently enabling the description of Web Service interfaces and the creation of Web Service repositories. Chapter 7 explores the potential of behavioral types, intended as abstract descriptions of Web Service behaviors, as natural generalizations of WSDL interfaces to realize sophisticated forms discovering, composition, and orchestration of Web Services.

- Chapter 8 illustrates the *design-by-contract* methodology for the development of possibly distributed, communicating systems. According to this methodology, behavioral types are used for describing, from a vantage point of view, the topology of the communication network, the communications that are supposed to occur, and in which order. Such global specifications serve multiple purposes: they are a valuable form of abstract specification of the overall behavior of a distributed system; they can be projected for describing the local behavior of the network participants to allow the modular type checking of complex systems; they enable the generation of monitors to verify, at runtime, that the participants of a heterogeneous distributed system behave as expected, even if only some or none of them have been type checked against their supposed or claimed behavior.

Overall, the survey provides substantial evidence that behavioral types have sprinkled a remarkable interest in the research community concerned with programming languages. The adoption of behavioral types beside the academic context proceeds more slowly, but nonetheless there are encouraging signals. As a matter of fact, it is known that programming languages tend to evolve slowly, especially when it comes to the integration of sophisticated typing disciplines. In this respect, approaches that rely on the *encoding* of behavioral types using conventional type constructors (§3.3), that allow for the verification of existing code (§4.2), or the type-driven generation of runtime monitors (Chapters 5 and 8), enable developers to fill the gap between theory

and practice of behavioral typing with little or no changes to their programming environment and development workflow. The survey also contains pointers to industrial projects in which behavioral types already play a key role: the Ocean Observatories Initiative, which aims at the realization of a planetary-scale network for the trasmission of environmental data (§8.2), and the programming language Sing#, developed by Microsoft, which offers behavioral types as a native and key feature (Chapter 6). These early examples of industrial applications of behavioral types indirectly hint at their effectiveness in supporting the development of complex, large-scale systems for which correctness and reliability guarantees are of paramount importance.

2

Object-Oriented Languages

Research on behavioral types for object-oriented languages has developed along two parallel lines. The first line originated from the session types community, with the aim of transferring types of the form described in §1 from process calculi to mainstream programming languages. The second line is the evolution of the concept of typestate, which uses type-theoretic ideas to specify and check the order in which operations are applied to objects. Session types can be seen as a special case of typestate, in which the ordered operations are the sends and receives on communication channels.

This chapter is organized into three parts. §2.1 describes theoretical work that adds communication channels and session types to core object-oriented calculi. §2.2 describes work that, as well as establishing theory, implements session types in extensions of Java. Some of this work, for example the Mungo and Mool languages, establishes a general typestate setting in which communication operations are treated in the same way as other methods. The distinctive aspect of these languages, in relation to other work on typestate, is that the constraints on the order of method calls on an object are expressed by an automaton-like session instead of by separate pre- and post-conditions for the methods.

§2.3 describes the evolution of typestate in its general sense, independently of communication.

2.1 Session Types in Core Object-Oriented Languages

The wide adoption of object-oriented paradigm for writing modern applications is the reason that motivates the research efforts towards integrating session types and session-oriented programming with object-oriented programming.

Object-oriented programs based on communication are implemented using *sockets* or *remote method invocation* primitives (as Java RMI or C♯ remoting). The former approach uses an abstraction of an untyped communication channel, therefore a great amount of dynamic checks of types is needed to ensure type safety of the exchanged data. The latter approach has the advantages of a standard method invocation in a distributed environment, which requires a method to be used according to its signature, but it suffers lack of flexibility to describe patterns of interaction that provide bidirectional message exchanges from both communication parties, interleaved by local computations.

Sessions and session types are then a good answer to the limitations of the previous approaches, taking into account the aim of writing concurrent and distributed applications with a better and, consequently, more solid structure.

The integration of session types into the object-oriented paradigms can be pursued:

- *by extending* standard object-oriented languages with ad-hoc primitives for session-based communication, as in languages MOOSE [Dezani-Ciancaglini et al., 2005, 2006, 2009], MOOSE$_{<:}$ [Dezani-Ciancaglini et al., 2007], and AMOOSE [Coppo et al., 2007].

- *by amalgamating* standard object-oriented methods and sessions in a unique, more expressive, construct, as in languages STOOP [Drossopoulou et al., 2007], $\mathsf{SAM}^{\mathsf{g}}$ [Capecchi et al., 2009], and $\mathsf{SAM}^{\vee}$ [Bettini et al., 2008a, 2013].

```
class Customer {
  Address addr;
  double price;
  bool loop := true;
  void buy( String journeyPref, double maxPrice ) {
    connect c1 placeOrder {
      c1.sendWhile (loop) {
        c1.send( journeyPref );
        price := c1.receive;
        loop := evalOffer(journeyPref,price);
        /* implementation of evalOffer omitted */
      }
      c1.sendIf( price <= maxPrice ) {
        c1.send( addr );
        Date date := c1.receive;
      }{
        null; /* customer rejects price, end of protocol */
      }
    } /* End connect */
  } /* End method buy */
```

Figure 2.1: The class `Customer`.

2.1.1 Moose dialects

MOOSE (Multi-threaded Object-Oriented calculus with Sessions) is the result of the embedding session types into object-oriented languages. MOOSE is a multi-threaded language with session types, thread spawning, iterative and higher-order sessions. Its design aims at consistently integrating the object-oriented programming style and sessions, and to treat various case studies from the literature.

Simple Communications: Value Sending/Receiving. Two parties may start communicating, provided the types attached to that communication —i.e., the corresponding session types— are dual of each other. Then, the type system is able to ensure soundness, in the sense that two communicating partners are guaranteed to receive/send sequences of values following the order specified by their session types.

Let us consider the example of a travel purchase, taken from Hu et al. [2008], involving a customer, an agency, and a travel service. The code of the customer is as in Figure 2.1. The class `Customer` contains a method `buy` for the purchase of a travel ticket according to the parameters `journeyPreferences` and `maxPrice`. The body of the method contains a `connect` request, which specifies a channel `c1`, a session type `placeOrder` and a block of instructions to be executed whenever the connection will be established. The session type of the connection is

```
session placeOrder = begin.!<!String.?double>*.
                          !<!Address.?Date.end,end>
```

The code of a compatible agency is as in Figure 2.2. The class `Agency` contains a method `requestEq(Int m1, Int m2)`. When the method is called, the `connect` on the channel `c1` with session type `acceptOrder` is executed and if another object is trying to connect on the same channel `c1` with a dual session type, then the connection will be established. The session type of the connection is

```
session acceptOrder = begin.?<?String.!double>*.
                           ?<?Address.!Date.end,end>
```

Let us now consider the following program fragment, which illustrates a communication between a customer and an agency:

```
spawn(new Customer.buy("London to Paris",300));
new Agency.sell()
```

The `spawn` expression triggers the execution of its body into a new parallel thread, so that the program evolves to

```
new Agency.sell() || new Customer.buy("London to Paris",300)
```

that is the parallel composition of two threads, respectively modeling the agency and the customer.

At this point two new objects are created, which are instances of classes `Agency` and `Customer`, respectively, and the two methods can be invoked on the corresponding receiver object. Notice that the formal parameter `journeyPref` is now replaced with the actual parameter `"London to Paris"`:

```
class Agency {
  String journeyPref;
  void sell() {
    connect c1 acceptOrder {
      c1.receiveWhile {
        journeyPref := c1.receive;
        double price := getPrice( journeyPref );
        /* implementation of getPrice omitted */
        c1.send( price );
      }
      c1.receiveIf { // buyer accepts price
        JourneyDetails journeyDetails := new JourneyDetails();
        spawn {
          connect c2 delegateOrderSession {
            c2.send( journeyDetails );
            c2.sendS( c1 );
          }
        }
      }{ null; /* receiveIf : buyer rejects */ }
    } /* End connect */
  } /* End method sell */
}
```

Figure 2.2: The class `Agency`.

```
class Service {
  void delivery() {
    connect c2 receiveOrderSession {
      JourneyDetails journeyDetails := c2.receive;
      c2.receiveS( x ) {
        Address custAddress := x.receive;
        Date date := new Date();
        x.send( date );
      }
    } /* End connect */
  } /* End method delivery */
}
```

Figure 2.3: The class `Service`.

```
connect c1 placeOrder {
  ...
  c1.send( "London to Paris" );
  price := c1.receive; ···}
```

```
connect c1 acceptOrder {
  ...
  journeyPref := c1.receive;
  c1.send( price ); ···}
```

Now we have two parallel threads, both trying to connect on the same channel `c1`. Since they expose dual session types, namely they in complementary ways, the connection is allowed. For guaranteeing the privacy of the communication, after the connection is established, a new private channel is created and used for the actual communications, instead of the connection channel `c1`.

At this point the exchange of data can begin.

Choices. Choices in MOOSE are modeled with the constructs

- `c.sendIf(e){e1}{e2}` where first the boolean expression `e` is evaluated, then its result is sent through channel `c`. If the value of `e` is `true`, it continues with `e1`, otherwise with `e2`;
- `c.receiveIf{e1}{e2}` which receives a boolean value via channel `c`, and if it is `true` then it continues with `e1`, otherwise with `e2`.

Similar constructs are used to model iterations: `c.sendWhile(e){e1}` and `c.receiveWhile{e1}`, where the evaluation of `e1` is repeated as long as the value of `e` is `true`.

In Figures 2.1 and 2.2 we can see both these constructs in use. Notice that in the original protocol (*cf* Figure 1.1) the choice is modeled with a label-based construct, so that the two branches are labeled with specific ACCEPT and REJECT labels, and the `Customer` chooses one branch or the other by sending the corresponding label to the `Agency`, while in Figure 2.1 the `Customer` sends just a Boolean value. Further details on choices are provided in the **Peculiarities** paragraph below.

Delegation. Delegation, namely the act of communicating a channel as a message, works as in standard session types and it is modeled through the constructs:

- `c.sendS(c1)`: the channel `c1` is sent over `c`;

- `c.receiveS(x){e}`: a channel is received from `c` and bound to `x` within the expression `e`.

In Figure 2.2 the `Agency` connects to the `Service` through channel `c2`, exposing a session type:

```
session delegateOrderSession =
  begin.!String.!(?Address.!Date.end).end
```

The delegation of a portion of communication to be held on channel `c1` is performed by means of the command `c2.sendS(c1)`. The `Service` by exposing a dual session type:

```
session acceptOrderSession =
  begin.?String.?(?Address.!Date.end).end
```

is a suitable candidate for accepting the delegated session.

The semantics of delegation is as expected, except that, when the channel is exchanged, the receiver spawns a new thread to handle the consumption of the delegated session. This strategy is necessary in order to avoid deadlocks in the presence of circular paths of session delegation (see Dezani-Ciancaglini et al. [2009]).

Peculiarities. In MOOSE, sessions have been added to an object-oriented language in a way to be as close as possible to the original π-calculus based sessions. Therefore most of the features are mere adaptations of the corresponding ones in session-based process calculi. A few differences, however, can be noticed. First of all, choices are based on boolean values instead on labels, in order to be closer to the standard programming constructs and habits. This feature does not affect expressivity in any way, since it corresponds to a binary labeled choice: choices with several branches can be easily encoded as nested binary choices, at the cost of sending multiple boolean values instead of just one label.

Other differences are more technical and concern the use of channels for the connection and communication. In the original calculus of sessions [Yoshida and Vasconcelos, 2007], two parties connect over a public channel, thus the connection construct mentions the public name and a variable binding the new private name in the scope of the session. In

here there is no such variable. Instead, the new private name replaces the public name (e.g. `c1` in Figure 2.1) in the body of the connection.

Moreover, in MOOSE there is only one `connect` primitive, as opposed to the dual `accept`/`request` primitives of the original π-calculus with sessions [Yoshida and Vasconcelos, 2007]. In MOOSE the discrimination of the two endpoints is realized by exploiting the duality of the associated session types.

Delegation, i.e., the communication of channel names, does not get along easily with the structural essence of session types. If wrong configurations are allowed, then type preservation under reduction (a property also called "subject reduction") may fail [Yoshida and Vasconcelos, 2007]. Two main solutions have been adopted: one is based on the use of two different private names, identifying the two endpoints of the communication; another one is based on the use of just one private name with the restriction that α-conversion is implicitly performed ahead of communication. This language does not fall in any of the two cases, because only one name is actually created at runtime and it does substitute the public name provided in the program. The type system ensures type safety by preventing interleaved connections, so that dangerous configurations cannot occur.

A further aspect, peculiar to the MOOSE approach, is that a session must begin and end (or be delegated) within a method, in contrast with what happens in other approaches. (See, for instance, the "modular session types" approach of Gay et al. [2015], Kouzapas et al. [2015] described in §2.2.)

Bounded polymorphism. Pursuing the intent of studying the integration of session types into a mainstream object-oriented programming language, one cannot ignore the feature of genericity. The natural course was to study bounded polymorphism for object-oriented sessions, by following the steps of Gay [2008], where bounded polymorphism is included into π-calculus sessions.

Thus, in Dezani-Ciancaglini et al. [2007] the language MOOSE$_{<:}$ is presented, which extends MOOSE with an adapted notion of bounded polymorphism for session types, inspired by Gay [2008], as a means

of describing the behavior of processes that operate uniformly over all subtypes of a given type.

For instance, it is possible to express the behavior of a process that receives an object of a subclass of a given class and then sends back a value of the same subclass. For example, the following session type

$$?(\texttt{X} <: \texttt{Image}).!\texttt{X}$$

specifies the behavior of a process that receives an image in some format and sends back an image in the same format as the one received. This behavior matches the one of a process that sends a `JPG` photo and expects a `JPG` photo, or the one of the process that sends a `GIF` image and expects back a `GIF` image.

Therefore a refined notion of duality is needed to correctly deal with bounded polymorphism, that associates to each session type more than one dual type.

Progress. Progress is the property that every terminating computation stops by returning a value (i.e., a result), that is, no computation may end stuck or deadlocked before it is completed. The paper by Coppo et al. [2007] gives a type system assuring progress for AMoose, an asynchronous variant of Moose.

2.1.2 Stoop dialects

In both Moose and its variants, sessions were added to the object-oriented language as an orthogonal feature. However sessions and methods show related, though different, features and this observation suggests that both could be derived from a more general notion of session associated to an object. Drossopoulou et al. [2007] propose a language that *amalgamates* the notion of session-based communication with the one of object-oriented programming. The approach is called Stoop (Session Types and Object-Oriented Programming). Stoop is only a core programming language in the sense that it is only concerned with the amalgamation of the object-oriented and the session paradigms, but is agnostic about issues that concern synchronization, distribution, copying of values across local heaps *etc.*.

	"traditional" session	"traditional" method	"amalgamated" session/method
request on	a thread	an object	an object
execution starts when	threads reach certain point	immediately	immediately
executed body is	the rest of the thread	determined by the class of the receiver	determined by the class of the receiver
execution	concurrent	same thread	concurrent
communication	any direction interleaved with computations	n-inputs then computation then one output	any direction interleaved with computations

Table 2.1: "Traditional" sessions, "traditional" methods, and "amalgamated" sessions/methods.

Stoop drives the amalgamation quite far, by unifying sessions and methods, and by basing the choices of alternative paths in communications on the object being sent or received [Dezani-Ciancaglini et al., 2007] rather than on labels, as in traditional session types. Stoop includes a rather general form of delegation, even if sessions are not higher order, that is, sessions cannot communicate sessions, a common feature in many session calculi.

The rationale of the method-session amalgamation The fundamental idea at the basis of the Stoop language is to amalgamate sessions and methods in one construct and it arises mostly from two observations:

1. sessions and methods share similar features; and
2. the integration of sessions and methods reflects well the intuition of a service.

This amalgamation comes out to be a very natural approach in an object-oriented setting: the abstractions provided by the object-oriented paradigm are much more intuitive than the ones of other

paradigms, supporting a more direct translation of the real problems that have to be resolved into the correspondent models. The running programs are objects, that come to life in a virtual world where they actively participate to a common goal, where each party carries out its own tasks, cooperating with the others through a reciprocal message exchange, that is exactly the essence of the object-oriented computation.

The notion of communication is already implied in the object-oriented paradigm and, from this point of view, sessions do not introduce any innovation: the immediate encoding of methods through sessions, that could be seen simply as a generalization of methods, will be a confirmation of this.

In Table 2.1, we compare "traditional" sessions and "traditional" methods from object-oriented languages with the "amalgamated" session/methods of Stoop.

Sessions are invoked on threads in a manner similar to Ada's rendezvous, and execution starts when two threads reach a certain point in their execution, where they can "serve" the session. The computation proceeds by executing in parallel the code of both threads. Sessions allow communication of any number of objects in any direction.

On the other hand, methods are invoked on an object, the body to run is defined in the class of the receiving object, execution is immediate and sequential, and it supports any number of inputs, followed by computation, followed by one output.

Stoop proposes "amalgamated" sessions/methods, which, for brevity, we shall call sessions from now on. Invocation takes place on an object, for instance a customer asks to withdraw money from a particular ATM machine, and execution of the corresponding session takes place immediately and concurrently with the requesting thread. The body is defined in the class of the receiving object, for instance the body of the withdraw session is defined in the ATM class, and any number of communications interleaved with computation is possible. Moreover no explicit mention of communication channels is required at source code level.

```
class Customer {
  Address addr;
  double price, maxPrice;
  bool loop := true;
  String journeyPref;
  new Agency.sell {
    sendWhile (loop) {
      send( journeyPref );
      price := receive;
      loop := evalOffer(journeyPref,price);
      // implementation of evalOffer omitted
    };
    sendCase( evalPrice(price,maxPrice) ) {
      ACCEPT ▷ send( addr ); Date date := receive;
      REJECT ▷ null; /* customer rejects price,
                                    end of protocol */ }
  } /* End method invocation */
```

Figure 2.4: The class `Customer`.

```
class Agency {
  String journeyPref;
  void acceptOrder sell {
    receiveWhile {
      journeyPref := receive;
      double price := getPrice( journeyPref );
      // implementation of getPrice omitted
      send( price );
    }
    receiveCase (x) { // buyer accepts price
      ACCEPT ▷ new Service • orderDelivery { } ,
      REJECT ▷ null;/* receiveCase : buyer rejects */ }
  } /* End method sell */
}
```

Figure 2.5: The class `Agency`.

```
class Service {
  void receiveOrderSession orderDelivery() {
    Address custAddress := receive;
    Date date := new Date();
    send( date );
  }
}
```

Figure 2.6: The class `Service`.

Simple Communications: Value Sending/Receiving Let us see how the ticket purchase example can be written in Stoop. The code of the `Customer` is listed in Figure 2.4. Notice that the code does not specify any channel. In Stoop all the channels are private and created at runtime.

In Stoop session invocations have a body that will be executed in parallel with the body of the session requested. The two bodies must have dual session types. This is checked at compile time, not at runtime as in Moose.

To understand how communication works in Stoop, let us consider the following invocation:

```
new Agency.sell {
  ...
  send( "London to Paris" );
  price := receive;
  ...
};
```

A new object of class `Agency` is created and a new thread is spawned in order to execute the body of the session. A pair of new private channels `k` and $\tilde{\texttt{k}}$ is created, which correspond to the two end points of the same private channel: they have dual session types. Every communication expression (`send` and `receive`) is now prefixed with a new channel, so that the program reduces to a configuration consisting of the following two threads running in parallel:

```
...
k.send( "London to Paris" );
price := k.receive;
...
```

```
...
journeyPref := k̃.receive;
...
k̃.send( price );
...
```

At the same time, two initially empty queues are created in the heap and associated with the two channels. The queues store messages that have been communicated but not yet read by the receiving party.

The communication begins with `k.send( "London to Paris")` and the value is put in the queue associated to the channel $\tilde{k}$, dual of the channel the value was sent over.

The communication is asynchronous, so the value may not be read right away by the partner. The residual program configuration becomes:

```
price := k.receive;
...
```

```
...
journeyPref := k̃.receive;
...
k̃.send( price );
...
```

Now the result can be read from the queue and stored into the field `journeyPref`. The queues associated to the channels ensure that messages are read in the same order they were sent.

Choices Choices are dealt with constructs similar to the one in MOOSE, except that the choice is based on the class of the exchanged object:

- $\texttt{sendCase}(\texttt{e})\{\texttt{C}_i \triangleright \texttt{e}_i\}_{i\in I}$ first evaluates expression `e` to an object and, depending the class $\texttt{C}_i$ of such object, proceeds as $\texttt{e}_i$;
- $\texttt{receiveCase}(\texttt{x})\{\texttt{C}_i \triangleright \texttt{e}_i\}_{i\in I}$ receives an object and continues with $\texttt{e}_i$ if the class of the object is $\texttt{C}_i$.

The type system guarantees that the class of the exchanged object is one of the $\texttt{C}_i$.

Similar constructs of the form $\texttt{sendWhile}(\texttt{e})\{\texttt{C}_i \triangleright \texttt{e}_i\}_{i\in I}$ and $\texttt{receiveWhile}(\texttt{x})\{\texttt{C}_i \triangleright \texttt{e}_i\}_{i\in I}$ are used to model iterations. In these constructs, a special variable `cont` occurring in some of the $\texttt{e}_i$ causes the iteration to start anew.

Delegation. The delegation in STOOP works in a quite different way than in traditional languages with sessions. It is not modeled by the exchange of a private channel, instead it uses a new construct:

$$\texttt{e•s\{ \}},$$

that means that the delegating object requests the session `s` in the class of the expression `e` to take temporarily the control of the communication with its partner.

We can see an instance of delegation in Figure 2.5, where the session is being delegated to a new `Service`: in the class in Figure 2.5 we see a delegation request to the session `offerDelivery` of `Service`.

That code is executed if the `Customer` accepts the purchase (sending to the `Agency` an object of class `ACCEPT`). Then the corresponding branch is selected and the runtime configuration is as follows (on the right we have the code of the `Customer` and on left the one of the `Agency`):

```
k.send( addr );                  ||  new Service•orderDelivery { }
Date date := k.receive;          ||
```

The `Customer` sends its address on channel `k` but the Agency is not set up to receive it. Instead, it delegates that part of the communication to a delivery `Service`. A new object is created, and the code of the session `orderDelivery` is retrieved and all the send/receive instructions are decorated with the private channel of the delegator object, namely channel $\tilde{\texttt{k}}$.

```
                                 ||  Address custAddress := k̃.receive;
Date date := k.receive;          ||  Date date := new Date();
                                 ||  k̃.send( date );
```

After the delegated code is executed, the control returns back to the delegator object, and the communication goes on as expected between the two original partners. In this case no further communication is expected and the session ends.

Union types. Choices based on the exchanged object, which are peculiar to the STOOP approach, have the advantage to be more object

oriented, thus the choice may be better integrated within the program. However, in many cases the particular object needed for the selection has to be expressly created for the purpose of the choice, thus treating objects as mere labels. With union types it is possible to express communications between parties which manipulate heterogeneous objects just by sending and receiving objects which belong to subclasses of one of the classes in the union. In this way the flexibility of object-oriented depth-subtyping is enhanced, by strongly improving the expressiveness of choices based on the classes of sent/received objects. In Bettini et al. [2008a, 2013] the use of union types for session-centered communications is formalized for $\mathsf{SAM}^{\vee}$, a core object-oriented language based on the Stoop approach.

Union types represent the least common supertype of all the types $\mathtt{T}_i$ forming the union $\bigvee_{i \in I} \mathtt{T}$. In object-oriented programming this is a useful way to enhance subtyping beyond the inheritance relation: two classes used in similar contexts, but placed apart in the class hierarchy, can have a common meaningful supertype. For example, let us consider the session type

$$!(\mathtt{NoMoney} \vee \mathtt{OK})$$

that describes the behavior of a process representing a bank that answers `yes` or `no` to a seller that wants to check the money availability of a client—where `yes` and `no` are objects of classes `OK` and `NoMoney`, respectively. Without union types, a superclass of both `OK` and `NoMoney` would be required: this superclass would allow the sending of objects of unrelated classes w.r.t `OK` and `NoMoney`.

Generic types. Analogously to the work done for Moose, also in Stoop the integration of polymorphism and session types has been studied. In Capecchi et al. [2009] the adoption of generic types for session-centered communications is formalized for $\mathsf{SAM}^{\mathtt{g}}$, a core object-oriented language based on the Stoop approach.

The use of generic types allows the code to be typed "generically", using variables instead of actual types, guaranteeing uniform behavior on a range of types. In an object-oriented language this corresponds to having parameterized classes and methods.

For instance, let reconsider the bounded polymorphism example shown before. We had a service with a behavior ?(X <: Image).!X that could interact with a client behaving as prescribed by !JPG.?JPG or with a client following the protocol represented by the type !GIF.?GIF. At the language level this means having two different classes of clients JPGcustomer and GIFcustomer. In the language with generic types we can implement this two clients with a single parameterized class Customer⟨X extends Image⟩, and then instantiate two objects of class Customer⟨JPG⟩ and class Customer⟨GIF⟩.

2.2 Behavioral Types in Java-like Languages

In this section we review attempts to integrate a form of behavioral types to Java or a Java-like language, usually using a couple of syntax extensions (typically to declare protocols) and some specific typing rules which are used to check behavior conformance. The features shared by all of them are: only objects of some specific classes are controlled by the behavioral type system; aliasing is disallowed for these objects; behavioral type-checking can in principle be implemented as a first pass before the file is passed to a regular Java compiler; syntactic extensions are either translated or erased after this pass.

In terms of actual implementation, most of the works presented here have only had a one-shot proof-of-concept implementation. SessionJ [SJ] is the largest project and was developed over several years.

These works draw from two different pre-existing lines of research: session types for channel-based communication, and type systems for non-uniform objects.

SessionJ [Hu et al., 2008, 2010, Ng et al., 2011, Alves et al., 2010] is based on the Moose calculus described in §2.1, with adaptations to Java and additional features; in this language, session channels are objects of one specific class. Usage of these objects is strictly controlled whereas objects of other classes are treated as in plain Java.

On the other side, Yak [Militão, Militão and Caires, 2009] allows adding a usage protocol to any class, but has no specific construct for channels or concurrency.

Mungo [Gay et al., 2015, Kouzapas et al., 2015] seeks to integrate both approaches: all classes can have specified usage protocols, and communication channels are objects of one specific class. The usage protocol for the channel class is not fixed: instances of that class are created by initializing a communication session, and their initial usage protocol is determined from the associated session type.

MOOL [Campos and Vasconcelos, 2010, Campos and Vasconcelos] has usage protocols and concurrency, but no explicit channels: message-passing between threads is done through regular method calls.

We now review SessionJ, Mungo and Mool in more detail. Yak's type system is very similar to Mungo's although the syntax is different; the main additional feature it has is handling of exceptions in the protocols. Mool uses essentially the same types for objects as Mungo, with the addition of qualifiers to control aliasing. SessionJ and Mungo are implemented as extensions of the Java language that essentially add protocol (in the form of session types) declarations. The new declarations have a profound impact on the Java type checking system, making it difficult to use the standard Java annotations.

2.2.1 Session Java

Session Java (SJ) is a Java extension to support session-typed channels, developed at Imperial College mainly by Raymond Hu. Several versions have been released with increasingly many features, described in several papers by Hu et al. [2008], Hu et al. [2010], Ng et al. [2011], Alves et al. [2010].

SJ is an implementation of the core programming language MOOSE described in §2.1.1, adapted to be an extension of Java. The syntax of SJ is very slightly different from that of MOOSE; the main difference is that SJ allows labeled branching and not just boolean branching. Thus, instead of MOOSE's `sendIf`/`receiveIf`, there is a construct of the form `outbranch(LABEL) {body}` to select a particular label and then execute the body part on the selecting side, and a `inbranch`/`case` construct on the receiving side. The `outbranch` construct is typically combined with a regular Java `if`, a switch/case, or a combination of the two, and there is no requirement that all choices allowed by the session type are

effectively present. The important part for the type system is that in every `c.outbranch(LABEL) {body}` call which appears, `body` uses channel `c` in conformance with the session type associated with label `LABEL`.

In SJ, a communication channel endpoint is an object of class `SJSocket`, which is implemented on top of a TCP socket. Thus, on the client side, this object is created by connecting to a specific host and port; on the server side, it is created by listening to a specific port. If we consider the class Customer of Figure 2.1, its method `buy` will start this way in SJ instead of the line `connect c1 placeOrder {` :

```
SJServerAddress agency =
  SJServerAddress.create(placeOrder, host, port);
SJSocket c1 = SJSocketImpl.create(agency);
c1.request();
```

(where `placeOrder` is the session type). Here `c1` is seen as a session-typed channel by the SJ system, which will statically check its correct usage throughout the method body before compilation; it is seen as an object of class `SJSocket` by the Java compiler and runtime. Note that `c1` *must* be a local variable of the `buy` method: in Moose and SJ, a session-typed channel cannot escape the method in which it is created except by being *delegated*. Delegation is implemented transparently in SJ: the `SJSocket` object embodying the session-typed channel is passed as argument to the `send` method of another one. There is no need to distinguish `send` and `sendS` thanks to overloading.

As an example of all this, in SJ the `sell()` method of class `Agency` (Figure 2.2) looks like this:

```
void sell() {
  SJServerSocket ss =
    SJServerSocketImpl.create(acceptOrder, port);
  SJSocket c1 = ss.accept();
  c1.inwhile() {
    String journeyDetails = s.receive();
    // calculate the price
    s.send(price);
  }
  c1.inbranch() {
    case ACCEPT: {
      SJServerAddress service =
```

```
      SJServerAddress.create(delegateOrderSession, host, port);
    SJSocket c2 = SJSocketImpl.create(service);
    c2.request();
    c2.send(c1);
  }
  case REJECT: {}
}
```

Note that it is also possible for a session channel to be passed as an argument to a regular method call, so that the remainder of the session is delegated to another local object rather than to another site. A method expecting a channel endpoint as argument declares the expected session type as its argument type (rather than `SJSocket`).

Additional features. In addition to implementing MOOSE, SJ implements several protocols for session delegation, each of which has advantages and drawbacks depending on the network configuration, and allows choosing which one to use.

Furthermore, the latest version of the language, called ESJ, combines all the features discussed above with event-driven programming.

2.2.2 Modular session types for objects / Mungo

Gay et al. [2015] present a clean incorporation of session types in a Java-like language, where sessions control the order in which methods are called, but also the choices imposed on clients by virtue of values returned by methods. Type checking is strictly static and sessions do not exist in the semantics, thus in practice this can be implemented by a verification pass on annotated Java source code just before compilation. Kouzapas et al. [2015] implement these ideas in a language/tool called Mungo and an associated tool called StMungo which translates channel session types, expressed in the Scribble language [Honda et al., 2011], into Mungo object sessions. Mungo is closely based on [Gay et al., 2015], using a nominal type system.

The aim of this approach to object-oriented session types is modularity. This means that the implementation of a session on a communication channel can be separated into several methods, each of which

expects the channel to be in a certain state (corresponding to a session type) and leaves the channel in another state (corresponding to a later point in the session type). It follows that such methods cannot be called in arbitrary orders, and this leads to the concept of a session or protocol for an object. This generalises the approach of SJ, in which a session on a channel must be completely implemented within a single object. In Mungo, channels and their session types are not native. Instead, a channel session type is translated into an object protocol that specifies the allowed sequence of calls of send/receive methods on an object that encapsulates a channel endpoint. This translation is done by a separate tool, StMungo.

The combination of Mungo and StMungo can be seen as a unification of channel session types with the more general concept of typestate (§2.3), in which method availability depends on the state of an object. The distinctive feature of Mungo's approach to typestate is the use of session-type-like syntax for constraints on sequences of method calls, instead of explicit pre- and post-conditions on methods.

In the following, we first show a straightforward adaptation of the running example, keeping the same structure as for SJ/Moose. We then show how a class can divide the implementation of the protocol between several methods.

The translation of channel session types into object session types is only defined, in the paper, for a language of session types which does not include the while construct (![]* and ?{}*) of SJ and Moose. It is however possible to encode this while construct into session types with branching and recursion:

```
acceptOrder =
  &{QUERY: ?[String].![Double].acceptOrder;
    ACCEPT: ?[Address].![Date].end;
    REJECT: end}
```

In the Mungo/StMungo methodology, the starting point is a Scribble global protocol describing the sequences of allowed interactions between the involved roles (in the example, Agency and Customer):

```
global protocol Travel(role Agency, role Customer) {
  rec Loop {
    choice at Customer {
```

```
      QUERY() from Customer to Agency;
      journey(String) from Customer to Agency;
      price(Double) from Agency to Customer;
      continue Loop;
    } or {
      ACCEPT() from Customer to Agency;
      address(Address) from Customer to Agency;
      date(Date) from Agency to Customer;
    } or {
      REJECT() from Customer to Agency;
    }
  }
}
```

The global protocol is projected to give a local protocol for each role. First is the local protocol for Customer, which corresponds to the session type `acceptOrder`.

```
local protocol Travel(self Agency, role Customer) {
  rec Loop {
    choice at Customer {
      QUERY() from Customer;
      journey(String) from Customer;
      price(Double) to Customer;
      continue Loop;
    } or {
      ACCEPT() from Customer;
      address(Address) from Customer;
      date(Date) to Customer;
    } or {
      REJECT() from Customer;
    }
  }
}
```

StMungo generates an enumerated type definition to represent the choice labels:

```
enum ChoiceLabel { QUERY, ACCEPT, REJECT }
```

and translates the local protocol into the following object session type, which will be the type of the channel endpoint object once the connection is established. The method names have been simplified here; in practice they also contain the name of the role that sends the message.

Also, Scribble allows data to be associated with a choice label, meaning that the payload of the next message can be incorporated into the choice message; for simplicity we do not take advantage of this feature here.

```
AcceptOrderSession = {
ChoiceLabel receiveChoice() :
  <
    QUERY: {String receiveString() :
      {void sendDouble(Double) : AcceptOrderEndpoint}};
    ACCEPT: {Address receiveAddress() :
      {void sendDate(Date) : {}}};
    REJECT: {}
  >
}
```

Mungo extends the syntax of Java to link a class to a session:

```
class AcceptOrderEndpoint typestate AcceptOrderSession { ... }
```

and the file `AcceptOrderSession.mungo` contains the definition of `AcceptOrderSession` above.

The curly braces in `AcceptOrderSession` indicate the set of methods which can be called at a given point when using the object. The angle brackets indicate a choice point where the client must examine the value returned by the last method call in order to know how to continue using the object.

`AcceptOrderSession` is the type the channel object will have when first created, and states that the only method initially available (there is only one thing in the outermost curly braces) is `receiveChoice()`. This method has no arguments and its return type is `ChoiceLabel` (in the theoretical calculus [Gay et al., 2015], the type is `linkthis`) which means the return value will be a label indicating how to continue using the object. After the colon, we have, between angle brackets, the possible return values, each of them associated with a session type. If the returned value is QUERY then the only method call available is `receiveString()`, which will return a `String`, after which a call to `sendDouble` will be possible, with an argument of type `Double`; this call will return nothing and the object will get back to session type `AcceptOrderEndpoint`.

If the returned value is ACCEPT, then method `receiveAddress()` is available. If the return value is REJECT, no method can be called anymore.

The dual channel session type, on the customer side, is:

```
placeOrder =
  +{QUERY: ![String].?[Double].placeOrder;
    ACCEPT: ![Address].?[Date];
    REJECT: end}
```

and this is expressed in Scribble as a local protocol, also obtained from the global protocol by projection.

```
local protocol Travel(role Agency, self Customer) {
  rec Loop {
    choice at Customer {
      QUERY() to Agency;
      journey(String) to Agency;
      price(Double) from Agency;
      continue Loop;
    } or {
      ACCEPT() to Agency;
      address(Address) to Agency;
      date(Date) from Agency;
    } or {
      REJECT() to Agency;
    }
  }
}
```

This is translated into an object session type as follows (note that the duality between the two endpoint types is not obviously visible anymore), and linked to a class `PlaceOrderEndpoint` as before.

```
PlaceOrderSession = {
  void sendQUERY():
    {void sendString(String):
      {Double receiveDouble(): PlaceOrderEndpoint}};
  void sendACCEPT():
    {void sendAddress(Address): {Date receiveDate(): {}}};
  void sendREJECT(): {}
}
```

The code of the client, without taking advantage of modularity, could look like:

```
PlaceOrderEndpoint c = agencyAccesspoint.request();
boolean decided = false;
while(!decided) {
  c.sendQUERY();
  c.sendString(journeyDetails);
  Double cost = c.receiveDouble();
  //set decided to true or change details and retry
}
if (want to place an order) {
  c.sendACCEPT();
  c.sendAddress(address);
  Date dispatchDate = c.receiveDate();
} else {
  c.sendREJECT();
}
```

where `agencyAccesspoint` is a wrapper around a socket connection.

The other endpoint would be obtained by calling `accept()` on the same access point, and would have the translation of the channel session type itself (not its dual), that is, `AcceptOrderSession`.

Delegation. The language described in the paper allows delegation, like Moose and SJ. However, delegation is not implemented in Mungo; this would require integrating the SJ runtime system for mobile socket connections. We show how the code would be written nevertheless, using the structural type system defined by Gay et al. [2015]. For brevity we work with channel session types instead of Scribble protocol definitions, and simplify the method names further to `send` and `receive`.

Let us suppose a global access point `serviceAccessPoint` has been declared with channel session type `?[?[Address].![Date]]`. This channel session type gives the translation:

```
ReceiveOrderSession = {
  {Address receive(): {Null send(Date): {} } } receive(): {}
}
```

for the `accept` side: only `receive()` can be called which returns an object with type `{Address receive(): {Null send(Date): {} } }`.

For the `request` side, the translation of the dual is:

```
DelegateOrderSession = {
  Null send({Address receive(): {Null send(Date): {} } }): {}
}
```

The code of the agency could look like:

```
AcceptOrderEndpoint s_ac = agencyAccesspoint.accept();
switch(s_ac.receive()) {
  QUERY:
    String journeyDetails = s_ac.receive();
    // calculate the price
    s_ac.send(price);
  ACCEPT:
    DelegateOrderSession s_as = serviceAccessPoint.request();
   s_as.send(s_ac);
  REJECT: null;
}
```

and the code of the service:

```
ReceiveOrderSession s_sc = serviceAccessPoint.accept();
Address custAddr = s_sc.receive();
s_sc.send(dispatchDate);
```

Modularity. Up to now, the code written is very close to the SJ and MOOSE examples, despite the type system being different. We now outline the definition of a class `CustomerWrapper` which provides a different interface to the operations of `PlaceOrderEndpoint`. This demonstrates the modularity supported by Mungo.

```
class CustomerWrapper {
// Session declaration
  session
    Init = {
      void connect(AgencyAccessPoint):
        {Double getPrice(String): S}
    }
    S = {
      Double getPrice(String): S;
      Date placeOrder(Address): Init;
      void cancel(): Init;
    }
```

```
  PlaceOrderEndpoint c;

// Method definitions
  void connect(AgencyAccessPoint agency) {
    c = agency.request();
  }
  Double getPrice(String journeyDetails) {
    c.sendQUERY();
    c.send(journeyDetails);
    return(c.receive());
  }
  Date placeOrder(Address address) {
    c.sendACCEPT();
    c.send(address);
    return(c.receive());
  }
  void cancel() {
    c.sendREJECT();
  }
}
```

The system works by taking the session type of the whole class and inferring from the method bodies the types the fields will have after each method call. Here the only field initially has a null type because it is not initialized. Then the first method called must be `connect` with an argument type of `AgencyAccessPoint`. The body of `connect` is typable in this context and changes the type of the field to the initial state of `PlaceOrderSession`.

Then the session type of `CustomerWrapper` says that the next method to be called will always be `getPrice` with an argument of type `String`. The body of `getPrice` is typechecked knowing that the `s_ca` field has type `PlaceOrderEndpoint` before the call and the type it gets to afterwards is inferred. This form of checking continues until the session type of `CustomerWrapper` terminates or loops.

2.2.3 Mool

With the definition of Mool, Campos and Vasconcelos [2010] extend modular session types for objects in two ways. (1) Mungo treats com-

munication channels shared by different threads as objects, by hiding channel primitive operations in an API from where clients can call methods. Mool eliminates channels in a programming language that relies on a simpler communication model—message passing in the form of method calls, both in sequential and concurrent settings. (2) Mungo deals with linear annotated classes only. Mool deals with linear types as well as shared ones, treating them in a unified framework.

Classes written in Mool are annotated with a usage descriptor that structures method invocation, enhanced by lin/un qualifiers for aliasing control. Mool defines a single category for objects that may evolve from a linear status into an unrestricted (or shared) one.

Example Taking advantage of objects, the Customer-Agency protocol can be split over several classes and methods. The interaction takes place through an object of type `Order` that the `Agency` sets up and returns to the `Customer`:

```
class Order {
  usage lin init; Sale
  where Sale = lin getPrice; lin{accept; end + reject; end};
  // the class fields
  Service service; String journeyDetails; double price;
  unit init(Service service, String journeyDetails,
            double price) {
    ... // sets field values
  }
  double getPrice() {
    price;
  }
  Date accept(Address address) {
    service.dispatch(journeyDetails, price, address);
  }
  unit reject() {
    // clean up and end the protocol
    unit;
  }
}
```

The usage type of class `Order` defines a sequential composition of available methods, starting with the linear "constructor" method

`init()`, and including a choice (given by `+`) for the caller (a `Customer` instance) to accept or reject a journey based on its price. In either case, the protocol is finished, `end` being an abbreviation for an unrestricted empty set of methods. The type system provides crucial information to object deallocation, enforcing that the type of an `Order` object is consumed to the end.

Class `Customer` receives an object of type `Order[Sale]` when querying the agency for the journey in method `acceptOrder()`. The type says that the object has advanced to a state where `getPrice` is the next available method, that is, `Order[Sale]` abbreviates `lin getPrice; lin{accept; end + reject; end}`.

```
class Customer {
  usage lin init; Order
  where Order = lin acceptOrder; <getDate; end + Order>;
  Agency[Order] agency; Date date;
  unit init(Agency[Order] agency) {
    this.agency = agency;
  }
  boolean acceptOrder(String journeyDetails, double maxPrice) {
    Order[Sale] order = agency.placeOrder(journeyDetails);
    if(order.getPrice() <= maxPrice) {
      date = order.accept();
      true; // return true
    } else {
      order.reject();
      false; // return false
    }
  }
  Date getDate() {
    date; // return date
  }
}
```

The usage type defined in class `Customer` allows an unlimited number of attempts to book journeys. `<getDate; end + Order>` denotes a variant type, indexed by the boolean values returned by method `acceptOrder()`. A caller of this method should test the result of the call: if true is returned the journey was booked, and the next available method is `getDate()`, otherwise the interaction can be repeated.

The type for the class guarantees that `getDate()` always returns an initialized value (set by method `acceptOrder()`).

Finally, the `Agency` usage type makes available the linear "constructor" method `init()` that sets up the service, after which the usage defines a new state `Agency[Order]` that abbreviates the recursive branch type given by `*placeOrder`. In turn, `*placeOrder` abbreviates type `T` such that `T = un placeOrder; T`. In state `Agency[Order]`, an `Agency` instance can be shared by an unrestricted number of customers. `Order` objects are then returned to `Customer` objects to establish the interaction.

```
class Agency {
  usage lin init; Order
  where Order = *placeOrder;
  // the only field
  Service service;
  init(Service service) {
    this.service = service;
  }
  Order[Sale] placeOrder(String journeyDetails) {
    Order order = new Order();
    order.init(service, journeyDetails,
               getPrice(journeyDetails));
    order; // return order
  }
  double getPrice(String journeyDetails) {
    ... // implementation omitted
  }
}
```

2.3 Typestate

Whereas the type of an object specifies all operations that can be performed on the object, typestates identify subsets of these operations that can be performed on the object in particular abstract states. When an operation is applied to the object, the typestate of the object may change, thereby dynamically changing the object's set of permitted operations. A *typestate pre-condition* must hold for an operation to be applicable, and a *typestate post-condition* reflects the possible types-

tates after the operation has been applied. Typestates were introduced by Strom and Yemini [1986], who applied typestates as abstractions over the states of data structures to control the initialization of variables (with the two typestates "uninitialized" and "initialized") and defined a static checker for typestates in this context. Another simple example is that of an object representing a file. The file must be opened before the "read" operation can be applied. The "close" operation can be used at any time after opening, but when the file has been closed, the only available operation is to open it again.

Strom and Yemini observe that although unrestricted aliasing and concurrency make the static checking of typestates impossible, it is still possible to apply static checking for controlled concurrency and dynamic process creation. Much work in the area of typestate concerns the development of static type systems for alias control.

Fähndrich and DeLine [2002] developed Vault, a typestate-checking system for an extension of C. Their alias control system is based on an association between "keys" and "tracked resources". A resource with a typestate specification must be tracked, which means that the type system attaches (via an existential type) a unique key to it. Aliases for the resource can be created freely, but applying a typestate-changing operation to a resource also requires the correct key; thus the problem of alias control is transferred to the keys. A system called "adoption and focus" allows temporary aliases to be created within a local scope and checks that they are destroyed by the end of the scope. Keys exist only in the static type system, and are not required at runtime.

Later work by DeLine and Fähndrich [2004] developed Fugue, a modular verification system for specifying and statically checking typestate properties for .NET programs. Its innovation was to adapt Strom and Yemini's typestates to object-oriented programs. In Fugue, a typestate is an abstraction over the concrete state of an object; it is specified by a predicate defined over the fields of the object. This approach has two main challenges: (1) the actual definition of a typestate depends on the subclass relation, and (2) the typestates must be uniform. These challenges are solved in Fugue by frame typestates, which define the property corresponding to a typestate for each subclass,

and by sliding methods, which ensure that subclasses override methods of superclasses which change the typestate, such that the change also applies in the subclass. To address aliasing, Fugue uses the adoption and focus model and distinguish two modes for object references: `NotAliased` and `MayBeAliased`. References which are `NotAliased` may become `MayBeAliased` and the typestate of `MayBeAliased` objects cannot change.

The next stage of development of typestate systems was Plural [Bierhoff and Aldrich, 2007, Beckman et al., 2008]. It is an extension of Java, implemented as an Eclipse plug-in. For alias control it uses a system of fractional permissions [Boyland, 2003, 2013] which allows a single reference to an object, with a certain access permission, to be split into fractions which are later recombined to restore the full permission. Plural also supports concurrency by means of synchronization and atomic blocks, and allows typestate specifications to be defined over collections of interacting objects.

Moving on from Plural, Aldrich et al. [2009] proposed typestate-oriented programming, with the goal of integrating typestates directly into a new language design instead of adding typestates to the features of existing languages. They argue that this approach leads to a cleaner language design and ultimately to better code. Plaid is an object-oriented language following this approach, developed by Sunshine et al. [2011a,b]. In contrast to Fugue and Plural, the typestates in Plaid are not predicates over concrete states. Typestates are declared in a way which is very similar to classes. Different typestates in Plaid may have the same values for the fields of the concrete state, but the fields of different typestates need not be the same. An object can change its typestate by means of an assignment, written `this <- NewTypestate(...)`. This can be seen as a dynamic constructor which replaces the current object by an instance of `NewTypestate` (the arguments to the constructor are used to initialize the declared fields of `NewTypestate`). To implement the Customer/Agency example, a `Customer` could have three different substates, reflecting if it is in the process of `Ordering` from an `Agency a`, if it is `Accepting` an offer from the `Agency` or if it is `Rejecting` all offers. A Plaid implementation of

such a `Customer` is given below, using the syntax of Aldrich et al. [2009] (in more recent papers, the syntax is slightly different):

```
state Customer {
  Agency a;
}

state Ordering extends Customer {

  void init () [ Ordering >> (Accepting || Rejecting) ] {
    Double d = getPrice("string"); ... ;
    if (good_offer("string",d)){
       this<-Accepting(a,"string"); a.accept("string")
    } else { this<-Rejecting; a.reject(); }
  }
}

state Accepting extends Customer {
  String s;
  ...
}

state Rejecting extends Customer {...}
```

If the client is in typestate `Ordering`, it can start the session by calling the `init` method. Note that the internal choice is modeled by a conditional and that the successful `"string"` argument is passed to the `Accepting` state, to initialize the field `s` which makes sense in this state only. The annotation `Ordering >> (Accepting || Rejecting)` next to the `init` method records the fact that the method can be invoked only when `Customer` is in typestate `Ordering` and that after the invocation the typestate may have changed to either `Accepting` or `Rejecting`, depending on the internal choice of `Customer`. This information is used by the type checker to verify that methods are invoked accordingly to the protocol of the receiver object.

Analogously, the `Agency` could consist of the typestates `OrderSession`, `Accept`, and `Reject`.

```
state Agency {
  Service service;
}
```

```
state OrderSession extends Agency {
  Double getPrice(String s){...} // calculate the price

  Date accept(Address a) [OrderSession >> Accept] {
    this<-Accept;
    Session s = service.createSession();
    return s.deliveryAddress(a);
  }

  Void reject() [OrderSession >> Reject] {
    this<-Reject;
  }
}

state Accept extends Agency {
  Double getPrice(String s) [Accept >> OrderSession] {
    this<-OrderSession; ... // calculate the price
  }
}

state Reject extends Agency {
  Double getPrice(String s) [Reject >> OrderSession] {
    this<-OrderSession; ... // calculate the price
  }
}
```

Here, we see how external choice is captured by different methods. Since the object can accept any number of `getPrice` calls when it is in the typestate `OrderSession`, this method does not change the typestate. By introducing the typestates `Accept` and `Reject`, calls to the methods `accept` and `reject` must be interleaved with calls to `getPrice`. To create a new `Session`, the `Service` creates an instance of typestate `Session` which accepts calls to the method `deliveryAddress`. Thus the `accept` method of typestate `OrderSession` can delegate to the `Session` object in a standard way.

```
state Service {
  Session createSession(){return new Session();}
}

state Session {
```

```
  Date deliveryAddress(String s){...}
}
```

It is also possible to specify that a method changes the typestate of one of its parameters, by means of annotations similar to those associated with `Customer` and `Agency` classes described above. For example, below method `m` calls the `reject` method on its parameter, resulting in a typestate change.

```
void m(OrderSession >> Reject agency) {
  agency.reject();
}
```

2.4 Related Work

The most recent work on typestate-oriented programming [Garcia et al., 2014] describes the system of access permissions to control aliasing. References with `full` permission have exclusive write access, references with `shared` permission have write access, and references with `pure` permission have read-only access. This version of the theory also integrates gradual typing by Siek and Taha [2007], which was first added to typestate by Wolff et al. [2011]. Gradual typing allows typestate declarations to be added progressively to a program. Whatever typestate information is present is used for static checking, and dynamic checks are inserted by the compiler to verify correct state transitions and access controls in untyped parts of the program. This approach supports a progression from a prototype program with many dynamic checks to a robust version in which almost all typestate properties are verified statically.

Crafa and Padovani [2015] have discovered intriguing analogies between typestate-oriented programming and the idiomatic modeling of objects in the Join Calculus. Following such analogies, they have put forward the Join Calculus as a formal model for typestate-oriented programming in a concurrent setting, whereby objects can be concurrently accessed and modified by several processes. A behavioral type system makes sure that objects are used according to their protocol.

3

Functional Languages

The integration of sessions and of session types in functional languages poses two main challenges. The first one is independent of the specific evaluation strategy (either call-by-value or call-by-name) and concerns the fact that, by their own definition, session types describe entities (channel endpoints) whose type may change after each usage. This feature is at odds with the conventional notion of type used in functional languages, which is meant to statically describe the nature of *values*. In particular, an arrow type $t \to s$ describes functions in terms of what they accept as argument (values of type t) and of what they produce as result (values of type s), but says nothing on how the function acts on channels possibly used when the function is applied to an argument. The second challenge concerns the fact that the call-by-name evaluation strategy adopted in lazy functional languages such as Haskell makes it difficult to predict the order in which expressions are evaluated. This contrasts with the need to perform input/output operations over channel endpoints in an order which is precisely determined by a session type. More generally, this is yet another instance of the recurring tension between the need of purity implied by laziness and the need to perform side-effects in order for a program to be useful.

The first challenge has been addressed in three different ways, either by drawing direct inspiration from effect systems (§3.1), or by using explicit continuation for channels (§3.2), or by defining a suitable monad for session communications (§3.3). Monads are also useful for addressing the second challenge, pretty much in the same way the `IO` monad in Haskell allows the structuring of programs doing generic input/output.

3.1 Effects for Session Type Checking

Vasconcelos et al. [2006] have investigated the integration of sessions into an ML-like language by devising a session type system that keeps track of the *effect* of a function on the channel it uses. Like traditional effect systems, the session type system of [Vasconcelos et al., 2006] decorates arrow types with information on the "effect" of a function. Unlike traditional effect systems, in this case the decorations are solely meant to capture the *change in the type* of channels passed to the function, when the function is applied.

As an illustration, we show the modeling of Agency below:

```
1  agency :: ⟨Agency⟩^a → Unit
2  agency agencyAccess = sell (accept agencyAccess)
3
4  sell :: s : Agency; Chan s → Unit; s : End
5  sell s =
6    case s {
7      QUERY   ⇒ let journeyDetails = receive s in
8                send (cost journeyDetails) on s
9                sell s
10     ACCEPT  ⇒ let address = receive s in
11               send (date journeyDetails) on s
12     REJECT  ⇒ ()
13   }
```

The type $\langle \texttt{Agency} \rangle^a$ on line 1 indicates that the `agency` function takes as argument a shared channel `agencyAccess` on which it accepts connections from customers through the `accept` primitive. The `sell` function implements the `Agency` protocol of the agency by reading and writing messages on the private session channel `s` by means of the

`send` and `receive` primitives. The `case s` construct is also related to communication: it waits for a label from channel `s` (one of `QUERY`, `ACCEPT`, or `REJECT` in the example) and evaluates the corresponding code.

The type of `sell` on line 4 consists of three parts: an arrow type `Chan s` $\rightarrow$ `Unit` with two decorations `s : Agency` and `s : End`. The arrow type indicates that `sell` accepts a channel as argument and returns the unit value, but it also gives a name `s` to the channel. This way, the decoration `s : Agency` before the domain type gives the expected session type (`Agency`) of the channel `s` when the function is applied, while the decoration `s : End` after the codomain type gives the session type (`End`) of the channel `s` after the function has returned. In the example, `Agency` is a name for the (recursive) session type:

```
Agency = &⟨QUERY: ?String.!Double.Agency,
           ACCEPT: ?String.!Date.End,
           REJECT: End⟩
```

With this information, the type checker can verify that the body of `sell` uses `s` according to the protocol described by `Agency`. In particular, the type checker updates the type of `s` in accordance with the operations performed on `s`. The occurrence of `s` on line 6 has type `Agency` and is used to receiving a label that indicates the operation chosen by the customer: if the label is `QUERY`, the channel `s` is used for receving a `String` with the details of the journey (line 7), then to send back the estimated cost of the journey (line 8), and then recursively according to the `Agency` protocol again (line 9); if the label is `ACCEPT`, the channel `s` is used for receiving a `String` with the customer's address (line 10) and then for sending back the estimated delivery date (line 11); finally, if the label is `REJECT`, the interaction with the customer terminates (line 12). Observe that the sequences of input/output operations in `sell` match those described by `Agency` and that `Agency` indicates that, whenever the function terminates, the session channel has type `End` and is not supposed to be used for further operations.

3.2 Sessions and Explicit Continuations

Inspired by the observation that channels are linear resources that must be used exactly once, Gay and Vasconcelos [2010] take a rather different approach that fits better the conventional type systems, where the type of an entity does *not* change. The idea is that a function that takes a channel as argument *consumes* the channel and *produces* a continuation of the same channel with a possibly different type. Following this style, the `sell` function in §3.1 is rewritten thus:

```
sell :: Agency → End
sell s =
  case s {
    QUERY   ⇒ λs.let (journeyDetails, s) = receive s in
                let s = send (cost journeyDetails) s in
                sell s
    ACCEPT  ⇒ λs.let (address, s) = receive s in
                let s = send (date journeyDetails) s in
                s
    REJECT  ⇒ λs.s
  }
```

Compared to the previous implementation of `sell`, we observe two main differences. At the type level, we note that `sell` has an ordinary-looking arrow type with domain `Agency` and codomain `End`. In particular, `sell` consumes a channel of type `Agency` and produces a continuation channel of type `End`. The second main difference concerns the body of `sell`, where we observe a series of subsequent *rebindings* of the channel `s`. Indeed, the communication primitives `send` and `receive` consume the channel on which they operate and produce its continuation. Note also that the `case` construct now expects functions on the right hand side of ⇒'s, which are applied to the continuation of `s` (next to the `case` keyword) after the label has been received.

It should be noted that, despite the multiple rebindings, at runtime there is just one session channel which is re-used over and over again after each operation. The rebindings are thus meaningful only at the type level, allowing each occurrence of `s` to be associated with a possibly different type. This is evident looking at the type schemes of `send` and `receive` primitives, shown below:

```
receive :: ?T.S → (T, S)
send    :: T → !T.S → S
```

The type of `receive` denotes the fact that `receive` consumes a channel of type $?T.S$ and produces a pair made of the message (of type T) received from the channel and a continuation channel (of type S) on which the communication may continue. The function `send`, on the other hand, takes a message of type T, consumes a channel of type $!T.S$ by sending the message on it, and produces a continuation channel of type S.

The mechanisms we have described can be used to keep track of the type of a channel that is the argument of a function, but not of the type of *free channels* that occur within its closure. This problem emerges with *partial application*, an idiomatic feature of (most) functional languages. For example, assuming that `s'` is a channel of type T (which is a session type), the partial application

```
send s' :: !T.S → S
```

denotes a function that, when applied to another channel `s` of type $!T.S$, delegates `s'` over `s` and then returns the continuation of `s`, having type S. The problem is that a conventional type system does not recognize a value of type $!T.S \to S$ as a *linear value* that *must* be used exactly once. Therefore, the closure resulting from the partial application `send s'` might be discarded or used multiple times, compromising any communication that is supposed to occur on `s` or, possibly worse, violating the protocol specified by the session type of `s'`. To solve this problem, Gay and Vasconcelos [2010] introduce a *linear arrow type* $\multimap$ that denotes functions that *must* be used exactly once. In particular, the partial application above is typed thus:

```
send s' :: !T.S ⊸ S
```

thereby preventing (the value of) this expression from being discarded or duplicated.

3.3 Monadic Approaches to Session Type Checking

Support for sessions and session types in Haskell has been investigated by Neubauer and Thiemann [2004], Pucella and Tov [2008], Imai et al.

[2010]. Incorporating primitives for session interaction, which rely on input/output operations, into a lazy functional language requires special care, so that their execution order becomes predictable. Therefore, all of the mentioned approaches define an appropriate monad (related to the `IO` monad) representing computations that may perform actions in a session. The use of a monad dedicated to session interactions also addresses the *aliasing problem*. If session channels were treated as ordinary Haskell values, and output on the channel were implemented through a `send` function with one of the types discussed above, nothing would prevent the *same* channel to be used multiple times, violating the protocol specified in its session type. For example, it could be possible to evaluate

```
send 74 c
```

twice, even if the type of `c` is `!Int.End` which allows only one integer to be sent over `c`. The monad for session interaction hides the actual channel from the programmer, and prevents the creation of aliases that could grant non-linear access to the channel. This makes it possible to statically enforce affine usage of session channels solely using the features of Haskell's type system, which makes no provision for linear values and, particularly, linear arrow types, which are necessary for the soundness of the approach described in §3.2. In the specific case of [Pucella and Tov, 2008], the abstract type

```
Session st st' a
```

denotes an action of the `Session` monad that transforms a session channel from type `st` to type `st'`, at the same time producing a value of type `a`. For instance, `receive` and `send` have the polymorphic types

```
receive :: Session (Cap e (a :?: r)) (Cap e r) a
send    :: a → Session (Cap e (a :!: r)) (Cap e r) Unit
```

which are analogous to the ones we have discussed earlier, except that there is no explicit argument denoting the channel on which these operations act. The actual channel is encapsulated within the `Session` monad, whose definition is private to the library and not accessible to the programmer. Here, `Cap` is a *phantom type constructor* that stores a type environment `e` (used for handling recursive protocols) and a proper

session type obtained through other type constructors `:?:` and `:!:` (for input and output), `:&:` and `:+:` (for binary branches and selections), and `Eps` (which plays the same role as `End`). For instance, `a :?: r` and `a :!: r` are respectively the encodings of the session types `?a.r` and `!a.r`. The agency server above is coded in Haskell like this (for the sake of simplicity, we implement a session that accepts exactly one query from the customer):

```
agency :: Rendezvous
  (String :?: Double :!:
   (Eps :&: (String :?: Date :!: Eps))) → IO Unit
agency agencyAccess = accept agencyAccess agencyOnce

agencyOnce :: Session
  (Cap e (String :?: Double :!:
          (Eps :&: (String :?: Date :!: Eps))))
agencyOnce = do journeyDetails <- receive
                send (cost journeyDetails)
                offer close
                      (do address <- receive
                          send (date journeyDetails)
                          close)
```

Note that `agencyOnce` makes no explicit reference to the session channel being used, which is instead supplied by `accept`. In the code, the basic actions `offer` and `close` respectively implement basic constructs for session branching and closing.

An analogous technique for avoiding aliasing is used in [Neubauer and Thiemann, 2004]. Pucella and Tov [2008] describe other extensions, including the encoding of recursive session types and the interleaving of multiple channels, and they claim that their encoding of session types scales without major obstacles to other polymorphic, typed languages such as ML and Java.

In general, the encodings proposed in [Neubauer and Thiemann, 2004, Pucella and Tov, 2008] produce cumbersome types. Fortunately, Haskell infers them in most cases. A more advanced session type inference technique is described by Imai et al. [2010].

A monadic approach is also taken for an integration of multiparty session types into OCaml/F#. This technique has been used for the se-

cure implementation of sessions [Corin et al., 2008, Corin and Deniélou, 2008, Bhargavan et al., 2009] as well as for the study of dynamic multirole sessions [Deniélou and Yoshida, 2011]. It consists of using a compiler that reads a session specification in order to automatically generate the appropriate monadic session API. As an example, the travel agency example could be specified as the following protocol between two roles `c` (for customer) and `a` (for agency):

```
session Travel =
  roles c, a
  global main =
    loop:
    choice from c to a {
      Journey(string).(Quote (int) from a to c; loop)
    | Accept.(Details(string) from c to a;
              Address(string) from c to a;
              Date(string) from a to c)
    | Reject.(RejectS from a to c)
    }
```

Starting from this specification, the compiler can generate a module with a CPS-API that prevents, by typing, any API user to derail from the specified behavior. The generated API contains one function per-role, whose argument must be a well-typed continuation structure. For example, the API for the agency is:

```
type result_a = unit
type a_start = a_2
and  a_2 = { hJourney : string → a_4 ;
             hAccept : unit → a_6 ;
             hReject : unit → a_13 }
and  a_4 = Quote of int * a_2
and  a_6 = Details of string * result_a
and  a_13 = RejectS of unit * result_a
val a : id → a_start → result_a
```

Each internal choice is represented as a sum-type where the constructor is the message label and each external choice a product-type (as a record). The generated function `a` is then in charge of reading the user's choice of message to send, and calling the appropriate continuation when a message is received.

The use of this technique is especially justified by security implementation reasons: there are no generic `send` and `receive` primitives since security verifications depend on the session specification.

3.4 Related Work

Bono et al. [2013] have extended the type system in [Gay and Vasconcelos, 2010] with support for polymorphism *à la ML*. In particular, in [Bono et al., 2013] it is possible to associate the `receive` and `send` functions with the types

```
receive :: ∀a. ∀A. ?a.A → (a, A)
send    :: ∀a. ∀A. a → !a.A → A
```

where a and A respectively stand for type and session type variables. In this way, the constants that implement communication primitives need not be treated with *ad hoc* type checking rules. Further decorations are allowed on quantifiers and arrow types for detecting potentially harmful state configurations that lead to memory leaks (more details on this issue are provided in Chapter 6).

The subsequent re-bindings of session channels that characterize the approach by Gay and Vasconcelos [2010] are reminiscent of the compilation scheme of sessions into pure π-calculus channels [Dardha et al., 2012]. Padovani [2015] has shown how to exploit this compilation scheme to encode a session type system in any ML-like functional language, supporting complete session type inference and using runtime checks for verifying that session channels are used at most once.

Continuations are also related to the monadic handling of state, whereby the "state" `s` is threaded in a strictly sequential way and the rebinding boilerplate code is implicit and hidden within the monad definition. The monadic integration of functions and sessions is not a prerogative of lazy languages, but makes sense also for strict languages to separate pure computations from side effects (communications). This approach has been formally studied by Toninho et al. [2013] and fosters the adoption of session-based communication in mainstream programming languages, since it does not require the host language to support linear types.

4

High-Performance Message-Passing Systems

The Message Passing Interface library specification Forum [2012] is the *de facto* standard for programming high-performance parallel applications. The standard's first version was published in 1994 and included bindings for the Fortran and C programming languages. Since then, several implementations of the standard have been developed for different platforms that support hundred of thousands of processing units.

MPI programs adhere to the Single Program, Multiple Data paradigm (SPMD), in which a single program specifies the behavior of the various processes, each working on different data and running on a different processor/core. MPI offers different forms of communication, including point-to-point, collective, and one-sided communication. The communication can be characterized along two orthogonal directions: (1) synchronous and asynchronous and (2) blocking and non-blocking (also referred as immediate). The standard includes primitives for all four possible combinations that arise from the fact that MPI communications account for the duration of the transmission. For example, in MPI it is possible to communicate synchronously in a blocking manner, as well as synchronously following a non-blocking approach. Non-blocking communications may overlap with computation.

Point-to-point communication specifies the interaction between two different processes, one sender and one receiver. Collective operations are executed synchronously by all (or a group of) processes. These operations include, for instance, the ability to broadcast a buffer to all (or a group of) processes, the ability to scatter or gather a buffer among processes, and reducing operations on values from all (or a group of) processes. Collective operations facilitate the writing of complex behaviors and allow the MPI implementation to optimize the performance of communication. Although a broadcast and n-send/-receive operations from one process to the others may be seen as having the same behavior, MPI implementations exploit the knowledge that a collective operation is taking place in order to optimize the way communication is handled, taking into account the network topology. One-sided communication allows a process to remotely access the memory of another process (RMA) for storing and retrieving values directly on the other's process memory. It differs from the previous modes of communication because the process issuing the request may do so without the collaboration of the other process involved, unlike point-to-point and collective communications.

High-performance computing applications can exhibit complex message passing behaviors and are often deployed in computing infrastructures that include thousands of processors/cores, costing serious money on computing power. The MPI standard, for instance, describes hundreds of primitives that can be used along the computation and that express far from trivial behaviors. It is, in fact, very easy to write an MPI application that deadlocks or that exhibits race conditions just by following a wrong communication protocol. In this context, behavioral types can be of great help to developers of MPI applications since they allow dedicated tools to check that programs comply to given communication protocols.

Example Below is an implementation of the travel agency example in C using MPI primitives (C+MPI). The programmer must craft the communication between Customer and Agency (in this simple case, by sending and receiving messages) in order to realize the desired behavior.

```
#define CUSTOMER 0
#define AGENCY 1
#define MSG_SIZE 100
#define ADDRESS_SIZE 100
#include <mpi.h>
int main(int argc,char **argv){
 int rank;    /* process rank */
 MPI_Status status;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 do {
  char journeyDetails[MSG_SIZE];
  float price;
  if (rank == CUSTOMER) {
   journeyDetails = generateMessageToSend();
   MPI_Send(journeyDetails,MSG_SIZE,MPI_CHAR,AGENCY,
            0,MPI_COMM_WORLD);
   MPI_Recv(&price,1,MPI_FLOAT,AGENCY,0,
            MPI_COMM_WORLD,&status);
   processPrice(price);
  } else if (rank == AGENCY) {
   MPI_Recv(journeyDetails,MSG_SIZE,MPI_CHAR,
            CUSTOMER,0,MPI_COMM_WORLD,&status);
   price = computePrice(journeyDetails);
   MPI_Send(&price,1,MPI_FLOAT,CUSTOMER,0,
            MPI_COMM_WORLD);
  }
 } while (moreQuestionsToAsk());

 int decision;
 char deliveryAddress[ADDRESS_SIZE];
 int date[3];   /* format: year, month, day */
 if (rank == CUSTOMER) {
  decision = decidesToAccept();
  MPI_Send(&decision,1,MPI_INT,AGENCY,0,
           MPI_COMM_WORLD);
  if (decision) {
   deliveryAddress = getDeliveryAddress();
   MPI_Send(deliveryAddress,ADDRESS_SIZE,MPI_CHAR,
            AGENCY,0,MPI_COMM_WORLD);
```

```
    MPI_Recv(date,3,MPI_INT,AGENCY,0,MPI_COMM_WORLD,
             &status);
   }
  } else if (rank == AGENCY) {
    MPI_Recv(&decision,1,MPI_INT,CUSTOMER,0,
             MPI_COMM_WORLD,&status);
    if (decision) {
     MPI_Recv(&deliveryAddress,ADDRESS_SIZE,MPI_CHAR,
              CUSTOMER,0,MPI_COMM_WORLD,&status);
     date = computeDate(deliveryAddress);
     MPI_Send(date,3,MPI_INT,CUSTOMER,0,
              MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();
  return 0;
}  /* main */
```

The program defines the behavior of both participants at the same time, according to the SPMD paradigm. It starts by initializing the MPI library and by retrieving the process rank, an integer that uniquely identifies each process involved in a parallel computation. In the present case, the process ranked 0 is the customer, whereas the process ranked 1 is the travel agency. The behavior of each participant is distinguished by testing the process rank and by choosing different control flows for each participant. Then, the program enters a loop where an unbound number of queries are posted to the travel agency. The sender of a message has to specify the buffer holding the data, its size and type, the rank to whom it is addressed to, a tag that may be used to distinguish messages (in the example we always use zero), and the communicator that specifies the group and topology of processes involved in the communication (in the example we always use the predefined `MPI_COMM_WORLD` that includes all processes). The receiver has to specify the buffer where the message is going to be stored, the size and type of the incoming message, its sender's rank, the tag, the communicator, and the status structure that contains information about the message being received. MPI is shutdown with a call to the `MPI_Finalize` function.

Several approaches based on behavioural types have been studied and implemented to verify and/or guarantee the correctness of MPI ap-

plications; in the rest of this chapter, we illustrate three of them: one that makes use of high-level, session-based communication primitives (§4.1), one based on verification of C+MPI code with the help of user-provided annotations (§4.2), and another one based on code generation (§4.3). All these approaches rely on so-called *global types*, a particular kind of behavioral types, that allow for the specification of communication protocols from a neutral viewpoint. A detailed presentation of global types is deferred to Chapters 5 and 8. For the time being, we anticipate that a global type can be *projected* to an endpoint protocol describing the behavior of a single participant in an interaction. The theory of session types guarantees, to this point, that the protocol is deadlock free and communication safe by construction. The final step is to guarantee that each process behaves according to each endpoint protocol.

The global type corresponding to the Customer-Agency example is shown below, using the syntax of Scribble (see §8.2):

```
protocol PurchaseATrip(role Customer,
                       role TravelAgency)
{
  rec tripProcurement {
    JourneyDetails from Customer to TravelAgency;
    Price from TravelAgency to Customer;
    tripProcurement;
  }
  choice at Customer {
    AcceptTrip from Customer to TravelAgency;
    DeliveryAddress from Customer to TravelAgency;
    date from TravelAgency to Customer;
  } or {
    RejectTrip from Customer to TravelAgency;
  }
}
```

Notice that the endpoint projections of this global type correspond to those presented in §2.2.1 and to the behavior of the C+MPI code shown earlier.

4.1 Session C

Session C [Ng et al., 2012] is a multiparty session-based programming environment for C that enforces deadlock-freedom, communication safety, and global progress through static type checking. This approach starts with the specification of a global protocol, using a protocol description language such as Scribble, that captures the communication pattern of the parallel algorithm to be implemented. From this protocol, the projection algorithm generates endpoint protocols that guide the design and implementation of each endpoint C program. The endpoint protocol can be further optimized through subtyping for asynchronous communication, preserving the original safety properties. The underlying theory can ensure that the complexity of the toolchain stays in polynomial time on the size of programs.

Session C represents an enhancement from SJ (§2.2.1) in the sense that it can handle multiparty communications directly, whereas SJ treats only binary sessions, and it offers primitives for chaining the binary constructs on different multiple sessions together. As for session delegation, Session C does not handle it since there is less motivation to use session delegation in the multiparty setting; on the other hand, SJ supports delegation. At runtime, Session C offers a significant speed-up (60%) when compared to SJ and to MPI for Java.

The programming environment is made up of two main components: a session type checker and a runtime library. The session type checker takes an endpoint protocol and a source code program as input and validates the source code against its endpoint protocol. The library offers a simple but expressive enough interface for session-based communications programming.

The Customer-Agency example written in Session C Our running example can be sketched in Session C as follows. Below is the customer code.

```
#include <libsess.h>
...
int main(int argc, char **argv) {
  session *s;
```

```
  join_session(&argc, &argv, &s, "Customer.spr");
  const role *agency = s->get_role(s,"TravelAgency");
  do {
    send_string(agency, generateMessageToSend());
    processPrice(recv_float(agency));
  } while(outwhile(moreQuestionsToAsk()));
  if (outbranch(decidesToAccept())) {
    send_string(agency, getDeliveryAddress());
    int date[3] = recv_int_array(agency, 3);
  }
  end_session(s);
}
```

The code for the travel agency is shown below:

```
#include <libsess.h>
...
int main(int argc, char **argv) {
  session *s;
  join_session(&argc,&argv,&s,"Travel_Agency.spr");
  const role *customer = s->get_role(s, "Customer");
  do {
    send_float(customer,
               computePrice(recv_string(customer)));
  } while(inwhile(customer));
  switch (inbranch(customer, &rcvd)) {
    case Accept:
      send_int_array(customer,
                 computeDate(recv_string(customer)));
      break;
    case Reject: break;
  }
  end_session(s);
}
```

A Session C program is a C program that calls the session runtime library. The code above implements the behavior of both the customer and the travel agency. We focus on the customer code. In the `main` function, `join_session` indicates the start of a session, whose arguments (`argc` and `argv` from the command line) are a session handle of type `session *` and the location of the endpoint Scribble file. The `join_session` establishes connections to other participating processes in

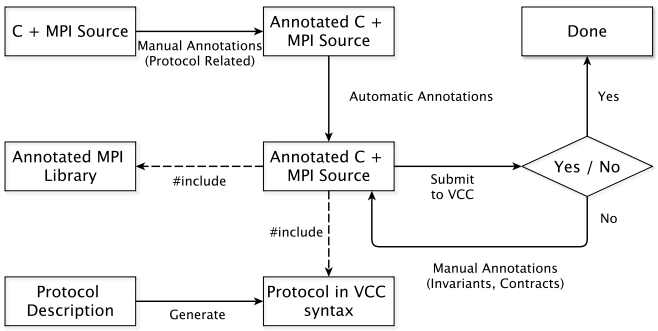


Figure 4.1: Approach for verifying C+MPI programs

the session, according to a connection configuration information such as the host/port for each participant, automatically generated from the global protocol. Next, the lookup function `get_role` returns the participant identifier of type `role *`. Then, we have a series of session operations such as `send_`*type* or `recv_`*type*. Iteration and branching in Session C are declared explicitly with the use of `inwhile`, `outwhile` and `inbranch`, `outbranch`, respectively, similarly to what has been described for Session Java.

4.2 Deductive Verification of C+MPI Code

This approach directly verifies C+MPI programs against session types [Honda et al., 2012, Marques et al., 2013b, López et al., 2015], in contrast with Session C where programmers use a particular library of communication primitives.

To verify the conformance of C+MPI programs against protocol specifications the programmer starts by capturing the application's communication global protocol description. Afterwards, the protocol is translated into a term written in the language of VCC [Cohen et al., 2009], a software verifier tool for the C programming language (refer to the example below). The translation is done automatically using a tool that verifies that the protocol is well formed, guaranteeing global deadlock freedom. The C+MPI code imports the protocol definition (in VCC form) and a VCC-annotated MPI library with session type con-

tracts for the various MPI primitives. Depending on the specifics of the C code, further manual annotations may be required. In this setting, VCC is invoked to check whether the C code follows the communication type. The overall workflow process is depicted in Figure 4.1.

The verification deals with point-to-point and collective operations. For that, the protocol specification departs from Multiparty Session Types and Scribble by introducing collective decision primitives, allowing for behaviors where all participants decide to enter or to leave a loop, or to choose one of two branches of a choice input. These two patterns are impossible to describe in Scribble, but are a standard practice in C+MPI programs. The communication types language includes specific MPI collective operations, as well as a dependent functional type constructor.

The verification process checks the program from MPI initialization (call to `MPI_Init`) to shutdown (`MPI_Finalize`). There is the need to add state and behavior to perform the verification. This is known as ghost data and code, and is only available for the verification process. A ghost `type_func` function, representing the protocol, parametric on the rank, returns the endpoint projection of the global type for a given rank. This endpoint type is assigned to a ghost variable and the verification proceeds by progressively reducing the protocol, i.e., by changing the ghost variable through the contracts of MPI primitives or as a result of the annotations that handle program control flow. The goal is that the ghost variable reaches a state congruent to `end()` at the shutdown point (the call to `MPI_Finalize`).

As for control flow, collective choices, and loops in particular, direct annotations are necessary in the program body. These are partially generated by a tool. Here, we focus now on its meaning, using the collective loop of our running example.

```
_(ghost SessionType body = loopBody(type);)
_(ghost SessionType continuation = head(type);)
do { _(ghost type = body;)
     ...
     _(assert congruent(type, end())
} while (moreQuestionsToAsk());
_(ghost type = continuation);
```

The fragment illustrates the extraction of the protocols corresponding to the loop `body` and its `continuation` from the endpoint type stored in ghost variable `type`. The protocol for the body must be a `loop` type. The verification procedure asserts that the loop protocol body is reduced to a term congruent to `end()`. After the loop, verification proceeds by using the loop `continuation` as the type.

The VCC theory put forward by Marques et al. [2013a] is divided in two parts: the first is a contract-annotated version of the MPI function signatures that ensures the conformance of the program operations against a protocol; the second encodes the type reduction relation (omitted here for brevity). We illustrate contract annotation using the `MPI_Send` function.

A significant part of the required program annotations are introduced automatically by a tool that uses the Clang/LLVM framework to traverse the syntactic tree of a C program and generate a new, annotated, version.

Verifying the Customer-Agency C+MPI program using VCC

Our running example protocol can be sketched using a VCC datatype value as follows. The function contains the global type ready to be projected, depending on the rank parameter.

```
_(ghost _(pure) \Type type_func(int rank)
  _(requires 0 <= rank && rank < 2)
  _(ensures \result ==
    loop (
      rank == 0 ?
        (comm(send(1,MPI_CHAR,100), ...);
        comm(recv(1,MPI_FLOAT,1), ...)) :
      rank == 1 ?
        (comm(recv(0,MPI_CHAR,100), ...);
        comm(send(0,MPI_FLOAT,1), ...)) :
        end()
    );
    rank == 0 ?
      comm(send(1,MPI_INT,1), ...) :
    rank == 1 ?
      comm(recv(0,MPI_INT,1), ...) :
```

```
      end();
    choice(
      (rank == 0 ?
        (comm(send(1,MPI_CHAR,100), ...);
        comm(recv(1,MPI_INT,3), ...)) :
      rank == 1 ?
        (comm(recv(0,MPI_CHAR,100), ...);
        comm(send(0,MPI_INT,3), ...)) :
        end())
      end(),
    end())))))
```

In what follows we present an excerpt of the annotations required to the C+MPI program presented in the beginning of this section.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
_(ghost type = type_func (rank))

_(ghost \Type loop_body = loopBody(type);)
_(ghost \Type loop_continuation = reduce(type);)
do {
  _(ghost type = loop_body;)
  ...
  _(assert congruent(type, end())
} while (moreQuestionsToAsk());
_(ghost type = loop_continuation;)
...
_(ghost \Type choice_true = choiceTrue(type);)
_(ghost \Type choice_false = choiceFalse(type);)
_(ghost \Type choice_continuation = reduce(type);)
if (decision) {
  _(ghost type = choice_true;)
    ...
   _(assert congruent(type, end())
}
_(ghost type = choice_continuation;)
...
MPI_Finalize();
```

The ghost annotations inserted into the C+MPI code introduce the ghost variable `type`, projecting it for a particular `rank`. The verification proceeds, either directed by the MPI function contract annotations or

by the manual annotations inserted before the loops and before the collective choices. More sophisticated constructs, here omitted since they are not used in our running example, include collective operations and foreach loops.

4.3 MPI Code Generation

A protocol-driven MPI code generation framework [Pabble-MPI] was presented by Ng et al. [2015] for type-safe and deadlock-free MPI programs based on Parameterised Multiparty Session Types [Deniélou et al., 2012] using the Pabble protocol description language [Ng and Yoshida, 2015], which itself is based on Scribble. The code generation process starts with the definition of the global topology using a protocol specification language based on parameterised multiparty session types (MPST). From this, an MPI parallel program backbone is automatically generated. The backbone code can then be merged with the sequential code describing the application behaviour, resulting in a complete MPI program. This merging process is fully automated through the use of an aspect-oriented compilation approach. In this way, programmers only need to supply the intended communication protocol and provide sequential code to automatically obtain parallelised programs that are guaranteed to be free from communication mismatch, type errors, and deadlocks. The code generation framework also integrates an optimisation method that overlaps communication and computation, and can derive not only representative parallel programs with common parallel patterns (such as ring and stencil), but also distributed applications from any MPST protocols.

The framework represents a top-down, correct-by-construction approach to applying behavioural types to MPI programming, an alternative to the bottom-up deductive verification in the previous section.

4.4 Related Work

Session C needs to be extended to support more conventional parallel programming runtime library. Message Passing Interface (MPI) has been identified as the ideal API to target because of its comprehen-

siveness, because it is standardized, and, to a lesser extent, because of its popularity in the HPC community. Also, the approach of Session C is only as useful as the expressiveness of the protocol language (Scribble) being type-checked against. In order to support MPI as the programming API for Session C, Scribble is currently being extended to a dependent language, with the primary aim of supporting a scalable way of addressing participants using numeric indexes. The challenges of this is to keep type checking decidable, while increasing the complexity of the parallel programming/MPI primitives that Session C supports. One idea being developed is to use code generation in place of static type checking, which sees communication safe code generated from a well-formed Scribble protocol.

The HPC community makes extensive use of non-blocking and one-sided communications. Non-blocking operations allows for the overlapping of computation and communication, while one-side communications allows for a participant to remotely access the memory of another participant. The remote access happens without a corresponding operation from the remote participant, as it is the case with point-to-point and collective operations. Governing these kind of interactions deserves further investigation.

New programming languages have been introduced in the recent past aimed at HPC, notably X10 [Charles et al., 2005], Chapel [Chamberlain et al., 2007], and Fortress [Steele, 2006], that propose new interaction models, in particular the asynchronous partitioned global address space [Saraswat et al., 2010], that introduces challenging open issues.

5

Multiagent Systems

Multi-agent systems (MASs [Jennings et al., 1998]) have been proved to be an industrial-strength technology for integrating and coordinating autonomous and heterogeneous systems. MASs are open, highly dynamic, and unpredictable; for these reasons, ensuring conformance of the agents' actual behavior to a given interaction protocol is of paramount importance to guarantee the participants' interoperability and security.

In this chapter we focus on the problem of verifying protocol conformance for Jason [Bordini et al., 2007], one of the most widespread implementations of the logic-based agent oriented programming language AgentSpeak [Rao, 1996]. Static verification for logic-based agent oriented programming languages is very challenging because of the intrinsically dynamic nature of such languages: no type discipline is statically enforced, heterogeneous data can be freely mixed together, and there is no clear separation between data (terms) and code (atoms). For this reason, runtime verification is preferred over static type analysis to check protocol conformance of MASs. As happens in choreographies (Chapter 8), interaction protocols between agents are specified globally with behavioral types called *global types*. Unlike choreographies,

however, the correct implementation of interaction protocols is always checked at runtime. This makes it possible to adopt more expressive global type languages since the undecidability issues typical of static analysis are not a concern.[1]

From global types, monitor agents are automatically generated to dynamically check that the conventional agents of the system implement the intended interaction protocols correctly. The behavior of monitor agents directly depends on the global type that specifies the interaction protocol to be checked, and on the semantics of global types, which is expressed by a labeled transition system, implemented by all monitor agents.

The main challenges of this proposed approach concern efficiency of dynamic checking and the expressiveness and conciseness of global types. For the approach to be effective, dynamic checks need to be performed efficiently, because a system has to be monitored for a considerable (ideally, for an indefinite) amount of time. For this reason, the time complexity of dynamic checking protocol conformance should be linear in the length of the sequence of exchanged messages. Expressive sets of type operators allow one to describe complex protocols more concisely, with beneficial effects on the readability of the protocol specification and on the scalability in terms of the dimension of the required global types.

In this chapter we work with the notion of global type introduced by Ancona et al. [2012, 2013a,b] and Mascardi and Ancona [2013].

5.1 Global Types for MAS Monitoring

Global types can be easily represented as cyclic Prolog terms, and a mechanism for verifying that a sequence of messages complies with a global type has been designed and implemented in Prolog. By exploiting these features, a monitor has been developed on top of Jason; such a monitor is able to verify at run-time that the actual conversation among

[1] Most decision problems are already undecidable for context-free languages, whereas the global types we have devised for defining interaction protocols are strictly more expressive than context-free grammars.

agents in the MAS complies to the interaction protocol specified by a global type in the formalism described in this section.

Interactions. An interaction occurring between two agents is represented as a 4-tuple consisting of two agent identifiers (the sender and the receiver of the message), the performative (explained below) expressed in some agent communication language the agents agree upon, such as FIPA-ACL [Foundation for Intelligent Physical Agents] or KQML [Mayfield et al., 1995] (in the Jason implementation the latter is used), and the actual content of the message expressed in some content language shared among the agents (in the Jason implementation Prolog terms are used). For instance in the interaction specified by `ca(seller, buyer, tell, price(pasta,10))`, `seller` and `buyer` denote the sender and the receiver, respectively, of the message, `tell` is its performative, and the term `price(pasta,10)` is its content which expresses the fact that `buyer` intends to sell `pasta` at the `price` of `10` euros. Performatives are defined in the "speech acts theory" [Austin, 1962], which is part of the philosophy of language, as sentences which are not only passively describing a given reality, but are changing the social reality they are describing. In the Agent Communication Languages (ACL) research field, the performative denotes the type of the communicative act of the ACL message, such as telling, asking, recommending, etc. Each ACL, such as FIPA-ACL and KQML, defines its own set of performatives. For instance, a message with performative `tell` changes the knowledge base of the receiver, whereas a message with performative `ask` corresponds to a query on the knowledge base of the receiver. The set of interactions is denoted by $\mathcal{A}$ throughout this chapter.

Interaction types. In the specification of a global type we use interactions types to model which kind of message pattern is expected at a certain point of the conversation. This gives us the freedom to specify the expected content type, such as an integer, a string, or a complex term, the sender and receiver type, and the performative type, possibly using free variables and additional conditions for modeling protocols in which, for example, we do not care which are the agents that interact as long as the interaction has a certain performative and the sender

and the receiver are two different agents. An interaction type α is a predicate on interactions, hence its interpretation is the set of interactions that verify α; we write $a \in \alpha$ to mean that α is true on a, and we also say that a has type α.

Global types. A global type τ represents a set of possibly infinite sequences of interactions, and is defined on top of the following type constructors:

- λ (empty sequence), representing the singleton set $\{\epsilon\}$ containing the empty sequence ϵ.

- $\alpha{:}\tau$ (*seq*), representing the set of all sequences whose first element is an interaction a matching type α ($a \in \alpha$), and the remaining part is a sequence in the set represented by τ.

- $\tau_1 + \tau_2$ (*choice*), representing the union of the sequences of τ_1 and τ_2.

- $\tau_1 | \tau_2$ (*fork*), representing the set obtained by shuffling the sequences in τ_1 with the sequences in τ_2 .

- $\tau_1 \cdot \tau_2$ (*concat*), representing the set of sequences obtained by concatenating the sequences of τ_1 with those of τ_2.

The semantics of global types is defined by a labeled transition system, where states are global types and labels are interactions.

At runtime, a monitor agent M stores in its knowledge base the global type τ that corresponds to the current state of the interaction protocol P checked by M; whenever M intercepts an interaction a between two agents that is pertinent to the checked protocol P, it verifies whether there exists a transition from τ to τ' labeled with a; if so, it updates in its knowledge base the current state of the protocol with the new global type τ', otherwise, it detects a runtime error and handles it properly.

Example

The Customer-Agency protocol described in §1 can be modeled by the following global type:

$$CustomerAgency = Sell \cdot AcceptOrReject$$

$$Sell = query : propose : (Sell + \lambda)$$

$$AcceptOrReject = (Accept + Reject)$$

$$Accept = confirm : sendAddress : forwardAddress : sendDate : forwardDate : \lambda$$

$$Reject = reject : \lambda$$

Interactions types. In global types, only *interaction types* appear. Actual interactions taking place in the environment are expected to have one of the foreseen interaction types, but the link between actual communication actions and their types is kept separate from the global type definition. Decoupling interaction types from actual communication actions allows the global type designer to concentrate on the description of the communication protocol among the involved parties, abstracting from the actual agent communication language used by them.

The coupling must be defined if the global type is to be used in practice, for monitoring a real multiagent system. For example, in the Customer-Agency protocol we might state that an actual interaction $ca(Customer, Agency, cfp, journey(Dest))$ has type *query* **iff** it models a call for proposal for an offer concerning a journey and *Customer* is the identifier of the customer, *Agency* is the identifier of the agency, and *Dest* is a `string`. Another actual interaction $ca(Agency, Customer, propose, journey(Price))$ could have type *propose* **iff**, besides constraints similar to those in the previous example, *Price* is a `double`.

As interactions in MASs are usually very complex and are not in the scope of this document, we do not enter into the details of actual communications and we limit ourselves to model their types.

Customer-Agency Global Type. The global type *CustomerAgency* is defined by means of the equation $CustomerAgency = Sell \cdot AcceptOrReject$, meaning that it is a composite type consisting of the global type *Sell* followed by the global type *AcceptOrReject* ($\cdot$ is the global type concatenation operator).

Sell is in turn defined as

$$Sell = query : propose : (Sell + \lambda)$$

meaning that it consists of the query about some destination *Dest* from *Customer* to *Agency*, followed by the price proposal (*propose*) from *Agency* to *Customer* (: is the sequence operator whose first operand is an interaction, and the second is a global type), further followed by a choice between repeating *Sell* (+ is the choice operator) or stopping (λ is the empty global type). Iteration is implemented by allowing a variable to appear in the equation defining the variable itself.

AcceptOrReject is defined as a choice between two global types ($Accept + Reject$), where *Accept* consists of a message from *Customer* to *Agency* to accept the proposal, followed by a message to inform *Agency* of the delivering address of the tickets, followed by a message from *Agency* to *Service* requesting to purchase the tickets to *Customer*, followed by the message from *Service* to *Agency* to inform it about the ticket purchase date which is forwarded by *Agency* to *Customer*. The final λ means that this branch of the global type ends here.

Reject just consists of a message from *Customer* to *Agency*, rejecting all the proposals made so far.

In the MAS frameworks where we exploited/plan to exploit the monitor, which include Jason but also JADE [Bellifemine et al., 2007] and possibly others, agents are usually aware of the receiver of the messages they send, and of the sender of the messages they receive. Hence, delegation as described in page 3, step 4, of [Hu et al., 2008],

> *Customer then sends a delivery address (unaware that he/she is now talking to Service)*

is not supported by the formalism.

5.2 Advanced Constructs for Protocol Specification

One of the distinguishing features of the global types presented here is their coinductive interpretation. This means that it is possible to specify and verify protocols that are not allowed to terminate. In particular, the monitor agent checks also agents responsiveness by means of time-outs; three different scenarios may occur:

1. if the current state of the monitor corresponds to the empty protocol (that is, the protocol must terminate), then the monitor reports an error as soon as an interaction is detected (independently of the time-out);

2. if the current state is final, but does not correspond to the empty protocol (that is, the protocol may terminate, but it can also continue), then the monitor reports a warning if a valid interaction is detected after the time-out has expired (if an invalid interaction is detected, then an error is reported independently of the time-out);

3. if the current state is not final (that is, the protocol is not allowed to terminate), then the monitor reports a warning as soon as the time-out expires, if no interaction is detected (an error is reported in case an invalid interaction is detected before the time-out).

If, on the one hand, static verification is able to ensure strongest guarantees on the correct behavior of a MAS, on the other hand, dynamic verification allows the adoption of more expressive languages. For instance, since global types are recursive and support concatenation, context-free languages can be specified.[2] Furthermore, since context-free languages are not closed under shuffle, global types are strictly more expressive. The expressive power of the formalism is further increased by the ability of constraining the shuffle operator, by specifying that two or more interaction types must correspond to the same event; in this way, languages that cannot be expressed with Petri nets can be

[2]Given the coinductive nature of global types, this claim holds if only finite sequences are considered.

specified with global types. For instance, the typical example of non context-free language[3] $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ can be easily specified (see the next section).

Specific examples. We consider the ABP, in the version defined by Deniélou and Yoshida [2012]. Four different interactions may occur: Alice sends msg1 to Bob (interaction type msg_1), Alice sends msg2 to Bob (interaction type msg_2), Bob sends ack1 to Alice (interaction type ack_1), Bob sends ack2 to Alice (interaction type ack_2). Also in this case the protocol is an infinite iteration, but the following constraints have to be satisfied for all occurrences of the interactions:

- The n-th occurrence of msg_1 must precede the n-th occurrence of msg_2.
- The n-th occurrence of msg_1 must precede the n-th occurrence of ack_1, which, in turn, must precede the $(n+1)$-th occurrence of msg_1.
- The n-th occurrence of msg_2 must precede the n-th occurrence of ack_2, which, in turn, must precede the $(n+1)$-th occurrence of msg_2.

The type defined below by the variable $AltBit_1$ is a correct specification of the ABP:

$$
\begin{aligned}
&AltBit_1 = msg_1{:}M_2\\
&AltBit_2 = msg_2 : M_1\\
&M_1 = (msg_1 : A_2) + (ack_2 : AltBit_1)\\
&A_1 = (ack_1 : M_1) + (ack_2 : ack_1 : AltBit_1)\\
&M_2 = (msg_2 : A_1) + (ack_1 : AltBit_2)\\
&A_2 = (ack_2 : M_2) + (ack_1 : ack_2 : AltBit_2)
\end{aligned}
$$

The type is reasonably compact, but it is harldy readable, and it takes time to understand what protocol is specified; also, it is not trivial to prove that ABP is correctly specified by the type.

[3] Again, given the coinductive interpretation, the languages must contain also the infinite sequence a^∞.

Another problem is that the size of the type grows exponentially with the number of interaction types; for instance, if we extend the ABP to three messages and three acknowledges, then we get the following type defined by the variable $AltBit_3$:

$$
\begin{aligned}
&AltBit_3 = msg_1 : S_1\\
&S_1 = (msg_2 : S_2) + (ack_1 : msg_2 : S_6)\\
&S_2 = (ack_1 : S_6) + ((ack_2 : S_4) + (msg_3 : S_3))\\
&S_3 = (ack_1 : S_7) + ((ack_2 : S_8) + (ack_3 : S_5))\\
&S_4 = (ack_1 : msg_3 : ack_3 : AltBit_3) + (msg_3 : S_8)\\
&S_5 = (ack_1 : ack_2 : AltBit_3) + (ack_2 : ack_1 : AltBit_3)\\
&S_6 = (msg_3 : S_7) + (ack_2 : msg_3 : ack_3 : AltBit_3)\\
&S_7 = (ack_3 : ack_2 : AltBit_3) + (ack_2 : ack_3 : AltBit_3)\\
&S_8 = (ack_1 : ack_3 : AltBit_3) + (ack_3 : ack_1 : AltBit_3)
\end{aligned}
$$

To see how constrained shuffle enhances the expressive power of the language, let us start with the following basic global type representing a naive and incorrect solution to the specification of the ABP:

$$
\begin{aligned}
&WAB = MA_1|MA_2\\
&MA_1 = msg_1{:}ack_1{:}MA_1\\
&MA_2 = msg_2{:}ack_2{:}MA_2
\end{aligned}
$$

The interpretation of WAB is a proper superset of the ABP; for instance, it contains sequences starting with $msg_2\ msg_1\ ack_2\ ack_1\ \ldots$ which do not meet the protocol, because the first occurrence of msg_2 must follow the first occurrence of msg_1.

This is due to the fact that the shuffle operator performs an unconstrained shuffle of the set of sequences (belonging to the interpretation) of the two operand types, while all correct sequences of the ABP must verify the additional constraint that the i-th occurrence of msg_2 must follow the i-th occurrence of msg_1 and precede the $i+1$-th occurrence of msg_1, for all natural numbers i. In other words, a correct sequence of the ABP must yield the infinite sequence $msg_1\ msg_2\ msg_1\ msg_2\ \ldots$, specified by the global type $MM = msg_1{:}msg_2{:}MM$, when restricted to the interactions msg_1 and msg_2.

However, the type $MA_1|MA_2|MM$ is not a correct fix to WAB, since the interactions generated from MM are considered different from those

generated from MA_1 and MA_2. To avoid this problem, we introduce two different kinds of interaction types, called *producers* and *consumers*, respectively. In global types extended with constrained shuffle (extended global types, for short) producer interaction types play the same role as interaction types in basic global types: each occurrence of a producer interaction type must correspond to the occurrence of a new event; in contrast, consumer interaction types correspond to the same event specified by a certain producer interaction type. The purpose of consumer interaction types is to impose constraints on interaction sequences, without introducing new events.

Differentiating producer and consumer interaction types allows us to express in a quite intuitive and simple way the ABP:

$$\begin{aligned} ABP &= MA'_1 | MA'_2 | MM \\ MA'_1 &= msg_1^1 : ack_1^0 : MA'_1 \\ MA'_2 &= msg_2^1 : ack_2^0 : MA'_2 \\ MM &= msg_1 : msg_2 : MM \end{aligned}$$

Global types MA'_1 and MA'_2 contain just producer interaction types, whereas MM contains only consumer interaction types. A consumer is an interaction type, whereas a producer is an interaction type α equipped with a natural superscript n specifying the number n of corresponding consumers that coincide with the same event; hence, n is the least required number of times $a \in \alpha$ has to be "consumed" to allow a transition labeled by a.

Hence msg_1^1 and msg_1 in MA'_1 and MM respectively, always correspond to the same event (and analogously for msg_2^1 and msg_2 in MA'_2 and MM). Since no constraint relates ack_1 and ack_2, the corresponding producers in MA'_1 and MA'_2 are super-scripted by 0.

As a final example, let us consider the protocol where Alice first sends n (with n arbitrary, and possibly infinite) messages to Bob (interaction type msg_1), then Bob send n messages to Carol (interaction type msg_2), and, finally, Carol sends n messages back to Alice (interaction type msg_3). This can be expressed by the global type T defined

as follows:

$$T = M_{1,2}|M_{2,3}$$
$$M_{1,2} = \lambda + ((msg_1^0{:}M_{1,2}) \cdot (msg_2^1{:}\lambda))$$
$$M_{2,3} = \lambda + ((msg_2{:}M_{2,3}) \cdot (msg_3^0{:}\lambda))$$

Since the two interaction types msg_2^1 and msg_2 in $M_{1,2}$ and $M_{2,3}$, respectively, must coincide with the same event, the number of messages exchanged between the three partners must always be the same.

5.3 Related Work

The notion of global type presented here is similar to that defined by Castagna et al. [2012]. There, global types model protocols in terms of atomic actions (interactions) and composite actions, essentially denoting a "language of legal interactions that can occur in a multi-party session". A protocol can consist of the empty sequence, a single interaction between a sender and a receiver, the concatenation, the shuffle, or the union of two global types. Interactions of arbitrary but finite length are defined with the Kleene star operator.

Whereas that paper focuses on "local" *session types*, which represent the *projections* of the global type on single entities (actors, agents), here only the global perspective is relevant. Also, the interpretation of global types is inductive: only interactions where the number of messages exchanged is arbitrary but always finite can be modeled. This is a radical difference with the formalism presented here, where infinite interactions can be modeled as well. Finally, constrained shuffle is not supported, and types cannot be recursive, hence the language is less expressive.

An approach similar to [Castagna et al., 2012] is described by Deniélou and Yoshida [2012] where the authors explore the connection between session types (which again are intended as projections of a global type to single participants) and communicating automata or Communicating Finite State Machines (CFSMs, [Brand and Zafiropulo, 1983]), and give a new syntax for global types.

Using the global types in [Deniélou and Yoshida, 2012] the ABP can be specified in a reasonably compact way, and the size of the type grows linearly if the protocol is extended. However their solution is less

simple than the specification presented in the previous section. Furthermore, the notion of global type as described here is more amenable to be directly translated in Prolog as a finite collection of unification equations having regular terms as solutions.

An interesting proposal for overcoming the limitations of dynamic protocol verification based on a centralized monitor is proposed by Chen et al. [2012]. There, a formal model of run-time safety enforcement for large-scale, cross-language distributed applications with possibly untrusted endpoints is proposed, whose underlying theory is based on multiparty session types with logical assertions (MPSA). MPSA is an expressive protocol specification language that supports run-time validation through monitoring. Given the global specifications based on MPSAs which the participants should obey, distributed monitors use local specifications, projected from global specifications, to detect whether the interactions are well-behaved and take appropriate actions, such as suppressing illegal messages. The main difference between that work and ours lies in this projection stage that, having a centralized monitor, we do not need to perform.

The Jason monitor is discussed by Ancona et al. [2012], whereas Mascardi and Ancona [2013] discuss the adoption of global types extended with attributes in the more general context of logic-based MASs. The theoretical underpinning of global types have been investigated by Ancona et al. [2013b], a working prototype and implemented examples are also available [GTV].

6

Singularity OS

Singularity OS [Hunt et al., 2005, Fähndrich et al., 2006, Singularity OS] is the prototype of a reliable operating system where software-isolated processes (SIPs) share the same address space, called *exchange heap*. Process interaction occurs solely through message exchange over asynchronous, FIFO channels and the communication overhead is reduced thanks to *copyless message passing*: only *pointers* to messages are physically exchanged between processes. Static analysis enforces *process isolation*, that is, every process can only access the memory it owns in an exclusive manner. Programs running in Singularity OS are written in Sing#, a programming language derived from C# specifically designed for developing Singularity applications. In this chapter we review the support for behavioural typing provided by Sing# (§6.1) and a refinement of its behavioural typing discipline aimed at preventing memory leaks (§6.2).

6.1 Channel Contracts in Sing#

In Singularity OS, channels are formed as pairs of related *endpoints*, called the channel *peers*. A message sent over a peer is received from

```
1  void Agency([Claims] imp<C:START> in ExHeap c,
2              [Claims] exp<D:START> in ExHeap d) {
3    switch receive {
4      case c.Query(String de):
5        /* compute a price pr for de */
6        c.Price(pr);
7        Agency(c, d);
8      case c.Reject():
9        c.Close();
10       d.Close();
11     case c.Accept():
12       d.Delegate(c);
13       d.Close();
14   }
```

Figure 6.1: Agency.

the other peer. Each peer is equipped with a FIFO buffer storing the messages sent to that peer that have not been read yet. This means that communication is asynchronous (send operations are non-blocking) and process synchronization is realized via handshaking protocols. Channel communication is, indeed, ruled by *channel contracts*, verified at compile-time, that describe messages, message argument types, and valid message interaction sequences as finite state machines. In essence, channel contracts are a syntactic variation of session types. We now take a closer look at Sing# by means of the Customer-Agency example.

The pseudocode snippet in Figure 6.1 defines a function `Agency` that encodes the behavior of the process Agency. The function accepts two arguments: a `c` endpoint serving as one of the peers of the session channel used to interact with the Customer (which the other peer endpoint belongs to); a `d` endpoint representing one peer of the channel exploited to *delegate* `c` to the Service at the appropriate moment. Indeed, Sing# supports delegation, as it permits sending endpoints as messages (i.e., endpoints are first-class values). The `switch receive` construct (lines 3–14) is used to receive messages from an endpoint, and to dispatch the control flow to various cases depending on the kind of message that is received. Each `case` block specifies the endpoint from which a message

```
void Customer([Claims] exp<C:START> in ExHeap c,
              String destination, String address) {
   c.Query(destination);
   switch receive {
     case c.Price(double p):
       if ( /* p not ok and negotiate */ )
         Customer(c, destination, address);
       else if ( /* p not ok and reject */ ){
         c.Reject();
         c.Close();
       } else {
         c.Accept();
         c.Address(address);
         switch receive {
           case c.Date(String d):
           c.Close();
         }
       }
   }
}
```

Figure 6.2: Customer.

```
void Service([Claims] imp<D:START> in ExHeap d) {
   switch receive {
     case d.Delegate(<C:ADDRESS> in ExHeap x):
       switch receive {
         case x.Address(String a):
           /* produce a date da for a */
           x.Date(da);
           x.Close();
       }
       d.Close();
   }
}
```

Figure 6.3: Service.

is expected and the tag of the message. If a request of details is received (message `Query` on line 4), a proposal is sent (line 6), and the function `Agency` is invoked recursively (line 7), so that the negotiation can continue. In the case a `Reject` message is received (line 8), both endpoints are closed, as the negotiation failed. If an `Accept` message is received, the endpoint `c` is delegated to Service by sending it over the endpoint `d` (whose peer belongs to Service), then `d` is closed. From this point on, the communication will be between Customer and Service, the former unaware of the change of interlocutor. The operational semantics of the processes Customer and Service, encoded as functions `Customer` and `Service` and shown in Figure 6.2 and Figure 6.3, should be self-explanatory.

We now describe the meaning of the type annotations and their relevance with respect to static analysis. Static analysis of Sing# programs aims at providing strong guarantees on the absence of errors deriving from communications and the usage of heap-allocated objects. The `in ExHeap` annotation states that a name denotes a pointer to an object allocated on the exchange heap. Regarding communications, the correctness of this code fragment relies on the assumption that the process(es) using the peer endpoints of `c` and `d` are able to deal with the message types as they are received/sent from within `Agency`. To this end, Sing# provides channel contracts describing the communication patterns that are permitted on a given endpoint. Consider, for example, the contracts for the Costumer-Agency example:

```
contract C {
  message Query(String);
  message Price(double);
  message Reject();
  message Accept();
  message Address(String);
  message Date(String);
  state START { Query! → REC_PRICE;
                Accept! → ADDRESS;
                Reject! → END; }
  state REC_PRICE { Price? → START; }
  state ADDRESS { Address! → DATE; }
  state DATE { Date? → END; }
```

```
  state END { }
}

contract D {
  message Delegate(<C:ADDRESS> in ExHeap);
  state START { Delegate! → END; }
  state END { }
}
```

A contract is made of a finite set of *message specifications* and a finite set of *states* linked by *transitions*. Each message specification starts with the keyword `message`, followed by the *tag* of the message and the type of its arguments. The state of an endpoint is determined by the state of the contract associated to it. This defines which messages can be sent and received. Communication errors are ruled out by associating the two peers of a channel with complementary types, that is, describing complementary actions. This is modeled in Sing# with the `exp<C:`*s*`>` and `imp<C:`*s*`>` type constructors that, given a contract `C` and a state *s* of `C`, stand respectively for the *exporting* and *importing* views of `C` when it is in state *s*. Think of the exporting view as of the type of the *provider* of the actions specified in the contract, and of the importing view as of the type of the *consumer* of the actions described in the contract.

Going back to the Costumer-Agency example, note that Service waits for an endpoint value in `x` (Figure 6.3, line 3) that must be in the state `ADDRESS` of the contract `C`, in order to conclude the transaction with Customer correctly.

From the previous discussion, it is sensible to formalize Sing# exploiting a process calculus equipped with an apt session type system for session types. There are, indeed, analogies between contracts and endpoint types: the contract defines an interaction between two processes in terms of states and transitions, from the viewpoint of one of the two processes; the session type defines the actions of a single process taking part to the interaction; the exporting and importing views of a contract correspond to the notion of duality in session types. Bono et al. [2011] have formalized a session-oriented process calculus in which it is straightforward to encode the Costumer-Agency example.

6.2 Behavioral Types for Memory Leak Prevention

The distinctive feature of the copyless paradigm is that objects are not copied from the sender to the receiver, but, instead, pointers to allocated objects are sent around. Unfortunately, this feature may invalidate the invariant requiring that at any time each object is owned by one and only one process. Therefore, the candidate type system should pay attention to the *ownership* of the allocated objects and the fact that, whenever the pointer to an object is sent as a message, its ownership is actually moved from the sender to the receiver, too. There are two cases for ownership of parameters: either the ownership is transferred back to the caller (no annotation), or it is kept by the callee (annotation `[Claims]`), after the execution. In the example of Figure 6.1, the function `Agency` owns its two parameters and retains their ownership. In fact, either it closes both endpoints in the case of rejection of the negotiation (lines 9 and 10), or it sends endpoint `c` away (transferring the ownership to the receiver) and closes endpoint `d` in the case of acceptance (lines 12 and 13).

It might seem feasible to enforce the ownership invariants of heap-allocated objects by means of a *linear* typing discipline. However, linearity alone is *too weak* to guarantee the absence of *memory leaks*, occurring when every reference to a heap-allocated object is lost. This is illustrated in the function below:

```
void leak([Claims] imp<C:START> in ExHeap e,
          [Claims] exp<C:START> in ExHeap f)
{ e.Arg(f); e.Close(); }
```

which accepts two endpoints `e` and `f` allocated in the heap, sends endpoint `f` as an `Arg`-tagged message on `e`, and closes `e`. One of the two arguments is sent away in a message, while the other is properly deallocated within the function, hence the `[Claims]` annotations in the function header. The key observation is that this function may introduce a leak if `e` and `f` are the peer endpoints of the same channel. In this case, only the `e` endpoint is deallocated, while every pointer to `f` is lost and it will never be deallocated. The `leak` function behavior is in accordance with the Sing$^{\#}$ contract

```
contract C {
  message Arg(exp<C:START> in ExHeap);
  state START { Arg? → END; }
  state END { }
}
```

This contract shows an (apparent) anomaly, which is the implicit recursion in the type of the argument of the `Arg` message, referring to the contract `C` being defined.

An encoding of this example in the calculus by Bono et al. [2011] is straightforward:

$$\text{LEAK} = \texttt{open}(e : T, f : S).e!\texttt{Arg}(f).\texttt{close}(e)$$

where:

$$T = !\texttt{Arg}(S).\mathsf{end}$$
$$S = \texttt{rec}\ \alpha.?\texttt{Arg}(\alpha).\mathsf{end}$$

The types T and S are dual of each other and, in particular, S corresponds to the contract `C` shown above.

In order to avoid memory leaks, it might be tempting to rule out types with a recursive form such as the one of S. However, this is too restrictive, as the implicit recursion in the type of the argument of the `Arg` message is harmless. The actual problem is the fact that LEAK creates a cycle in the exchange heap: the endpoint f is stored in its own FIFO queue, as it is the argument of a message that will be never read. In order to avoid cycles of this nature, the intuition is that such a message queue containing a loop has an infinite "size". In this context, the "size" of a queue is not the number of messages in it, but rather the measure of the longest chain of pointers originating from the messages in the queue. To avoid confusion, this measure is called *weight* by Bono et al. [2011]. The type system requires endpoint types so that they only denote endpoints whose queue has finite weight.

Notably, the `leak` function is ill typed also in Sing$^{\#}$ [Fähndrich et al., 2006], even though there the motivations for rejecting `leak` arise from the implementation details of the ownership transfer rather than from the possible presence of memory leaks. Having pinpointed the actual reason why the function `leak` is faulty is the main contribution of the formalization of Sing$^{\#}$ [Bono et al., 2011].

6.3 Related Work

Fähndrich et al. [2006] address how the language, verification, and runtime system features of copyless message passing make it suited for actual use as the only mechanism of communication between processes in Singularity. The authors show, by means of an advanced programming language and appropriate verification techniques, that it is possible to yield strong system-wide invariants that endow efficient communication and low-overhead, software-based process isolation. An (informal) overview of Singularity OS specifications is given by Hunt et al. [2005].

Bono et al. [2011] present a calculus that models a form of process interaction based on copyless message passing, in the style of Singularity OS. The calculus is equipped with a type system ensuring that well-typed processes are free from memory faults, memory leaks, and communication errors. The type system is essentially linear, but linearity alone is shown not be adequate, because it does not prevent scenarios where well-typed processes leak memory. Linearity is then supported with the notion of weight discussed above, as a mean to rule out such processes.

Bono and Padovani [2012] extend [Bono et al., 2011] by adding *bounded polymorphism* to endpoint types, along the lines of [Gay, 2008], while preserving all the properties of the monomorphic version. With polymorphic endpoint types it is possible to type a (polymorphic) variant of the process LEAK without exploiting recursive types. The notion of weight extends to type variables: when α occurs in a constraint $\alpha \leqslant t$, the weight of α is approximated to the weight of t.

Stengel and Bultan [2009] show that contracts are implementable without deadlocks if they are *deterministic* and *autonomous*. The first condition states that there cannot be two transitions different only in the target state. The autonomous condition requires that every two transitions starting from the same state are either two sends or two receives. These conditions make it possible to separate contracts into pairs of dual session types.

Villard et al. [2009, 2010] study an extension of separation logic for verifying correct communications and absence of memory leaks in programs using copyless message passing in the style of Singularity OS.

Jakšić and Padovani [2012, 2014] study an extension of [Bono et al., 2011, Bono and Padovani, 2012] with exceptions. The semantics of processes is inspired to software transactional memories: a transaction is a process that must accomplish a message exchange and that should either be executed completely, or should have no observable effect if killed by an exception.

Bono et al. [2013] extend the technique presented in [Bono et al., 2011, Bono and Padovani, 2012] for detecting memory leaks into a language with first-class functions. These results are part of the stream of work on functional languages (see §3). The technique based on weights mentioned above does not work directly in a language with first-class functions. The problem is that function types only describe the function input and output, but not which other (heap-allocated) objects the function may use: this information is fundamental for defining a sound notion of weight for (linear) arrow types. The solution is to equip linear arrow types with an explicit annotation providing an upper bound to the weight of the types of all endpoints present in the function body. Once again, the weight approximates the length of chains of pointers in the heap: it is safe to send a function value over an endpoint only if its weight is bounded.

7

Web Services

7.1 Behavioral Interfaces for Web Services

Service Oriented Computing (SOC) is based on services, intended as autonomous and heterogeneous components that can be published and discovered via standard interface languages and publish/discovery protocols. Web Services is the most prominent service-oriented technology: Web Services publish their interface expressed in the Web Service Description Language (WSDL), they are discovered through the UDDI protocol, and they are invoked using SOAP.

Services are often developed as combination of other existing services, by using so-called orchestration languages, such as WS-BPEL [OASIS, 2007]: executable languages which perform activities by means of local computations combined with invocations to other services. In this context, behavioral abstractions, which can be seen as an analogous of behavioral types extracted from a program written in an orchestration language [Boreale and Bravetti, 2011], have been studied in order to reason about correctness of service composition. Examples of these languages are Abstract WS-BPEL and behavioral contracts [Fournet et al., 2004, Bravetti and Zavattaro, 2007, 2008a,b]. Such abstract languages make it possible to check whether the retrieved

services and the client invocation protocol are actually compliant/complementary. For instance, they make it possible to check whether the overall composition of the client protocol with the invoked services is stuck-free [Fournet et al., 2004], deadlock-free [Castagna et al., 2009] or successfully terminates [Bravetti and Zavattaro, 2007, 2008a,b].

Session type theories make it possible to extract such behavioral descriptions (in the form of types) from the actual service code (type inference) or to check that service code conforms to a given behavioral description (type checking). In turn, type checking crucially relies on the notion of duality (correspondence of invokes and receives), guaranteeing service compliance in an interaction involving multiple services, and on a sub-typing relation between session types (see compliance testing preorder [Bravetti and Zavattaro, 2007, 2008b]). The sub-typing relation is defined to be the coarsest one that preserves the desired termination properties, so to be as permissive as possible when typing code (we will discuss these aspects with examples in §7.3). This form of sub-typing, called semantic sub-typing, is more permissive compared to the syntactic ones commonly adopted in session types, and plays a key role in addressing the problem of service discovery. Since it is unrealistic in general to be able to find a service that matches *exactly* a given behavioral description, sub-typing enables more relaxed forms of queries whereby any service with a description that is related by subtyping (but not necessarily equal) to the requested one can be returned as result. Despite the increased expressiveness granted by such more permissive forms of sub-typing, type checking of programming languages remains feasible. Bravetti and Zavattaro [2007, 2008b] describe a decidable sound characterization of compliance testing preorder.

In order to be able to perform this kind of checks, it is necessary for services to expose in their interface also the description of their behavior (obtained, as we mentioned, by applying the type system on the service code). In general, a service interface description language used in directory services like UDDI can expose both *static* and *dynamic* information about Web Services. The former deals with the signature (name and type of the parameters) of the invocable operations; the latter deals with the correct order of invocation of the provided opera-

tions in order to correctly complete a session of interaction. The WSDL, which is the standard Web Services interface description language, is essentially concerned just with static information. Behavioral contracts provide a formal basis for generalizing these descriptions to also include the protocol to be used to successfully interact with the service. Thanks to flexibility of semantic sub-typing, we can use behavioral contracts as search keys for discovering web services possessing some desired behavior at a directory (UDDI) service (which matches services whose published behavioral contract is a sub-type of the desired one).

In the following we will deal with languages for representing concrete and abstract service orchestrations. Concrete orchestrations are presented to show how services can be programmed in terms of invocations of other services and as a starting point to then extract abstract orchestrations, used to express and reason about interaction with other services. Such abstract representations can then be used to enrich the information provided in WSDL.

7.2 Languages for Service Composition

We already mentioned WS-BPEL as an executable standard language for programming service orchestrations. Jolie (Java Orchestration Language Interpreter Engine) is a general-purpose programming language based on the Service-Oriented Computing paradigm [Montesi et al., 2014, development team]. It was originally presented by Montesi et al. [2007] as an orchestration language for Web Services alternative to the standard language WS-BPEL, with the advantage of being based on formal models from the start and consequently enabling abstract reasoning on the behavior of Jolie programs; this is in contrast with WS-BPEL, whose reference implementations are based on informal specifications. Other advantages of Jolie are that it is equipped with a friendly syntax similar to C/Java and that it integrates behavioral primitives for orchestration with architectural primitives for programming the organization of a network; the result of this integration is that these architectural primitives can be used to set up, e.g., load balancers, proxies, or monitors that can be reused independently of the orches-

tration behavior of the services that they compose, or even in settings where different communication technologies or protocols are used (e.g., HTTP instead of SOAP). Jolie is also equipped with a rather sophisticated fault handling mechanism [Guidi et al., 2009]: compensation handlers can be dynamically updated taking under consideration information available only at runtime. Moreover, if a fault occurs during a bidirectional request-response interaction, the correct interruption and compensation of both communicating processes is guaranteed. Despite Jolie was initially designed as a language for Web Services orchestration, during its development the language has evolved to a general-purpose tool that can be applied to different scenarios, from multi-core computing to web applications [Montesi, 2013a, Montesi et al., 2014].

We exemplify service composition using orchestration by implementing the Customer-Agency use case in Jolie. Our implementation includes two programs, one for the customer and one for the agency. We first discuss the program for the customer, reported below:

```
main
{
  start@Agency()(m.sid);
  satisfied = false;
  for(i = 0, !satisfied && i < 5, i++) {
    showInputDialog@SwingUI("Name")(m.product);
    askPrice@Agency(m)(price);
    showYesNoDialog@SwingUI(string(price))(answer);
    satisfied = !bool(answer)
  };
  showYesNoDialog@SwingUI
    ("Buy "+m.product+" for "+price+"?")(answer);
  satisfied = !bool(answer);
  if (satisfied) {
    accept@Agency(m);
    order@Agency(m)(date)
  } else {
    reject@Agency(m)
  }
}
```

The customer program starts by sending a message for operation `start` to `Agency`, an external reference implementing the agency. Operation

`start` will start a fresh session in the agency service, identified by a new session identifier that we expect to receive as a reply. Such session identifier is stored in the variable `m.sid`, and the data structure `m` in the rest of the code refers to the correct session inside the agency. Lines 5–10 implement the loop of the use case; here, the customer requests the price for a product at most 5 times or until she is satisfied. In line 6, an external service `SwingUI`, provided by the Jolie standard library, is used for asking the user the product she desires to purchase. In line 7, a request is sent to the agency for the price of the product the user wishes to buy; then, the user can select whether the price is acceptable or not. After the loop ends, in lines 11–12, the user is asked whether she wishes to proceed with the purchase of the last selected product. In this case, in lines 15–16 a message for the `accept` operation offered by the agency is sent and the order is placed using another message; when placing the order, the expected delivery date for the product is sent as reply. Otherwise, if the user does not wish to proceed, the operation `reject` is invoked on the agency and the session terminates (line 18).

Below is the code for the agency:

```
1  main
2  {
3    start()(csets.sid) { csets.sid = new };
4    continue = true;
5    while(continue) {
6      [ askPrice(m)(double(price)) {
7        showInputDialog@SwingUI(m.product)(price)
8      } ] { nullProcess }
9      [ accept(m) ] {
10       order(m)(date) {
11         date = string(int(m.product))
12       };
13       continue = false
14     }
15     [ reject(m) ] {
16       continue = false
17     }
18   }
19 }
```

The agency is willing to start a new session when invoked on operation `start`; when such operation is invoked, in line 3, the agency creates a new session identifier `csets.sid` and sends it back to the invoker. Thereafter, the agency enters a loop in which it offers three possibilities (expressed by the input choice construct `[ ] { } ... [ ] { }`). If the customer invokes the operation `askPrice`, then the agency calculates the price for the received product, sends it back to the invoker, and continues in its loop. The loop terminates only when either operation `accept` or operation `reject` is invoked; in the first case, the agency also waits for an order request and sends back the expected delivery date (here calculated with a toy example); otherwise, the loop simply terminates.

7.3 Abstract Service Descriptions and Behavioral Contracts

The orchestration language WS-BPEL can be used to describe so-called abstract processes, that is behavioral descriptions that include unspecified parts, hence may represent just the externally visible communicating behavior of a service. Such Abstract WS-BPEL representations are not meant to be executable: they can be exposed to the service users in order to determine how to successfully interact with it.

As we already mentioned, using a process algebraic approach, it is possible to define how to extract the externally observable behavior (behavioral contract/session type) from the actual executable behavior of a service [Boreale and Bravetti, 2011].

The achieved abstraction, expressed in a process algebraic language similar to Abstract WS-BPEL, is then enough informative to enable analysis of certain properties of the actual service (when interacting with other services), including stuck freedom [Fournet et al., 2004], deadlock freedom [Castagna et al., 2009], termination (under fairness assumptions) [Bravetti and Zavattaro, 2007, 2008a,b]. In particular, such analysis is often carried out by resorting to more low-level semantic descriptions of service behaviors (essentially labeled transition systems) called *behavioral contracts*. One of the most important aspects of the service contract technology is considered to be *correctness*

of composition or, more simply, *compliance*: given any set of services, it should be possible to prove that their composition is correct (according to the above mentioned termination properties) knowing only their contracts, i.e. in the absence of complete knowledge about the internal details of the services behavior.

We exemplify abstract process representation by providing the description of the Customer-Agency use case with abstract WS-BPEL. In order to avoid writing obscure and verbose XML code we adopt the more intuitive notation of BPELscript. The representation of the Customer is the following (`A` denotes the agency service, `C` the customer service).

```
while ( !satisfied(price) ) {
  price=askPrice (A);
};
if (confirm(price)) {
  accept (A); date=order (A);
} else
  reject (A);
```

Notice that functions *satisfied* and *confirm* are underspecified and, thus, assumed to non-deterministically yield a boolean return value. The representation of the Agency is the following.

```
continue = true;
while (continue) {
  pick {
    onMessage(C,askPrice) {reply(C,askPrice,price);}
    onMessage(C,accept) {
      receive(C,order);
      reply(C,order,date);
      continue = false; }
    onMessage(C,reject) { continue = false; }
  }
}
```

Again, the generation of *price* and *date* values is underspecified.

This can be seen as the behavioral abstraction of the Jolie code given in the previous section, where, also, the $i < 5$ constraint in the *for* loop is disregarded. Notice that, in order to reason about such service abstraction, we can resort to a behavioral contract, i.e., a more semantic

notation which expresses the service behavior, in an even more abstract way, merely in the form of (finite-state) transition systems, where labels are of the kind $\{\overline{a}_l, a, \tau\}$ representing invocations of operation a on service l, reception of a message on operation a (inputs do not have a service subscript), and internal computations τ.

For example, using the familiar notation of regular expressions, we can specify the Customer service contract as

$$(\overline{askPrice}_A; price)^*; (\overline{accept}_A; \overline{order}_A; date) + \overline{reject}_A\,,$$

and the Agent service contract as

$$(askPrice; \overline{price}_C)^*; (accept; order; \overline{date}_C) + reject\,.$$

Such contracts can be also derived by projection from the choreographic description given in §8.1.

An important topic in this regard is service substitutability. It is a fundamental notion in behavioral contract theory and corresponds to, so-called, contract refinement (subcontract relation). Such a notion permits to determine when, given the contract describing an expected service behavior, a given service can be used to play that role, based on its contract. Intuitively, a contract C' refines a contract C if any C' is compliant (successfully interacts) with any environment (set of contracts of other services) which is compliant with C. As we already mentioned, in the context of session types, where behavioral descriptions are used as types for actual service code, compliance is guaranteed by duality of types and contract refinement corresponds to the definition of sub-typing. It is thus quite immediate to observe that one of the main challenges is to define contract refinement so that it is the coarsest possible pre-order that preserves the desired termination properties, so to be as permissive as possible when typing checking code against a given type.

In the following we show, with a couple of examples, how underlying contracts can be used to reason about service compositions.

It is not difficult to see that the parallel composition of the behavioral contracts above for the *Customer* and the *Agency* is a correct service composition: it is both stuck-free in the sense of [Fournet et al.,

2004] and always leads to termination of all interacting contracts (assuming fairness) [Bravetti and Zavattaro, 2007, 2008a,b].

It is also interesting to observe that in the Customer service we can establish a maximum number of invocations to the *askPrice* service without breaking the correctness of the system:

```
i=0;
while (!satisfied(price) && i<5) {
  price=askPrice (A);
  i++;
};
if (confirm(price)) {
  accept (A);
  date=order (A);
} else
  reject (A);
```

This can be seen as the behavioral abstraction of the Jolie code for the Customer given in the previous section.

The corresponding behavioral contract turns out to be the labeled transition system denoted, e.g., by the regular expression E_1, where

$$\begin{array}{rcl} E_i & = & (\overline{askPrice}_A; price); E + E_{i+1} \quad \text{for} \quad 1 \leq i \leq 4 \\ E_5 & = & (\overline{askPrice}_A; price); E \end{array}$$

and, finally, E is the following

$$E = (\overline{accept}_A; \overline{order}_A; date) + \overline{reject}_A$$

According to the the theory in [Bravetti and Zavattaro, 2007, 2008a,b], the contract for this service is a refinement of the contract for the previous unbounded Customer service (because it is compliant with any environment which is compliant with the unbounded Customer service).

On the contrary, the contract for the unbounded Customer service is not a refinement of the contract for this service because there exists a context for which it is not a correctness preserving substitute. Consider, for instance, an Agency service that can only perform the pick activity for 5 cycles: it would cause the Customer service to get stuck (and not to reach termination).

Finally we show a substitute service of the original agency service. For instance, the following alternative agency service gives rise to a refinement of the contract of the one above.

```
continue = true;
while (continue) {
  pick {
    onMessage(C,askPrice) { reply(C,askPrice,price); }
    onMessage(C,accept) {
      receive(C,order);
      reply(C,order,date);
      continue = false; }
    onMessage(C,loan) {continue = false; }
    onMessage(C,reject) {continue = false; }
  }
}
```

The corresponding behavioral contract is the labeled transition system denoted by the regular expression:

$$(askPrice; \overline{price}_C)^*; (accept; order; \overline{date}_C) + loan + reject$$

The reason for originating a refined contract is that it simply differs for an additional input on the *loan* channel, modeling the possibility for the Agency to receive a loan request.

7.4 Related Work

Concerning inclusion of abstract service descriptions in WSDL, for instance SAWSDL [Kopecký et al., 2007] provides a mechanism for adding semantic annotations in WSDL.

Other kind of checks can be obtained by enriching information included in WSDL descriptions. For instance, Allison et al. [2012] deal with negotiation between a web service requester and a web service provider. The negotiation is performed for privacy reasons (i.e., the requester specifies privacy preferences that should be met by the provider). Specifically, the policy languages employed in such negotiations are relevant to WG3 (e.g., eXtensible Access Control Markup Language - XACML).

We have already remarked that the availability of repositories of Web service descriptions enables forms of dynamic Web service discovery using contracts as search keys. Sometimes, the services available in a repository expose a contract that is quite similar to the searched one, but is not a refinement according to the behavioral preorders/equivalences that have been mentioned throughout this chapter. For example, it may expose unnecessary operations or it may perform the required operations in an order that differs from the required one. In these cases, it is sometimes possible to devise a *mediator process* – a simple form of *orchestrator* – that adapts the behavior of the actual service to the one required in a particular context. Castagna et al. [2009] and Padovani [2009, 2010] investigate these aspects defining the semantics of such mediators in terms of weaker behavioral equivalences. In this way, the orchestrator that realizes the adaptation plays the role of an explicit coercion that can be automatically synthesized.

Unlike traditional testing preorders [De Nicola and Hennessy, 1984], the definition of compliance testing (that is semantic sub-typing) requires both the test and the system under test to succeed [Bravetti and Zavattaro, 2007, 2008b]. The complete characterization of compliance testing remains an open problem. Recent work in this direction has been done by Bernardi and Hennessy [2013], where however compliance testing is characterized only for controllable processes (i.e. processes for which there exists a compliant test) and, differently from [Bravetti and Zavattaro, 2007, 2008b], fairness assumption is not considered. This is practically convenient in order not to discard looping interactions, but technically more complex to deal with [Rensink and Vogler, 2007].

8

Choreographies

Choreographies are syntactic descriptions of the overall coordination of a system, in terms of interactions between autonomous principals. A choreography captures how two or more endpoints (or nodes) exchange messages during execution from a global viewpoint, instead of a collection of programs that define individually the behavior of each endpoint. As an example of a choreography, consider the following pseudo-code (whose syntax is a variant of the "Alice and Bob" security protocol notation from Needham and Schroeder [1978]):

1. Customer -> Agency : *product*;
2. Agency -> Customer : *price*

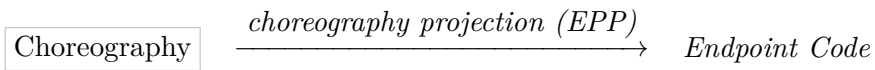
The choreography above describes the behavior of two endpoints, Customer and Agency: Customer sends to Agency a *product* name (line 1); then, Agency replies to Customer with the *price* of the product she requested (line 2).

The following discusses two approaches to developing communication-based software using choreographies. In §8.1, choreographic programming is a paradigm where programmers write a choreography to generate system that is "safe by design", since it describes directly the intended communications in the system: a

choreography can be seen as the formalization of the communication flow intended by the programmer. Moreover, each communication is treated as atomic: the sending and receiving actions of the respective sender and receiver endpoints cannot be seen separately, preventing typical concurrency bugs such as deadlocks. In §8.2, a choreography is treated as a global specification of an asynchronous communication protocol that is used to verify, either statically through type checking or dynamically through decentralized run-time monitoring, the conformance of each endpoint process to the intended protocol.

8.1 Choreography Programming Languages

A recent line of research advocates the development of safe distributed systems with Choreographic Programming, a programming paradigm in which developers write system implementations using choreographies. The executable code for each endpoint (which we call endpoint code) is then automatically projected from a choreography, using a procedure known as Endpoint Projection (EPP). The key idea is to formally prove that the definition of EPP is correct, i.e., it preserves the intended behavior of a projected choreography in the produced endpoint code; in other words, executing the endpoint code produced by EPP leads exactly to the communications defined in the originating choreography. This property enables a development methodology in which developers write a choreography and then distributed software implementing the choreography is automatically generated. We can depict such methodology thus:



The main aspect of the methodology above is that the produced endpoint code is safe by construction: since the EPP procedure is correct, it follows that the communications defined by the programmer are implemented faithfully and without errors. Such methodology has been investigated in many theoretical works, e.g., by Carbone et al. [2006], Qiu et al. [2007], Bravetti and Zavattaro [2007], Lanese et al. [2008].

Mendling and Hafner [2005] informally discuss how to project choreographies to endpoint code using the real-world choreography language WS-CDL [CDL] and the endpoint process language WS-BPEL [OASIS, 2007]. A formalization of WS-CDL is provided by Carbone et al. [2012]. Montesi and Yoshida [2013] show how choreography models can be extended to support the integration of (i) choreographies developed separately and (ii) choreographies with externally provided services that have been developed using the typical programming of endpoint programs. Carbone et al. [2014] investigate the logical foundations of choreographies by using linear logic [Girard, 1987], from which they derive a procedure for inferring the choreography implemented by an arbitrary system of endpoint programs as long as these programs can be typed using linear logic as in [Caires and Pfenning, 2010].

Currently, the most renown implemented choreography languages are WS-CDL [CDL] and BPMN [BPMN], which do not come with a behavioral typing discipline. More recently, Carbone and Montesi [2013] have proposed a choreographic programming model for the development of distributed systems based on multiparty sessions and asynchronous messaging that can be checked for respecting protocols specified as multiparty session types [Honda et al., 2008]. We depict such methodology below:

Building on the model proposed by Carbone and Montesi [2013], the Chor language offers an Integrated Development Environment (IDE) based on Eclipse for developing systems with the methodology above [Chor].

Example. Below, we report an implementation of the Customer-Agency example in the Chor language.

```
1 program customer_agency;
2
3 protocol PurchaseProtocol {
```

```
4     Customer -> Agency: askPrice(string);
5     CheckPrice
6   }
7
8   protocol CheckPrice {
9     Agency -> Customer: price(int);
10    Customer -> Agency: {
11    askPrice(string);
12      CheckPrice,
13    accept(void);
14      Customer -> Agency: order(string);
15      Agency -> Customer: date(string),
16    reject(void)
17    }
18  }
19
20  public agency_url: PurchaseProtocol
21
22  define checkPrice(c,a)(s[CheckPrice:c[Customer],
23                           a[Agency]])
24  {
25    ask@a(prod,price);
26    a.price -> c.price: price(s);
27    ask@c(price,satisfied);
28    if(satisfied == "Yes")@c {
29      ask@c("Confirm?",confirm);
30      if (confirm == "Yes")@c {
31        c -> a: accept(s);
32        c.prod -> a.prod: order(s);
33        ask@a(prod,date);
34        a.date -> c.date: date(s)
35      } else {
36        c -> a: reject(s)
37      }
38    } else {
39      ask@c("Product Name",prod);
40      c.prod -> a.prod: askPrice(s);
41      checkPrice(c,a)(s)
42    }
43  }
44
45  main
```

```
46 {
47   c[Customer] start a[Agency]: agency_url(s);
48   ask@c("Product Name",prod);
49   c.prod -> a.prod: askPrice(s);
50   checkPrice(c,a)(s)
51 }
```

The program starts by defining `PurchaseProtocol`. In Chor, protocols are behavioral types describing the structure of the communication flow between some roles, `Customer` and `Agency` in this case. In protocol `PurchaseProtocol`, role `Customer` sends a message to role `Agency` on operation `askPrice`, asking the price for a product; then, the protocol proceeds as protocol `CheckPrice`. In protocol `CheckPrice`, the agency sends the price for the product to the customer. The customer then selects one of three available choices on the agency: (i) ask again for the price of another product, in which case the protocol recurs; (ii) accept the price and order the product, in which case the agency replies with a delivery date; (iii) reject the price, in which case the protocol terminates.

After the protocol definitions is the choreography adhering to them. Procedure `main` is the choreography entry-point of execution. A session `s` between a customer process `c` and an agency process `a` is started (line 47). The two processes synchronize on the public URL `agency_url`. The customer internally computes (by asking its user through a user interface) the product `prod` to buy (line 48); then, it asks the agency for the price of the product (line 49) and the whole system proceeds as defined by procedure `checkPrice`.

Procedure `checkPrice` implements protocol `CheckPrice`. The procedure is declared together with the processes and sessions it uses, respectively `c,a` and `s` (lines 22–23). All parameters are behaviorally typed, indicating which protocol each session should implement and which role each process plays in the sessions. In this case, the declaration states that session `s` implements protocol `CheckPrice` using process `c` as the customer and process `a` as the agency. The body of `checkPrice` follows the structure of the protocol `CheckPrice` (lines 25–42). Note the usage of concrete data values and the conditional construct `if` since this is actual code, not merely a protocol description.

Using Chor, the choreography above can be automatically translated into an executable implementation in the Jolie language [development team] equivalent to that presented in §7. Observe that Chor uses multiparty session types [Honda et al., 2008] as protocol specifications and repetition is thus expressed through recursion, while the notation used in §7, inspired from [Lanese et al., 2008, Bravetti and Zavattaro, 2013], used the Kleene star.

Implementation challenges. We use Chor and our example to describe three challenges that are encountered in the development of languages for choreographic programming. We refer the reader to [Montesi, 2013b] and [Carbone and Montesi, 2013] for more detailed descriptions (especially regarding session mobility, which is not discussed here).

The first challenge is supporting session communications. In general, a session may have many participating processes, which need to be able to communicate with each other at runtime. In Chor, a session among some processes is created using primitive `start`. In line 46 of our example, the processes `c` and `a` are involved in the creation of a new session `s` through the public channel `agency_url`. This is a high-level abstraction that may be implemented differently, depending on the target technology. Since Chor uses a service-oriented language such as Jolie, the public channel is actually implemented as an always-available service that can be used by processes in the network to create new sessions [Montesi, 2013b] (each public channel has its own service implementation). Such start service implements a variant of the two-phase commit protocol for synchronising all the processes involved in the session, and communicate to all of them the necessary binding information (e.g., IP addresses) for reaching each other. This binding information also contains correlation data, a generalisation of session tokens inspired by WS-BPEL [OASIS, 2007].For each role that a process implements in a choreography, the Chor compiler generates a separate correlation set for the code of the process. Each correlation set identifies a separate message queue managed by the process; at runtime, messages for different sessions that the process is involved in will be stored in separate queues, allowing the process to distinguish where

messages come from and consume them as specified by the original choreography.

The second challenge is the integration with existing paradigms. Language models for choreographies typically focus on minimality and give only a high-level description of how the code compiled from a choreography should behave. Typically, these models are inspired to the π-calculus [Sangiorgi and Walker, 2001] in order to facilitate their formal investigation. In practice, these high-level descriptions must be suitably mapped to concrete executable code found in the target implementation language. For instance, session communications in Chor always include the name of the operation invoked by the sender (e.g., `price`, in line 25). This is not the case in the theoretical model that Chor follows. However, all communications in service-oriented computing happen over operations and thus this change is a necessary addition when compiled code is to be executed in a service-oriented architecture. Different modifications would be needed for other paradigms; in general, they should be made with particular care, since any changes to the original theoretical models of choreographic programming risk breaking their safety properties (e.g., deadlock-freedom).

Lastly, the third challenge is about reliability. To the best of our knowledge, all implemented choreography languages work on the assumption that communications will succeed (the network is "perfect") and execution units never fail. Therefore, the safety properties of choreographies are guaranteed only if these assumptions are not broken at runtime. We envision that much future work in the area will go towards creating choreography models where nodes and communications may fail during execution. Such models may include primitives that allow programmers to specify what should happen in case of failure, possibly including the notion of time. Another strategy could be to develop a dedicated middleware that monitors the execution of a choreography in an unreliable environment and makes it reliable by automatically managing and restarting the resources that it needs.

8.2 Scribble

The Scribble project [Honda et al., 2014, 2011] is a collaboration between session types researchers and architects and engineers from industry towards the application of session types principles and techniques to current engineering practices. Building on the theory of multiparty session types (MPST) [Honda et al., 2008, Bettini et al., 2008b], this ongoing work tackles the challenges of adapting and implementing session types to meet real-world usage requirements. This section gives an overview of the current version of the Scribble framework for the MPST-based development of distributed software. In the context of Scribble, we use the terms *session* and *conversation* interchangeably.

The main elements of the Scribble framework are as follows.

The Scribble language is a platform-independent description language for the *specification* of asynchronous, multiparty message passing protocols [Honda et al., 2014, 2011]. Scribble may be used to specify protocols from both the global (neutral) perspective and the local perspective of a particular participant (abstracted as a role); at heart, the Scribble language is an engineering incarnation of the notation for global and local types in formal MPST systems and their correctness conditions.

The Scribble Conversation API provides the local communication operations for *implementing* the endpoint programs for each role natively in various mainstream languages. The current version of Scribble supports Java and Python Conversation APIs with both standard socket-like and event-driven interfaces for initiating and conducting conversations.

The Scribble Runtime is a local platform library for executing Scribble endpoint programs written using the Conversation API. The Runtime includes a conversation monitoring service for *dynamically verifying* [Hu et al., 2013, Bocchi et al., 2013, Neykova et al., 2013, Demangeon et al., 2015] the interactions performed by the endpoint against the local protocol for its role in the conversation. In addition to internal monitors at the endpoints, Scribble

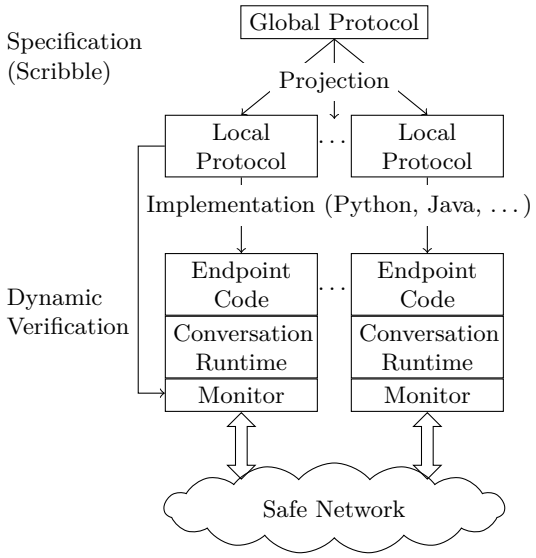


Figure 8.1: The Scribble framework for distributed software development.

also supports the deployment of external conversation monitors within the network [Chen et al., 2012].

In addition to the generic Java/Python Conversation APIs, it is also possible to generate protocol-specific endpoint APIs for the target language from a source protocol [Hu and Yoshida, 2016].

The Scribble framework combines these elements to promote the MPST-based methodology for distributed software development depicted in Figure 8.1. Below, we first illustrate an example protocol specification in the Scribble language, and then describe the stages of the Scribble framework, explaining the design challenges of applying session types to practice and recent research threads motivated by this work [Scribble].

Online Travel Agency example To demonstrate Scribble as a multiparty session types language, Figure 8.2 lists the Scribble specification of the global protocol for an extended version of the running Online Travel Agency example. If Customer decides to accept a travel quote

```
module TravelAgency;

type <java> "java.lang.Double" from "rt.jar" as Double;
type <java> "java.lang.String" from "rt.jar" as String;
type <java> "java.util.Date" from "rt.jar" as Date;

// C = Customer, A = Agent, S = Service
global protocol BookJourney(role C, role A, role S) {
  rec LOOP {
    choice at C {
      query(journey:String) from C to A;
      price(Double) from A to C;
      info(String) from A to S;
      continue LOOP;
    } or {
      choice at C {
        ACCEPT() from C to A;
        ACCEPT() from A to S;
        Address(String) from C to S;
        (Date) from S to C;
      } or {
        REJECT() from C to A;
        REJECT() from A to S;
} } } }
```

Figure 8.2: Scribble specification of a global protocol for the Online Travel Agency use case.

from Agency, the exchange of address details and the ticket dispatch date is now conducted between Customer and Service, representing the transport service being brokered by Agency. The Scribble is read as follows:

- The first line declares the Scribble module name. Although this example is self-contained within a single module, Scribble code may be organized into a conventional hierarchy of packages and modules. Importing payload type and protocol declarations between modules is useful for factoring out libraries of common payload types and subprotocols.

- The design of the Scribble language focuses on the specification of protocol *structures*. With regards to the payload data that may be carried in the exchanged messages, Scribble is designed to work orthogonally with external message format specifications and data types from other languages. The `type` declaration on Line 3 declares a payload type based on Java object serialization format, specifically `java.lang.Double` objects, whose definition (i.e. class) is to be imported from the file `rt.jar`, and is aliased as `Double` within this Scribble module. Data type formats from other languages, as well as XML or various IDL based message formats, may be used similarly. A single protocol definition may feature a mixture of message types defined by different formats.

- Line 8 declares the signature of a global protocol called `BookJourney`. This protocol involves three roles: Customer (`C`), Agency (`A`), and Service (`S`).

- Lines 9–24 define the interaction structure of the protocol. Line 11 specifies a basic message passing action. `query(journey:String)` is a message signature for a message with header (label) `journey`, carrying one payload element within the parentheses. A payload element is an (optional) annotation followed by a colon and the payload type, e.g. `journey` details are recorded in a `String`. This message is to be dispatched by `C` to be received by `A`.

- The outermost construct of the protocol body is the `rec` block with label `Loop`. Similarly to labeled blocks in e.g. Java, the occurrence of a `continue` for the same label within the block causes the flow of the protocol to return to the start of the block. The first `choice` within the `rec`, decided by `C`, is to obtain another quote (lines 11–14: send `A` the `query` details, receive a `price`, and `continue` back to the start), or to accept/reject a quote. The latter is given by the inner `choice`, with `C` sending `ACCEPT` to `A` in the first case and `REJECT` in the second. In the case of `ACCEPT` (lines 17–20), `A` forwards the confirmation to `S` before `C` and `S` exchange `Address` and `Date` messages; otherwise, `A` forwards the `REJECT` to `S` instead.

The Scribble framework. The Scribble development workflow starts from the explicit specification of the required global protocols (such as `BookJourney` above), similarly to the existing, informally applied approaches based on prose documentation, such as Internet protocol RFCs, and common graphical notations, such as UML and sequence diagrams. Designing an engineering language from the formal basis of MPST faces the following challenges.

- To developers, Scribble is a new language to be learned and understood, particularly since most developers are not accustomed to formal protocol specification in this manner. For this reason, we have worked closely with our collaborators towards making Scribble protocols easy to read, write and maintain. Aside from the core interaction constructs that are grounded in the original theory, Scribble features extensions for the practical engineering and maintenance of protocol specifications, such as subprotocol abstraction and parameterized protocols [Honda et al., 2014] (demonstrated in the examples below).

- As a development step (as opposed to a higher-level documentation step), developers face similar coding challenges in writing formal protocol descriptions as in the subsequent implementation steps. IDE support for Scribble and integration with other devel-

```
module TravelAgency_BookJourney_C;

type <java> "java.lang.Double" from "rt.jar" as Double;
type <java> "java.lang.String" from "rt.jar" as String;
type <java> "java.util.Date" from "rt.jar" as Date;

local protocol BookJourney_C(self C, role A, role S) {
  rec LOOP {
    choice at C {
      query(journey:String) to A;
      price(Double) from A;
      continue LOOP;
    } or {
      choice at C {
        ACCEPT() to A;
        Address(String) to S;
        (Date) from S;
      } or {
        REJECT() to A;
} } } }
```

Figure 8.3: Scribble local protocol for `Customer` projected from the `BookJourney` global protocol.

opment tools, such as the Java-based tooling in Red Hat JBoss, are thus important for developer uptake.

- Although session types have proven to be sufficiently expressive for the specification of protocols in a variety of domains, including standard Internet applications [Hu et al., 2010], parallel algorithms [Ng et al., 2012], actor distributed applications [Neykova and Yoshida, 2014] and Web Services [CDL], the evaluation of Scribble through our collaboration use cases has motivated the development of new multiparty session type constructs, such as asynchronous conversation interrupts [Hu et al., 2013] (demonstrated below) and subsession nesting [Demangeon and Honda, 2012], which were not supported by the pre-existing theory.

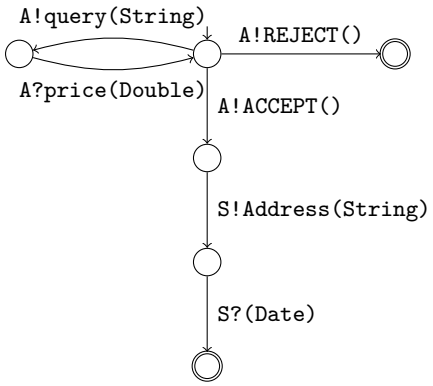


Figure 8.4: FSA generated from the local protocol by the Scribble conversation monitor.

After the specification of the global protocols, the next step of the Scribble framework (Figure 8.1) is the *projection* of local protocols from the global protocol for each role. In comparison to languages implemented from binary session types, such as SJ and Mungo (§2.2) or Sing$^{\#}$ (Chapter 6), this additional step is required to derive local specifications for the endpoint implementation of each role process from the central global protocol specification. Scribble projection follows the standard MPST algorithmic projections, with extensions for the additional features of Scribble, such as the subprotocols and conversation interrupts mentioned above.

Figure 8.3 lists the local protocol generated by the Scribble tools as the projection of the `BookJourney` for the `Customer` role, identified in the local protocol signature as the `self` role. Projection preserves the dependencies of the global protocol, such as the payload types used, and the core interaction structures in which the target role is involved, e.g. the `rec` and `choice` blocks, as well as payload annotations and similar protocol details. The well-formedness conditions on global protocols allow the projection to safely discard all message actions not involving `C` (i.e. messages between `A` and `S`).

As for the binary session languages cited above, it is possible to statically type check role implementations written in endpoint languages

with appropriate MPST programming primitives against the local protocols following the standard MPST theory: if the endpoint program for every role is correct, then the correctness of the whole multiparty system is guaranteed. The endpoint languages used in the Scribble industry projects, however, are mainstream engineering languages like Java and Python that lack the features, such as first-class communication channels with linear resource typing or object alias restriction, required to make static session typing feasible. In Scribble practice, the Conversation API is used to perform the relevant conversation operations natively in these languages, making static MPST type checking intractable. In general, distributed systems are often implemented in a mixture of languages, including dynamically typed languages (e.g. Python), and techniques such as event-driven programming, for which the static verification of strong safety properties is acknowledged to be difficult.

For these reasons, the Scribble framework, differently from the above session languages, is designed to focus on *dynamic verification* of endpoint behavior [Hu et al., 2013]. Endpoint monitoring by the local Conversation Runtime is performed by converting local protocols to communicating finite state automata, for which the accepted languages correspond to the I/O action traces permitted by the protocol. The conversion from syntactic Scribble local protocols to FSA extends the algorithm in [Deniélou and Yoshida, 2012] to support subprotocols and interrupts, and to use nested FSM for parallel conversation threads to avoid the potential state explosion from constructing their product. Figure 8.4 depicts the FSA generated by the monitor from the `Customer` local protocol. The FSA encodes the control flow of the protocol, with transitions corresponding to the valid I/O actions that `C` may perform at each state of the protocol.

Analogously to the static typing scenario, if every endpoint is monitored to be correct, the same communication-safety property is guaranteed [Bocchi et al., 2013]. In addition, since the monitor verifies both messages dispatched by the endpoint into the network and the messages inbound to the endpoint from the network, each conversation monitor is able to protect the local endpoint within an untrusted network and

```
1  global protocol CustomerOptions(role C, role A, role S) {
2    choice at C {
3      do GetQuote(C, A, S);
4    } or {
5      do Forward<ACCEPT()>(C, A, S);
6      do ServiceCall<Address(String), (Date)>(C, S);
7    } or {
8      do Forward<REJECT()>(C, A, S);
9  } }
10
11 global protocol GetQuote(role C, role A, role S) {
12   do ServiceCall<query(String), price(Int)>(C, A);
13   info(String) from A to S;
14   do CustomerOptions(C, A, S);
15 }
16
17 global protocol ServiceCall<sig Arg, sig Res>(role C, role S) {
18   Arg from C to S;
19   Res from S to C;
20 }
21
22 global protocol Forward<sig M>(role X, role Y, role Z) {
23   M from X to Y;
24   M from Y to Z;
25 }
```

Figure 8.5: Decomposition of the `BookJourney` global protocol using parameterised subprotocols

vice versa. The internal monitors embedded into each Conversation runtime function perform synchronous monitoring (the actions of the endpoint are verified synchronously as they are performed); Scribble supports mixed configurations between internal endpoint monitors and asynchronous, external monitors deployed within the network (as well as statically verified endpoints, where possible) [Chen et al., 2012].

Further examples The following gives two further examples to demonstrate additional features of Scribble motivated by application in practice.

```
1 global protocol InterruptibleServiceCall(role C, role S) {
2   Arg from C to S;
3   interruptible {
4     Res from S to C;
5   } with {
6     cancel() by C;
7 } }
```

Figure 8.6: Revision of the `ServiceCall` global protocol with a request cancel interrupt

The first example demonstrates the abstraction of protocol declarations as *subprotocols*, and the related feature of *parameterised* protocol declarations. Figure 8.5 gives an alternative specification for the Travel Agency example that is decomposed into four smaller global protocols.

`ServiceCall` specifies a generic call-return pattern between a `Client` and a `Server`. The message signatures of the two communications are abstracted by the `Arg` and `Res` parameters, declared by the `sig` keyword inside the angle brackets of the protocol signature.

`Forward` specifies a generic forwarding pattern between three roles, from `X` to `Y` and then `Y` to `Z`. The intent is for `Y` to forward a copy of the same message, so the signatures of the two communications are abstracted by the same `M` parameter.

`CustomerOptions` is the main protocol in this version of the Travel Agency specification, with the same signature as `BookJourney` in Figure 8.2. It starts with the `choice` of `C` to get another quote, accept a quote, or reject. The main interactions are now built by composing instances of the `Forward` and `ServiceCall` subprotocols. For example, `do Forward<ACCEPT()>(C, A, S)` on line 5 states that the `Forward` protocol should be performed with the target roles `X`, `Y` and `Z` played by `C`, `A` and `S`, respectively, and `ACCEPT()` as the concrete message signature in place of the `M` parameter; `C` sends `ACCEPT` to `A`, who forwards it to `S`. After this, `C` and `S` engage in a `ServiceCall` subprotocol to exchange the `Address` and `Date` messages.

`GetQuote` performs the quote query case of the choice between `C` and `A`, and loops back to the overall start of the protocol. The quote exchange is specified by instantiating the `ServiceCall` with the appropriate role and message signature parameters. To return to the start of the protocol, we recursively `do` the main protocol `CustomerOptions`. The loop is thus specified by the mutual recursion between these two protocol declarations.

The final example demonstrates the Scribble feature for asynchronously interruptible conversations. Unlike the previous features, which involve the integration of session types with useful, general programming language features (code abstraction and parameterization), conversation interrupts require extensions to the core design of session types [Hu et al., 2013]. The motivation for interrupts comes from our collaboration use cases, featuring patterns such as asynchronously interruptible streams and interaction timeouts,which could not be directly expressed in the standard MPST formulations.

Figure 8.6 gives a very simple revision of the `ServiceCall` protocol that allows the `Client` to `cancel` the call by interrupting the `Server`'s reply. A key design point is that interruptible conversation segments do not incur any additional synchronization overhead from the explicit messaging actions (i.e. interrupts are themselves communicated as regular messages). Due to asynchrony between `C` and `S`, the interrupt can cause various communication race conditions to arise, e.g. `C` sending `cancel` before `S` processes the initial `Arg` or after `S` has already dispatched the `Res`. The Scribble Runtime is designed to handle these issues by tracking the progress of the local endpoint through the protocol (as part of the monitoring service). This allows the Runtime to resolve the communication races by discarding messages that are no longer relevant due to the local role raising an interrupt or receiving an interrupt message from another role.

8.3 Related Work

The development of the Scribble framework and its application in real-world use cases is ongoing work. The two main use case projects men-

tioned in the above are:

Savara is a JBoss project developed by Red Hat.Savara relies on Scribble as an intermediate language for representing protocols, to which high-level notations, such as BPMN2, are translated to perform endpoint projections and various refactoring tasks. Savara provides a suite of tools for testing of service specifications against the initial project requirements. The testing is based on simulations between the former, represented in Scribble, and the latter, expressed as sequence diagram traces.

The Ocean Observatories Initiative is an NSF-funded project to develop the infrastructure for the remote, real-time acquisition and delivery of data from a large sensor network deployed in ocean environments to users at research institutions. The Scribble framework, including Conversation Runtime monitoring, has been integrated into the Python-based OOI platform. So far, the OOI cyberinfrastructure is mainly running on an RPC-based architecture. The current Scribble integration is accordingly primarily used for the specification of RPC service and application protocols, and the dynamic verification of the Python client/server endpoints.

Below, we summarize some of the active threads in regards to these projects.

- The Savara project is examining formal encodings between the specification languages commonly used in practice and Scribble (the current translation by Savara is not yet formalized), which is motivating further extensions to Scribble, such as dynamic introduction of roles during a conversation and fork-join conversation patterns. In general, adapting MPST and Scribble to graphical representations will increase the expressiveness of the protocol specification language. Using the native semantics of formal graphical formats for concurrency, such as communication automata [Deniélou and Yoshida, 2012] and Petri nets [Fossati et al., 2014], to provide global execution models of conversations is an

interesting direction for integrating Scribble protocol specifications with specifications of other system aspects, such as internal endpoint workflows.

- The current phase of the OOI project includes the development of a framework for actor-based interactions over the existing service infrastructure. To support the specification and verification of higher-level application properties above the core message passing protocol, Scribble is being extended with a framework for annotating protocols with assertions and policies in third-party languages. Annotations may be associated to individual messages, interaction steps, control flow structures, roles or protocols as a whole; examples range from basic constraints on specific message values and control flow (e.g. recursion bounds) to more complicated logics for security or contractural obligations of roles. The Scribble framework will accept plugins for parsing and projecting the annotation language, and evaluating the annotations at run-time. This allows the Scribble tools and monitors to be extended modularly with application- and domain-specific annotations, and the dynamic verification approach enables the enforcement of properties that would be difficult or impossible to verify statically without conservative restrictions.

- The Savara and OOI use cases implement the Scribble language, meaning the syntax, well-formedness (valid protocol) conditions and projections, as defined by the central language reference. Both implementations also necessarily conform to baseline communication model of Scribble, namely asynchronous but reliable and role-to-role ordered messaging. The Scribble project is currently working on defining an accompanying Conversation Runtime specification. This will provide the reference for Scribble runtime libraries and platforms, including the specification of the key system protocols for conversation initiation, message formats (conversation and monitoring message meta data) and more advanced features such as conversation delegations [Hu et al., 2008]. This work is towards full interoperability of Scribble endpoints

running on different platforms, such as the Java and Python platforms of the above use cases, supported by platform-independent monitoring. This interoperability will also extend to safely combining dynamically and statically verified endpoints within conversations.

As a converse of choreographic programming, the construction of graphical choreographies, or global graphs, is also a research topic of interest. Global graphs are general multiparty session specifications featuring expressive constructs such as forking, merging, and joining for representing application-level protocols. Global graphs can be directly translated into modelling notations such as BPMN and UML. The approach has been investigated by Deniélou and Yoshida [2013], Lange et al. [2015] who present algorithms to construct a global graph from the asynchronous interactions represented by communicating finite-state machines (CFSMs). An implementation of these algorithms is also available [GMC]. These works have been extended by Bocchi et al. [2014, 2015] to the timed settings as Communicating Timed Automata (CTAs) to build timed global specifications from systems of CTAs.

Acknowledgments

This work has been supported by COST Action IC1201 *Behavioural Types for Reliable Large-Scale Software Systems* (BETTY). The authors are grateful to all members of the BETTY Working Group on Programming Languages (WG3) for interesting related discussions and to the anonymous reviewer who provided precious comments and suggestions for improving an early version of this survey.

Mario Bravetti, Elena Giachino and Einar Broch Johnsen have been supported by the EU project FP7-610582 Envisage: Engineering Virtualized Services (`http://www.envisage-project.eu`).

We warmly acknowledge the enormous influence of Kohei Honda (1959-2012) on the research area covered by this survey, on the process of establishing COST Action IC1201, and on our own careers.

References

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems Languages and Applications*, pages 1015–1022. ACM, 2009.

David S. Allison, Miriam A. M. Capretz, Hany F. E. L. Yamany, and Shuying Wang. Privacy protection framework with defined policies for service-oriented architecture. *Journal of Software Engineering and Applications*, 5 (3):200–215, 2012.

Nuno Alves, Raymond Hu, Nobuko Yoshida, and Pierre-Malo Deniélou. Secure execution of distributed session programs. In *Proceedings of the 3rd Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software*, volume 69 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–11, 2010.

Davide Ancona, Sophia Drossopoulou, and Viviana Mascardi. Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason. In *Proceedings of the 10th International Workshop on Declarative Agent Languages and Technologies*, volume 7784 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 2012.

Davide Ancona, Matteo Barbieri, and Viviana Mascardi. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1377–1379. ACM, 2013a.

Davide Ancona, Viviana Mascardi, and Matteo Barbieri. Global types for dynamic checking of protocol conformance in multi-agent systems. Technical report, Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi, Università di Genova, 2013b.

John Langshaw Austin. *How to Do Things with Words*. Oxford: Clarendon Press, 1962.

Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–244. ACM, 2008.

Fabio L. Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.

Giovanni Bernardi and Matthew Hennessy. Mutually testing processes - (extended abstract). In *Proceedings of the 24th International Conference on Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2013.

Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Session and Union Types for Object Oriented Programming. In Rocco De Nicola, Pierpaolo Degano, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 659–680. Springer-Verlag, 2008a.

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *Proceedings of the 19th International Conference on Concurrency Theory*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008b.

Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Deriving session and union types for objects. *Mathematical Structures in Computer Science*, 23(6):1163–1219, 2013.

Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, pages 124–140. IEEE, 2009.

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–320. ACM, 2007.

Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *Proceedings of the 8th International Federated Conference on Distributed Computing Techniques*, volume 7892 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2013.

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *Proceedings of the 25th International Conference on Concurrency Theory*, volume 8704 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2014.

Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *Proceedings of the 26th International Conference on Concurrency Theory*, volume 42 of *Leibniz International Proceedings in Informatics*, pages 283–296. Schloss Dagstuhl, 2015.

Viviana Bono and Luca Padovani. Typing Copyless Message Passing. *Logical Methods in Computer Science*, 8:1–50, 2012.

Viviana Bono, Chiara Messa, and Luca Padovani. Typing Copyless Message Passing. In *Proceedings of the 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.

Viviana Bono, Luca Padovani, and Andrea Tosatto. Polymorphic Types for Leak Detection in a Session-Oriented Functional Language. In *Proceedings of the 8th International Federated Conference on Distributed Computing Techniques*, volume 7892 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2013.

Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

Michele Boreale and Mario Bravetti. Advanced mechanisms for service composition, query and discovery. In Martin Wirsing and Matthias M. Hölzl, editors, *Results of the SENSORIA Project*, volume 6582 of *Lecture Notes in Computer Science*, pages 282–301. Springer, 2011.

John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Proceedings of the 10th International Symposium on Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

John Boyland. Fractional permissions. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2013.

BPMN. Business Process Model and Notation. `http://www.omg.org/spec/BPMN/2.0/`, 2011.

Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of ACM*, 30:323–342, 1983.

Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.

Mario Bravetti and Gianluigi Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In Roberto Bruni and Karsten Wolf, editors, *Proceedings of the 5th International Workshop on Web Services and Formal Methods*, volume 5387 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2008a.

Mario Bravetti and Gianluigi Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae*, 89(4):451–478, 2008b.

Mario Bravetti and Gianluigi Zavattaro. Service discovery and composition based on contracts and choreographic descriptions. In Guadalupe Ortiz and Javier Cubo, editors, *Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solutions*, pages 60–88. IGI Global, 2013.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21th International Conference on Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

Joana Campos and Vasco T. Vasconcelos. MOOL. Available at `http://gloss.di.fc.ul.pt/mool/`, accessed May 21, 2016.

Joana Campos and Vasco T. Vasconcelos. Channels as objects in concurrent object-oriented programming. In *Proceedings of the 3rd Workshop on Programming Language Approaches to Concurrency and communicationcEntric Software*, volume 69 of *Electronic Proceedings in Theoretical Computer Science*, pages 12–28, 2010.

Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theoretical Computer Science*, 410:142–167, 2009.

Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 263–274. ACM, 2013.

Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming, 2006. Available at `http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf`.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems*, 34(2):8, 2012.

Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *Proceedings of the 25th International Conference on Concurrency Theory*, volume 8704 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2014.

Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A Theory of Contracts for Web Services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8(1), 2012.

CDL. W3C Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`, 2002.

Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM, 2005.

Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *Trustworthy Global Computing*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2012.

Chor. Programming Language. Available at `http://www.chor-lang.org/`, accessed May 21, 2016.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *Proceedings of the 9th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2007.

Ricardo Corin and Pierre-Malo Deniélou. A protocol compiler for secure sessions in ML. In *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2008.

Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.

Silvia Crafa and Luca Padovani. The Chemical Approach to Typestate-Oriented Programming. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 917–934. ACM, 2015.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 139–150. ACM, 2012.

Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

Romain Demangeon and Kohei Honda. Nested protocols in session types. In *Proceedings of the 23rd International Conference on Concurrency Theory*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012.

Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 46(3):197–225, 2015.

Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 435–446. ACM, 2011.

Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *Proceedings of the 21st European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.

Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013.

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.

Jolie development team. Jolie Programming Language. Available at `http://www.jolie-lang.org/`, accessed May 21, 2016.

Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. I_{doos}: a Distributed Object-Oriented language with Session types. In *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 299–318. Springer, 2005.

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 207–245. Springer-Verlag, 2007.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and Session Types. *Information and Computation*, 207(5):595–641, 2009.

Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, and Mario Coppo. Amalgamating the Session Types and the Object Oriented Programming Paradigms. In *Proceedings of the Workshop on Multiparadigm Programming in Object-Oriented Languages*, 2007.

Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24. ACM, 2002.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 177–190. ACM, 2006.

MPI Forum. *MPI: A Message-Passing Interface Standard—Version 3.0*. High-Performance Computing Center Stuttgart, 2012.

Luca Fossati, Raymond Hu, and Nobuko Yoshida. Multiparty session nets. In *Trustworthy Global Computing*, volume 8902 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2014.

Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Available at `http://www.fipa.org/specs/fipa00061/SC00061G.html`, accessed May 21, 2016.

Cédric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. In Rajeev Alur and Doron Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2004.

Ronald Garcia, Eric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 2014.

Simon Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

Simon Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

GMC. From communicating machines to graphical choreographies. Available at `https://bitbucket.org/julien-lange/gmc-synthesis`, accessed 21 May 2016.

GTV. Global Types Verification. Available at `http://www.disi.unige.it/person/MascardiV/Software/globalTypes.html`, accessed 21 May 2016.

Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284. ACM, 2008.

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Proceedings of the 7th International Conference on Distributed Computing and Internet Technology*, volume 6536 of *Lecture Notes in Computer Science*, pages 55–75. Springer, 2011.

Kohei Honda, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Verification of mpi programs using session types. In *Proceedings of the 19th European MPI Users' Group Meeting on Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 291–293. Springer, 2012.

Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, and Nobuko Yoshida. Structuring communication with session types. In Gul A. Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura, editors, *Concurrent Objects and Beyond - Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, volume 8665 of *Lecture Notes in Computer Science*, pages 105–127. Springer, 2014.

Raymond Hu and Nobuko Yoshida. Hybrid session verification through API generation. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016.

Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010.

Raymond Hu, Rumyana Neykova, Nobuko Yoshida, and Romain Demangeon. Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *Proceedings of the 4th International Conference on Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 130–148. Springer, 2013.

Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in Haskell. In *Proceedings of the 3rd Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software*, volume 69 of *Electronic Proceedings in Theoretical Computer Science*, pages 74–91, 2010.

Svetlana Jakšić and Luca Padovani. Exception Handling for Copyless Messaging. In *Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 151–162. ACM, 2012.

Svetlana Jakšić and Luca Padovani. Exception Handling for Copyless Messaging. *Science of Computer Programming*, 84:22–51, 2014.

Nicholas R. Jennings, Katia P. Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Xplore: IEEE Internet Computing*, 11(6):60–67, 2007.

Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Mungo. `http://www.dcs.gla.ac.uk/research/mungo`, 2015.

Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*, pages 323–332. IEEE, 2008.

Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232. ACM, 2015.

Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Protocol-based verification of message-passing parallel programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 280–298. ACM, 2015.

Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco T. Vasconcelos, and Nobuko Yoshida. Specification and verification of protocols for MPI programs. Available at `http://www.di.fc.ul.pt/~vv/papers/marques.martins_specification-verification-mpi.pdf`, 2013a.

Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, and Nuno Martins. Towards deductive verification of MPI programs against session types. In *Proceedings of the 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, volume 137 of *Electronic Proceedings in Theoretical Computer Science*, pages 103–113, 2013b.

Viviana Mascardi and Davide Ancona. Attribute global types for dynamic checking of protocols in logic-based multiagent systems. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), 2013.

James Mayfield, Yannis Labrou, and Tim Finin. Evaluation of KQML as an agent communication language. In *Proceedings of the Workshop on Agent Theories, Architectures, and Languages*, volume 1037 of *Lecture Notes in Computer Science*, pages 347–360. Springer, 1995.

Jan Mendling and Michael Hafner. From inter-organizational workflows to process execution: Generating bpel from ws-cdl. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, volume 3762 of *Lecture Notes in Computer Science*, pages 506–515. Springer, 2005.

Filipe Militão. Design and implementation of a behaviorally typed programming system for web services. Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia. Available at `http://run.unl.pt/handle/10362/1792`, accessed May 21, 2016.

Filipe Militão and Luís Caires. An exception aware behavioral type system for object-oriented programs. In *Proceedings of the Simpósio de Informática*, 2009. Available at `http://www.cs.cmu.edu/~foliveir/papers/corta2009.pdf`.

Fabrizio Montesi. Process-aware web programming with Jolie. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 761–763. ACM, 2013a.

Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013b. Available at `http://www.fabriziomontesi.com/files/choreographic_programming.pdf`.

Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *Proceedings of the 24th International Conference on Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2013.

Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing services with Jolie. In *Proceedings of the 5th IEEE European Conference on Web Services*, pages 13–22. IEEE Computer Society, 2007.

Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21 (12):993–999, 1978.

Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *Proceedings of the 16th International Conference on Coordination Models and Languages*, volume 8459 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2014.

Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In *Proceedings of the 4th International Conference on Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer, 2013.

Nicholas Ng and Nobuko Yoshida. Pabble: parameterised Scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015.

Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. Safe parallel programming with session java. In *Proceedings of the 13th International Conference on Coordination Models and Languages*, volume 6721 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 2011.

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session c: Safe parallel programming with message optimisation. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2012.

Nicholas Ng, Jose G. F. Coutinho, and Nobuko Yoshida. Protocols by default: Safe MPI code generation based on session types. In *Proceedings of the 24th International Conference on Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 212–232. Springer, 2015.

OASIS. Web Services Business Process Execution Language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, 2007.

Pabble-MPI. MPI Generation Framework. Available at `https://github.com/sessionc/pabble-mpi`, accessed May 21, 2016.

Luca Padovani. *Contract-based Discovery and Adaptation of Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 213–260. Springer, 2009.

Luca Padovani. Contract-Based Discovery of Web Services Modulo Simple Orchestrators. *Theoretical Computer Science*, 411:3328–3347, 2010.

Luca Padovani. A Simple Library Implementation of Binary Sessions. Technical Report hal-01216310, Dipartimento di Informatica, Università di Torino, Italy, 2015. Available at `https://hal.archives-ouvertes.fr/hal-01216310`.

Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 25–36. ACM, 2008.

Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web*, pages 973–982. ACM, 2007.

Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

Arend Rensink and Walter Vogler. Fair testing. *Information and Computation*, 205(2):125–198, 2007.

Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. In *Proceedings of the 1st Workshop on Advances in Message Passing*, 2010.

Scribble. Available at `http://www.scribble.org`, accessed May 21, 2016.

Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007.

Singularity OS. Available at `http://singularity.codeplex.com/`, accessed May 21, 2016.

SJ. Session J. Available at `http://code.google.com/p/sessionj/`, accessed May 21, 2016.

Guy L. Steele. Parallel programming and parallel abstractions in fortress. In *Proceedings of the 8th International Symposium on Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 1–1. Springer, 2006.

Zachary Stengel and Tevfik Bultan. Analyzing Singularity Channel Contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 13–24. ACM, 2009.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 713–732. ACM, 2011a.

Joshua Sunshine, Sven Stork, Karl Naden, and Jonathan Aldrich. Changing state in the Plaid language. In *Proceedings of the Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 37–38. ACM, 2011b.

Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013.

Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1–2):64–87, 2006.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving Copyless Message Passing. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2009.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking Heaps That Hop with Heap-Hop. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 275–279. Springer, 2010.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483. Springer, 2011.

Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, 2007.